Error-Correcting Codes

Matej Boguszak

Contents

1 Introduction

Error-correcting codes have been around for over 50 years now, yet many people might be surprised just how widespread their use is today. Most of the present data storage and transmission technologies would not be conceivable without them. But what exactly are error-correcting codes? This first chapter answers that question and explains why they are so useful. We also explain how they do what they are designed to do: correct errors. The last section explores the relationships between different types of codes as well the issue of why some codes are generally better than others. Many mathematical details have been omitted in this chapter in order to focus on concepts; a more rigorous treatment of codes follows shortly in chapter 2.

1.1 Why Coding?

Imagine Alice wants to send her friend Bob a message. Cryptography was developed to ensure that her message remains private and secure, even when it is sent over a non-secure communication channel, such as the internet. But there is another problem: What if the channel is not so much non-secure but noisy? In other words, what if Alice's original message gets distorted on the way and Bob receives an erroneous one? In many cases Bob may be able to tell what error has occurred from the context of her message, but this is obviously not guaranteed. Hence, two potential problems can arise:

1. An error has occurred and Bob detects it, but he is unable to correct it. Thus the message he receives becomes unreadable.

2. An error has occurred and Bob does not detect it. Thus the message he mistakenly perceives to be correct is in fact erroneous.

So what can be done about this? The answer and solution lies in cunning coding.

Terminology

Communication over a *noisy channel*, as described in the introductory example above, is the motivating problem behind *coding theory*. *Error-correcting codes* in particular provide a means of encoding a message in a way that makes it possible for the receiver to detect and correct the errors that might have occurred during transmission. The following diagram sums up the transmission process across a noisy channel:

$$\text{Alice} \xrightarrow{\mathbf{m}} \text{ENCODER} \xrightarrow{\mathbf{c}} \textit{noisy channel} \xrightarrow{\mathbf{d}} \text{DECODER} \xrightarrow{\mathbf{m}} \text{Bob}$$

A message is usually not transmitted in its entirety, but rather sent piecewise in blocks of a fixed length called *message words*. Each message word **m** is encoded into a *codeword* **c** using a suitable coding algorithm, and then sent across the noisy channel. The collection all codewords (i.e., all possible message words encoded) is called a *code* and is often denoted by C.

While being transmitted, the codeword **c** may get corrupted by an *error* **e** into a word **d**, which is not necessarily a codeword (in fact, we hope that it isn't). The decoder then has several jobs. First, it needs to detect that **d** is not a codeword, identify the error **e** that has occurred, and correct it accordingly to get the original codeword **c**. Once **c** is identified, it can easily be decoded to obtain the original message word **m** as output for the receiver. The details of how all this is done will be discussed in the next section.

Applications

It is important to note that different communication channels require different types of coding mechanisms, depending on the sorts of errors that are expected to happen. For example, errors may occur randomly throughout the message or in *bursts*, meaning that numerous errors occur within only a short distance from each other. An effective error-correcting code must be designed with the nature of the specific noisy channel in mind.

There are many other areas besides communications where error-correcting codes are used and the same ideas are applicable. Digital storage media such as compact discs use error-correcting codes to protect data from "noise" caused by equipment deterioration (e.g., scratches). Some identification standards such as the ISBN use them to help prevent errors that might creep in by human mistake. Some of these applications will be discussed in subsequent chapters, but first, let us turn to how error-correcting codes work.

1.2 Error Detection and Correction

Let us go back to Alice and Bob. Since almost all electronic equipment today operates in binary mode, we will assume for the moment that Alice's messages simply consist of 0's and 1's. Suppose, for example, that Alice wants to send the following message, consisting of four 4-bit binary message words, to Bob:

```
1011 0110 0001 0101
```

Let's say that Alice forgets to encode her message and simply sends it as it is. Because the communication channel is noisy, an error in the third digit of the second word occurs, and the 1 gets changed to a 0:

$$\begin{array}{c}\texttt{1011 0110 0001 0101}\\ \downarrow \mathbf{e}\\ \texttt{1011 0100 0001 0101}\end{array}$$

Now Bob has little way of telling whether the second or any other word is in fact what Alice originally sent! After all, she might very well have sent the word 0100 instead of 0110. So how can Alice encode her message so that Bob will be able to detect and then correct the errors that may occur on the way?

One obvious method would be to simply repeat each word that is sent a few times, so as to make sure that if one copy of the word gets corrupted, the others probably won't. Though this probabilistic method can be as accurate as one desires (the more repetition the greater its accuracy), it is clearly an inefficient coding scheme, as it increases the size of the transmitted message tremendously.

Hamming's Breakthrough

In 1948, Richard Hamming, a mathematician working at the Bell Telephone Laboratories, came up with a more clever idea to encode messages for error-correction. To obtain a codeword, he added a few extra digits, so-called *check-digits*, to each message word. These digits allowed him to introduce a greater "distance" between codewords. To understand what that means, we need to define the notion of more precisely.

Definition 1.1 *The* **Hamming distance** *between two codewords* $\mathbf{c}$ *and* $\mathbf{c}'$*, denoted by* $d(\mathbf{c}, \mathbf{c}')$*, is the number of digits in which they differ.*

For example, $d(0110, 0100) = 1$ is the distance between the message word 0110 that Alice originally sent and the erroneous word 0100 that Bob received. This distance is 1 because these two words differ in exactly one digit.

Definition 1.2 *The* **minimum distance** *of a code* C*, denoted by* $d(C)$*, is the smallest Hamming distance between any two codewords in* C*, i.e.* $d(C) := \min\{d(\mathbf{c}, \mathbf{c}') \mid \mathbf{c}, \mathbf{c}' \in C\}$*.*

If Alice doesn't encode her message words at all and uses them as codewords, then her code C will simply consist of all 4-digit binary words, and clearly, $d(C) = 1$. But this is exactly our problem! Now when an error occurs, we have no way of telling because it will simply change one codeword into another: When the codeword 0110 gets changed to another codeword, 0100, Bob cannot tell that there was an error.

That is why Hamming, when he worked with 4-bit binary words just like the ones in our example, introduced three extra check digits in a clever way[1] to create a 7-bit code C with $d(C) = 3$, the famous Hamming(7,4) code. Now that the distance between any two codewords was 3, he could be sure that if an error occurred in a given codeword $\mathbf{c}$ (i.e., one digit got altered), the resulting word $\mathbf{d}$ would *not* be a codeword. This is because $d(\mathbf{c}, \mathbf{d}) = 1$, but the Hamming distance between $\mathbf{c}$ and any other codeword $\mathbf{c}'$ is at least 3. In this way, a single error could be easily detected.

What's more, assuming that only one error occurred, Hamming would also have a way to correct it by simply matching $\mathbf{d}$ to its nearest codeword neighbor. This *nearest-neighbor decoding* would be unambiguous and always yield the correct original codeword $\mathbf{c}$, for $\mathbf{c}$ is the only codeword that could be at such a close distance to $\mathbf{d}$. However, had 2 errors occurred, resulting in a word $\mathbf{d}'$ at a distance 2 from $\mathbf{c}$, then our decoding scheme fails because $\mathbf{c}'$, not $\mathbf{c}$, is now the nearest neighbor. Take a look at figure 1, which illustrates this concept. The nearest codeword neighbor of $\mathbf{d}$ is the center of the "sphere" in which it lies.

Some Mathematics

The obvious limitation of the Hamming(7,4) code, and seemingly any code with minimum distance three, is that it is only one-error-correcting. Naturally, we should ask ourselves: Can we design a code that corrects any given number of errors per codeword? The answer, as the reader might suspect, is yes, and has to do with increasing the minimum distance of the code. Before we get into any details, however, we need to take a closer look at the Hamming distance.

[1] To be discussed in chapter 2. Refer to [3] for historical details.

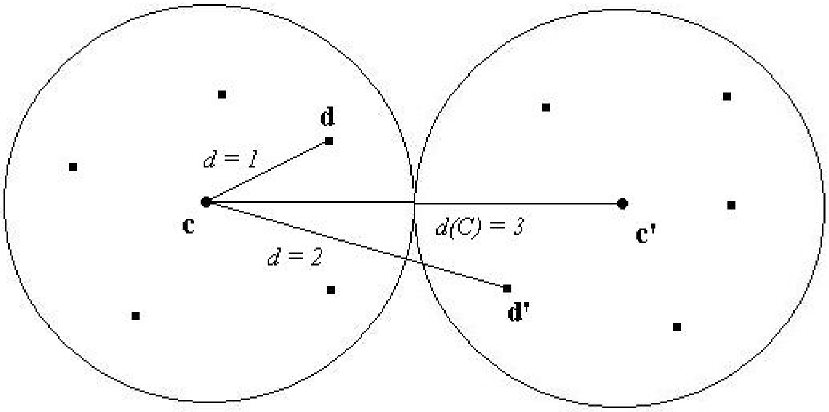

Figure 1: Hamming distances in a code C with minimum distance 3.

Theorem 1.1 *The Hamming distance is a metric on the space of all words of a given length n, call it W. In other words, the function $d : W \times W \to \mathbb{Z}$ satisfies for all words $\mathbf{x}, \mathbf{y}, \mathbf{z} \in W$:*

i) $d(\mathbf{x}, \mathbf{y}) \geq 0 \ \wedge \ d(\mathbf{x}, \mathbf{y}) = 0 \Leftrightarrow \mathbf{x} = \mathbf{y}$

ii) $d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$

iii) $d(\mathbf{x}, \mathbf{z}) \leq d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z})$

Proof. Recall that $d(\mathbf{x}, \mathbf{y})$ is defined as the number of digits in which the two words differ. (i) and (ii) are both trivially true by this definition, so let's turn to (iii). Denote $\mathbf{x} = (x_1, ..., x_n)$, $\mathbf{y} = (y_1, ..., y_n)$, and $\mathbf{z} = (z_1, ..., z_n)$. Now consider the i^{th} digit of each word: x_i, y_i, and z_i.

CASE 1: $x_i = z_i$. In this case $\mathbf{x}$ and $\mathbf{z}$ do *not* differ in their i^{th} digit: The digits are either both 0 or both 1. Therefore, they must either both differ from or both concur with the i^{th} digit of $\mathbf{y}$. This means that this i^{th} digit contributes 0 to $d(\mathbf{x}, \mathbf{z})$, and either 0 or 2 to $d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z})$. Clearly, $0 \leq 0$ and $0 \leq 2$.

CASE 2: $x_i \neq z_i$. In this case $\mathbf{x}$ and $\mathbf{z}$ *do* differ in their i^{th} digit: One digit is 0 and the other is 1. Therefore, exactly one of them differs from the i^{th} digit of $\mathbf{y}$, while the other concurs with it. This means that the i^{th} digit contributes 1 to $d(\mathbf{x}, \mathbf{z})$ and also 1 to $d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z})$. Clearly, $1 \leq 1$.

This argument holds for all digits of the words $\mathbf{x}, \mathbf{y}$ and $\mathbf{z}$. By adding up all the contributions, we indeed get the triangle inequality (iii).

□

Now that we have established the nature of the Hamming distance, we can return to our original question: How big does a code's minimum distance have to be so that it corrects a given number of errors, say t? The answer to this question is relatively straightforward, and is summarized by the following theorem.

Theorem 1.2 *A code C is t-error-correcting if and only if $d(C) \geq 2t+1$.*

Proof. First, assume that $d(C) \geq 2t+1$. Let $\mathbf{c} \in C$ be the codeword that was transmitted, $\mathbf{d}$ the possibly erroneous word that was received, and $\mathbf{c}'$ any other codeword in C. Assuming that no more than t errors occurred, can we correct them? In other words, will we decode $\mathbf{d}$ correctly by identifying $\mathbf{c}$ as its unique nearest neighbor? Let's see:

$$\begin{array}{rcll}
2t+1 & \leq & d(C) & \text{by assumption} \\
 & \leq & d(\mathbf{c},\mathbf{c}') & \text{by definition of } d(C) \\
 & \leq & d(\mathbf{c},\mathbf{d}) + d(\mathbf{d},\mathbf{c}') & \text{by triangle inequality} \\
 & \leq & t + d(\mathbf{d},\mathbf{c}') & \text{since at most } t \text{ errors occurred.} \\
\text{Hence, } d(\mathbf{d},\mathbf{c}') & \geq & t+1 & \\
 & > & t & \\
 & \geq & d(\mathbf{d},\mathbf{c}). &
\end{array}$$

So the distance between the received word $\mathbf{d}$ and the sent codeword $\mathbf{c}$ is less than the distance between $\mathbf{d}$ and any other codeword. Therefore, assuming up to t errors, we can safely correct $\mathbf{d}$ as its nearest neighbor $\mathbf{c}$, and thus the code C is t-error-correcting.

Conversely, assume that $d(C) < 2t+1$ and let $\mathbf{c}, \mathbf{c}'$ be codewords at minimum distance from each other: $d(\mathbf{c},\mathbf{c}') = d(C)$. As before, $\mathbf{c}$ will be the codeword initially transmitted. Now introduce the following errors to $\mathbf{c}$: If $d(C)$ is even, pick $\frac{d(C)}{2}$ digits in which $\mathbf{c}$ differs from $\mathbf{c}'$ (there are $d(C)$ total such digits) and replace them with the corresponding digits in $\mathbf{c}'$. If $d(C)$ is odd, do the same with $\frac{d(C)+1}{2}$ digits. Notice that since $d(C) < 2t+1$, the number of errors introduced in this manner is at most t in either case.

The result of these errors is again the word $\mathbf{d}$. In the even case, we have $d(\mathbf{c},\mathbf{d}) = \frac{d(C)}{2}$ since this is the number of errors introduced, but also $d(\mathbf{c}',\mathbf{d}) = \frac{d(C)}{2}$ because of the way those errors were introduced (half of the original $d(C)$ differing digits still differ from $\mathbf{c}'$). Thus, the received word $\mathbf{d}$ does not have a unique closest neighbor; the decoder cannot decide between $\mathbf{c}$ and $\mathbf{c}'$, both of which are at the same distance from $\mathbf{d}$. In the odd case, we get similarly $d(\mathbf{c},\mathbf{d}) = \frac{d(C)+1}{2}$ but $d(\mathbf{c}',\mathbf{d}) = \frac{d(C)-1}{2}$, so this time the decoder would actually make the mistake of decoding $\mathbf{d}$ as $\mathbf{c}'$ because it would be its closest neighbor. Therefore, the code C cannot be considered t-error-correcting.

□

1.3 Efficiency Considerations

So far, we have looked at how a code's error-correcting capability can be improved by increasing its minimum distance. There are, however, other criteria to consider when evaluating the overall usefulness of a code. One of these is *efficiency*, which refers to the ability of a code to transmit a message as quickly and with as little redundancy (as represented by check digits) as possible. An important efficiency measure is *information rate*, which is designed to measure the proportion of each codeword that actually carries the message as opposed to "just" check digits for error-correction.

Definition 1.3 *The information rate ρ of a q-ary code C consisting of $M := |C|$ codewords of length n is defined as*

$$\rho := \frac{1}{n}\log_q M.$$

Clearly, we would like the information rate to be as high as possible. This means that we should strive for short codewords (small n) and/or large codes (big M). Let us fix n and the minimum distance $d(C)$ for now and ask ourselves: How big can M get?

If the size of our alphabet is q (a generalization of the binary codes discussed so far), then a trivial upper bound for M is certainly q^n, for there are q choices for each digit and n digits in each codeword. But this is just the number of all q-ary words of length n, so except for the rather useless case $d(C) = 1$, the actual number of codewords will be smaller. To get a better (i.e. smaller) upper bound on M as well as some more intuition for the concept, let us introduce some geometric language.

Consider the "sphere" of all words at a Hamming distance $\leq t$ from a chosen center $\mathbf{c}$, such as the one if figure 1, defined by

$$S(\mathbf{c}) := \{\mathbf{d} \mid d(\mathbf{c}, \mathbf{d}) \leq t\}.$$

Now look at the set of all such spheres, where $\mathbf{c}$ is a codeword. If C is a q-ary t-error-correcting code, notice that no two of these spheres centered on codewords can intersect, otherwise a word $\mathbf{w}$ in both $S(\mathbf{c_1})$ and $S(\mathbf{c_2})$ could not be unambiguously decoded. So no word belongs to more than one sphere, but there may be words that belong to no sphere.

The number of words at a distance i from a codeword $\mathbf{c}$ is $\binom{n}{i}(q-1)^i$, for there are $q-1$ choices for alphabet symbols in each of the i positions where a word differs from $\mathbf{c}$, and $\binom{n}{i}$ choices for these positions. Hence, the number of words in each sphere is $\sum_{i=0}^{t}\binom{n}{i}(q-1)^i$. Since all spheres are pairwise disjoint, the number of words in all spheres is $M\sum_{i=0}^{t}\binom{n}{i}(q-1)^i$ (recall that M is the number of codewords), and this number is obviously $\leq q^n$, the number of all possible words, some of which might not be elements of any sphere. Thus we get the following upper bound on M:

Theorem 1.3 *The* **sphere packing bound** *on the number of codewords M in a q-ary t-error-correcting code C is*

$$M \leq q^n \left(\sum_{i=0}^{t}\binom{n}{i}(q-1)^i\right)^{-1}$$

.

Using a similar reasoning and calculation, it can be shown that a possible lower bound on the number of codewords is $M \geq q^n\left(\sum_{i=0}^{d-1}\binom{n}{i}(q-1)^i\right)^{-1}$. This result is known as the *Gilbert-Varshmanov bound*. It is an area of ongoing research to find increasingly tighter bounds, as they allow us to set ambitious yet theoretically feasible goals for M when designing a code.

Generally, *if* our goal is to increase M as much as possible, we should aim for the following type of codes:

Definition 1.4 *A q-ary t-error-correcting code is called* **perfect** *if and only if strict equality holds for the sphere packing bound.*

Geometrically, this definition expresses that each word an element of some sphere, and so the word space is partitioned into non-overlapping spheres centered $S(\mathbf{c})$ centered on codewords. Besides the fact that perfect codes have a maximum possible number of codewords M and are aesthetically pleasing in a geometric sense, they also have a very nice error-correcting property.

Theorem 1.4 *A perfect t-error-correcting code cannot correctly decode a word with more than t errors.*

Proof. Let $\mathbf{c}$ be the codeword sent and $\mathbf{d}$ be the erroneous word received, which has more than t errors: $d(\mathbf{c}, \mathbf{d}) > t$. But if the code is perfect, then $\mathbf{d} \in S(\mathbf{c}')$ for *some* codeword $\mathbf{c}' \neq \mathbf{c}$, and thus will be decoded as $\mathbf{c}'$, not as $\mathbf{c}$. Clearly, the code therefore does not correct more than t errors.

□

When we test the performance of some specific codes in chapter 4, we will come back to the concept of efficiency, and see that there is usually a trade-off between a code's error-correcting capability and its efficiency. But first we need to define codes, of which we spoke very vaguely and generally in this chapter, in a more rigorous way.

2 Linear Codes

In this chapter we turn our attention to the mathematical theory behind error-correcting codes, which has been purposely omitted in the discussion so far. Notice that although we explained the concept of error-correction via check digit encoding and nearest neighbor decoding, we have not even hinted at how these methods are actually implemented. We now turn our attention to these practical issues as we define codes more rigorously and learn how to manipulate words and codewords as necessary.

Many practical error-correcting codes used today are examples of so-called *linear codes*, sometimes called linear *block* codes, a concept to be defined in the first section. Among the benefits of linearity are simple, fast, and easy-to-analyze encoding and decoding algorithms, which will be the subject of the next two sections. The last section introduces *Hamming codes* as one of the oldest and still widely used class of error-correcting codes. Basic knowledge of linear and abstract algebra on part of the reader is assumed.

2.1 Basic Concepts

Recall that *words* sent across a noisy channel are simply blocks of some length n consisting of 0's and 1's, or more generally, of elements from some alphabet of size q. From now on, we shall take $\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$ as our alphabet, where q is a prime so as to ensure that $\mathbb{Z}_q$ is a finite field. Any word of length n can therefore be written as a (row) vector $\mathbf{w} = (w_1, \ldots, w_n)$, where each $w_i \in \mathbb{Z}_q$. In other words $\mathbb{Z}_q^n$, a vector space over the finite field $\mathbb{Z}_q$, will be our space of all words, and we are free to use standard vector addition and scalar multiplication (mod q).

Not all words need necessarily be *codewords*, however. Codewords are only those words $\mathbf{c}$ that are elements of a code C. We are especially interested in a particular class of codes:

Definition 2.1 *A* **linear code** C *over* $\mathbb{Z}_q$ *is a subspace of* $\mathbb{Z}_q^n$, *i.e.*

i) $C \neq \emptyset \qquad (\mathbf{0} \in C)$

ii) $\mathbf{c_1}, \mathbf{c_2} \in C \Rightarrow \mathbf{c_1} + \mathbf{c_2} \in C$

iii) $\lambda \in \mathbb{Z}_q, \mathbf{c} \in C \Rightarrow \lambda \mathbf{c} \in C$

There are non-linear codes in use today as well and it turns out that they have properties very similar to those of linear codes, but we shall not discuss them any further in this text. The advantages of defining a code as a linear subspace are apparent, as we can fully utilize the language of linear algebra when dealing with error-correcting codes. Concepts such as linear independence (of codewords), basis (of a code), and orthogonality (of codewords as well as codes) can be easily understood in this new setting.

The dimension of the word space $\mathbb{Z}_q^n$ is n (as long as q is a prime), but for a linear code $C \subseteq \mathbb{Z}_q^n$, its dimension is smaller except for the rather useless case where $C = \mathbb{Z}_q^n$. We denote a linear code C of word length n and dimension k as an $[n, k]$ code. Notice that M, the number of its codewords, only depends on k:

Lemma 2.1 *If $dim(C) = k$, then $M = q^k$.*

Proof. Let $\mathcal{B} = \{\mathbf{b_1}, \dots, \mathbf{b_k}\}$ be a basis for C. Then its codewords are of the form $\mathbf{c}_i = \sum_{i=1}^{k} \lambda_i \mathbf{b}_i$, where each $\lambda_i \in \mathbb{Z}_q$ can be chosen from among q distinct symbols. Hence, there are q^k distinct codewords $\mathbf{c}_i$.

□

The errors that occur during transmission can be represented as words as well. For example, an error $\mathbf{e} \in \mathbb{Z}_2^7$ is simply a word of the same length as the codewords of a binary code $C \subseteq \mathbb{Z}_2^7$. The result of an error $\mathbf{e}$ acting on a codeword $\mathbf{c}$ is then the erroneous word $\mathbf{d} = \mathbf{c} + \mathbf{e}$:

$$\begin{aligned} \mathbf{c} &= (1,0,1,1,0,1,0) \\ \mathbf{e} &= (0,0,1,0,0,0,0) \\ \mathbf{d} = \mathbf{c} + \mathbf{e} &= (1,0,0,1,0,1,0) \end{aligned}$$

Finally, let us introduce the notion of *weight*, which is not only useful for the proofs of the next two lemmas, but also several other concepts that follow.

Definition 2.2 *The* **weight** *of a word* $\mathbf{w}$*, denoted by* $w(\mathbf{w})$*, is the number of its non-zero digits.*

Lemma 2.2 $d(\mathbf{w_1}, \mathbf{w_2}) = w(\mathbf{w_1} - \mathbf{w_2})$ *for any words* w_1, w_2*.*

Proof. $w(\mathbf{w_1} - \mathbf{w_2})$ is the number of non-zero digits in $\mathbf{w_1} - \mathbf{w_2}$, which is the number of digits in which $\mathbf{w_1}$ and $\mathbf{w_2}$ differ. But that is precisely $d(\mathbf{w_1}, \mathbf{w_2})$!

□

Lemma 2.3 *Let C be a linear code. Then $d(C)$ is the smallest weight of any non-zero codeword and is denoted by w_{min}.*

Proof. Let $\mathbf{c_1}, \mathbf{c_2}$ be the codewords closest to each other, such that $d(\mathbf{c_1}, \mathbf{c_2}) = d(C)$, and let $\mathbf{c_3}$ be the codeword with the smallest weight $w(\mathbf{c_3}) = w_{\text{min}}$. Now,

$$\begin{array}{rrcll} & d(C) & = & d(\mathbf{c_1}, \mathbf{c_2}) & \\ & & = & w(\mathbf{c_1} - \mathbf{c_2}) & \text{by lemma 2.2} \\ & & = & w(\mathbf{c}) & \mathbf{c} \text{ is a codeword} \\ & & \geq & w_{\text{min}}, & \\ \text{and} & w_{\text{min}} & = & w(\mathbf{c_3}) & \\ & & = & w(\mathbf{c} - \mathbf{0}) & \\ & & = & d(\mathbf{c}, \mathbf{0}) & \\ & & \geq & d(C). & \end{array}$$

So we have $d(C) \geq w_{\text{min}}$ and $w_{\text{min}} \geq d(C)$, which implies equality.

□

The significance of lemma 2.3 is that it is easy to compute the minimum distance for linear codes: Simply find the word with the smallest weight!

2.2 Encoding

Having laid the mathematical groundwork, we can finally address the practical issue of encoding a message. Remember that a message has to be split into *message words* of some length k. We will use $\mathbb{Z}_q^k$ as our message space, which means that each $\mathbf{m} \in \mathbb{Z}_q^k$ is a message word. To encode $\mathbf{m}$, recall that we need to add *check digits* to it to obtain a codeword of length $n \geq k$. The key to this is to use a linear code C with word length n and dimension k – an $[n, k]$ code.

Specifically, take a $k \times n$ matrix G, whose rows are the codewords of any basis $\mathcal{B} = \{\mathbf{b_1}, \dots, \mathbf{b_k}\}$ of C. Then, to encode $\mathbf{m}$, simply multiply it by G to get a codeword $\mathbf{c} \in C$:

$$\begin{aligned} \mathbb{Z}_q^k &\longrightarrow C \subseteq \mathbb{Z}_q^n \\ \mathbf{m} &\longmapsto \mathbf{c} := \mathbf{m}G = \sum_{i=1}^{k} m_i \mathbf{b_i} \end{aligned}$$

Definition 2.3 *A* **generator matrix** *for an $[n,k]$ code C is a $k \times n$ matrix G, whose rows are the codewords of any basis of C.*

Notice that an $[n,k]$ code C is uniquely defined by a given generator matrix, though usually it has more than one. Take, for example, the $[7,4]$ code defined by the generator matrix

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

According to the prescribed algorithm, the message word $\mathbf{m} = (0,1,1,0)$ under this code is encoded as

$$\mathbf{c} = \mathbf{m}G = (0,1,1,0) \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} = (0,1,1,0,1,0,1).$$

Now take a look at G. Clearly, its four rows are linearly independent and thus constitute a basis for a code C with $dim(C) = 4$. This particular generator matrix, however, has a special feature: It is of the form $G = [I_k|A]$, where I_k is the $k \times k$ identity matrix. Such generator matrices are said to be in *standard form*, and the advantage of using them is that they leave the message word intact and simply "add" $n-k$ check digits (here: $1,0,1$) to it to obtain the codeword – the process we originally described in chapter 1. For actual decoders on the other side of the channel, this is a big time-saving feature because only the first k digit of each received word need to be checked for errors. Only if errors are found in the first k digits do we need to examine the check digits in our decoding process.

2.3 Decoding

While encoding is a fairly straightforward process using the generator matrix, decoding is not. Oftentimes, a theoretically efficient and powerful error-correcting code cannot be used in practice because there is no good algorithm for decoding in a feasible amount of time. Recall that the way we decode a received word **d** is by identifying the closest codeword **c**. Before we can describe how to do that, we need to introduce a few more pieces of mathematics.

The Parity Check Matrix

A parity check matrix of a code C can be used as a tool to identify whether a received word **d** is a codeword of C.

Definition 2.4 *A* **parity check matrix** *for a linear* $[n, k]$ *code* C *is a* $(n-k) \times n$ *matrix* H *that satisfies these two conditions:*

i) all of its rows are independent;

ii) $\mathbf{c}H^T = \mathbf{0} \Leftrightarrow \mathbf{c} \in C$.

As we can see from the definition, **d** is a codeword if and only if the product of **d** and the transpose of H is the zero word. But does every code have a parity check matrix?

Theorem 2.1 *Let* C *be a* $[n, k]$ *code with a standard form generator matrix* $G = [I_k|A]$. *Then a parity check matrix for* C *is* $H = [-A^T|I_{n-k}]$.

Proof. Let the generator matrix of C be

$$G = [I_k|A] = \left[\begin{array}{ccc|ccc} 1 & \cdots & 0 & a_{11} & \cdots & a_{1,n-k} \\ \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 1 & a_{k1} & \cdots & a_{k,n-k} \end{array}\right],$$

and let

$$H = [-A^T|I_{n-k}] = \left[\begin{array}{ccc|ccc} -a_{11} & \cdots & -a_{k1} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ -a_{1,n-k} & \cdots & -a_{k,n-k} & 0 & \cdots & 1 \end{array}\right].$$

Is H really a parity check matrix for C? Clearly, H has the right size, namely $(n-k) \times n$, and its rows are independent, so it remains to show that $\mathbf{c}H^T = \mathbf{0} \; \forall \; \mathbf{c} \in C$.

Recall that every codeword can be written as $\mathbf{c} = \sum_{i=0}^{k} m_i \mathbf{b_i}$, where $m_i \in \mathbb{Z}_q$ are the message word digits and $\mathbf{b_i} \in C$ are the rows of the generator matrix. So to get $\mathbf{c}H^T = m_i \mathbf{b_i} H^T = \mathbf{0}$ we need to ensure that every row of G is orthogonal to every row of H. But the inner product of the i^{th} row of G with the j^{th} row of H is

$$0 + \ldots + 0 + (-a_{ij}) + 0 + \ldots + 0 + a_{ij} + 0 + \ldots + 0 = 0,$$

which means that they are indeed orthogonal.

□

So every linear code has a parity check matrix, which can be derived from a generator matrix using the simple formula in theorem 2.1. Going back to the example in the previous section, the $[7,4]$ code discussed would have a parity check matrix H below. Notice that in the binary case, $-A = A$.

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

A parity check matrix is extremely useful. Among many other things, it allows us to identify the respective code's minimum distance.

Theorem 2.2 *The minimum distance $d(C)$ of a linear code C is the size of the smallest linearly dependent set of columns of its parity check matrix H.*

Proof. Let $\mathbf{c} \in C$ be the codeword with the smallest weight, which is $w_{\min} = w(\mathbf{c}) = d(C)$ by lemma 2.3. Then $\mathbf{c} = (c_1, \ldots, c_n)$, where all but $d(C)$ digits are zero. Next, let $H = [\mathbf{h_1} \cdots \mathbf{h_n}]$ be the parity check matrix of C, where $\mathbf{h_i}$ is the i^{th} column of H. Now, by definition of a parity check matrix,

$$\mathbf{c}H^T = c_1\mathbf{h_1} + \ldots + c_n\mathbf{h_n} = \mathbf{0},$$

but only $d(C)$ of the above terms are non-zero. So the size of the trivially dependent set $\{\mathbf{h_i} \mid c_i \neq 0\}$ of columns of H is $d(C)$.

To show that this is indeed the smallest such set, consider an arbitrary set of t dependent columns: $\{\mathbf{h}_{\sigma(\mathbf{1})}, \ldots, \mathbf{h}_{\sigma(\mathbf{t})}\}$. By definition of linear dependence, there are constants $\lambda_1, \ldots, \lambda_t$, not all zero, such that $\lambda_1 \mathbf{h}_{\sigma(\mathbf{1})} + \ldots + \lambda_t \mathbf{h}_{\sigma(\mathbf{t})} = \mathbf{0}$. Now define the word $\mathbf{c}'$ to have the entries λ_i in its $\sigma(i)^{\text{th}}$ positions and 0 in the remaining positions. Clearly, $\mathbf{c}'H = \mathbf{0}$, so $\mathbf{c}'$ is a codeword, and it is by its definition of weight $\leq t$. But $t \geq w_{min} = d(C)$, so our original set $\{\mathbf{h_i} \mid c_i \neq 0\}$ of size $d(C)$ is indeed the smallest set of linearly dependent columns of H.

□

Hence, to construct a t-error-correcting $[n, k]$ code, it suffices to find an $(n-k) \times n$ parity check matrix such that every set of $2t$ columns is linearly independent (recall theorem 1.2).

Finally, let us make one more definition before we go on to show how decoding is done.

Definition 2.5 *Let* $\mathbf{w}$ *be any word and let* H *be a parity check matrix for a linear code. Then the* **syndrome** *of* $\mathbf{w}$ *is*

$$s(\mathbf{w}) := \mathbf{w}H^T$$

Cosets and Errors

Let $C \subseteq \mathbb{Z}_q^n$ be an $[n, k]$ code. Recall from abstract algebra that the *coset* of C associated with the word (element) $\mathbf{a} \in \mathbb{Z}_q^n$ is defined by

$$\mathbf{a} + C = \{\mathbf{a} + \mathbf{c} \mid \mathbf{c} \in C\}.$$

Further recall that the word space $\mathbb{Z}_q^n$, consisting of q^n words, is partitioned into a finite number of (non-overlapping) cosets of C. This number, given by Lagrange's Theorem, is $|\mathbb{Z}_q^n|/|C| = q^n/q^k = q^{n-k}$. Now, let $\mathbf{d}$ be the received word obtained from the original codeword $\mathbf{c}$ after the error $\mathbf{e}$ occurred: $\mathbf{d} = \mathbf{c} + \mathbf{e}$ Then we have

$$\begin{aligned} \mathbf{c} &= \mathbf{d} - \mathbf{e} \in C \\ \Rightarrow \mathbf{d} + C &= \mathbf{e} + C. \end{aligned}$$

So since both cosets are the same, it follows that both the received word $\mathbf{d}$ and the error word $\mathbf{e}$ are in the same coset. However, in a real-world situation we don't know what $\mathbf{e}$ is; we simply receive $\mathbf{d}$. So which word in the coset $\mathbf{d} + C$ is the error that occurred?

Remember, we are using nearest neighbor decoding, which means that the original codeword $\mathbf{c}$ must be the closest codeword to $\mathbf{d}$: $d(\mathbf{d}, \mathbf{c}) \leq d(\mathbf{d}, \mathbf{c}')$ for all other codewords $\mathbf{c}'$. In other words, the number of digits in which $\mathbf{d}$ and $\mathbf{c}$ differ must be minimal. Therefore, the error word $\mathbf{e}$ must be the word in $\mathbf{d} + C$ with the smallest weight $w(\mathbf{e})$ possible! So, having received the word $\mathbf{d}$, we can simply go through the coset $\mathbf{d} + C$, find the word with the smallest weight, which will be our error $\mathbf{e}$, and then correctly decode $\mathbf{d}$ as $\mathbf{c} = \mathbf{d} - \mathbf{e}$.

This type of algorithm, however, though correct, would be enormously costly in terms of both running time and storage space. Not only would we need to store the entire word space $\mathbb{Z}_q^n$, its elements ordered by cosets, but we would also need to browse through this large set in order to find $\mathbf{d}$ and thus its coset $\mathbf{d} + C$. Fortunately, there is a better way, which will be paved by the next two lemmas.

Lemma 2.4 *The words* $\mathbf{v}, \mathbf{w}$ *belong to the same coset iff* $s(\mathbf{v}) = s(\mathbf{w})$.

Proof. Let $\mathbf{v}, \mathbf{w}$ belong to the same coset. Then

$$\mathbf{v} - \mathbf{w} \in C \Leftrightarrow (\mathbf{v} - \mathbf{w})H^T = \mathbf{0} \Leftrightarrow \mathbf{v}H^T = \mathbf{w}H^T \Leftrightarrow s(\mathbf{v}) = s(\mathbf{w}).$$

□

Lemma 2.5 *Let* C *be a* t*-error-correcting linear code and let* $\mathbf{e_1}, \mathbf{e_2}$ *be correctable errors. Then* $s(\mathbf{e_1}) \neq s(\mathbf{e_2})$.

Proof. Aiming to get a contradiction, suppose that the two arbitrary errors lie in the same coset: $\mathbf{e_1} + C = \mathbf{e_2} + C$. Then $\mathbf{e_1} - \mathbf{e_2} \in C$. But $w(\mathbf{e_1} - \mathbf{e_2}) \leq 2t$, because both errors are correctable and thus of weight at most t each. Therefore, using the worst case scenario that their non-zero digits are in different positions, we get that their difference must be of weight not more than $2t$. Now,

$$\begin{array}{rcll} w(\mathbf{e_1} - \mathbf{e_2}) & \leq & 2t & \\ & < & 2t + 1 & \\ & = & d(C) & \text{by theorem 1.2} \end{array}$$

But $w(\mathbf{e_1} - \mathbf{e_2}) < d(C)$ is a contradiction because $w_{\min} = d(C)$ by lemma 2.3. Therefore, we have $\mathbf{e_1} + C \neq \mathbf{e_2} + C$, i.e. no two correctable errors lie in the same coset. Hence, by the previous lemma 2.4, $s(\mathbf{e_1}) \neq s(\mathbf{e_2})$.

□

A Better Decoding Algorithm

Given a t-error-correcting $[n, k]$ code C with a parity check matrix H in standard form, we can use the following algorithm to decode a possibly erroneous received word $\mathbf{d}$. First, make a table of all the correctable errors $\mathbf{e_i}$ (these are just all the words of weight $\leq t$) and their syndromes $s(\mathbf{e_i})$ and store it. Then,

1. *Calculate* $s(\mathbf{d}) = \mathbf{d}H^T$.
2. *Identify the error* $\mathbf{e}$ *in the table as the word with the same syndrome. If none found,* $\mathbf{d}$ *is not decodable by* C.
3. *Decode to the codeword* $\mathbf{c} = \mathbf{d} - \mathbf{e}$*, and drop the last* $n - k$ *digits to obtain the original message word* $\mathbf{m}$.

This algorithm is obviously more efficient than the previously described scheme because we need to store much less data: merely a table of correctable errors along with their syndromes. Furthermore, we do not need to look for $\mathbf{d}$, but simply compute its syndrome and then look for the same syndrome in a much smaller list of error syndromes. But why exactly does this algorithm work?

First, recall that both the error $\mathbf{e}$ and the received word $\mathbf{d}$ must be in the same coset. Therefore, by lemma 2.4, they have the same syndrome. Hence, we can simply look up $\mathbf{e}$ in the stored table by finding $s(\mathbf{e}) = s(\mathbf{d})$, which we previously computed. The only thinkable problem would be if two errors had the same syndrome, and thus the error look-up was not unique. But this is not the case, as lemma 2.5 asserts. So indeed we have a fast, clean, and most importantly correct nearest-neighbor decoding algorithm.

We close this section with an example (from [1]) illustrating how to use this algorithm. Let C be a ternary $[7, 3]$ code with the parity check matrix

$$H = \begin{bmatrix} 1 & 2 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 2 & 2 & 2 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Check that no pair of columns is dependent, but the first, second, and fourth columns are since

$$2\begin{pmatrix} 1 \\ 1 \\ 0 \\ 2 \end{pmatrix} + 1\begin{pmatrix} 2 \\ 1 \\ 0 \\ 2 \end{pmatrix} + 2\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \mathbf{0}.$$

Therefore, these three columns constitute the smallest set of dependent columns, and so by theorem 2.2, $d(C) = 3$. From this it follows, by theorem 1.2, that C is 1-error-correcting. So $t = 1$ and thus the correctable error words are $\mathbf{0}$ and all words of weight 1. We construct the error-syndrome look-up table accordingly:

$\mathbf{e}$	$s(\mathbf{e})$	$\mathbf{e}$	$s(\mathbf{e})$	$\mathbf{e}$	$s(\mathbf{e})$
1000000	1102	2000000	2201	0000000	0000
0100000	2102	0200000	1201		
0010000	0112	0020000	0221		
0001000	1000	0002000	2000		
0000100	0100	0000200	0200		
0000010	0010	0000020	0020		
0000001	0001	0000002	0002		

Now suppose that we receive the word $\mathbf{d} = (1,0,1,0,1,2,2)$. Simply follow the steps of the algorithm outlined above:

1. $s(\mathbf{d}) = \mathbf{d}H^T = (1,0,0,0)$.

2. In the table we locate $(1,0,0,0)$ as the syndrome of the error $\mathbf{e} = (0,0,0,1,0,0,0)$.

3. Therefore, we decode $\mathbf{c} = \mathbf{d} - \mathbf{e} = (1,0,1,0,1,2,2) - (0,0,0,1,0,0,0) = (1,0,1,2,1,2,2) \Rightarrow \mathbf{m} = (1,0,1,2)$.

2.4 Hamming Codes

Despite all the talk about encoding and decoding, we have still not presented a specific error-correcting code, except partially in the last example. We shall now discuss one class of codes, the *Hamming codes*, more in depth. Hamming codes, one of the oldest class of codes, were first implemented in the 1950s by Richard Hamming, whom we already mentioned in the first chapter. Though these codes only correct one error per codeword, they do so with great efficiency, and are widely used today in DRAM (main memory on PCs) error-correction to protect against hardware failure.

Definition 2.6 *Let $r \geq 2$. A linear code over $\mathbb{Z}_q$ whose parity check matrices have r rows and n columns, where n is the greatest possible number of columns such that no two of them are linearly dependent, is called a* **Hamming Code** *with redundancy r.*

Note that in the binary case $q = 2$, the condition of column independence boils down to all columns being distinct and non-zero, for no two distinct non-zero binary columns can be linearly dependent. The number of distinct binary columns of length r is 2^r, and by subtracting 1 for the **0**-column, we get:

$$n = 2^r - 1$$

in the binary case. The dimension k of the code will therefore be

$$k = n - r.$$

From this we can see the natural role for the *redundancy* r: It is simply the number of check digits in a codeword.

We denote the set of all Hamming codes over $\mathbb{Z}_q$ with redundancy r by $\mathrm{Ham}(r, q)$. Notice that there are several choices for a code for any given parameters q, r, but that they all produce similar results, so-called *equivalent* codes, which only differ in inessential ways (e.g., one can be obtained from the other by performing a positional permutation on the digits of its codewords). Rather than focusing on these details, however, we will turn our attention to why the Hamming codes have been such a success.

Theorem 2.3 *All Hamming codes are have a minimum distance 3 and thus are 1-error-correcting.*

Proof. This directly follows from the definition of a Hamming code in $\mathrm{Ham}(r, q)$. Since n is the *greatest* number of columns of its parity check matrices consistent with the condition that no *two* columns are dependent, the size of the smallest set of *dependent* columns is 3. Therefore, by theorem 2.2, $d(C) = 3$, and so by theorem 1.2, C is 1-error-correcting.

□

Theorem 2.4 *All Hamming codes are perfect.*

Proof. We need to show that for equality holds with the sphere-packing bound of theorem 1.3 for any Hamming code. Since all Hamming codes are 1-error-correcting by the previous theorem, we use $t = 1$ in our formula:

$$\frac{q^n}{\sum_{i=0}^{1} \binom{n}{i} (q-1)^i} = \frac{q^n}{1 + (q^r - 1)} = q^{n-r} = q^k.$$

But q^k is precisely the number of codewords M of a Hamming code because its dimension is k. Thus the desired equality holds, and so the code is by definition perfect.

□

Even though the Hamming codes are "only" 1-error-correcting, this is not so bad for small codewords and completely sufficient for certain applications. Their strength lies in their obvious simplicity and speed with which codewords can be encoded and decoded. Furthermore, note that *all* Hamming codes are 1-error-correcting, no matter what their redundancy is. Redundancy should, however, be chosen carefully based on the code's application, as it greatly affects efficiency. Such issues will be discussed in greater depth in chapter 4.

Again, let us end this last section with a hopefully illuminating example, the already mentioned Hamming(7,4) code in $\text{Ham}(3,2)$. Since this is a binary code, a parity check matrix can be constructed by simply listing all the $2^3 - 1 = 7$ distinct non-zero binary columns of length 3. To get the matrix in standard form, let the last three columns for the identity matrix I_3. One possible parity check matrix is

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

From this, we get the generator matrix using theorem 2.1:

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

Suppose Alice wants to send Bob the message word $\mathbf{m} = (1,0,0,1)$. She will encode it as $\mathbf{c} = \mathbf{m}G = (1,0,0,1,0,1,0)$. Now suppose an error in the third digit occurs, and Bob receives $\mathbf{d} = (1,0,1,1,0,1,0)$. He will compute $s(\mathbf{d}) = \mathbf{d}H^T = (0,1,1)$. In the error-syndrome lookup table (not pictured here), he will find that $\mathbf{e} = (0,0,1,0,0,0,0)$, and thus correctly decode to $\mathbf{c} = (1,0,0,1,0,1,0) \Rightarrow \mathbf{m} = (1,0,0,1)$.

3 Cyclic Codes

Cyclic codes are a subclass of linear codes that have an additional condition imposed upon them – the cyclic property, which we will discuss shortly. The advantage of cyclic codes is that their codewords can be represented as polynomials – as opposed to vectors – over a finite field (again, usually over $\mathbb{Z}_2$). Whereas the language of the linear block codes discussed in the previous chapter is linear algebra, cyclic code properties are generally phrased in terms of ring theory. The reader will, however, notice many similarities, as most of the concepts are analogous.

The Hamming codes from section 2.4 turn out to be cyclic in fact, though we will not discuss this representation here. Instead, we will present the cyclic version of the binary *Golay Code*, which is capable of correcting 3 errors per word. We will also briefly introduce the so-called *Fire Codes*, which are especially efficient at correcting errors that occur in *bursts* rather than randomly.

3.1 Polynomial Code Representation

Let us first consider a regular linear code C with codeword length n. The crucial idea here is that of a *cyclic shift.* If $\mathbf{c} = (c_0, \ldots, c_{n-1}) \in C$, then the cyclic shift of $\mathbf{c}$ is defined as

$$\mathcal{C}(\mathbf{c}) = (c_{n-1}, c_0, \ldots, c_{n-2}).$$

$\mathcal{C}$ shifts all digits to the right by one and places the last digit in the first position. Using this notation, we now define a special class of linear codes.

Definition 3.1 *A linear code C is called a* **cyclic code** *if and only if*

$$\mathbf{c} \in C \Rightarrow \mathcal{C}(\mathbf{c}) \in C.$$

On a first glance, this cyclic property for linear codes seems rather arbitrary and not very useful. Its significance will become apparent, however, when we change our word space.

So far, we have represented words as vectors in $\mathbb{Z}_q^n$, a vector space over a finite field. This turned out to be a convenient representation, but let's try something different now. Let us identify the word $\mathbf{w}$ of length n with the polynomial of degree $n-1$, whose coefficients are the digits of $\mathbf{w}$:

$$\mathbf{w} = (w_0, w_1, \ldots, w_{n-1}) \longmapsto w(x) = w_0 + w_1 x + \ldots + w_{n-1} x^{n-1}$$

The motivating reason for this polynomial representation is that cyclic shifts are particularly easy to perform. Specifically, notice what happens when we multiply a word $w(x) = \sum_{i=0}^{n-1} w_i x^i$ by x:

$$\begin{aligned} x \cdot w(x) &= \qquad w_0 x + w_1 x^2 + \ldots + w_{n-2}x^{n-1} + w_{n-1}x^n \\ &\equiv w_{n-1} + w_0 x + w_1 x^2 + \ldots + w_{n-2}x^{n-1} \qquad \pmod{x^n - 1} \\ &= \mathcal{C}(w(x)). \end{aligned}$$

So if we take as our word space the quotient ring

$$R_n := \mathbb{Z}_q[x]/(x^n - 1),$$

whose elements we will regard simply as polynomials of degree less than n rather than their equivalence classes, then multiplying a word by x corresponds to a cyclic shift of that word. Similarly, multiplying by x^m corresponds to a m cyclic shifts (a shift through m positions).

In this new setting, there is an algebraic characterization of cyclic codes that is quite similar to the definition (2.1) of linear codes.

Theorem 3.1 *A code C is cyclic if and only if it is an ideal in R_n, i.e.*

i) $0 \in C$

ii) $c_1(x), c_2(x) \in C \Rightarrow c_1(x) + c_2(x) \in C$

iii) $r(x) \in R_n, c(x) \in C \Rightarrow r(x)c(x) \in C$

Proof. First, suppose that C is a cyclic code, as defined by definition 3.1.

(i) Since C is a linear code, the zero word is one of its elements.

(ii) Similarly, by def. 3.1, the sum of two codewords is again a codeword.

(iii) Let $r(x)c(x) = r_0 c(x) + r_1(xc(x)) + \ldots + r_{n-1}(x^{n-1}c(x))$. Notice that in each summand, the codeword $c(x)$ is first cyclically shifted by the according number of positions, and then multiplied by a constant $r_i \in \mathbb{Z}_q$. But both of these operations, as well as the final addition of summands, are closed in C by its definition as a linear code, so we indeed have $r(x)c(c) \in C$.

Conversely, suppose that C is an ideal in R_n, and thus (i),(ii),(iii) hold. If we take $r(x)$ to be a scalar in $\mathbb{Z}_q$, these conditions imply that C is a linear code by definition 2.1. If we take $r(x) = x$, (iii) implies that C is also cyclic.

□

3.2 Encoding

Just as linear block codes were constructed using a generator matrix G, cyclic codes can be constructed using a *generator polynomial*, denoted by $g(x)$. But before we introduce it by its name, let us look at some of the properties of this polynomial.

Theorem 3.2 *Let C be a non-zero cyclic code in R_n. Then*

i) there exists a unique monic polynomial $g(x) \in C$ of smallest degree;

ii) the code is generated by $g(x)$: $C = \langle g(x) \rangle$;

iii) $g(x)$ is a factor of $(x^n - 1)$.

Proof.

(i) Clearly, since C is not empty, a smallest degree polynomial $g(x)$ must exist. If it is not monic, simply multiply it by a suitable scalar. Now suppose it was not unique, that is, suppose there are two monic polynomials $g(x), h(x) \in C$ of smallest degree. Then $g(x) - h(x) \in C$, again multiplied by a suitable scalar, is monic and of a smaller degree than $\deg(g(x))$, which is a contradiction.

(ii) Let $c(x) \in C$, and divide it by $g(x)$. By the division algorithm, we get $c(x) = q(x)g(x) + r(x)$, where $\deg(r(x)) < \deg(g(x))$. Now, $c(x), g(x) \in C$ and $q(x) \in R_n$. Since C is an ideal, we can conclude that $r(x) = c(x) - q(x)g(x) \in C$. But since we said that $g(x)$ is the smallest degree polynomial in C, the only possibility for the remainder is $r(x) = 0$. Hence, $c(x) = q(x)r(x)$ for any codeword in C, so we must have $c(x) \in \langle g(x) \rangle$ and thus $C = \langle g(x) \rangle$.

(iii) Again, by the division algorithm, $x^n - 1 = q(x)g(x) + r(x)$, where $\deg(r(x)) < \deg(g(x))$. But $r(x) \equiv -q(x)g(x) \pmod{x^n - 1}$, so $r(x) \in \langle g(x) \rangle = C$. By the minimality of $\deg(g(x))$, we must again have $r(x) = 0$. So $x^n - 1 = q(x)g(x)$, which means that $g(x)$ is a factor of $x^n - 1$.

□

Definition 3.2 *The* **generator polynomial** *of a cyclic code C is the unique monic polynomial $g(x) \in C$ of smallest degree.*

Theorem 3.2 tells us that such a polynomial (i) exists and is unique, (ii) is indeed a generator of C in the algebraic sense, and (iii) can be found by factoring the modulus $(x^n - 1)$. The following table (taken from [2]) shows some of these factorizations for small n:

n	factorization of $(x^n - 1)$ over $\mathbb{Z}_2$
1	$(1 + x)$
2	$(1 + x)^2$
3	$(1 + x)(1 + x + x^2)$
4	$(1 + x)^4$
5	$(1 + x)(1 + x + x^2 + x^3 + x^4)$
6	$(1 + x)^2(1 + x + x^2)^2$
7	$(1 + x)(1 + x + x^3)(1 + x^2 + x^3)$
8	$(1 + x)^8$

Now the natural question becomes: Which of the factors of $x^n - 1$ should we pick as a generator polynomial? The answer depends on what we want the dimension of our code to be. The following theorem clarifies this.

Theorem 3.3 *Let C be a cyclic code of codeword length n and let $g(x) = g_0 + g_1 x + \ldots + g_r x^r$ be its generator polynomial of degree r. Then a generator matrix for C is the $(n - r) \times n$ matrix*

$$G = \begin{bmatrix} g_0 & g_1 & \cdots & g_r & 0 & \cdots & 0 \\ 0 & g_0 & g_1 & \cdots & g_r & & \vdots \\ \vdots & & \ddots & \ddots & & \ddots & 0 \\ 0 & \cdots & 0 & g_0 & g_1 & \cdots & g_r \end{bmatrix}$$

and therefore $k := dim(C) = n - r$.

Proof. First, it is important to note that $g_0 \neq 0$. Suppose it were; then $g(x) = g_1 x + \ldots + g_r x^r$ and since C is cyclic, $x^{n-1} g(x) = x^{-1} g(x) = g_1 + \ldots + g_r x^{r-1} \in C$ would also be a codeword, but of degree $r - 1$, which contradicts the minimality of $\deg(g(x)) = r$.

So the constant coefficient g_0 is indeed non-zero. It follows that the rows of the matrix G are linearly independent because of the echelon of g_0's. It remains to show that the rows constitute a basis of C, i.e. that any codeword can be expressed as a linear combination of these rows, which represent the codewords $g(x), xg(x), x^2 g(x), \ldots, x^{n-r-1} g(x)$.

By theorem 3.3 part (ii), we already know that any codeword $c(x)$ can be written in the form $c(x) = q(x)g(x)$, for some word $q(x)$. Now, $\deg(c(x)) <$

$n \Rightarrow \deg(q(x)) < n - r$, and thus we can write

$$\begin{aligned} c(x) &= q(x)g(x) \\ &= (q_0 + q_1 x + \ldots + q_{n-r-1}x^{n-r-1})g(x) \\ &= q_0[g(x)] + q_1[xg(x)] + \ldots + q_{n-r-1}[x^{n-r-1}g(x)] \end{aligned}$$

which is indeed a linear combination of the rows of G. Therefore, the linearly independent rows are a basis for C, and so by definition 2.3, G is a generator matrix for this code.

□

Let us look at the simple case of binary cyclic codes of codeword length 3. From the factorization table above we can see that $(x^3 - 1) = (1 + x)(1 + x + x^2)$. If we want a code with dimension $k = 2$, then we must pick as generator polynomial the factor with degree $r = 3 - 2 = 1$, which is $1 + x$. We then get the following generator polynomial $g(x)$, generator matrix G, and cyclic code C:

$$g(x) = 1 + x, \quad G = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}, \quad C = \{000, 011, 110, 101\}.$$

On the other hand, if we want a code with dimension 1, we get

$$g(x) = 1 + x + x^2, \quad G = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}, \quad C = \{000, 111\}.$$

3.3 Decoding

The reader will probably have noticed that the generator matrix G we constructed is not in standard form, so we cannot use the method from theorem 2.1 to find a parity check matrix H. Nevertheless, there is a very natural choice for H, much like there was for G.

Definition 3.3 *Let C be an cyclic $[n, k]$ code with the degree r generator polynomial $g(x)$. Then a* **check polynomial** *for C is the degree $k = n-r$ polynomial $h(x)$ such that*

$$g(x)h(x) = x^n - 1$$

Notice that since $g(x)$ is a factor of $x^n - 1$ by theorem 3.2 part (iii), $h(x)$ must exist: It is the "other" factor of $x^n - 1$, which corresponds to 0 in R_n.

Furthermore, since $g(x)$ is by definition monic, $h(x)$ must be monic also, for else we would not get the coefficient 1 for x^n in their product. Finally, the name check polynomial was not chosen at random; $h(x)$ has some familiar properties reminiscent of the parity check matrix H.

Theorem 3.4 *Let C be a cyclic code in R_n with the generator and check polynomials $g(x)$ and $h(x)$, respectively. Then*

$$c(x) \in C \Leftrightarrow c(x)h(x) = 0.$$

Proof. First, suppose that $c(x) \in C$. Since $C = \langle g(x) \rangle$, we have

$$\begin{array}{rcll} c(x) & = & a(x)g(x) & \text{for some } a(x) \in R_n \\ \Leftrightarrow c(x)h(x) & = & a(x)g(x)h(x) & \text{multiply by } h(x) \\ & = & a(x)(x^n - 1) & \text{by definition of } h(x) \\ & \equiv & a(x) \cdot 0 & \text{in } R_n \\ & = & 0 & \text{as desired.} \end{array}$$

Conversely, suppose that $c(x)h(x) = 0$. If we divide $c(x)$ by $g(x)$, then by the division algorithm, $c(x) = q(x)g(x) + r(x)$, where $\deg(r(x)) < n - k$. So we get

$$\begin{array}{rcll} r(x) & = & c(x) - q(x)g(x) & \\ \Leftrightarrow r(x)h(x) & = & c(x)h(x) - q(x)g(x)h(x) & \\ & = & 0 - 0 = 0 & \text{by def. of } h(x) \text{ and assumption.} \end{array}$$

Note that in R_n this really means $r(x)h(x) \equiv 0 \pmod{x^n - 1}$. But we have

$$\deg(r(x)h(x)) = \deg(r(x)) + \deg(h(x))$$

since these are polynomials over $\mathbb{Z}_q$, a finite field. So, $\deg(r(x)h(x)) < (n - k) + k = n$, which means that we really have $r(x)h(x) = 0$ in $\mathbb{Z}_q[x]$. Therefore, the only possibility is $r(x) = 0$, since $h(x)$ is clearly non-zero. Hence, $c(x) = q(x)g(x) \in \langle g(x) \rangle = C$.

□

Just as we could express the generator matrix using the generator polynomial, we can express the parity check matrix using the check polynomial in much the same way.

Theorem 3.5 *Let C be an $[n,k]$ cyclic code with the check polynomial $h(x) = h_0 + h_1x + \ldots + h_kx^k$. Then a parity check matrix for C is*

$$H = \begin{bmatrix} h_k & h_{k-1} & \cdots & h_0 & 0 & \cdots & 0 \\ 0 & h_k & h_{k-1} & \cdots & h_0 & & \vdots \\ \vdots & & \ddots & \ddots & & \ddots & 0 \\ 0 & \cdots & 0 & h_k & h_{k-1} & \cdots & h_0 \end{bmatrix}$$

Proof. We know that $h(x)$ is a monic polynomial and therefore $h_k = 1$, so the h_k's form an echelon in H and thus its rows are linearly independent. We still need to show that $\mathbf{c}H^T = \mathbf{0}$ for all $\mathbf{c} \in C$.

Let $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$, which corresponds to $c(x) = c_0 + c_1x + \ldots + c_{n-1}x^{n-1}$, be any codeword of C. By the definition of $h(x)$, we have $c(x)h(x) = 0$. If we write this out carefully, we see that

$$\begin{aligned} c(x)h(x) &= (c_0 + c_1x + \ldots + c_{n-1}x^{n-1})(h_0 + h_1x + \ldots + h_kx^k) \\ &= c_0h_0 + \ldots \\ &\quad +(c_0h_k + c_1h_{k-1} + \ldots + c_kh_0)x^k \\ &\quad +(c_1h_k + c_2h_{k-1} + \ldots + c_{k+1}h_0)x^{k+1} + \ldots \\ &\quad +(c_{n-k-1}h_k + c_{n-k}h_{k-1} + \ldots + c_{n-1}h_0)x^{n-1} \\ &= 0 \end{aligned}$$

implies in particular that each of the coefficients of $x^k, \ldots, x^{n-1}$ must be 0:

$$\begin{aligned} c_0h_k + c_1h_{k-1} + \ldots + c_kh_0 &= 0 \\ c_1h_k + c_2h_{k-1} + \ldots + c_{k+1}h_0 &= 0 \\ &\vdots \\ c_{n-k-1}h_k + c_{n-k}h_{k-1} + \ldots + c_{n-1}h_0 &= 0 \end{aligned}$$

But notice that this is precisely equivalent to saying that $\mathbf{c}H^T = \mathbf{0}$, which is what we wanted to show.

□

Instead of talking about the generalities of how to encode and decode with cyclic codes, how to find their minimum distance, etc., we will focus on two specific cyclic codes, which have proved to be very useful and effective. This will be the topic of the next two sections.

3.4 The Binary Golay Code

The binary Golay code, though not of much practical use anymore due to innovation in coding, is quite interesting from a mathematical point of view as it leads into related mathematical topics such as group theory and combinatorics (see [3], for instance). That is not to say that it is completely useless, however. It was successfully used in the 1970s on the Voyager 2 spacecraft during its Mission to Jupiter and Saturn, and it is in fact one of the oldest known error-correcting codes. The Golay code is perfect just like the Hamming codes, yet has a greater minimum distance and thus error-correcting capability. It should be noted that there are different ways of constructing the binary Golay code; we will do so via polynomials, since that is the topic of this chapter.

We start by introducing some terminology and an important result. If $f(x) = a_0 + a_1x + \ldots + a_nx^n$ is a polynomial in R_n, then $\bar{f}(x) = a_n + a_{n-1}x + \ldots + a_0x^n$ is called the *reciprocal* of $f(x)$. Using this notation, we state an important fact:

Lemma 3.1 *Let $x^p - 1 = (x-1)g(x)\bar{g}(x)$, where p is a prime, and let $c(x) \in \langle g(x) \rangle$ be a codeword of weight w. Then:*

i) $w^2 - w \geq p - 1$ *if w is odd;*

ii) $w \equiv 0 \pmod 4$ *and $w \neq 4$ unless $p = 7$* *if w is even.*

For a proof of this general result please refer to [1] pp 153-155 or [4] pp 154-156. The discussion is relatively lengthy and not necessary for understanding the rest of this section.

Let us consider the case where $p = 23$, where we get the following factorization (over $\mathbb{Z}_2$):

$$\begin{aligned}(x^{23}-1) &= (x-1)\\ &\quad\cdot(x^{11}+x^{10}+x^6+x^5+x^4+x^2+1)\\ &\quad\cdot(x^{11}+x^9+x^7+x^6+x^5+x+1)\\ &=: (x-1)g(x)\bar{g}(x).\end{aligned}$$

Notice that the two degree 11 polynomials are indeed reciprocals. Next, consider the code generated by $g(x)$.

Definition 3.4 *The binary cyclic code in R_{23} with generator polynomial $g(x) = 1+x^2+x^4+x^5+x^6+x^{10}+x^{11}$ is called the binary* **Golay Code***, denoted by G_{23}.*

With the help of the previous lemma, we will now prove the crucial property of the Golay code.

Theorem 3.6 *G_{23} is a perfect three-error-correcting $[23, 12]$ code.*

Proof. By theorem 3.3, G_{23} is a $[23, 12]$ code because its generator $g(x)$ of degree 11 is a product of $x^{23} - 1$, as shown above. Let $c(x) \in G_{23}$ be a codeword with minimum weight $w_{\min}$. If this weight is odd, then by lemma 3.1.(i) we must have

$$\begin{array}{rrcl} & w_{\min}^2 - w_{\min} & \geq & 23 - 1 \\ \Rightarrow & w_{\min}(w_{\min} - 1) & \geq & 22 \\ \Rightarrow & w_{\min} & \geq & 7. \qquad \text{(try it!)} \end{array}$$

On the other hand, lemma 3.1.(ii) implies that G_{23} has no words of even weight < 8, because $p = 23 \neq 7$. Finally, we can see that $w(g(x)) = 7$, so the minimum distance $d(G_{23}) = 7$. By theorem 1.2, the binary Golay code is therefore 3-error-correcting.

It remains to show the G_{23} is perfect. Checking the sphere packing bound (theorem 1.3) for $t = 3$, we get:

$$\begin{aligned} \frac{2^{23}}{\sum_{i=0}^{3} \binom{23}{i}(2-1)^i} &= \frac{2^{23}}{1 + 23 + \frac{23\cdot 22}{2} + \frac{23\cdot 22\cdot 21}{6}} \\ &= \frac{2^{23}}{1 + 23 + 253 + 1771} \\ &= \frac{2^{23}}{2048} = \frac{2^{23}}{2^{12}} = 2^{11} \end{aligned}$$

which is exactly 2^k, the number of codewords M, and thus G_{23} is perfect.

□

3.5 Burst Errors and Fire Codes

So far, we have not addressed the issue of what types of errors occur in a noisy channel. We have simply assumed that errors, in the form of binary digit changes, occur randomly throughout a message with some small, uniformly distributed probability for any given digit. Our only criterion for a code's error-correcting capability so far is its minimum distance, from which we can tell how many errors per codeword can be corrected. However, if we know that only certain types of errors will occur, we can greatly improve our coding efficiency (in terms of information rate).

Among particularly common types of errors are so-called *burst errors*. A burst error $\mathbf{e}$ of length t is simply an error in t consecutive codeword digits:

$$\mathbf{e} = (0, \ldots, 0, \underbrace{1, \ldots, 1}_{t \text{ digits}}, 0, \ldots, 0)$$

As a polynomial, a burst error can be expressed as

$$e(x) = x^i b(x) = x^i (b_0 + b_1 x + \ldots + b_{t-1} x^{t-1})$$

i.e. a cyclic shift by i digits of a polynomial $b(x)$ with $\deg(b(x)) = t - 1$ and non-zero coefficients b_j. Such errors often occur in digital data storage devices such as CDs, for example, where physical scratches and other distortions are usually the cause. It turns out that burst errors can be corrected with much less overhead than random errors of the same size.

The *Fire codes*, originally discovered by P. Fire in 1959[2], are one class of such burst-error-correcting codes. In fact, these codes are considered to be the best single-burst-correcting codes of high transmission rate known, according to [5].

Definition 3.5 *A* **Fire Code** *of codeword length n is a cyclic code in R_n with generator polynomial*

$$g(x) = (x^{2t-1} - 1)p(x)$$

where $p(x)$ is an irreducible polynomial over $\mathbb{Z}_q$ that does not divide $(x^{2t-1} - 1)$ and with $m := \deg(p(x)) \geq t$, and n is the smallest integer such that $g(x)$ divides $(x^n - 1)$.

For example, the $[105, 94]$ code with generator polynomial $g(x) = (1 + x^7)(1 + x + x^4)$ is a Fire code, but notice that a Fire code need not (and usually does not) exist for any given n. Nevertheless, it turns out that the

[2] see [10]

number t in the definition above is precisely the maximum correctable burst length.

Lemma 3.2 *A Fire code has codeword length* $n = \mathrm{lcm}(2t-1, q^m - 1)$.

Proof. Let us denote $g(x) = a(x)p(x)$, where $a(x) = x^{2t-1} - 1$. We want n to be the smallest integer such that $g(x) \mid (x^n - 1)$. But by definition, $p(x)$ is irreducible and does not divide $a(x)$, which implies that $\gcd(a(x), p(x)) = 1$. Therefore,

$$\begin{aligned} & a(x) \mid (x^n - 1) \\ \text{and} \quad & p(x) \mid (x^n - 1) \end{aligned}$$

imply that $g(x) \mid (x^n - 1)$, so we will find the smallest n that satisfies these two conditions.

The smallest integer e for which $p(x) \mid (x^e - 1)$ is $e = q^m - 1$, where $m = \deg(p(x))$. Clearly, $e \mid \mathrm{lcm}(2t-1, e)$, so $(x^e - 1) \mid (x^{\mathrm{lcm}(2t-1,e)} - 1)$. But

$$p(x) \mid (x^e - 1) \text{ and } (x^e - 1) \mid (x^{\mathrm{lcm}(2t-1,e)} - 1) \Rightarrow p(x) \mid (x^{\mathrm{lcm}(2t-1,e)} - 1).$$

On the other hand, $(2t-1) \mid \mathrm{lcm}(2t-1, e)$, so

$$(x^{2t-1} - 1)) \mid (x^{\mathrm{lcm}(2t-1,e)} - 1) \Leftrightarrow a(x) \mid (x^{\mathrm{lcm}(2t-1,e)} - 1).$$

Therefore, (1) and (2) are satisfied by $n = \mathrm{lcm}(2t-1, e) = \mathrm{lcm}(2t-1, q^m-1)$, and since this is the *least* common multiple, they cannot by satisfied by any smaller n.

□

Theorem 3.7 *A Fire code can correct all burst errors of length* t *or less.*

Proof. We base our argument on the presentation in [11], pp 480-481. Let $d(x) = c(x) + e(x)$ be the received word, where $c(x)$ is the original Fire code codeword and $e(x) = x^i b(x)$ is a burst error of length t or less. We want to show that we can uniquely identify this burst error given only our received word $d(x)$. In order to do that, we need to find

- the burst pattern $b(x)$ of length $\leq t$ and thus degree $\leq t - 1$, and
- the position i of this burst pattern in a word of length n.

Since $i < n$, it suffices to find $i \pmod{n}$. But by lemma 3.2, $n = \operatorname{lcm}(2t-1, q^m - 1)$, so it suffices to find $i \pmod{2t-1}$ and $i \pmod{q^m-1}$.

First, let us find $b(x)$ and $i \pmod{2t-1}$. Define

$$\begin{array}{rcll} s_1(x) & := & d(x) \pmod{x^{2t-1}-1} & \\ & = & c(x) + e(x) \pmod{x^{2t-1}-1} & \\ & = & e(x) \pmod{x^{2t-1}-1} & \text{because } c(x) \text{ is a multiple of } g(x), \\ & & & \text{which is a multiple of } x^{2t-1}-1 \\ & = & x^i b(x) \pmod{x^{2t-1}-1}. & \end{array}$$

Since n is a multiple of $2t-1$, our error word will look like this:

$$e(x) = (\overbrace{0,\ldots,0}^{2t-1},\overbrace{0,\ldots,0}^{2t-1},\ldots,\overbrace{0,\ldots,\quad\underbrace{1,\ldots,1}_{b(x)\text{ at position } i}\quad,\ldots,0}^{2t-1},\ldots,\overbrace{0,\ldots,0}^{2t-1})$$

Taking this burst error modulo $(x^{2t-1}-1)$ will simply reduce all exponents to values less than $2t-1$, and shift the burst pattern into the first $2t-1$ digits without changing it:

$$s_1(x) = (\overbrace{0,\ldots,0,\quad\underbrace{1,\ldots,1}_{b(x)\text{ at position } i'}\quad,0,\ldots,0}^{2t-1}).$$

Here, $i' := i \pmod{2t-1}$, and so we have $s_1(x) \equiv x^{i'}b(x) \pmod{x^{2t-1}-1}$, where both $b(x)$ and i' are uniquely determined by the received word $d(x)$. Thus we have found the burst pattern $b(x)$ and its position $i \pmod{2t-1}$.

It remains to show that $i \pmod{q^m-1}$ is uniquely determined as well. To that end, write

$$i = i' + k(2t-1)$$

for some multiple $k < \frac{n}{2t-1}$. We already know i', so we only need to find $k(2t-1) \pmod{q^m-1}$. In other words, we need $k \left(\operatorname{mod} \frac{q^m-1}{\gcd(2t-1,q^m-1)}\right)$. Define

$$\begin{array}{rcll} s_2(x) & := & d(x) \pmod{p(x)} & \\ & = & c(x) + e(x) \pmod{p(x)} & \\ & = & e(x) \pmod{p(x)} & \text{because } c(x) \text{ is a multiple of } g(x) \\ & & & \text{which is a multiple of } p(x) \\ & = & x^i b(x) \pmod{p(x)} & \\ & = & x^{i'+k(2t-1)} b(x) \pmod{p(x)} & \\ & = & q(x)p(x) + x^{i'+k(2t-1)}b(x) & \text{for some polynomial } q(x). \end{array}$$

Now let $\alpha \in \mathbb{F}_{q^m}^{\times}$ be a root of $p(x)$, which must have order $e = q^m - 1$, for that is the smallest integer for which $p(x)$ divides $x^e - 1$ (recall lemma 3.2). Then

$$s_2(\alpha) = q(\alpha) \cdot 0 + \alpha^{i'+k(2t-1)} b(\alpha),$$

and since $p(x)$ is irreducible and $\deg(b(x)) < t \leq m = \deg(p(x))$, we have $b(\alpha) \neq 0$, and so

$$\alpha^{i'+k(2t-1)} = \frac{s_2(\alpha)}{b(\alpha)}.$$

Now we can use this equation to find $k \pmod{\frac{q^m-1}{\gcd(2t-1,q^m-1)}}$. First we compute the right-hand side and then we match the exponent to the left-hand side, where α is a generator of the cyclic group $\mathbb{F}_{q^m}^{\times}$.

Hence, we have found all that we needed to recover the burst error $e(x) = x^i b(x)$ of length t or less from any received word $d(x)$. Therefore, the Fire codes are indeed t-burst-error-correcting.

□

An alternative proof of theorem 3.7 can be found in texts such as [5]. Having established the error-correcting capabilities of Fire codes, we will next test the performance of the Fire code with $t = 3$ and see how it measures up to the other codes we have discussed. This will be the topic of the subsequent, final chapter.

4 Code Performance

This final chapter is very different from the previous ones in that we do not introduce any new theory. Instead, the sole focus will be on the performance of the three classes of codes we have described so far – the Hamming codes, the binary Golay code, and the Fire codes – given different types and frequencies of errors. To this end, we have used a test program that implements these various codes and tests their error-correcting capabilities.

The first section discusses some of the implementation details of this program. The next section fully describes the performance test and presents the data obtained from it. The final section then discusses several highlights in the results and compares the test data to theoretically expected values.

Although there is by no means any scientific breakthrough here, the test results we discuss here are nevertheless interesting in so far as they highlight the inverse relationship between a code's error-correcting capability (as measured by its *correction rate*) and coding efficiency (as measured by its *information rate*). Finding new codes with a higher error-correcting capability while maintaining efficiency and keeping implementation overhead low is an area of ongoing research.

4.1 Implementation Issues

The program used to test code performance was written in Java. Hence, it was natural to approach the problem from an object-oriented point of view.

There is a `Word` class for binary words, which stores the digits of the word in an array, along with the word's length and weight. In addition, the class implements the addition and scalar product of two words. All coding operations in the program are done in terms of `Word` objects, which in particular implies that all codes are binary.

A `Code` class is used as a template for a generic linear error-correcting code with codeword length `n`, dimension `k`, redundancy `r`, generator matrix `G`, and parity check matrix `H`. In addition, a table of correctable errors hashed by their syndromes, such as the one from our example in section 2.3, is stored. The methods for encoding and decoding are implemented in this class as well.

Extending the `Code` class are the `Hamming` and `Cyclic` classes, each of which implements a constructor for the particular kind of code desired. The job of the constructor is to build appropriate generator and parity check matrices as well as an error table for some given input parameters.

The only parameter for the `Hamming` constructor is an integer r for the code redundancy, which is stored as `r`. All other information can be derived from r, as discussed in section 2.4:

- $n = 2^r - 1$
- $k = n - r$
- H is the $r \times n$ matrix whose columns are all the non-zero binary words of length r, arranged in the standard form $H = [A|I_r]$;
- G is the $k \times n$ matrix $[I_k|A^T]$ (by theorem 2.1 - binary case);
- The error table contains all errors of weight 1 (plus the zero word) since all codes in $\mathrm{Ham}(r, 2)$ are 1-error-correcting.

An interesting real-world implementation issue is the question of how to efficiently generate the columns of H. In other words, we want to generate all the distinct non-zero binary words of length r. A nice way to do that is to use a *Gray Code*[3], which is a method of listing all binary words of a given length in an order such that not more than one digit changes from one word in the list to the next. For example, compare the "standard" way of listing consecutive binary integers of length 3 in ascending order to the corresponding Gray Code:

$0_2 - 7_2$	Gray Code
000	000
00**1**	00**1**
0**10**	0**1**1
01**1**	01**0**
100	**1**10
10**1**	11**1**
1**10**	1**0**1
11**1**	10**0**

The digits that have changed from one word to the next are highlighted on both sides. Note that while only one digit changes in the Gray Code, up to all three digits change in the left column. For this reason, the `Hamming` class uses a Gray Code when constructing the parity check matrix (a special `GrayCode` class was written for this).

The `Cyclic` constructor needs significantly more information than just redundancy to construct a cyclic code. Its input parameters are the codeword

[3] see e.g. [9]

length n, a generator polynomial $g(x)$ and a check polynomial $h(x)$. Using this information, the desired `Cyclic` code object is built.

4.2 Test Description and Results

We tested the following six error-correcting codes for their performance. Note that they are all linear $[n, k]$ codes.

1. `H7`: A Hamming $[7, 4]$ code with redundancy 3.
2. `H15`: A Hamming $[15, 4]$ code with redundancy 4.
3. `H31`: A Hamming $[31, 26]$ code with redundancy 5.
4. `H63`: A Hamming $[63, 57]$ code with redundancy 6.
5. `G23`: The binary Golay $[23, 12]$ code with generator polynomial $g(x) = 1 + x^2 + x^4 + x^5 + x^6 + x^{10} + x^{11}$.
6. `F35`: The $[35, 27]$ Fire code with generator polynomial $g(x) = (x^{2\cdot 3-1} - 1)p(x) = (1+x^5)(1+x+x^3) = 1+x+x^3+x^5+x^6+x^8$.

We have seen that the Hamming codes are all 1-error-correcting, the Golay code is 3-error-correcting, and this particular Fire code can correct all burst errors up to burst length $t = 3$. Table 1 below lists the codes' respective information rates. Recall that $\rho = \frac{1}{n} \log_q M$, and in all cases we have $M = q^k$ (by lemma 2.1) and $q = 2$.

The test is then conducted as follows. First, an arbitrary message consisting of 10,000 bits is encoded into codewords for each of the six codes. Then, *one* of the following two types of errors is introduced throughout the entire message:

- A **random error pattern**, where the probability that any given codeword digit c_i gets corrupted is p. In other words, the probability of each error word of weight 1 (of the form `0...010...0`) occurring is p. Note, however, that it is possible that more than one such error occurs per codeword. This is obviously more likely when codewords are longer.
- A **burst error pattern**, where the probability that a burst of length 3 occurs starting at any given digit c_i is p. In other words, the probability of each burst error word of weight 3 (of the form `0...01110...0`) occurring is p. Notice that in this scenario, it is possible that bursts of

length greater or less than three appear as a result of the conceivable event of two or more bursts in one codeword (burst error underlined):
$\mathtt{1101000} \xrightarrow{\mathbf{b_1}=0111000} \mathtt{1\underline{010}000} \xrightarrow{\mathbf{b_2}=0000111} \mathtt{1\underline{010111}}$, vs.
$\mathtt{1101000} \xrightarrow{\mathbf{b_1}=0111000} \mathtt{1\underline{010}000} \xrightarrow{\mathbf{b_2}=0011100} \mathtt{1\underline{0}0\underline{01}00}$.

Once the errors are introduced, the possibly erroneous words are decoded, and the program checks how many of these words have been decoded correctly, i.e. restored to their original codewords. We call the percentage of received words that has been correctly decoded the *correction rate*, denoted by $\mathcal{R}$. This measure of error-correcting capability is computed for each of the six codes for different values of p, ranging from 0 to 0.06. The process is repeated 30 times (for accuracy) and mean values of $\mathcal{R}$ are reported. The results of this test are presented in figures and tables 2 and 3 on the subsequent pages.

TABLE 1. *Information rates.*

code	`H7`	`H15`	`H31`	`H63`	`G23`	`F35`
ρ	57.142%	73.333%	83.871%	90.476%	52.174%	77.143%

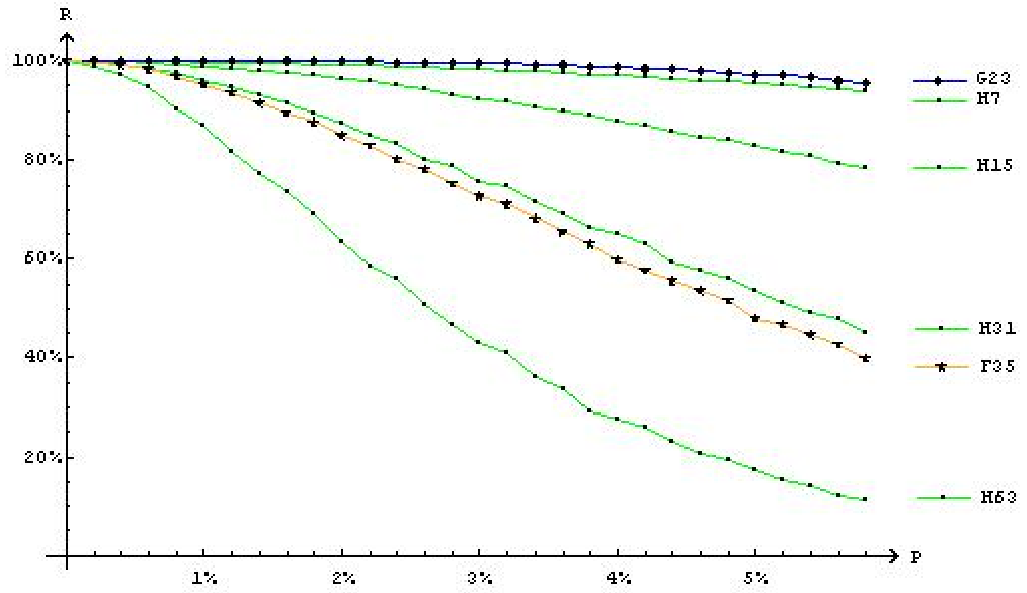

Figure 2: Code performance, given the **random** error pattern described.

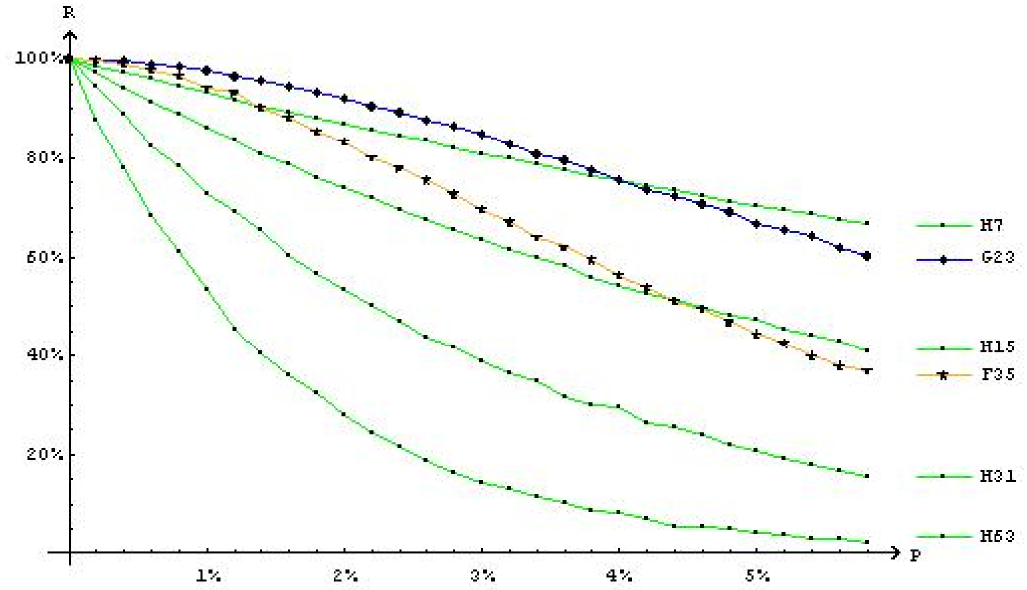

Figure 3: Code performance, given the **burst** error pattern described.

TABLE 2. *Data for the* ***random*** *error pattern in figure 2.*

p	H7	H15	H31	H63	G23	F35
0.000	100.000%	100.000%	100.000%	100.000%	100.000%	100.000%
0.002	99.992%	99.969%	99.823%	99.318%	100.000%	99.827%
0.004	99.968%	99.835%	99.423%	97.261%	100.000%	99.191%
0.006	99.918%	99.624%	98.499%	94.205%	99.998%	98.156%
0.008	99.863%	99.363%	97.512%	90.909%	100.000%	97.084%
0.010	99.804%	99.066%	96.114%	86.943%	99.986%	95.321%
0.012	99.722%	98.626%	94.805%	82.682%	99.981%	93.822%
0.014	99.607%	98.158%	92.758%	77.648%	99.990%	91.369%
0.016	99.498%	97.640%	91.429%	72.898%	99.952%	89.806%
0.018	99.381%	96.993%	89.205%	69.136%	99.930%	87.542%
0.020	99.258%	96.499%	87.055%	64.045%	99.921%	85.062%
0.022	99.039%	95.752%	84.592%	59.955%	99.878%	83.084%
0.024	98.903%	95.042%	83.164%	54.966%	99.799%	80.119%
0.026	98.660%	94.196%	80.618%	50.898%	99.765%	78.151%
0.028	98.503%	93.327%	78.660%	47.068%	99.669%	75.391%
0.030	98.294%	92.811%	76.530%	42.716%	99.525%	73.488%
0.032	98.054%	91.932%	73.564%	40.045%	99.465%	70.588%
0.034	97.869%	90.930%	72.083%	36.580%	99.314%	67.741%
0.036	97.566%	90.068%	68.748%	33.182%	99.062%	64.997%
0.038	97.404%	88.905%	66.914%	30.625%	98.945%	63.321%
0.040	96.988%	88.075%	64.935%	27.614%	98.727%	60.620%
0.042	96.811%	86.866%	62.431%	24.773%	98.532%	58.092%
0.044	96.554%	86.123%	59.725%	23.023%	98.305%	55.332%
0.046	96.204%	85.167%	58.151%	21.102%	97.964%	53.499%
0.048	95.910%	84.143%	55.548%	18.977%	97.830%	50.846%
0.050	95.499%	82.752%	54.390%	16.875%	97.456%	48.739%
0.052	95.236%	81.712%	51.003%	15.614%	97.079%	46.965%
0.054	94.806%	80.692%	49.538%	13.886%	96.751%	44.253%
0.056	94.576%	79.393%	47.434%	11.898%	96.403%	42.863%
0.058	94.170%	78.473%	45.210%	11.727%	95.986%	40.749%

TABLE 3. *Data for the* ***burst*** *error pattern in figure 3.*

p	H7	H15	H31	H63	G23	F35
0.000	100.000%	100.000%	100.000%	100.000%	100.000%	100.000%
0.002	98.606%	97.059%	93.735%	88.477%	99.926%	99.779%
0.004	97.334%	94.244%	87.906%	77.182%	99.624%	99.089%
0.006	95.808%	91.327%	83.117%	68.693%	99.108%	97.914%
0.008	94.597%	88.629%	78.255%	60.170%	98.345%	96.415%
0.010	93.225%	86.097%	73.060%	53.080%	97.683%	94.706%
0.012	91.933%	83.429%	68.894%	45.761%	96.715%	92.814%
0.014	90.750%	80.998%	64.732%	41.602%	95.667%	90.566%
0.016	89.416%	78.347%	60.992%	36.261%	94.552%	87.903%
0.018	88.155%	76.347%	57.294%	32.398%	93.333%	85.569%
0.020	87.025%	73.593%	53.740%	28.705%	92.007%	83.445%
0.022	85.664%	71.771%	50.566%	23.977%	90.451%	80.431%
0.024	84.419%	69.554%	47.060%	21.636%	89.432%	77.784%
0.026	83.373%	67.464%	44.930%	18.523%	87.777%	75.111%
0.028	82.110%	65.503%	41.673%	17.523%	86.046%	72.162%
0.030	81.290%	63.532%	38.997%	15.034%	84.072%	69.040%
0.032	79.966%	61.224%	36.499%	13.307%	82.350%	66.776%
0.034	78.681%	60.404%	34.919%	11.102%	81.048%	64.210%
0.036	77.692%	57.954%	31.953%	9.295%	79.245%	61.687%
0.038	76.766%	56.270%	29.860%	8.125%	77.213%	59.224%
0.040	75.559%	54.501%	27.600%	7.591%	75.957%	56.582%
0.042	74.409%	52.952%	26.831%	6.852%	73.847%	54.059%
0.044	73.311%	51.066%	25.143%	5.466%	72.089%	50.658%
0.046	72.502%	49.798%	23.616%	5.307%	70.276%	48.809%
0.048	71.272%	47.956%	21.881%	4.330%	68.662%	47.067%
0.050	70.402%	47.068%	20.660%	3.489%	67.168%	44.361%
0.052	69.531%	45.459%	19.506%	3.670%	65.643%	42.097%
0.054	68.352%	43.690%	18.343%	3.330%	63.633%	39.655%
0.056	67.334%	42.338%	16.919%	2.614%	61.739%	38.852%
0.058	66.566%	41.365%	16.156%	2.193%	60.345%	36.081%

4.3 Conclusions

First, we observe that the experimental data is quite close to theoretical expected values. Take, for example, $p = 0.02$ in the random error pattern. Now consider, say, the Hamming code `H31`, which is a 1-error-correcting code. The probability that one error or less occurs per codeword is

$$\sum_{i=0}^{1} \binom{31}{i} 0.02^i 0.98^{31-i} = 0.98^{31} + 31 \cdot 0.02 \cdot 0.98^{30} = 87.277\%$$

which is close to its measured error-correcting capability, 87.055% (see table 2). As another example, consider the 3-error-correcting Golay code `G23`. Its expected error-correcting capability is

$$\begin{aligned} & \textstyle\sum_{i=0}^{3} \binom{23}{i} 0.02^i 0.98^{23-i} \\ = \; & 0.98^{23} + 23 \cdot 0.02 \cdot 0.98^{22} + 253 \cdot 0.02^2 \cdot 0.98^{21} + 1771 \cdot 0.02^3 \cdot 0.98^{20} \\ = \; & 99.896\% \end{aligned}$$

as compared to the measured 99.921%. These sample calculations indicate that the test went as it was intended.

Next, we make some qualitative observations, and note that the results we got in figures 2 and 3 should not be very surprising. The graphs clearly show that a code's correction rate $\mathcal{R}$ decreases as the error rate p increases – this makes perfect sense. Furthermore, as figure 2 unambiguously illustrates, the general trend is that increasing error-correcting capability carries with it an explicit cost in terms of efficiency.

For example, on a first glance `G23` might seem like the best code because of its consistently high $\mathcal{R}$ value. But a look at table 1 reveals that it is also the least efficient, with its information rate at only 52%. In fact, by comparing table 1 and figure 2, we can see that the higher a code is on the graph (the greater its $\mathcal{R}$), the lower is its efficiency ρ – with the sole exception of `F35`, which performs marginally *worse* than `H31`, despite *lower* efficiency.

That is not to say, however, that `F35` is a particularly poorly performing code. On the contrary, it proves to perform significantly better relative to other codes when faced with the types of burst errors it was designed for, as can be seen in figure 3. Its performance is especially strong for small p, as this decreases the probability of more than one burst error occurring in one of its comparatively long codewords. Not only does the Fire code now outperform `H31` for all p, but also `H15` for $p < 4.4\%$ and `H7` for $p < 1.4\%$, whose information rates (73% and 57%, respectively) are considerably lower than its own 77%.

Finally, this brings us to the question of why relative error-correcting capability changes with p (i.e., why the graphs cross). The reason has to do with the codeword length n of each code. Take, for example, `F35` and `H7` for the burst error pattern in figure 3. The Fire code corrects all burst errors of length 3 or less per codeword, but no more. The Hamming Code on the other hand only corrects one single error per codeword. All errors that occur in this part of the experiment are burst errors of length 3 that occur with a probability p in each of the n digits of a transmitted codeword. Therefore, `F35` correctly decodes a word $\mathbf{d}$ with up to one such error. `H7`, however, correctly decodes $\mathbf{d}$ if and only if *no* such error occurs (or only one of the three distorted digits lies in $\mathbf{d}$ – the first burst digit in position $n-1$ or the last burst digit in position 0).

For small error rates, the probability that *more* than one burst error occurs in a word of length 35 is smaller than the probability that one burst error occurs in a word of length 7. Consequently, `F35` initially outperforms `H7`. However, as p increases, the probability that the 35-digit codewords of `F35` get corrupted with more than one burst error becomes greater than the probability that the shorter 7-digit codewords of `H7` get uncorrectably corrupted. Hence, for $p = 1.4\%$ and greater, the short codewords of the Hamming code become a more valueable asset than burst-error-correcting capability of the Fire code, and `H7` takes over as the code with the higher correction rate.

Final Remarks

These results reinforce the idea that it is very important to carefully choose an error-correcting code that is suitable for the particular noisy channel in our application. We need to ask ourselves:

- How often do errors occur? In other words, how big is p?
- What types of errors occur? Random, bursts, mixed, other?
- What is a tolerable error correction rate? Can I understand the message even if it is partially erroneous?
- How quick does the encoding and decoding algorithm have to be? Does it have to be nearly instantaneous (such as in cell phones) or is there more time to process data?

It is questions such as these that ultimately determine the choice of a code and the associated trade-off between efficiency and decoding accuracy. Clearly, both qualities are desirable and we want as much as possible of both, which is why coding theorists are constantly working to develop new, application-specific codes that offer the best of both worlds. In a world where an ever increasing amount of information is being transmitted and stored, the need for such codes is as apparent and as critical as ever.

A Glossary

- **burst error** – An error that corrupts two or more consecutive digits in a *codeword.* (p 33)

- **check digits** – Extra digits added to each *message word* to obtain a *codeword.* They are used to detect and correct *errors* in the message. (p 4)

- **check polynomial** – A polynomial $h(x)$ such that when any *codeword* $c(x)$ of a *cyclic code* is multiplied by it, we get $c(x)h(x) = 0$. (p 27)

- **code, error-correcting** – A collection of all *codewords* of a particular type, usually denoted by C. Error-correcting codes allow us to detect and correct unavoidable errors in messages sent across a *noisy channel.* (p 2)

- **codeword** – A *word* (element) of a code. Notice the following relation: *message word* + *check digits* = *codeword.* (p 3)

- **correction rate** – The percentage of received words that are correctly decoded by some *error-correcting code.* (p 39)

- **cyclic code** – A *linear code* with the property that a cyclic shift of a *codeword* **c** is also a *codeword.* The cyclic shift of $\mathbf{c} = (c_1, c_2, ..., c_{n-1})$ is simply $(c_{n-1}, c_1, ..., c_{n-2})$. Codewords of cyclic codes are often represented as polynomials over a finite field. (p 23)

- **equivalent codes** – Two *linear codes* are called equivalent if and only if the *generator matrix* of one code can be obtained from the other by using only row or column permutations, multiplying a row or column by a non-zero scalar, and adding a row to another. (p 21)

- **error** – A distortion of one or more digits in a *codeword* that is sent across a *noisy channel.* An error **e** in a codeword **c** is simply represented as a *word* of the same length as **c**. The resulting distorted word is $\mathbf{d} = \mathbf{c} + \mathbf{e}$. (p 3)

- **Fire Codes** – *Cyclic codes* that is especially suitable for correcting single *burst errors.* (p 32)

- **generator matrix** – A *linear code* C can be obtained by multiplying each message word $\mathbf{m}$ by its generator matrix G: $C = \{\mathbf{c} = \mathbf{m}G\}$. Generator matrices are used for encoding. (p 14)

- **generator polynomial** – A polynomial $g(x)$ that generates a *cyclic code* C: $C = \langle g(x) \rangle$. (p 25)

- **Gilbert-Varshmanov bound** – A lower bound on the number of *codewords* in a *code*. (p 9)

- **Golay Code, binary** – A 3-error-correcting perfect *cyclic code* that is largely archaic. (p 30)

- **Hamming Codes** – A *linear code* with the property that its *parity check matrix* has r rows and n columns, where r is the *redundancy* of the code, and n is the greatest possible number of columns consistent with the condition that no pair of columns is linearly dependent. By design, all Hamming Codes are *perfect* and one-error-correcting. (p 20)

- **Hamming distance** – The Hamming distance $d(\mathbf{c_1}, \mathbf{c_2})$ between the two *codewords* $\mathbf{c_1}$ and $\mathbf{c_2}$ is the number of digits in which they differ. (p 4)

- **information rate** – A measure of efficiency of a *code*. Given by $\rho = \frac{1}{n} \log_q M$. (p 8)

- **linear code** – A collection of all *codewords* such that the sum or scalar multiple of any codewords is also a codeword. Mathematically, a linear code can be represented as a subspace of a finite-dimensional vector space (*word space*) over a finite field. (p 11)

- **message word** – A *word* (piece) of a message before it is encoded. (p 3)

- **minimum distance** – The minimum distance $d(C)$ of a *linear code* C is the smallest *Hamming distance* between any two *codewords* of that code. (p 4)

- **noisy channel** – A communication channel with the property that information sent across it may potentially get distorted. In other words, *errors* may occur in a message. (p 2)

- **parity check matrix** – A *linear code* can be specified by its parity check matrix H, which has the property that $\mathbf{c}H^T = \mathbf{0}$. Parity check matrices are primarily used in decoding. (p 15)

- **perfect code** – A *code* is called perfect if and only if the number of its codewords is equal to the *sphere packing bound*. (p 9)

- **redundancy** – The number of *check digits* in each *codeword*. (p 21)

- **sphere packing bound** – An upper bound on the number of *codewords* in a *code* based on the idea of sphere packing. It states that $|C| \leq q^n(\sum_{i=0}^{t} \binom{n}{i}(q-1)^i)^{-1}$. (p 9)

- **standard form** – A *generator matrix* is in standard form when it is in the form $G = [I|A]$, where I is an identity matrix. (p 14)

- **syndrome** – The syndrome of a *word* $\mathbf{w}$ is $\text{syn}(\mathbf{w}) = \mathbf{w}H^T$, where H is the *parity check matrix* of the relevant code. If the word is a *codeword*, then its syndrome is $\mathbf{0}$. (p 17)

- **weight** – The weight $w(\mathbf{c})$ of a *codeword* $\mathbf{c}$ is the number of its non-zero digits. (p 12)

- **word** – A block of information consisting of digits in a certain alphabet. Mathematically, this is usually a vector in a finite-dimensional vector space over a finite field, or a polynomial representative in $\mathbb{F}_q[x]/(x^n - 1)$. (p 3)

B References

1. Baylis, J. (1998) *Error-Correcting Codes A Mathematical Introduction*, Chapman & Hall/CRC.

2. Varstone, S.A. and van Oorschot, P.C. (1992) *An Introduction to Error-correcting Codes with Applications*, Kluwer.

3. Thompson, T.M. (1984) *From Error-Correcting Codes Through Sphere Packings to Simple Groups*, MAA.

4. Hill, R. (1990) *A First Course in Coding Theory*, Oxford University Press.

5. Blahut, R.E. (2003) *Algebraic Codes for Data Transmission*, Cambridge University Press.

6. Trappe, W. and Washington, L.C. (2002) *Introduction to Cryptography with Coding Theory*, Prentice Hall.

7. Rhee, M.Y. (1989) *Error-Correcting Coding Theory*, McGraw-Hill.

8. Peterson, W.W. and Weldon, E.J. (1972) *Error-Correcting Codes*, MIT Press.

9. Brualdi, R. (1999) *Introductory Combinatorics*, Prentice Hall.

10. Fire, P. (1959) *A Class of Multiple-Error-Correcting Binary Codes for Nonindependent Errors*, Sylvania Report RSL-E-2, Sylvania Reconnaissance Systems Laboratory, Mountain View, CA.

11. Van Tilborg, H.C.A. (1992) *Fire Codes Revisited*, Discrete Mathematics Vol. 106/107, North-Holland.