Error-Correcting Codes

Michael Mo 10770518

6 February 2016

Abstract

An introduction to error-correcting codes will be given by discussing a class of error-correcting codes, called linear block codes. The main concept of block codes will also be deduced, which comes with a notion of how powerful a block code is. Following the concrete example of the Hamming[7, 4] code, some general methods to encode and decode linear block codes are given and at last we consider some aspects of codes, which can help determine how to rank the performance of a code and what other considerations have to be made.

1 Introduction

Moving some data from one place to another requires the data to move through some medium. What we wish is that the data which one party sends is exactly the same as the data which the other party receives. But in general, there is not always a guarantee that the data really stays the same as it goes through the medium. For example, a satellite sends data in the form of radiowaves to some station on earth, but some interference can easily change that signal. This means the data received at the station might not be the same as the original.

The goal of introducing error-correcting codes is to still be able to communicate reliably, even though the medium may not be reliable. To explain how this can be achieved, we first give a more detailed explanation of what an error-correcting code is. Then we consider the main concept of a class of error-correcting codes called block codes and show some practical ways on how to encode and decode. At last we compare some different codes in order to have some sense of which code is better.

2 Setting and Model

Communication between a source and destination can be summarized in the following diagram.

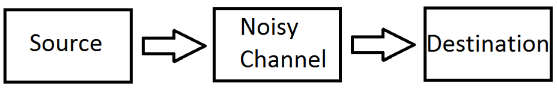


Figure 1: Standard setting of communication from a source to destination.

Whenever data from the source passes through a noisy channel, some noise could affect the data, meaning the data received at the destination could be different than the original data send.

One solution to overcome this problem is to introduce an encoding and decoding scheme. The encoding part will perform some operation on the data from the source, which probably involves adding some redundancy as we will see later. The decoding part, will work on any information received from the channel, with the goal to try to decode it back to the original data, even if some noise has altered the data. So our communication diagram now has two extra blocks:

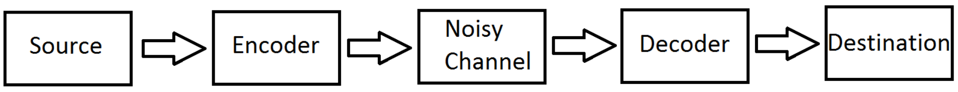


Figure 2: Communication with an encoding and decoding function.

Before we can introduce some error-detecting/correcting codes, we need a more detailed model for the diagram above. In this introduction, we assume that the source generates a binary bitstream. We furthermore assume that the noisy channel is a binary symmetric channel, which means that the only error which can be introduced is a bitflip, and that the chance for a 0 to turn into an 1 is the same as the chance for an 1 to turn into an 0.

For this model we will now consider a block code, which is an encoding and decoding scheme where the encoder works on a fixed number of data bits and always changes it into a fixed-length binary string.

3 Block Codes

To discuss block codes, we first need to have some kind of measure of how much two binary strings look like each other. Note that the following definitions for this measure are based on the model of the channel which tells us that the error only introduces bitflips. For other types of errors, for example a deletion of a bit, you might consider other distance functions.

Definition 3.1. The *Hamming weight* of a binary string x is defined as the total number of 1's which appear in x.

Definition 3.2. The *Hamming distance* between two binary strings x and y of the same length is defined as the number of positions in which x and y differ.

Remark 3.3. The Hamming distance between x and y is the same as the Hamming weight of $(x \oplus y)$. The symbol $\oplus$ means the bitwise XOR operator.

Suppose we have a block code with n data bits and k extra bits. What does the encoding function actually do? It is simply a function which works on the n data bits and creates a longer binary string, consisting of $n + k$ bits. So the encoder just maps each data string to a longer binary string of length $n + k$, which is called a codeword. It should also be clear that the encoder should be an injective function, since we do not want the same codeword for different data strings. Therefore k should not be negative. It also does not make sense to have $k = 0$, since you would then just permute the data strings.

So the problem of designing a block code really lies in choosing the codewords of the bigger set wisely. To demonstrate the main concept of what a good block code is, let us suppose we have two different data strings d_1 and d_2 which the encoder will respectively encode as codewords c_1 and c_2. In the next two figures we also assume that each circle in the set represents a binary string, and that they are ordered such that two circles are neighbours only if the Hamming distance between them is 1.

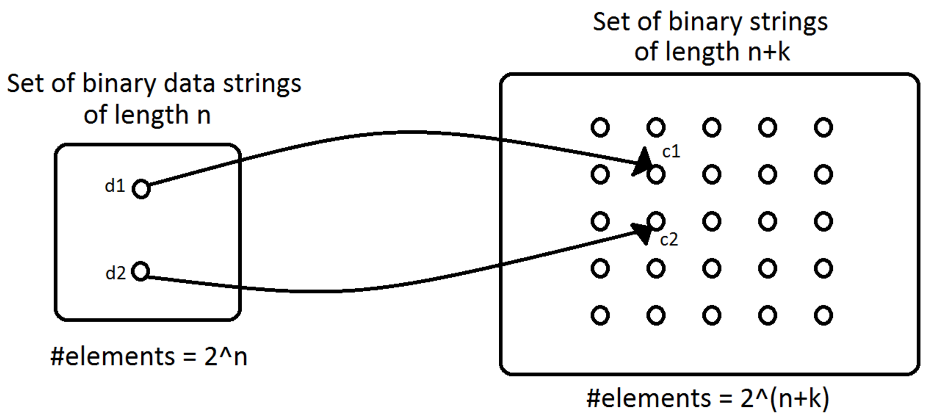

Set of binary data strings of length n
d1
d2
#elements = 2^n
Set of binary strings of length n+k
c1
c2
#elements = 2^(n+k)

Figure 3: Scenario 1.

Scenario 1: The Hamming distance of c_1 and c_2 is very small. In other words: The two codewords c_1 and c_2 look a lot like each other. This means that when the codeword c_1 goes through the noisy channel, a single bitflip can turn c_1 into c_2. So when the decoder gets the binary string c_2, the obvious strategy would be to decode c_2 as the data string d_2 which is wrong.

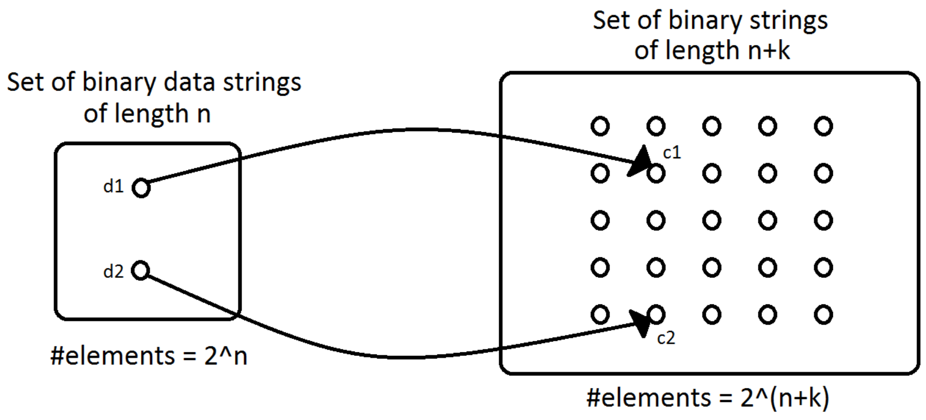

Set of binary data strings of length n
d1
d2
#elements = 2^n
Set of binary strings of length n+k
c1
c2
#elements = 2^(n+k)

Figure 4: Scenario 2.

Scenario 2: The Hamming distance of c_1 and c_2 is big. In this case, even if there is a bitflip, the corrupted codeword $\widetilde{c}_1$ will still be close to c_1. The decoder can then use the strategy: Pretend the received binary string is just the codeword which lies closest to it, and then continue decoding as normal. In this case, it means the corrupted codeword $\widetilde{c}_1$ will still be decoded as d_1, which is the error-correcting ability we want. If the strategy is to only accept binary strings which are exactly equal to a codeword, then the decoder would detect the error and we have an error-detecting code.

So we see that if the encoding function distributes the codewords in such a way that the Hamming distance for all pairs of codewords is as big as possible, then it will be more difficult for one codeword to turn into another codeword and the decoder is thus able to detect or correct more errors. Having this concept in mind, we define a property of a code which can tell us how powerful the code is with respect to detecting or correcting errors.

Definition 3.4. The *distance* of a code is defined as the minimum Hamming distance of all possible pairs of two different codewords.

Theorem 3.5. If a code has distance d, then:

- it can be used as a $d-1$ error-detecting code
- it can be used as a $\lfloor \frac{d-1}{2} \rfloor$ error-correcting code
- mix of the two above

This theorem immediately follows from Figure 4 when we generalize Scenario 2 with the two decoding strategies mentioned.

4 Hamming [7, 4] code

Before demonstrating the Hamming[7, 4] code, we first define what a parity bit is.

Definition 4.1. A parity bit p over the bits $b_1, \ldots, b_k$ is a bit with the value set such that the number of 1's in the string $b_1 \ldots b_k p$ is even. The parity bit p can thus be defined as $p = b_1 \oplus \ldots \oplus b_k$.

Example 4.2. The value of the parity bit p over the bits $\{1, 1, 0\}$ is 0 and the value of the parity bit p over the bits $\{0, 1, 1, 1\}$ is 1.

The Hamming[7, 4] code is a 1-bit error-correcting linear block code with a blocksize of 7 which consists out of 4 data bits. The encoding part of this code works like this: The data string $d_1 d_2 d_3 d_4$ gets encoded as the codeword $p_1 p_2 d_1 p_3 d_2 d_3 d_4$ where:

$$\begin{aligned} p_1 &= d_1 \oplus d_2 \oplus d_4 \\ p_2 &= d_1 \oplus d_3 \oplus d_4 \\ p_3 &= d_2 \oplus d_3 \oplus d_4 \end{aligned}$$

So you can see that the encoding function adds three extra parity-check bits, where each parity bit covers only a subset of the data bits. To have a more concrete feeling on how the decoder decodes a received binary string or corrects it, we can draw a figure of three intersecting circles which can represent our codeword and received binary string of length 7. As seen in Figure 5, each circle consists of one parity bit and the three data bits it covers. The number of 1's in each circle should therefore be even.

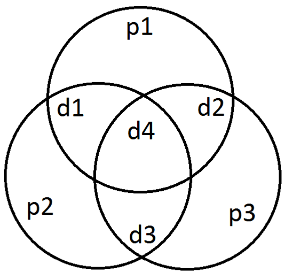


Figure 5: Representation of the codeword $p_1p_2d_1p_3d_2d_3d_4$.

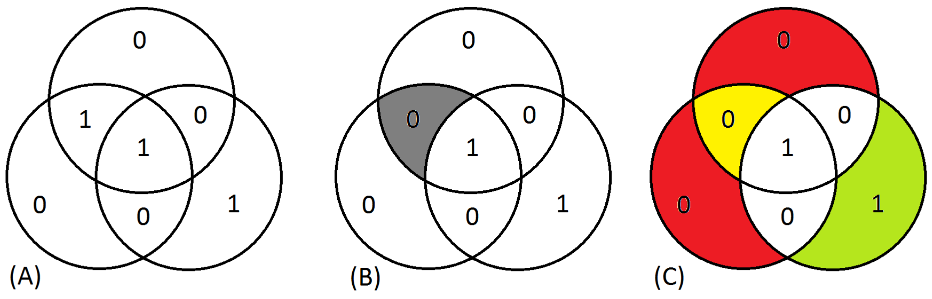


Figure 6: Example of 1-bit error correction.

An example of how a 1-bit error can be corrected is shown in Figure 6: The encoder sends the codeword (A), and the decoder receives the corrupted codeword (B) where d_1 was flipped. In (C) you can see that for each parity bit, the decoder marks with red or green whether the parity bit satisfies the parity-check equation or not. What you see is that for any combination of these markings, there will always be exactly 1 bit which when flipped will make all markings green. So in this case that bit would be d_1 (coloured with yellow), which indeed gives us back the original codeword.

Note that when there are two bitflips, the decoder will flip another bit thinking it has corrected an 1-bit error, when actually the original codeword has been converted to another codeword. An example is given in Figure 7: When both d_1 and p_3 have been flipped, we see that all three parity bits do not satisfy their parity-check equations. Using the 1-bit error-correcting algorithm, the decoder determines that d_4 is the bit that should be flipped, since flipping that bit will make all three markings green. The 'corrected' codeword the decoder gives us is therefore not the original codeword that was sent!

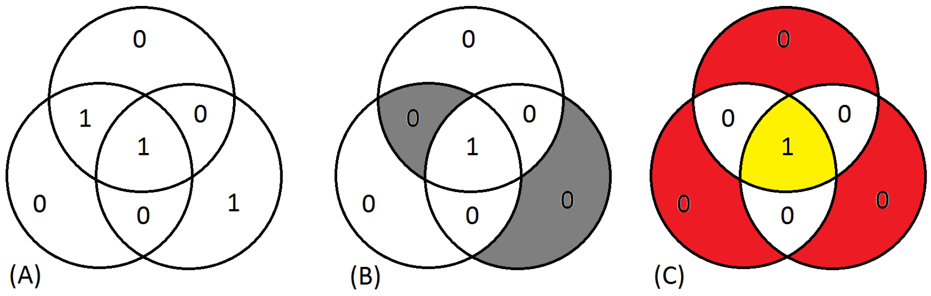


Figure 7: Example of a 2-bit error leading to a wrong correction.

4.1 Extended Hamming[8, 4] Code

We have seen that the Hamming[7, 4] code can correct a 1-bit error, but fails whenever 2 bitflips happen. The extended Hamming[8, 4] code is a small modification of the Hamming[7, 4] code, such that it can still correct up to 1-bit errors, but is also able to detect all 2-bit errors. The only change is that it now has an extra parity bit p_4 which checks the parity of all other 7 bits. Using the fact that the value of p_4 tells us whether either there is an odd or even number of errors makes it possible to know when there are 2 errors.

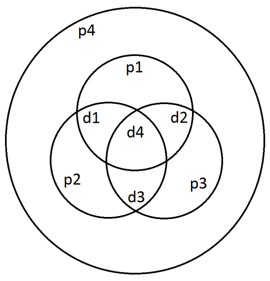


Figure 8: Representation of the codeword $p_1p_2d_1p_3d_2d_3d_4p_4$.

5 Linear block codes

We have seen how to encode and decode with the Hamming[7, 4] code. Now we analyse the Hamming[7, 4] code a bit more, and use the fact that it is linear to show a more general encoding and decoding algorithm, which also works for other linear block codes.

We can regard a bit as an element of $\mathbb{F}_2$, the finite field with two elements. A binary string of length n is then a vector of the vector space $(\mathbb{F}_2)^n$ and the encoding function of the Hamming[7, 4] code becomes:

$$\left(\mathbb{F}_2\right)^4 \to \left(\mathbb{F}_2\right)^7$$

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \mapsto \begin{pmatrix} x_1 + x_2 + x_4 \\ x_1 + x_3 + x_4 \\ x_1 \\ x_2 + x_3 + x_4 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$$

It is easy to see that any parity-check function is a linear function, so the Hamming code is indeed a linear block code. Therefore, the encoding function can be expressed as a matrix like this:

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

This matrix is called the *generator matrix*, since for any data string $x \in (\mathbb{F}_2)^4$ the corresponding codeword y can be calculated from the expression $y = Gx$. Note that if the block code was not linear, then there would not exist such a generator matrix. In that case the encoder would need to use a table lookup method which might result in gigantic tables.

All codewords Hamming[7, 4] code				
0000	0000000		1000	1110000
0001	1101001		1001	0011001
0010	0101010		1010	1011010
0011	1000011		1011	0110011
0100	1001100		1100	0111100
0101	0100101		1101	1010101
0110	1100110		1110	0010110
0111	0001111		1111	1111111

Figure 9: Codewords of the Hamming[7, 4] code.

With the help of all codewords seen in the table above, we can also prove that the distance of the Hamming[7, 4] code is three, which is of course in accordance with Theorem 3.5.

Theorem 5.1. The distance of a linear block code equals the minimum Hamming weight of all codewords c, where c is not the null vector/string.

Proof. First we show that the sum of two different codewords is a codeword not equal to the null string. Let c_1 and c_2 be two different codewords for the data strings x_1 and x_2 respectively. Since the code is linear, there exists a generator matrix G, and we see:

$$c_1 + c_2 = Gx_1 + Gx_2 = G(x_1 + x_2)$$

which means that $c_1 + c_2$ is the codeword for the data string $x_1 + x_2$. The codeword $c_1 + c_2$ can also not be the null string, since that would imply $c_1 = c_2$ giving a contradiction.

Now let C be the set of all codewords. For $x, y \in C$, let $D(x, y)$ be the Hamming distance between x and y, and $W(x)$ be the Hamming weight of x. Then by definition the distance d of the code is $\min\{D(x, y) | x, y \in C, x \neq y\}$.

If x and y are any two different codewords, using Remark 3.3 and the fact that $x + y$ is some codeword $z \neq 0$, we have:

$$\begin{aligned} D(x, y) &= W(x + y) \\ &= W(z) \\ &\geq \min\{W(z) | z \in C, z \neq 0\} \end{aligned}$$

This holds for any two different x and y, thus $d \geq \min\{W(z) | z \in C, z \neq 0\}$.

On the other hand, fixing one codeword to be the null codeword, we have:

$$\begin{aligned} d = \min\{D(x,y) | x, y \in C, x \neq y\} &\leq \min\{D(0,y) | y \in C, y \neq 0\} \\ &= \min\{W(y) | y \in C, y \neq 0\} \end{aligned}$$

Combining the two inequalities gives us $d = \min\{W(z) | z \in C, z \neq 0\}$. $\square$

Using the previous theorem and a quick scan of the table in Figure 9, we see that the Hamming[7, 4] code does indeed has distance 3.

For a more general decoding method, we first create another useful matrix called the *parity-check matrix*. The parity-check matrix H is the matrix which checks if all parity-check equations are correct. So the parity-check matrix for the Hamming[7, 4] code is exactly the 3×7 matrix, where each row corresponds with one of the three parity-check equations, and is thus given by:

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

To see why this matrix is useful in decoding, we note that $Hc = 0$ for any codeword c. This follows from the fact that any codeword is constructed such that it fulfills all parity-check equations.

Now suppose the codeword c is sent over the noisy channel, and the decoder receives the binary string v. If there are some errors introduced from the noisy channel, then we can write the received binary string as $v = c + e$, with e an error vector which only has 1's in the positions where there are bitflips.

The problem of decoding is to determine from all codewords which codeword has the smallest Hamming distance with v. The solution for a block code is to go through a list of all codewords, and calculate the Hamming distance for each codeword with v. But as we already saw in the encoding part, this is inefficient and even impossible if the number of codewords is very large. What we can do is to again make use of the property that the block code is linear, to come up with some smarter decoding algorithm called *syndrome decoding*.

Definition 5.2. The *syndrome* s of a received binary string v is defined as $s = Hv$.

What follows from the linear property is:

$$s = Hv = H(c + e) = Hc + He = 0 + He = He$$

So the syndrome of v basically gives us the parity-check matrix multiplied with the error vector. We want to know the error vector, since knowing the error vector can give us the original codeword back again with $v - e = (c + e) - e = c$. But solving $s = He$ for e often has many solutions. What we really want is to find the error vector e which was most likely to have happened i.e. the e with the smallest Hamming weight.

The standard way to find that e is to have a table which for each syndrome can tell you exactly which error vector of minimum weight can generate that syndrome. That table is called a syndrome table and an algorithm to construct a syndrome table goes like this:

Step 1: Begin with the integer $i = 0$ and an empty syndrome table.

Step 2: For each possible error vector e with a Hamming weight of i, multiply it with the parity-check matrix with to get a syndrome s and only put the pair (s, e) in the table if the syndrome s is not yet in the table.

Step 3: If the table contains all possible syndromes, then the syndrome table is complete, else we increment i and go back to step 2.

Syndrome table	
Syndrome	Error vector
000	0000000
100	1000000
010	0100000
110	0010000
001	0001000
101	0000100
011	0000010
111	0000001

Figure 10: Syndrome table for the Hamming[7, 4] code.

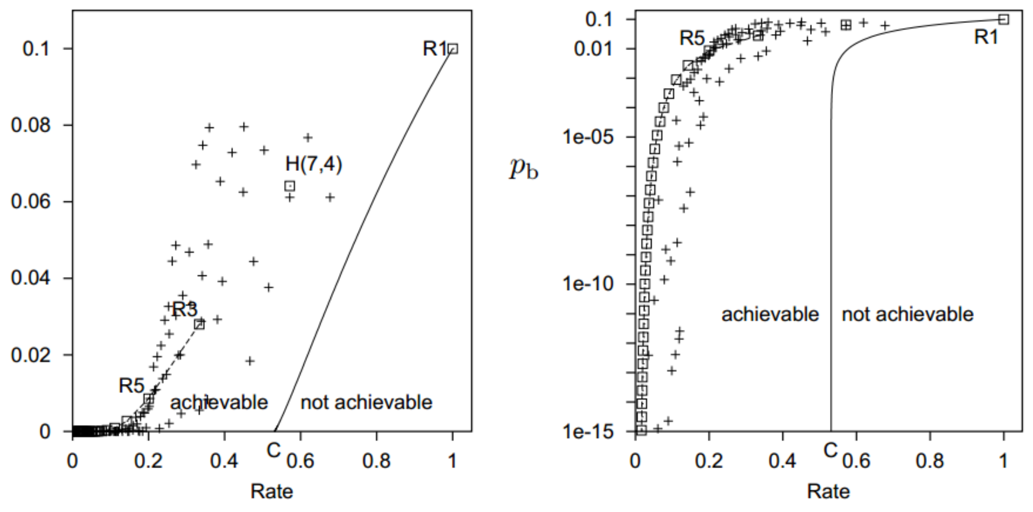
In this case we see that the syndrome table in Figure 10 has a nice pattern. When we see the syndrome as a binary number with the least significant bit on its left, the value of that number actually gives you the position of the error in the error vector! This is not a coincidence, but rather a consequence of the way the Hamming[7, 4] code was constructed with its strange positions for the parity bits. For general linear block codes, this will not be the case.

6 Performance

We have seen an example of a linear block code, called the Hamming[7, 4] code. But how do we rank all these different error-correcting codes, or how should one choose which one to use? Remember that the original goal of introducing error-correcting codes was to improve the reliability of sending data through a noisy channel. So of course we still need to measure how reliable the communication is when using the error-correcting code. Is is important that this also should not completely hamper the real goal of what we want, namely sending some data from the source to the destination. Information about this aspect is given by the rate of the code.

Definition 6.1. The *rate* R of a blockcode is defined as (#data bits / blocksize).

Definition 6.2. With f the chance of a bitflip, the *decoding error* of a data bit $p_b(f)$ is defined as the chance that a single data bit will be decoded wrongly.



$$C(f) = 1 - H_2(f) = 1 - \left[f \log_2 \frac{1}{f} + (1-f) \log_2 \frac{1}{1-f} \right]$$

Figure 11: The solid curve represents the Shannon limit defined by $R = C/(1 - H_2(p_b))$. (David J. C. MacKay. Information Theory, Inference, and Learning Algorithms [1]).

In Figure 11 we have a noisy channel with a bitflip chance of $f = 0.1$. The figure shows a graph of some different codes, where the decoding error of a data bit $p_b(f)$ is on the y-axis, and the rate of the used code is on the x-axis. Intuitively, we see what we expect: As the decoding error of a data bit gets lower, the rate also gets lower. In the graph we also see the number $C(f)$, which is the capacity of the channel as defined in Shannon's noisy channel theorem. This theorem actually tells us that there exists codes whose rate can approach C and also have an arbitrarily low decoding error. So we want codes which in the graph lie near C.

But this graph also does not tell you everything. For example, suppose we have some very good code which in the graph lies very close to the channel capacity C. Also suppose that the encoding and decoding function consists of a lot of complex calculations, and therefore take an enormous amount of time to complete. Then this code is not what we are searching for, since the actual transmission rate of data will be much lower, because you have to take into account the time taken to complete the encoding and decoding functions. And also, if the code is so complex, then these calculations might even use up all battery on board of the computer on our satellite, which is not what we want the satellite to do!

7 Conclusion

In this introduction to error-correcting codes, we have seen that when creating and comparing codes, there are many different aspects which have to be considered, even when we limit ourselves to block codes. We saw that even choosing the model of the channel might give you a complete different concept of what a desired property of a code should be. We also saw that for practical uses, a block code really has to have a linear encoding function in order to make encoding and decoding reasonable with respect of calculation time, which is a point that has to be considered when choosing an error-correcting code.

References

[1] David J. C. MacKay. *Information Theory, Inference, and Learning Algorithms.* Cambridge University Press, 2003.