Error-Correcting Code

Ken Lee

Abstract

In this expository paper, we will work with a special class of code called linear code, and construct methods of checking whether or not a certain linear code C is error-correcting or not. Our criteria for error-correcting codes will be that C must detect errors and correct them. Our primary focus will be on single-bit errors during transmission, but we will also provide a basic idea of characterizing how to detect and correct multiple errors.

1 $\mathbb{F}_2$ and the Field of Bits

To talk about code, we need to talk about binary systems. In our usual representation of numbers, we have:

$$314 = 3 \cdot 10^2 + 1 \cdot 10^1 + 4 \cdot 10^0.$$

However, we can also write 314 as the sum:

$$314 = 1 \cdot 2^8 + 0 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0.$$

Generally, we can write a number $x \in \mathbb{Z}$ as the sum $x = \sum_{i=0}^{n} a_i 10^i$, where a_i are numbers in the set $\{0, .., 9\}$. This is the denary expansion we are familiar with, but there is also a way for us to write x as $x = \sum_{i=0}^{m} b_i 2^i$, where b_i are numbers in the set $\{0, 1\}$. This is what is known as *binary expansion.*

Example. The number 314 is written as 1001110110 in binary, where the nth digit in the binary representation corresponds to whether or not we add 2^{n-1}. Naturally, 1 corresponds to adding, and 0 corresponds to doing nothing.

Computers are basically a large combination of switches and electrical signals which are either on or off. Thus, the binary system becomes incredibly useful in conforming to these Boolean rules, allowing computers to do a wide range of arithmetic. The way this arithmetic is defined is as follows:

i. $0 + 0 = 0$

ii. $0 + 1 = 1$

iii. $1 + 1 = 10$

where these numbers are written in binary representation. Essentially, this just translates to:

i. $0 + 0 = 0$

ii. $0 + 1 = 1$

iii. $1 + 1 = 2$

in terms of our more familiar denary tongue.

Example. $1001 + 1011 = 10100$ in binary, which translates to $9 + 11 = 20$ in denary.

When computers send messages, they send a binary number. Next, let us define addition and multiplication for an individual bit. This is more useful than considering these operations for full binary numbers since it will be easier to work with using Linear Algebra. We define addition as the same, except for rule iii., where we take $1 + 1 = 0$ instead. For the multiplication of bits, we have:

i. $0 \cdot 0 = 0$

ii. $0 \cdot 1 = 0$

iii. $1 \cdot 1 = 1$

where we have commutativity and associativity as we do with addition.

Note that with these rules of addition and multiplication, we get that the set of bits form the field $\mathbb{F}_2$, which is great, because in order to do linear algebra, we need a field and a vector space. That vector space in this case will be the vector space $\mathbb{F}_2^n$, where this space has vectors that are essentially n-bit messages.

Let us try some basic linear algebra with $\mathbb{F}_2$ and $\mathbb{F}_2^n$:

Example. $x+y = 0$ and $x-y = 1$ is a system of equations with no solutions. This is because $-y = y$ for all $y \in \mathbb{F}_2$ by rules of addition, and thus we simplify the system of equations into $x + y = 0$ and $x + y = 1$.

Example. $x + y = 1$ implies that $x = x + (y + y) = 1 + y$.

Example. We can multiply matrices whose elements are in $\mathbb{F}_2$ the same way we multiply matrices whose elements are in $\mathbb{R}$ or $\mathbb{C}$:

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot 1 + 1 \cdot 1 + 0 \cdot 0 & 1 \cdot 0 + 1 \cdot 0 + 0 \cdot 0 \\ 0 \cdot 1 + 0 \cdot 1 + 0 \cdot 0 & 0 \cdot 0 + 0 \cdot 0 + 1 \cdot 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

Example. Let P be any matrix of any finite dimension over $\mathbb{F}_2$. $P + P = 0$ by addition in $\mathbb{F}_2$.

Now that we have gotten used to using $\mathbb{F}_2$, let us move onto methods of correcting code.

2 Basic Error-Correcting Methods

As much as we want them to be, computers are rarely perfect. Sometimes, when data is being sent back and forth, a bit of the information can change from a 1 to a 0. This can happen for a variety of reasons. One such reason is the particles in cosmic rays. A famous example of this was during a Mario 64 speedrun. Here, cosmic rays hit a Nintendo 64 and switched a bit by altering the electrical signals within the circuitry. This then caused Mario to suddenly jump up to a platform, saving a tremendous amount of time in the speedrun. It would be very hard to prevent these types of errors, but there are plenty of ways for receivers of information to know when a bit has been changed.

Before talking about these methods, let us first set some definitions.

Definition 2.1 (Code)**.** A *code* C of length n is a subset of $\mathbb{F}_2^n$. Elements $c \in C$ are called *codewords* in C, and the collection of all codewords is called a *codebook*.

Definition 2.2 (Encoding)**.** A function $e : \mathbb{F}_2^k \to \mathbb{F}_2^n$ is an *encoding map*, if all messages in $\mathbb{F}_2^k$ are sent this way to a codeword in $e(\mathbb{F}_2^k)$; i.e. if $\text{range}(e) = C$ for some code $C \subset \mathbb{F}_2^k$.

Consider the simple example of sending the message 0. We can repeat this message three times to get 000, which is represented as the vector $(0, 0, 0)$. Here, our encoding map is just $e(x) = (x, x, x)$. Let us say that this message is received as 001, and the receiver knows that only 1 bit has changed. Then obviously, the correct message can be deduced as 000, and hence 0 by the sender. However, if the receiver isn't sure that only 1 bit has changed, then they won't know if 001 is a 2-bit change of 111 or a 1-bit change of 000.

Another way of dealing with errors is with what is called a parity check. Consider the $\mathbb{F}_2^2$. Let us use the encoding map $e(x, y) = (x, x, y, y, x + y)$, where we repeat each bit twice and include what is called a *parity bit* in the right-most digit. If we send 01, then the correctly received codeword would be 00111, with parity bit 1. Suppose that due to an error, the receiver gets the codeword 00011. Knowing that at most 1 bit has been altered, the receiver knows that they should have either 00111 or 00001. However, with the parity bit, they can confirm that the correct codeword is 0011, as 0000 does not produce the parity bit 1. A neat interpretation of the parity bit is counting the number of 1s and seeing if the number is odd or even; i.e. the parity.

Proposition 2.3. *The parity bit is 1 if there is an odd amount of 1s in the message being encoded, and 0 if there is an even amount.*

Proof. Let $c \in \mathbb{F}_2^n$ be a message with an odd amount of 1s and write $c = (c_1, ..., c_n)$. Then the parity bit is calculated by $\sum_{i=1}^n c_i$, which simplifies to the sum of 1s since the $c_j = 0$s don't matter. This then further simplifies to just 1, as any pair of 1s add up to 0, and there will be one 1 that is left in the end. $\square$

Let us repeat the previous example of sending the message 01 with the encoding map $e(x, y) = (x, x, y, y, x + y)$. If the receiver gets 11001 but doesn't know that at most 1 bit has been altered this time, then the parity bit becomes useless. The received code is what would be expected from $e(1, 0)$, as the repeated bits and the parity bits are correct, and thus the receiver would conclude that 10 is the intended message. In general, whenever an even amount of bits changes, the parity doesn't, so parity checks become redundant. One way of fixing this is known as a block parity check; this is essentially a 2-dimension parity check, but we will not get into this method.

3 Linear Codes and the Generator Matrix G

There are a variety of ways of getting around the problems that arise with parity checks. One of these methods involves what is called a *Generator Matrix*, but to talk about this method, we have to use a special class of code.

Definition 3.1 (Linear Code)**.** A code $C \subset \mathbb{F}_2^n$ is called *linear code* if C is a subspace of $\mathbb{F}_2^n$ over the field $\mathbb{F}_2$.

Example. $\{0\}$ is a code, but not a linear code of $\mathbb{F}_2$. $\{0, 1\}$ is a linear code of $\mathbb{F}_2$, as it is exactly $\mathbb{F}_2$, which is a vector-space over the field $\mathbb{F}_2$.

Example. Let $\mathbf{e}_i$ represent the vector in $\mathbb{F}_2^n$ with a 1 in the ith entry, and 0 elsewhere. Then span$(\mathbf{e}_1, \mathbf{e}_2)$ is clearly a linear code of length n.

The switch from considering subsets to linear subspaces proves to be incredibly useful, as we are doing *Linear* Algebra after all. Since linear codes satisfy properties of a vector space or a subspace, there must exist some basis $\mathbf{c} = \{\mathbf{c}_1, ..., \mathbf{c}_k\}$. Then, every message that we wish to encode can be written as a linear combination of the basis vectors. This doesn't do much good unless our encoding map itself preserves this linearity; i.e. unless the encoding map is a linear transformation. There are many ways to define linear transformations with a given basis, but the most useful way defines the Generator Matrix G:

Definition 3.2 (Generator Matrix)**.** Let $C \subset \mathbb{F}_2^n$ be a linear code of length k. A linear transformation $T : \mathbb{F}_2^k \to \mathbb{F}_2^n$ is called the *Generator* of C if the columns of $G = \mathcal{M}(T)$ form a basis of C. We call the matrix G the Generator Matrix.

Note that by the definition, we have range$(G) = C$, and that G is an $n \times k$ matrix.

Let us specify linear codes C with $[n, k]$, where n is the length of the codewords, and k is the length of the messages, which is also the dimension of C.

Furthermore, note that for a codeword of $[n, k]$ code, only k bits of information really matter, as they are the k bits of the message being encoded. We shall now refer to the remaining $n - k$ bits as check bits when necessary. Examples of check bits were the parity bit or the repetition bits.

Let us consider an example of G:

Example. Consider the Generator T of the linear-$[4, 3]$ code $C = \text{span}(\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3)$. We have:

$$G = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} I_3 \\ \mathbf{0} \end{pmatrix}.$$

In this example, G is written in what is called *systematic form*, where we have $G = \begin{pmatrix} I_k & A \end{pmatrix}^T$ for some matrix A that is $k \times (n-k)$. For any basis we choose, we can always simplify the matrix in this form. This comes from the fact that column operations do not change the column space:

Proposition 3.3. *Let A be an $m \times n$ matrix, and let E be a column operation. Then* $\text{range}(A) = \text{range}(AE)$.

Proof. The column operation E will scale a column or add two scaled columns together, or switch columns. Let $\mathbf{v}_1, ..., \mathbf{v}_n$ be the columns of A. If the column operation adds two scaled columns, WLOG we can assume that we get $\lambda\mathbf{v}_1 + \mu\mathbf{v}_2$. Then our new matrix AE has columns $\lambda\mathbf{v}_1 + \mu\mathbf{v}_2, \mathbf{v}_2, ..., \mathbf{v}_n$. Note that the span of these new columns is exactly the same as the span of the original columns. If the column operation switches two columns, then clearly the span is also not changed. If the column operation scales a column, then clearly the span is not changed. Thus $\text{range}(A) = \text{range}(AE)$ □

Note that by taking the transpose, we have the same argument for row operations. Additionally, we will need:

Proposition 3.4. *Let A be an $m \times n$ matrix and R its Reduced Column Echelon form. Then the rank r of A is the number of pivots p in R.*

Proof. First, note that the dimension of the output space of R is just the number of pivots. The Reduced Column Echelon form is achieved by applying column operations to A. Thus $R = AE_1 \cdots E_k$ for some column operations $E_1, ..., E_k$, and by Prop 3.3., we have $r = \dim \text{range}(A) = \dim \text{range}(R) = p$. □

Proposition 3.5. *Every linear-$[n, k]$ code admits a systematic generator matrix.*

Proof. Let C be a linear$[n, k]$ code with basis $\mathbf{c}_1, ..., \mathbf{c}_k$. Then G is rank-k, and thus the Reduced Column Echelon form R of G has k pivots by Prop 3.4.. Thus we can apply column-switching operations to R to get some matrix of the form $G' = \begin{pmatrix} I_k & A' \end{pmatrix}^T$. Since column operations do not change the output space by Prop 3.3., we have that $\text{range}(G) = \text{range}(R) = \text{range}(G') = C$. Then columns of G' must be a basis of C, making G' a Generator Matrix for C. □

Now that we know how to use simple G for practical computation and application, let us look at how we can use G to check for errors in transmission.

The idea behind using G and linear codes is to check whether the received code is a codeword or not. For regular code, this is not so simple, as there's no "rule" to follow. For linear codes, there are many rules: the rules of linearity and the rules of being in a vector space.

Suppose that we send a message $\mathbf{x} \in \mathbb{F}_2^k$ with the Generator Matrix G of linear-$[n, k]$ code C. We send a code word $\mathbf{y} \in C$. Suppose that for now, only one error has occurred; i.e. only one bit has switched. Then the receiver gets $\mathbf{y} + \mathbf{e}_i$ for some $1 \leq i \leq n$. If this is a codeword of C, then we can decompose it into a linear combination of G's columns, and that will give us a message $\mathbf{x}'$ the scalars in the combination.

Thus, we need to make sure that $\mathbf{y} + \mathbf{e}_i$ is not a codeword, so we don't mistake $\mathbf{x}$ with $\mathbf{x}'$. Since $\mathbf{y}$ is a codeword, this just means that we can't have $\mathbf{e}_i$ as a codeword.

Condition 3.6. *For C to be an error-correcting linear-$[n,k]$ code, the Generator Matrix G must have the following property:*

$$\mathbf{e}_i \notin \operatorname{range}(G) = C \textit{ for every } i, \textit{ where } \mathbf{e}_i \textit{ are the standard basis vectors in } \mathbb{F}_2^n.$$

We can interpret this condition in a very simple way. Let us consider G in its systematic form. If $\mathbf{e}_i$ is never in the column space, then that just means that the matrix A in $G = \begin{pmatrix} I_k & A \end{pmatrix}^T$ is nonzero. This means that when we do $G\mathbf{x}$, the last $n-k$ bits won't always be 0. Recall that there are $n-k$ check bits, so this is just saying that for an error-correcting code, we need to include check digits, which is pretty intuitive.

Example. Suppose we use the matrix

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

to send two messages 101 and 100. The receiver then gets the codewords 1010 and 1000. However, if the second codeword has an error with $1000 + \mathbf{e}_3 = 1010$, then it is impossible to know if the correct messages are $101, 101$ or $101, 100$.

Example. Suppose we use

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

to send two messages 101 and 100. This time, note that no standard basis vector is in the output space. The codewords should be 1010 and 1001. If an error occurred and the receiver got 1010 and $1001 + \mathbf{e}_3 = 1011$, then this time, since 1011 isn't in $\operatorname{range}(G)$, the receiver knows that there had to be an error here during transmission. For this specific G, not that we have our check bit as the parity bit. Thus, the receiver knows that the correct codewords can be 1001, 1111, 0011. Note that 1111 is also in $\operatorname{range}(G)$ but not the correct codeword. Thus Condition 3.6. itself is not enough to correct errors.

Next, we need to consider when we know that there is an error with $\mathbf{y} + \mathbf{e}_i$, but we don't know which $\mathbf{e}_i$ causes the error. We can fix this by adding $\mathbf{e}_j$ to get $\mathbf{y} + \mathbf{e}_i + \mathbf{e}_j$ and running through all j until we get something that is a codeword. However, if there are multiple $\mathbf{e}_j$ that make $\mathbf{y} + \mathbf{e}_i + \mathbf{e}_j$ a codeword, there is no way to know which one should be the correctly received codeword. Thus, we need to make sure that if $\mathbf{y} + \mathbf{e}_i + \mathbf{e}_j$ is a codeword, there is only one such $\mathbf{e}_j$ that allows it to be. Then we would naturally have $\mathbf{y} + \mathbf{e}_i + \mathbf{e}_j = \mathbf{y}$ or equivalently, $i = j$, by Condition 3.6..

Condition 3.7. *For C to be an error-correcting linear-$[n,k]$ code, the Generator Matrix G must have the following property:*

$$\textit{For all } \mathbf{x} \in \mathbb{F}_2^k,\ G\mathbf{x} + \mathbf{e}_i + \mathbf{e}_j \in \operatorname{range}(G) = C \textit{ implies that } \mathbf{e}_i = \mathbf{e}_j.$$

In other words, if we can fix an error into a codeword by switching some bit, then that bit is where the error occurred. If our Generator Matrix can't allow this condition, then we won't have an error-correcting code.

Example. Let us go back to our matrix

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

and send the messages 101 and 100. Let us consider the same error, where the codewords are received as 1010 and $1001 + \mathbf{e}_3 = 1011$. We know from using Condition 3.6. that 1011 is the error, but from that condition alone we don't know where the error is. Now, by using Condition 3.6., we can precisely find the error. $1011 + \mathbf{e}_1 = 0011$ is not a codeword as it is not in the output space. $1011 + \mathbf{e}_2 = 1111$ is also not in the output space. $1011 + \mathbf{e}_3 = 1001$ is in the output space. Thus by Condition 3.7., the error must have been $1001 + \mathbf{e}_3$, meaning the correct codeword must be 1001.

With the current two conditions, we can locate an exact error, and get the correct $\mathbf{y}$ codeword. The final step now is solving $G\mathbf{x} = \mathbf{y}$, but unless the solution is unique, the receiver might not always get the correct message. Thus all equations $G\mathbf{x} = \mathbf{y}$ must have only one solution. In other words:

Condition 3.8. *For C to be an error-correcting linear-$[n, k]$ code, the Generator Matrix G must be injective. That is, it must have the following property:*

$$G\mathbf{x} = G\mathbf{x}' \text{ if and only if } \mathbf{x} = \mathbf{x}'.$$

In summary, we have:

Theorem 3.9. *C is an error-correcting linear-$[n, k]$ code if and only if it's Generator Matrix G has the following properties:*

1. *$\mathbf{e}_i \notin \text{range}(G) = C$ for every i, where $\mathbf{e}_i$ are the standard basis vectors in $\mathbb{F}_2^n$.*
2. *For all $\mathbf{x} \in \mathbb{F}_2^k$, $G\mathbf{x} + \mathbf{e}_i + \mathbf{e}_j \in \text{range}(G) = C$ implies that $\mathbf{e}_i = \mathbf{e}_j$.*
3. *$G\mathbf{x} = G\mathbf{x}'$ if and only if $\mathbf{x} = \mathbf{x}'$.*

From these conditions, we can concur a variety of nice corollaries. One nice one is as follows:

Corollary 3.10. *$\mathbb{F}_2^k$ is not an error-correcting linear-$[k, k]$ code.*

This is easily proven by looking at number 2 in Theorem 3.9, however also by the fact that our Generator is an isomorphism. If we have an isomorphism, then any errors made with codewords will produce new codewords due to surjectivity. Then receivers will never know that an error was made since the new codeword has a unique corresponding message by injectivity.

4 Detecting Multiple Wrong Bits

So far, we have been dealing with the generous case of only 1 bit switching during transmission. However, in reality, this is a tall demand. Thus, we need to be more realistic. Luckily, the conditions for creating error-correcting code that detects multiple errors aren't so difficult; it shares a lot of similarities with the 1-bit case.

To investigate this general case of multiple errors, it will be beneficial to study a more mathematical and or abstract formulation of code and error-correcting code. We introduce what is called the *Hamming Distance*:

Definition 4.1 (Hamming Distance)**.** The Hamming Distance $\Delta(\mathbf{x}, \mathbf{y})$ between two strings is the number of positions at which the two strings differ. That is,

$$\Delta(\mathbf{x}, \mathbf{y}) = |\{i : x_i \neq y_i\}|.$$

For strings in $\mathbb{F}_2^n$, $\Delta(\mathbf{x}, \mathbf{y})$ is the number of 1s in $\mathbf{x} + \mathbf{y}$.

Note that this defines a clear metric on $\mathbb{F}_2^n$. Next, we define the distance of a code C:

Definition 4.2 (Code Distance)**.** Let C be a code. The distance of the code, $\Delta(C)$ is defined as the minimum hamming distance between two distinct codewords. That is,

$$\Delta(C) = \min_{\substack{\mathbf{x},\mathbf{y} \in C \\ \mathbf{x} \neq \mathbf{y}}} \Delta(\mathbf{x}, \mathbf{y}).$$

With this notation, we get a very nice generalization of our work so far.

Theorem 4.3. *Let C be some code. Suppose that $\Delta(C) = t$. Then we have:*

1. *C can be used to detect at most $t - 1$ errors.*
2. *C can be used to uniquely correct at most $\frac{t-1}{2}$ errors.*

Proof. Suppose that a code $C \subset \mathbb{F}_2^n$ has $\Delta(C) = t$. To prove that at most $t - 1$ errors can be detected, we need to prove that changing at most $t-1$ positions of a codeword can never create another codeword. Suppose that changing at most $t - 1$ positions of a codeword $\mathbf{y}$ does create a codeword $\mathbf{z}$. Then:

$$t = \Delta(C) \leq \Delta(\mathbf{y}, \mathbf{z}) \leq t - 1$$

which is a contradiction. Thus changing at most $t - 1$ bits in a codeword doesn't create another codeword, meaning we can detect up to $t-1$ errors. If at most $\frac{t-1}{2}$ errors occur, the receiver can uniquely decode the received message to a codeword as every received message has at most one codeword at distance $\frac{t-1}{2}$. Suppose that every received message $\mathbf{z}$ has at least two codewords at distance $\frac{t-1}{2}$, say, $\mathbf{x}, \mathbf{y}$. Then we have by the Triangle Inequality that:

$$t - 1 = \Delta(\mathbf{y}, \mathbf{z}) + \Delta(\mathbf{x}, \mathbf{z}) \geq \Delta(\mathbf{x}, \mathbf{y}) \geq \Delta(C) = t$$

which is a contradiction. Thus every message has at most one codeword that is of distance $\frac{t-1}{2}$ apart, meaning that up to $\frac{t-1}{2}$ errors can be corrected as a unique codeword exists for the error-ridden received message. □

From this theorem, it is clear that if we want to correct more and more errors, or even detect them, we need larger and larger code and hence longer and longer codewords.

Let's consider what this theorem means it terms of our investigations with linear code and Theorem 3.9. and its conditions. If the linear-$[n, k]$ code C has a minimum Hamming Distance of t, then what does that say about summing up at most $t-1$ standard basis vectors?

In Theorem 3.9., the idea was that $\mathbf{e}_i$ was never a codeword. The idea with Theorem 4.3. in terms of linear C is that the sum of at most $t-1$ basis vectors cannot be a codeword other than $\mathbf{0}$.

Proposition 4.4. *Let C be a linear-$[n, k]$ code that is error-correcting. To detect at most $t-1$ errors, we require that a sum of at most $t-1$ standard basis vectors can only be the $\mathbf{0}$-codeword, and no other codeword.*

Proof. Suppose that there was a sum of at most $t-1$ basis vectors that was a nonzero codeword. Call this sum $\mathbf{v}$. Then $\Delta(\mathbf{v}, \mathbf{0}) \leq t-1$ since the biggest difference from $\mathbf{0}$ is when the sum is of $t-1$ unique basis vectors with and hence when $\mathbf{v}$ has $t-1$ 1's. This then implies that $\Delta(C) \leq \Delta(\mathbf{v}, \mathbf{0}) \leq t-1$, so we cannot have $\Delta(C) = t$. Then by Theorem 4.3., C cannot be used to detect at most $t-1$ errors. Thus we cannot have that $\mathbf{v}$ is nonzero and a codeword. It must be the $\mathbf{0}$ codeword or no codeword. $\square$

This ensures that we may detect up to $t-1$ errors using C. This is because if $\mathbf{y}$ is a codeword that experiences at most $t-1$ errors, then the received message is $\mathbf{z} = \mathbf{y} + \text{bad stuff}$. However, "bad stuff" is some sum of $t-1$ standard basis vectors that is nonzero and hence not a codeword, meaning the received message cannot be a codeword per the rules of vector spaces. This will tell the receiver that there is an error. We need to ensure that the incorrectly received $\mathbf{z}$ is not a codeword, for if it was, then the receiver could trace it back to a valid message that is different from the correct one.

Recalling our investigation with correcting a single error using G, we have the condition that $G\mathbf{x} + \mathbf{e}_i + \mathbf{e}_j$ being a codeword means that $\mathbf{e}_i + \mathbf{e}_j = 0$. For the multiple error case, we have a similar condition by making use of $\Delta(C) = t$.

Proposition 4.5. *Let C be a linear-$[n, k]$ code that is error-correcting. To correct at most $\frac{t-1}{2}$ errors, we require that a sum of at most $t-1$ standard basis vectors can only be the $\mathbf{0}$-codeword, and no other codeword.*

Proof. Note that if $t-1$ standard basis vectors cannot sum to a nonzero-codeword, then we can detect up to $t-1$ errors or that $\Delta(C) = t$. This immediately shows that up to $\frac{t-1}{2}$ errors can be corrected by Theorem 4.3. and thus our proof is complete. $\square$

Thus, in summary, we have:

Theorem 4.6. *Let C be a linear-$[n, k]$ code with Generator Matrix G. Then C can detect at most $t-1$ errors and correct at most $\frac{t-1}{2}$ of them if:*

Any nonzero sum, $\mathbf{v}$, of at most $t-1$ standard basis vectors in C is such that
$$\mathbf{v} \notin \text{range}(G) = C.$$

Now that we've covered multiple errors, we are complete with basic ideas of how to make code error-correcting. Note that to make code error-correcting, we need a detection algorithm and a correction algorithm which is just an algorithm to locate the error: e.g. Condition 3.6. and Condition 3.7..

5 Check Matrix H: Streamlining G

Recall that G can be constructed to satisfy conditions of Theorem 3.9. in order to make an error-correcting linear code C.

For an error-correcting code C, we have to check if a received message $\mathbf{z}$ is a codeword or not, which is not always so simple for large code C. The first step in streamlining this step is to construct a matrix H that is such that $H\mathbf{z} = 0$ only when $\mathbf{z} \in \text{range}(G) = C$.

Definition 5.1 (Check Matrix)**.** Let C be a linear-$[n, k]$ code. The $(n-k) \times n$ check matrix H is given by:

$$H = \begin{pmatrix} A & I_{n-k} \end{pmatrix}$$

where A is the $(n-k) \times k$ matrix in the systematic Generator Matrix $G = \begin{pmatrix} I_k & A^T \end{pmatrix}^T$ of C. H is called C's *Check Matrix.*

Proposition 5.2. *With notation as above, $H\mathbf{z} = 0$ if and only if $\mathbf{z} \in \text{range}(G) = C$.*

Proof. Suppose that $\mathbf{z} \in \text{range}(G) = C$. Then there exists $\mathbf{x} \in \mathbb{F}_2^k$ such that $G\mathbf{x} = \mathbf{z}$. We have that:

$$HG = \begin{pmatrix} A & I_{n-k} \end{pmatrix} \begin{pmatrix} I_k \\ A \end{pmatrix} = AI_k + I_{n-k}A = A + A = 0$$

which means that $H\mathbf{z} = HG\mathbf{x} = 0(\mathbf{x}) = \mathbf{0}$.

Suppose that $\mathbf{z} \in \mathbb{F}_2^n$ and that $H\mathbf{z} = 0$. Then we have:

$$A\mathbf{z}_1 + \mathbf{z}_2 = 0$$

where $\mathbf{z}_1$ is the first k elements of $\mathbf{z}$ and $\mathbf{z}_2$ is the last $n-k$ elements of $\mathbf{z}$. Since we are working in $\mathbb{F}_2$, we have that $A\mathbf{z}_1 = \mathbf{z}_2$. Thus we can write:

$$\mathbf{z} = \begin{pmatrix} \mathbf{z}_1 \\ A\mathbf{z}_1 \end{pmatrix} = \begin{pmatrix} I_k \\ A \end{pmatrix} \mathbf{z}_1 = G\mathbf{z}_1$$

which shows that $\mathbf{z} \in \text{range}(G) = C$. □

Example. Let us go back to our matrix

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

for the linear-$[4, 3]$ code $C = \text{span}(\mathbf{e}_1 + \mathbf{e}_4, \mathbf{e}_2 + \mathbf{e}_4, \mathbf{e}_3 + \mathbf{e}_4)$. It's check matrix is:

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}$$

which gives:

$$H\left(\sum_{i=1}^{3} a_i(e_i+e_4)\right) = a_1+a_1+a_2+a_2+a_3+a_3 = 0.$$

Furthermore,

$$H(c_1,c_2,c_3,c_4) = c_1+c_2+c_3+c_4 = 0$$

will imply that $\mathbf{c}$ is a linear combination of G's columns after performing some algebra and computation.

Now that we have this important property of H from Prop 5.2., we can return to investigations of single-bit errors once more. If $\mathbf{y} = G\mathbf{x}$ is a codeword but receives the error and becomes $\mathbf{y}+\mathbf{e}_i$, we get:

$$H(\mathbf{y}+\mathbf{e}_i) = 0 + H\mathbf{e}_i$$

which is the ith column of H. This streamlines checking the second condition in Theorem 3.9., as rather than trying to add all the different $\mathbf{e}_j$ and checking if $G\mathbf{x}+\mathbf{e}_i+\mathbf{e}_j = G\mathbf{x}$, we can simply look at columns of H until we find a match. Both seem like they are not efficient, but remember that in practical applications, we would know H and its columns explicitly, while we would need some computation to figure out which $\mathbf{e}_j$ is such that $G\mathbf{x}+\mathbf{e}_i+\mathbf{e}_j = G\mathbf{x}$.

However, if H has two same columns, then we run into a problem, as we won't be sure which one of the multiple possible errors really happened. Thus we need to make sure H has different columns. In general, we have:

Theorem 5.3. *Let C be a linear-$[n,k]$ code, and H it's Check Matrix. Then C is error-correcting if and only if the following are satisfied:*

1. *H has all nonzero columns.*
2. *No two columns of H are the same.*
3. *G is injective*

Proof. Suppose that C is error-correcting. Then by Theorem 3.9., the Generator Matrix G is such that $G\mathbf{x}+\mathbf{e}_i+\mathbf{e}_j \in \text{range}(G) = C \implies \mathbf{e}_i = \mathbf{e}_j$ and that $\mathbf{e}_i \notin \text{range}(G) = C$ for all possible i. Note that:

$$H(G\mathbf{x}+\mathbf{e}_i+\mathbf{e}_j) = H\mathbf{e}_i + H\mathbf{e}_j = 0$$

means column i and j of H are the same. However, since C is error-correcting, we must have $i = j$ and thus we cannot have that two different columns are the same. Additionally,

$$H\mathbf{e}_i \neq 0$$

since $\mathbf{e}_i \notin \text{range}(G) = C$, so H has all nonzero columns.

Suppose that H satisfies the given properties. Having all nonzero columns means that $H\mathbf{e}_i \neq 0$ for all $\mathbf{e}_i$, so $\mathbf{e}_i \notin \text{range}(G) = C$. This means C satisfies Condition 3.6.. Furthermore, the second property of H implies that for $i \neq j$, we get:

$$H(\mathbf{e}_i+\mathbf{e}_j) = H\mathbf{e}_i + H\mathbf{e}_j \neq 0$$

since different columns must not be the same, and since the additive inverse of an element in $\mathbb{F}_2$ is the element itself. This means that $\mathbf{e}_i + \mathbf{e}_j$ is never a codeword unless $i = j$. In other words, if $G\mathbf{x} + \mathbf{e}_i + \mathbf{e}_j \in \text{range}(G) = C$, then we must have $\mathbf{e}_i = \mathbf{e}_j$, i.e. Condition 3.7.. Thus the two conditions on H along with G's injectivity match the three conditions on G for C to be error-correcting, which completes the proof. $\square$

Up to now, we've been constructing H by using G. This can also be done backwards. If we have H but wish to find a corresponding G, rather than using Definition 5.1., we can also just look at null(H) and find a basis. Doing so will yield the columns of G. The problem with this method is that our basis may not lead to G being in systematic form while working backwards by using Definition 5.1. will ensure G is in systematic form.

Note that the way we use H is very nice for computation since it's like the systematic form. What if H was in a different form other than $\begin{pmatrix} A & I \end{pmatrix}$? Luckily, row operations will not change H's code C:

Proposition 5.4. *Elementary row operations on H will not change it's corresponding code C.*

Proof. This comes easily from the fact that row operations preserve the nullspace and hence the code C that is sent to zero by H. This is because if E is an elementary row operation and A is a matrix, we have:

$$\mathbf{0} = A\mathbf{x} = IA\mathbf{x} = (E^{-1}E)A\mathbf{x} = E^{-1}EA\mathbf{x}.$$

By invertibility of row operations, they are injective and surjective so we must have that $EA\mathbf{x} = \mathbf{0}$. This is because $EA\mathbf{x} \in \text{null}(E^{-1})$ by above, and since injectivity implies $\dim(\text{null}(E^{-1}) = 0$, the only null-element must be $\mathbf{0}$. $EA\mathbf{x} = \mathbf{0}$ implies that the nullspace is unchanged since this is only true when $\mathbf{x} \in \text{null}(A)$. $\square$

Since row operations can be done on H to get another H' for the same code C, we can put messy forms of H into the form presented in Definition 5.1.. When constructing G from a messy H, we can avoid this row operating step and just consider a basis for null(H), which is in some sense simpler than cleaning H up and finding G via Definition 5.1..

6 Hamming Code

Lastly, let us investigate Hamming Code. This is a special, explicit example of error-correcting linear code. The most popular Hamming Code is the $[7, 4]$-Hamming Code. The Generator Matrix for this code is given by:

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}.$$

We may think of the corresponding code as being the original message along with three parity checks. More specifically, if we send $\mathbf{x} = (x_1, x_2, x_3, x_4)$, the codeword is $(\mathbf{x}, x_1 + x_2 + x_4, x_1 + x_3 + x_4, x_2 + x_3 + x_4)$. Let's look at the parity checks in particular. We have:

$$\begin{aligned} x_1 + x_2 + 0 + x_4 &= c_5 \\ x_1 + 0 + x_3 + x_4 &= c_6 \\ 0 + x_2 + x_3 + x_4 &= c_7 \end{aligned}$$

and $x_i = c_i$ for $i = 1, 2, 3, 4$. When the recipient gets a message, they just have to check that $c_5 + c_1 + c_2 + c_4 = 0$, and that $c_6 + c_1 + c_3 + c_4 = 0$, and that $c_7 + c_2 + c_3 + c_4 = 0$. Let's check that C is really error-correcting by going through Theorem 3.9. on G. Checking that $\mathbf{e}_i$ is not in the range of G can be tedious, so let us use H and Theorem 5.3. instead. We have:

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

which has no same columns and no nonzero columns. All that remains is to check that G is injective. We can do this with a nice fact about linear transformations.

Proposition 6.1. *If $T : V \to W$ is a linear transformation, it is injective if and only if $T\mathbf{v} = \mathbf{0}$ if and only if $\mathbf{v} = \mathbf{0}$; i.e. T is injective when it's nullspace is zero-dimensional.*

Proof. Suppose that T is injective. Then $T\mathbf{u} = T\mathbf{v}$ implies that $\mathbf{v} = \mathbf{u}$. Thus, if $T\mathbf{v} = 0$, since $T\mathbf{0} = \mathbf{0}$, we have $\mathbf{v} = \mathbf{0}$.

Suppose that $\dim \operatorname{null}(T) = 0$. Consider the case when $T\mathbf{u} = T\mathbf{v}$. Then $T(\mathbf{u} - \mathbf{v}) = T\mathbf{u} - T\mathbf{v} = 0$ by linearity, so we have $\mathbf{u} - \mathbf{v} \in \operatorname{null}(T) = \{0\}$, meaning $\mathbf{v} = \mathbf{u}$, so T is injective. $\square$

Note that G is a linear transformation (all matrices are) and that $G\mathbf{0} = \mathbf{0}$. Thus all conditions in Theorem 5.3. are satisfied, making the $[7, 4]$-Hamming Code an error-correcting linear code. This code is particularly nice due to the ratio $\frac{4}{7}$. For a $[n, k]$ code, the efficiency of the code can be interpreted as $\frac{k}{n}$. This efficiency tells us how much percent of the codeword is the message we're trying to send. When we use the encoding function $e(x) = (x, x, x)$, we have an efficiency of $\frac{1}{3}$ which is terrible for large messages. However, $\frac{4}{7} > \frac{1}{2}$, so it is quite efficient since more than half the codeword is something important.

An extension of the $[7, 4]$-Hamming Code is the $[15, 11]$-Hamming Code and generally the $[2^n - 1, 2^n - n - 1]$. This type of code is nice because the efficiency tends to 100 percent:

$$\lim_{n \to \infty} \frac{2^n - n - 1}{2^n - 1} = 1.$$

Naturally, if we have an infinitely long message and codeword, we can be a completely efficient method of data transmission. Unfortunately, no computer right now can possibly hold that much data.

References

[1] Jauregui, Jeff. Error–correcting codes with linear algebra. August 24, 2012.

[2] Fenyes, Aaron. Matrix Algebra and Error-Correcting Codes. University of Texas, October 2015.

[3] Venkatesan, Guruswami. Introduction to coding theory. Carnegie Mellon University, Spring 2010. http://www.cs.cmu.edu/˜venkatg/teaching/ codingtheory/