Error-Correcting Codes

Arjun Kolani

March 2024

1 Introduction

1.1 History of Error-Correction

Error-correcting codes are a beautiful application of maths that has revolutionized the transmission of data. Emerging as a pivotal aspect of coding and information theory in the 20^{th} century, these mathematical tools have become indispensable in our digital age. The transmission of data can be unreliable and messages can become corrupted. As usual, when there's a problem, there's a creative mathematical solution.

The first practical error-correcting code, Hamming code, was discovered by Richard Hamming at Bell Labs and meant that single-bit errors in the transmitted data could be automatically corrected. His colleague at Bell Labs, Claude Shannon, at the same time was laying down the theoretical foundation for information theory which then propelled the development of error-correcting codes forwards. After Hamming codes, many further codes were invented for correcting multiple errors in transmission such as Golay codes and Reed-Solomon codes. Reed-Solomon codes are used in CDs and DVDs which is why you can scratch a CD a lot and it still works - some pretty handy mathematics!

1.2 Errors in transmission

Typically data is stored and transmitted in binary, 1's and 0's, as this is what digital computers can understand best. Sometimes, when data is transmitted, the data can become corrupted by *noise*. Noise is the term that refers to unpredictable changes in the signal that can be caused by electromagnetic radiation for example. There are 2 general types of errors that can happen to the data:

- **Erasure Errors:** When you lose a bit e.g. a scratch on a CD can remove the bits.
- **Bit-flip errors:** When the noise corrupts the signal and flips a bit from a 1 to a 0 or vice-versa.

These errors mean we need to find a way of knowing if an error has occurred i.e. *error detection* and also how to know what the error was so we can correct it i.e. *error correction*.

1.3 Error detection

The task of error detection is much more trivial than error-correction. For example, we'll discuss a simple method of error-detection as we introduce the idea of parity that will be useful later for Hamming Codes.

If we add a single bit at the end of our message for sending, we can actually detect any odd number of errors. *Parity* in mathematics refers to whether an integer is odd or even and here it will refer to whether there are an odd or even number of 1s in the binary message. We can use this by making sure that valid messages have to have an even parity - an even number of 1's - otherwise, it is not a valid message. So, when we send a message, we send an extra parity bit which is set so the parity of the whole message is even.

For example, if we were sending the data 1011, we would add for our extra parity bit a 1 so that there would be an even number of 1's. It is important to note that **any bit-flip changes the parity**. This means that:

- If there are an *odd* number of bit-flips, we can tell that an error occurred since the parity of the message will be odd.

- However, if there were an *even* number of errors, we would not know, since this would still lead to a valid message. This is a limitation to this method and why many more robust error-detection systems are used in reality.

One important detail is that it is never possible to always detect or correct an error since there could be so many errors that one message turns into another valid message that the system accepts. The aim of error-correction is being able to correct as many errors as possible with as few redundant bits as possible.

1.4 Naive repetitions code

At first sight, there seems to be a simple method of error-correction which is repeating the message bits. Let's say you're trying to send a bit of information which can either be 0 or 1:

- We could send a message of 00 or 11 where: $0 \equiv 00$, $1 \equiv 11$. But this fails since if there is a bit-flip such as turning 00 into 01. Then, when you receive 01, you don't know which bit was flipped and whether the message was 00 (2nd bit flipped) or 11 (1st bit flipped).
- A simple solution to this is to repeat each bit three times so the valid messages are 000 or 111 which you send. Now the receiver can correct single bit-flips since a receiver can tell what was the valid message by taking the most common digit and knowing what bit of information was sent to him (e.g. 001 –> 000).

Before we start celebrating, we see that this means we have to repeat each bit in our message three times and this gives us an information rate of 1/3 (33.3%).

$$\text{Information rate} = \frac{\text{Number of data bits}}{\text{Total number of bits}}$$

Later, we look at Hamming codes which also allow for single-bit error-correction but have the highest possible information rate.

2 Hamming Codes

2.1 Intuition

Hamming codes are a way of being able to not only detect errors but also correct single-bit errors (so we're assuming throughout this section at maximum only 1 bit could have been flipped by noise in transmission).

Firstly, let's look at how you can make the Hamming(15, 11) code. A Hamming(15,11) code means: 15 bits are sent, with 11 of those bits being actual data bits and the other 4 being redundant, but needed for error-correction. Let's take an example message of 11001011011 as our 11 bit message we're trying to send.

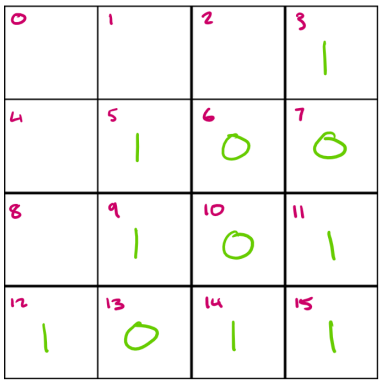

Figure 1: Data bits placed in order in 15 bit block leaving out the 0^{th} bit and the powers of 2

We take our 11 bits and place them into a 16 bit block, which is indexed with values from 0 to 15, as shown in Figure 1.

The 0^{th} bit doesn't actually exist but it helps to act like it does so that the maths works and then we can just not include it in the message we send. We place our 11 bits in order, but we leave the bits with an index of a power of 2 to act as our parity check bits that can locate where a single-bit error took place. Our parity bits have a group associated with them which we show in Figure 2 as each highlighted section:

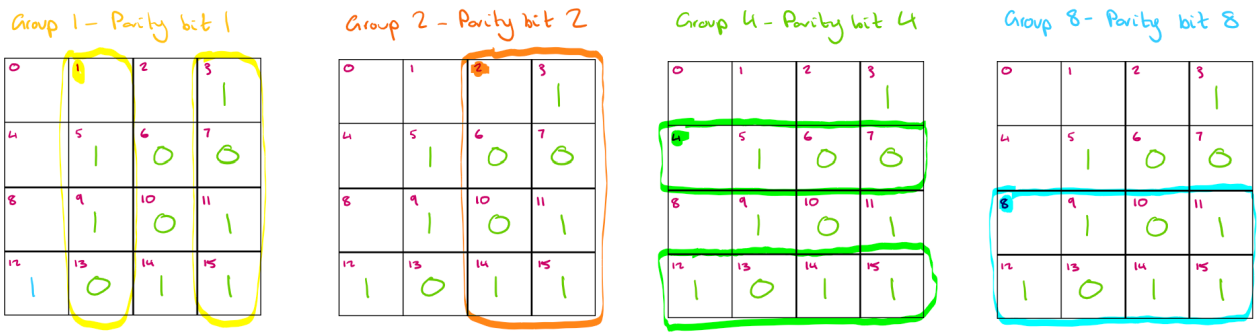

Figure 2: Groups for each parity bit

We set our parity bits so that the parity of the number of 1s in that group is even. For example, when setting parity bit 1, we count the number of 1s in the group, this is the leftmost block in Figure 2, and see there are 5 1s which mean we add a 1 to parity bit 1 so the parity of the whole group is even. Now let's take parity bit 2, there are 4 1s in this group so we set the parity bit to 0; this ensures there's an even number of 1s in the group. We can do this for the other examples leading to a completed block illustrated in Figure 3.

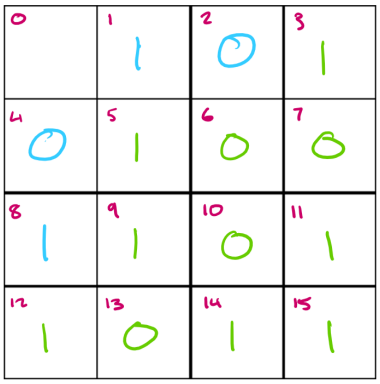

Figure 3: Data and parity bits placed in block

This means we have finished making our message and can send this block to the receiver. The message sent would be 110010110111001. Note that the first 11 bits are the original data bits and we've added the additional parity bits to the end. The receiver can place the parity bits back into the power of 2 indices if they want.

Let's say a single bit flips and our message becomes 100010110111001.
Now we can use a procedure to locate our error without any knowledge of what the sent message was. We can re-arrange this into our 16 bit block like before leaving us with the block in Figure 4 on the next page.

Rough Method for correcting the error

1. **Parity Checks:** Check parity of 4 groups.
2. **Is the error in the group?:** If the parity of the group is odd, the error's in the group, else, it's not.
3. **Find the error's location:** We can take the information from our 4 parity checks to narrow down the location to a single square on our block.

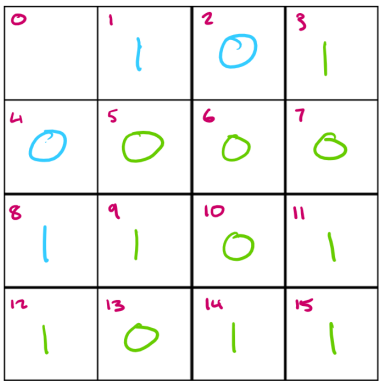

Figure 4: Our received message with some error, bit-fip, in it

Applying this method - with explanations
To find the error, we do a parity check on all the 4 groups. First we check the parity of group 1 (the group with parity bit 1) and see that there are an odd number of 1s, 5 1s. This means we know the error is in group 1 as the only way our parity could change was if a bit flipped. If the parity was even, we would know our error was in the opposite of group 1 - all the squares not in group 1 - since we're assuming at most 1 error can occur or there is no error. Initially in this explanation, we'll assume there was a single error and at the end we'll consider what happens if there is no error.

Next, we check the parity of group 2 (the group with parity bit 2) and see that it's even. This means the error isn't in group 2 and so must be in the squares not in group 2. If we take a step back, we can see that these two parity checks have actually located the column of the error since the only column which is the intersection of these groups is the column with indices 1, 5, 9, 13. With these 2 parity checks we can locate the column of the error no matter where the error occurred.

We can do a similar process with the groups with parity bits 4 and 8 to find the row of the error. Group 4 has an odd parity and Group 8 has an even parity. The intersection of group 4 and not group 8 is the row with indices 4, 5, 6, 7. The intersection of this column and row is square 5 and so we know that (assuming there was at most 1 bit-flip) we can flip the bit at this square to 1 and get the original message.

It seems that we've ignored the possibility of there being no errors but if we followed the same process if there were no errors, we'd find our point to be the intersection of not group 1, not group 2, not group 4 and not group 8. We can see from our previous image of the groupings, Figure 2, that there is only one square that isn't in any of the groups and that is square 0 which doesn't actually exist but we've just added it for ease of calculation. This means that if we get our error at square 0 we can say there were no errors.

2.2 Formalism

In this section, we'll show how this generalizes for message sizes being sent of $2^n - 1$ bits and how you only need n parity bits. We'll also prove that this always works when finding the 1-bit error.

Let our message size be of $2^n - 1$ bits. We know we need n parity bits as this is the number of powers of 2 that are less than $2^n - 1$. In the intuition section, we didn't formalise how the groups were assigned to parity bits. We can do this by representing the indices we use in binary, example shown in Figure 5 on next page.

We can create a rule for assigning the parity groups. Firstly, we can see that each parity bit index has only a single 1 in its binary form as they are a power of 2. This can be proved by knowing that shifting a binary digit left (adding a 0 at the end) multiplies the number by 2 and so any power of 2 can be made by starting with a 1 in binary (e.g. 0001) and shifting left a certain number of times. This proves there is only a single 1.

Now we assign squares to a group by saying that a group contains all the squares with indices in binary that have a 1 in the same location as the binary of the parity bit index. For example, the group with parity bit 1, which is

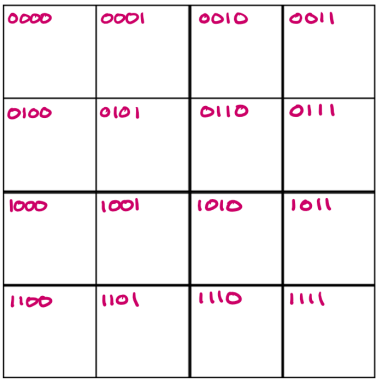

Figure 5: Block with indices in binary form

0001 in binary, has squares with a 1 in the right-most digit which is all the odd numbers as shown in the grouping diagram in Figure 2.

As the indices of parity bits only have a single 1 in their binary, they can only belong to a single group - using the rules for groups from above. Now we'll prove we can always locate the error.

In each parity check, we decide whether the error is in that specific group or not. We can do this because, assuming there is at most a single bit-flip error, if the parity of the group has changed (is odd), then an error must have happened in the group and if the parity is still even, then the error is not in the group or not happened (which we treat as happening in the 0^{th} bit). Now with our formalised grouping system, we can change our method of finding the error into deciding whether the index of the error in binary form has a 1 in the location of the 1 in the parity bit index we're checking. This is because if the error is in the group, the error's binary index has a 1 in this location since this is a requirement for being part of the group and otherwise it has a 0. This means we can construct the binary index of the error by adding a 1 or a 0 after each parity check until we're left with our final index in binary. If this sounded like gibberish, a demo is shown below Figure 6.

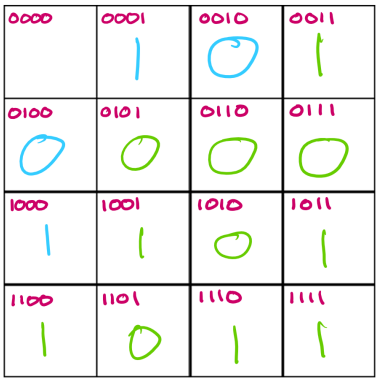

Figure 6: Block with an error and with indices in binary form

To demonstrate this process take the example from before, Figure 6.

Parity check on group 1 (0001) –> odd –> Error is in group –> Error's binary index is ...1
Parity check on group 2 (0010) –> even –> Error isn't in group –> Error's binary index is ..01
Parity check on group 4 (0100) –> odd –> Error is in group –> Error's binary index is .101
Parity check on group 8 (1000) –> even –> Error isn't in group –> Error's binary index is 0101

Therefore, 0101 is the square where the error was.

As in each of our n checks, we can add a 1 or a 0, we can construct a final index of anywhere in the whole 2^n bit block and so can locate any error.

2.3 Packing Codewords

A codeword is a term used when talking about error-correcting codes. It refers to the final message, after adding redundant bits, that is sent and so from the receiver's point of view it's any message you take as valid. The difference between the message and the codeword is that the message is the data bits you want to send and the codeword is the data bits and the redundant bits necessary.

Let's take a $2^n - 1$ bit codeword where n is a positive integer. A reasonable question to ask is, "What is the maximum number of codewords you can fit in so that you can do error-correction on at most a single bit-flip?" which loosely corresponds to "Are Hamming codes the most efficient and so have the highest information rate possible?". Let's have a look:

Let x be the number of codewords and assume that there are at most 1 bit-flips. The number of messages you can receive from a codeword that is sent is 2^n. This is because the codeword has $2^n - 1$ bits and so any one of the $2^n - 1$ bits could be flipped. Also we add an extra 1 from the possibility that there were no flips and you receive the original codeword. We can now make a set for each codeword which contains all the 2^n possible messages you can receive from that codeword. We can say that for a functional error-correcting there won't be any intersection between the sets, i.e. no message can be in multiple sets. This is because if there was a message in multiple sets then you wouldn't be able to know, when you receive the message, which of the codewords was sent which would mean the error-correction scheme isn't capable of dealing with 1 bit-flip errors. This means that in total, the number of messages that have to be represented by the $2^n - 1$ bits sent is at least $x \times 2^n$ as these are all the messages that could be received.

$$
\begin{aligned}
&\therefore x \times 2^n \leq \text{No. of } 2^n - 1 \text{ bit words} = 2^{2^n - 1} \\
&\implies x \times 2^n \leq 2^{2^n - 1} \\
&\implies x \leq 2^{2^n - 1 - n} \\
&\implies log_2(x) \leq 2^n - 1 - n
\end{aligned}
$$

x, the number of codewords, represents the number of messages you can send with $2^n - 1$ bits and so the number of data bits, not the redundant bits, is $\log_2(x)$ since each bit has 2 possible values. $\therefore$ The upper bound for the number of data bits is $2^n - 1 - n$ and so the lower bound for the number of parity bits needed is $2^n - 1 - (2^n - 1 - n) = n$.

If we look back at our generalized Hamming Code, we only used n parity bits for a message size of $2^n - 1$. Now we've proven that this is the minimum number of parity bits you need and so we've proven Hamming Codes have the highest information rate possible for 1 bit-flip errors.

2.4 Hamming Codes Overview

Hamming codes were a very important discovery as they showed that efficient solutions were possible for error-correcting codes. We've seen that Hamming Codes are as efficient as possible for single-bit errors but we can see how powerful they are if we consider a total message size of $2^{20} - 1 = 1,048,575$ bits and using our formula from before we know that of these bits, you only need 20 to be parity bits. This gives an information rate (to 5s.f.) of:

$$1048555/1048575 = \textbf{99.998\% !!!}$$

Despite this impressive result, in the real world, sending a message of chunks of this size increases the chance of multiple errors occurring on each chunk which can't be handled by Hamming codes. Also in practice, errors tend to happen in bursts, meaning close together. One way of countering this is by interlacing the bits of the blocks so that consecutive errors happen to different blocks and can be handled. Even with this, multiple errors are too common for Hamming codes being practical and so more complex codes like Reed-Solomon and Turbo codes tend to be used for transmission. Some applications of Hamming codes include DRAM memory chips and satellite communication hardware.

3 Reed-Solomon Codes

Reed-Solomon codes are a type of error-correcting code that can correct erasure errors, when bits are lost in transmission, and multiple bit-flip errors. In this essay we will just discuss erasure errors but the general idea is the same for bit-flip errors with a few adjustments. Reed-Solomon codes are a perfect demonstration of how an abstract part of pure mathematics can be applied to the real world.

3.1 Polynomials and Lagrange interpolation

The backbone of Reed-Solomon codes is the idea of utilising a characteristic of polynomials. This is that:

Given $n+1$ points $(x_1, y_1), ..., (x_{n+1}, y_{n+1})$ with all x_i distinct, there is a unique polynomial, $p(x)$, of degree at most n that passes through these points, meaning $p(x_i) = y_i$ for $1 \leq i \leq n+1$.

We'll go about proving this by first showing a method of finding such a polynomial - Lagrange interpolation - and then we'll prove that no other polynomial with a degree $\leq n$ can pass through these points.

3.1.1 Lagrange interpolation

The first step is to make a series of Lagrange polynomials. A Lagrange polynomial, $l_i(x)$, has the property that $l_i(x_i) = 1$ and $l_i(x_j) = 0$ for all j where $0 \leq j \leq n+1$ and $j \neq i$. Essentially this is a polynomial, l_i that passes through the x-axis as at all points apart from at x_i where it passes through y = 1. Consider the polynomial

$$q_i(x) = \prod_{j \neq i} (x - x_j)$$

This notation here refers to the product of n brackets $(x - x_1)(x - x_2)...(x - x_{n+1})$ not including the bracket with $(x - x_i)$. The range for j has not been included when we write out the product symbol but it can just be assumed, whenever we write it, to go over all the distinct inputs apart from x_i. We can see that this polynomial has roots for all our x values apart from x_i. To make this into a Lagrange polynomial we need to make it so that inputting in x_i returns an output of 1. At the moment if we input in x_i we'll get $q_i(x_i) = \prod_{j \neq i}(x_i - x_j)$. We know that this a non-zero constant since our x_is are distinct and so if $j \neq i$, then $x_i \neq x_j$. This means we can make our Lagrange polynomial as:

$$l_i(x) = \frac{q_i(x)}{q_i(x_i)} = \frac{\prod_{j \neq i}(x - x_j)}{\prod_{j \neq i}(x_i - x_j)}$$

The reason this works is for all inputs not equal to x_i, this will be a root for the polynomial on the numerator and dividing by the constant on the denominator will still leave you with 0 (we know the constant is non-zero so we can divide by it). In addition, for input x_i, the numerator and the denominator will be the same so $l_i(x_i) = 1$.

Now we can use these Lagrange polynomials as the foundation for constructing the polynomial we were initially interested in. This was a polynomial of degree at most n that passes through $n+1$ points $(x_1, y_1), ..., (x_{n+1}, y_{n+1})$ with all x_i distinct. We can now do the final magic step and make our polynomials as:

$$p(x) = \sum_{i=1}^{n+1} y_i l_i(x)$$

To understand why this works let's consider some general input x_a where $1 \leq a \leq n+1$. We know $l_i(x) = 0$ where $i \neq a$ and so we can see that $p(x_a) = y_a l_a(x_a)$ as all the other terms of the summation go to 0. We also know from before that a property of Lagrange polynomials means that $l_a(x_a) = 1$ and substituting this in leaves us with $p(x_a) = y_a$ which proves that this polynomial passes through the $n+1$ points as our input x_a could be any of the inputs.

Take the points (1, 2), (3, 2) and (4, -1). We can do the process of Lagrange interpolation to construct the polynomial of degree 2 to pass through these points. This can be seen in Figure 7 on the next page where the polynomials are on the left.

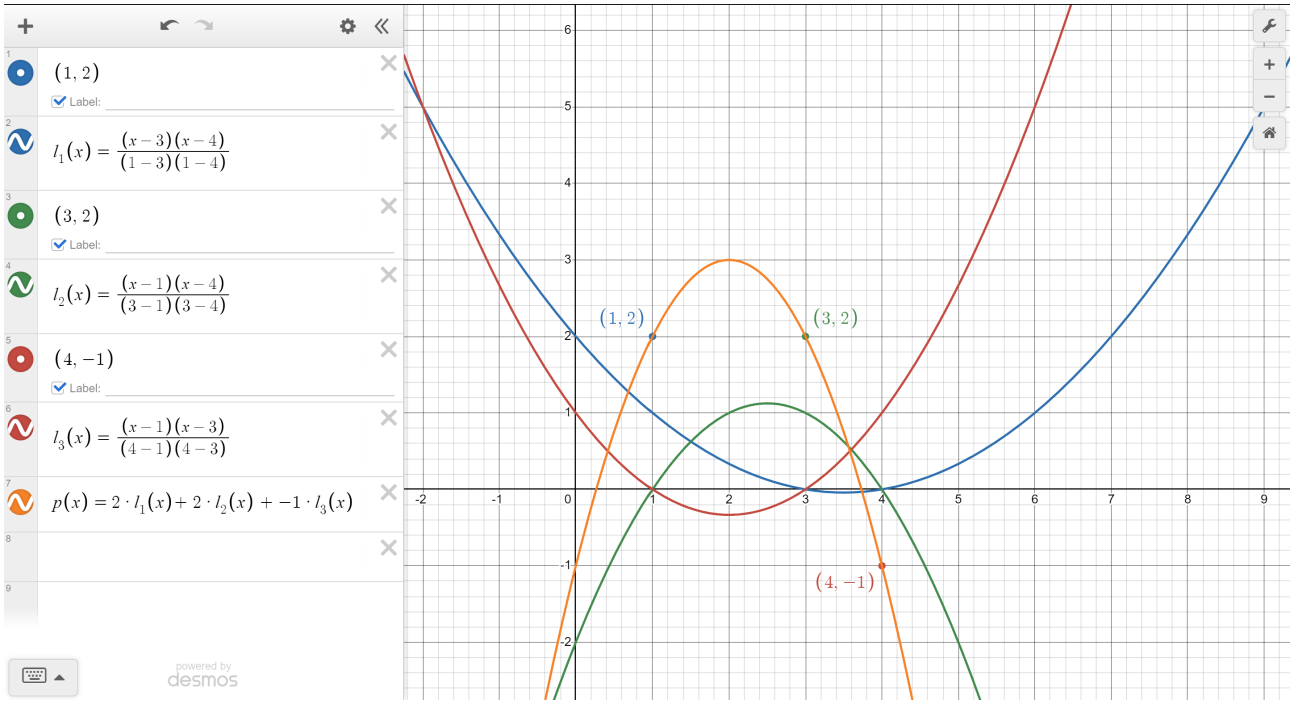

Figure 7: Graph of polynomials generated with Desmos

Finally, we just have to prove that our final polynomial $p(x)$ is of degree at most n. We can do this by proving that each Lagrange polynomial has degree of at most n. $l_i(x) = \frac{\prod_{j \neq i}(x - x_j)}{\prod_{j \neq i}(x_i - x_j)}$. The denominator is a constant term and the numerator will be the product of n brackets of $(x - x_j)$ where x_j is constant so we can see that each Lagrange polynomial has a degree of n. Then, since $p(x)$ is the summation of a constant term multiplied by a Lagrange polynomial we can conclude that $p(x)$ has degree of at most n.

3.1.2 Proof polynomial is unique

We can prove by contradiction that there is only 1 polynomial of degree at most n for passing through our previously mentioned distinct $n+1$ points. Suppose there is another polynomial of degree at most n, $f(x)$, that passes through these points. Then we know $p(x_i) = f(x_i) = y_i$ for $1 \leq i \leq n+1$. Let $r(x) = p(x) - f(x)$, we know $r(x) \neq 0$ because we've assumed that $p(x)$ and $f(x)$ are different polynomials. The degree of $r(x)$ is at most n since $p(x)$ and $f(x)$ have degrees of at most n.

However, if we take our previous equation of $p(x_i) = f(x_i)$ and re-arrange to get $p(x_i) - f(x_i) = 0$ which is equivalent to $r(x_i) = 0$ for all i such that $1 \leq i \leq n+1$, we see that this equation implies there are $n+1$ roots for $r(x)$. But, there is a theorem, which we haven't proved here but you can assume for this essay, that a non-zero polynomial of degree n has at most n roots. This means there's a contradiction as our non-zero polynomial of $r(x)$ is of degree at most n but has $n+1$ roots. $\therefore$ Another distinct polynomial can't exist meaning that our polynomial of having degree at most n is unique for passing through $n+1$ points.

3.2 Finite Fields

In doing Lagrange interpolation and proving that the polynomial is unique, we've been working with coefficients, values of x and y chosen from the real numbers. We could also do this with choosing just from rational and complex numbers. This is because for Lagrange interpolation and proving the polynomial is unique, the only property of numbers you need is that you can add, subtract, multiply and divide (not by 0) a pair of numbers from the group you're choosing from and end up with numbers from that group. This is a simple description for the algebraic structure of a field where these operations are well-defined. A more formal definition of a field is any set of elements that satisfy the field axioms in Figure 8.

name	addition	multiplication
associativity	$(a + b) + c = a + (b + c)$	$(a\,b)\,c = a\,(b\,c)$
commutativity	$a + b = b + a$	$a\,b = b\,a$
distributivity	$a\,(b + c) = a\,b + a\,c$	$(a + b)\,c = a\,c + b\,c$
identity	$a + 0 = a = 0 + a$	$a \cdot 1 = a = 1 \cdot a$
inverses	$a + (-a) = 0 = (-a) + a$	$a\,a^{-1} = 1 = a^{-1}\,a$ if $a \neq 0$

Figure 8: Axioms for a field, [3]

Subtraction and division are the additive and multiplicative inverses. Real, rational and complex numbers are examples of fields but integers are not. This is because when you do division on a pair of integers you won't always get an integer (e.g. $5/2 = 2.5$).

When applying our polynomial concepts into error-correcting codes we need to restrict our elements as we're working with limited memory/bits. This is where the idea of finite fields/Galois fields comes in. These are fields with a finite number of members and to create a finite field we can use modular arithmetic and apply modulo p, where p is a prime. Applying modulo p means, for a value, you take the remainder when you divide by p. Working with numbers in modulo p means we're working over a finite field and so it turns out that Lagrange interpolation and the polynomial being unique also hold with modulo p.

This fact proves to be incredibly useful since when making our error-correcting codes we want to compute extra values of a polynomial and if these values are extremely high they can't be stored with a limited number of bits.

In the finite field **modulo p**, the identity is 1 since $a \times 1 \equiv a\ mod\ p$. When p is prime, it is a fact that there is an inverse for every value in modulo p. This means any value such as a has some multiplicative inverse f in modulo p such that $a \times f \equiv 1 \bmod p$. The definitions of identity and inverse used here can be checked from Figure 8.

An important part of this finite field for the next part is the idea of a multiplicative inverse. The reason division by a non-zero value is defined in modulo p is we take it as a multiplicative inverse and there is a multiplicative inverse for every term in the finite field. The statement $x/2$ is the same as $x \times 2^{-1}$ where 2^{-1} is the inverse of 2 - we're taking division as the inverse of multiplication. The inverse of 2 in modulo 7 is 4 since $2 \times 4 \equiv 1 \bmod 7$. This means we can say the statement is also equivalent to $x \times 4 \pmod 7$. This is how we'll convert the division in our Lagrange polynomials into multiplication over our finite field.

This was meant to be a brief explanation of finite fields since a rigorous exploration of this topic deserves its own essay.

3.3 Fixing Erasure Errors

Let's say you want to send a message which is made up of n packets, blocks of data which we treat here in denary but in reality are made of bits (binary), across an unreliable channel. You might lose some packets in transmission. But there's no need to panic as we'll see how an extra k redundant packets protect our message for up to k packets being lost.

We assume that each packet has a header so you know the order of the packets and which packets were lost. Furthermore, we will be working over the finite field of modulo p meaning coefficients, inputs and values are reduced modulo p. We'll pick p to be a prime so that it's larger than any of the values in the message packets and also such that $n + k \leq p$ which ensures all our distinct inputs can be uniquely included with the restriction of modulo p. However, we also need p to be small enough so possible values are restricted so to what the available number of bits can represent.

We denote the message to be sent as packets m_1, m_2, ..., m_n. We know from before that we can reconstruct a unique polynomial, $p(x)$, of degree at most $n-1$ such that $p(i) = m_i$ for all i where $1 \leq i \leq n$ with Lagrange interpolation. We'll then denote the codeword we actually send to be packets $c_1 = p(1)$, $c_2 = p(2)$, ..., $c_{n+k} = p(n+k)$ - we've computed an extra k values above n from the polynomial $p(x)$. We would also to need to apply modulo p to these extra values. The point of these redundant bits is that if we lose at most any k packets, we'll still be left with at least n packets that have a unique polynomial of degree at most $n-1$ passing through them which we can find with Lagrange interpolation.

But wait! This polynomial has to be the same as $p(x)$ since $p(x)$ went through these points before and it was unique. This means we can just compute $p(x)$ for the indices of our missing packets and we're done. We can also be sure that modulo p won't mess up our values since we picked p so that it was greater than these values at the beginning so we know our answer has to be restricted to modulo p.

Let's give a demonstration on a simple example. Let $m_1 = 3$, $m_2 = 1$, $m_3 = 5$, $m_4 = 0$. We can work over modulo 7 since it meets the conditions as described before. Now we need to use Lagrange interpolation on these packets which we think of as the points (1, 3), (2, 1), (3, 5), (4, 0). This means we need to find the Lagrange polynomials $l_1(x)$, $l_2(x)$, $l_3(x)$, $l_4(x)$ and to do this we can use our formula from before of $l_i(x) = \frac{\prod_{j \neq i}(x - x_j)}{\prod_{j \neq i}(x_i - x_j)}$ and apply modulo 7. Our Lagrange polynomials are:

$$
\begin{aligned}
l_1(x) &= \frac{(x-2)(x-3)(x-4)}{(1-2)(1-3)(1-4)} \\
&= \frac{(x-2)(x-3)(x-4)}{-6} \equiv 1(x-2)(x-3)(x-4) \pmod 7 \\
l_2(x) &= \frac{(x-1)(x-3)(x-4)}{2} \equiv 4(x-1)(x-3)(x-4) \pmod 7 \\
l_3(x) &= \frac{(x-1)(x-2)(x-4)}{-2} \equiv 3(x-1)(x-2)(x-4) \pmod 7 \\
l_2(x) &= \frac{(x-1)(x-2)(x-3)}{6} \equiv 6(x-1)(x-2)(x-3) \pmod 7
\end{aligned}
$$

Note: We've used the fact that the multiplicative inverses of {-6, 2, -2, 6} are {1, 4, 3, 6} respectively when working in modulo 7. These can be checked by multiplying each pair together and seeing you get 1 after applying modulo 7 - the product of the pair is 1 more than a multiple of 7. Then we'd find our degree 3 polynomial, $p(x)$ with:

$$p(x) = 3 \cdot l_1(x) + 1 \cdot l_2(x) + 5 \cdot l_3(x) + 0 \cdot l_4(x)$$

When we expand the brackets and apply modulo 7 to our coefficients we get left with a polynomial of:

$$p(x) = x^3 + 4x^2 + 5$$

.
If you substitute our inputs and apply modulo 7 you'll get the corresponding output. For protecting against 2 erasure errors we'd add packets $p(5) = 6$ and $p(6) = 1$ and send these 6 packets off as our codeword. Now if at most 2 packets get lost we could do a similar procedure with 4 of the remaining packets to get our original polynomial of degree 3 and compute the missing inputs to recover the lost packets.

3.4 Reed-Solomon codes overview

This solution is particularly efficient as the number of extra redundant bits you send is constant to how many errors you want to be able to handle and remains constant even if you wanted to send 1,000,000 packets.

We haven't covered Reed-Solomon codes handling a corrupted message, multiple bit-flips, but we can do this and it uses the same application of polynomials with some slight differences in implementation though. Nevertheless, we explored the beauties of polynomials and Lagrange interpolation and described a method for handling erasure

errors which is applicable for CDs if bits get scratched off. General Reed-Solomon error-corrections are commonly used for digital storage and satellite communications - in 1977 NASA Voyager mission used Reed-Solomon codes to enable accurate deep-space communication!

4 Conclusion

The solutions to errors in transmission now may seem quite simple but this is typical after seeing a solution. The idea of creativity in maths is often overlooked but these two error-correction codes are fantastic illustrations of this. The study of polynomials and Galois fields seem to be only useful for pure maths but it was Reed and Solomon who were able to make the link and change the world of transmission for the better.

Perhaps the most recent and exciting progress in error-correction is its application in quantum computers where even small amounts of noise can cause quibits (quantum bits) to lose their quantum information. The first quantum error-correction codes were invented by Shor and Steane but more research in this area is required to make the idea of quantum computers closer to reality.

Bibliography

[1] 3Blue1Brown. *But what are Hamming codes? The origin of error correction*. 2020. URL: https://youtu.be/X8jsijhllIA?si=CdePWQsZZLhVHxYu.

[2] 3Blue1Brown. *Hamming codes part2: The one-line implementation*. 2020. URL: https://youtu.be/b3NxrZOu_CE?si=U7HCHRkzVp5Ut5LX.

[3] Wolfram MathWorld. *Field Axioms*. URL: https://mathworld.wolfram.com/FieldAxioms.html.

[4] Alistair Sinclair Sanjit Seshia. *Discrete Mathematics and Probability Theory, Note 8*. 2024. URL: https://www.eecs70.org/assets/pdf/notes/n8.pdf.

[5] Alistair Sinclair Sanjit Seshia. *Discrete Mathematics and Probability Theory, Note 9*. 2024. URL: https://www.eecs70.org/assets/pdf/notes/n9.pdf.

[6] Madhu Sudan. *Hamming Codes, Distance, Examples, Limits, and Algorithms*. 2017. URL: https://madhu.seas.harvard.edu/courses/Spring2017/scribe/lect01.pdf.

[7] vcubingx. *What are Reed-Solomon Codes?* 2022. URL: https://youtu.be/1pQJkt7-R4Q?si=9NbPigg5RdK0KC0l.