Introduction to Error Correcting Codes

Jakob Dahl Andersen

DTU Fotonik

Technical University of Denmark

Version 0.2

02 March 2021

Contents

1. Introduction

1.1 Parity Check

When we transmit a digital signal on any media – which we will call a channel – errors occur. In some cases a bit transmitted as a 1 will be received as a 0 or vice versa. This is the case for radio transmission, transmission in optical fibers or writing and reading from a RAM.

In this note we will see how we can detect or even correct some of these transmission errors.

A main assumption in the following is that the errors are random and independent, i.e. whether a certain bit is hit by noise or not does not depend on the situation for the neighbour bits. We do not deal with fading or disrupted channels in this context. We will concentrate on the Binary Symmetric Channel which is described by the error probability p as shown in Table 1. Also, we assume that 0 and 1 are transmitted with the same probability, i.e. 0.5.

Transmitted	Received	probability
0	0	1-p
0	1	p
1	0	p
1	1	1-p

Table 1 Binary Symmetric Channel

The errors can only be detected if we have some restrictions on the transmitted signal.

If a byte of data can be anything we have no way of seeing an error, but if we only allow half of the byte values - like the ones with even parity (an even number of 1's) - we can detect all single errors. A single error will change a 0 to a 1 or a 1 to a 0, in both cases the parity will change from even to odd. Of course two errors will just bring us to another valid byte and the errors can not be detected. But since the error probability is expected to be fairly low 2 errors are much less likely than 1, so detecting one will be a good start.

Example (one error)

11001001 (allowed) → $1\underline{0}001001$ (not allowed)

Of course our data source wants to transmit whatever information they have. There can be no restrictions on the byte values. So what we will do in practice is to append a 9'th bit to the data byte giving even parity.

Example (extra parity bit)

11001001 → 110010010

10001001 → 100010011

This is in fact what is done in many RAM circuits.

In the above example we can detect all single errors because the minimum distance between the "codewords" is 2. I.e. we have to change at least two positions to get from one codeword to another.

To correct a single error we need a distance of 3, otherwise single errors could give the same result from two different codewords and we would not know which was transmitted.

Example (single errors from different codewords with simple parity – i.e. minimum distance 2)

$$110010010 \rightarrow \underline{0}10010010$$

$$010010011 \rightarrow 01001001\underline{0}$$

The most primitive single error correcting code is the repetition code where we simply repeat the bit three times.

Information	Codeword
0	000
1	111

Table 2 Repetition code – Hamming (3,1)

It is easy to see that any single error can be both detected and corrected.

With error correcting codes we always have a choice whether to use them for error detection only, for error correction only or in some cases a combination.

For the repetition code we can chose to detect all single and double errors (triple errors lead to the other codeword and can not be detected) or to correct all single errors. If we decide to correct single errors we will make a wrong decoding in case of double errors.

The repetition code is actually the first one in a class of very clever codes called Hamming Codes, which is the topic of the next section.

1.2 Hamming Codes

Of course it is easy to decode the repetition code, Hamming (3,1), just by inspection, and if you need to make a program you would just make a table with the 8 possible received values. But what if we decide to multiplied the received vector by a matrix

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

Since we transmit and receive **bits** and in general are concerned about hardware implementations we will do the calculation modulo 2.

For modulo 2 calculation we have addition and multiplication as shown in Table 3.

+	0	1
0	0	1
1	1	0

×	0	1
0	0	0
1	0	1

Table 3 Modulo 2 calculation

Notice that 1+1=0, the addition is really an XOR. When we have addition and multiplication we also have subtraction and division. Subtraction is the same as addition and since division with zero, as always, is illegal since the results is unknown we only have division with 1 which is the identity.

Now we are ready to multiply a received vector by H and you can easily verify that if the result – which we will call a **syndrome** – is interpreted as a binary number we get exactly the error position (when the positions are named (3,2,1)). When the word is a legal codeword (which is not the same as error free) the syndrome is 0.

Example

$$[101] \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = [1 \cdot 1 + 0 \cdot 1 + 1 \cdot 0, 1 \cdot 1 + 0 \cdot 0 + 1 \cdot 1] = [10]$$

The trick is that the whole process is linear. Adding noise to our transmitted codeword c is the same as adding (xor'ing) an error vector e. We say that the received vector r is

$$r = c + e$$

When we multiply by H

$$r \cdot H = (c+e) \cdot H = c \cdot H + e \cdot H = 0 + e \cdot H$$

Since c·H=0 for all codewords the resulting syndrome is e·H and only dependent of the errors.

Example

$$(c + e) \cdot H = ([111] + [010]) \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = [0\ 0] + [10] = [10]$$

Now we are ready for the next Hamming code, Hamming (7,4). We will simply use three bits to give the position instead of just two. The parity matrix becomes

$$H_1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Now what are the codewords in this code ?

That is all vectors of length 7 where c·H=0. We rearrange the rows in H_1 to

$$H_2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Which just mean that the syndrome [011] now indicates an error on position 4 instead of position3 , and the syndrome [100] indicates an error on position 3 instead of position 4.

Now it is clear that we can choose the first 4 bits at random (information bits) and then calculate the remaining 3 bits (parity bits) to get a zero syndrome.

$$c_3=c_7+c_6+c_5$$

$$c_2=c_7+c_6+c_4$$

$$c_1=c_7+c_5+c_4$$

Where we denote the codeword (c_7,c_6,c_5,c_4,c_3,c_2,c_1). This means we have a total of 16 codewords.

We usually do the encoding by matrix multiplication with the matrix

$$G_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

If x is the four bit information vector we have

$$x \cdot G = c$$

Of course $G_2 \cdot H_2$ is the all zero matrix. The 16 codewords are shown in Table 4

0 0 0 0 - 0 0 0	1 0 0 0 - 1 1 1
0 0 0 1 - 0 1 1	1 0 0 1 - 1 0 0
0 0 1 0 - 1 0 1	1 0 1 0 - 0 1 0
0 0 1 1 - 1 1 0	1 0 1 1 - 0 0 1
0 1 0 0 - 1 1 0	1 1 0 0 - 0 0 1
0 1 0 1 - 1 0 1	1 1 0 1 - 0 1 0
0 1 1 0 - 0 1 1	1 1 1 0 - 1 0 0
0 1 1 1 - 0 0 0	1 1 1 1 - 1 1 1

Table 4 The 16 codewords in Hamming (7,4) as generated by G_2

As indicated by the indexes 1 and 2 many "different" versions of a Hamming code exist. There are two possible parameters.

1. Different order of the bits in the codeword. This is done by reordering the columns in G and the corresponding rows in H.

2. Different mapping of the information vectors x to the codewords c. This is done by row operations in G, but the set of codewords remain the same.

Notice that

$$H_2 = \begin{bmatrix} A \\ I \end{bmatrix} \ \ and\ G_2 = \ [I \quad A]$$

Where I is the identy matrix and A a matrix for the specific version of the Hamming code.

The best way to calculate G form H or vice versa is often to bring it on the form with I and A submatrices by the 2 rules mentioned above. There various transformations can then be done in reverse.

How do we know that the code corrects all single errors ?

Well, all the codewords give zero syndromes that is how they are constructed and all error vectors with six 0’s and one 1 will give a syndrome that is exactly one of the rows in H. Since the rows in H are all distinct we can correct the error.

If we write out all 16 codewords we can also see that there is a minimum distance d_{min}= 3, i.e. to go from one codeword to another we need to change at least three bits. Actually, this may not be that easy to see, but if we notice that the code is linear, meaning that the sum of two codewords is also a codeword, and that the all zero word is always a codeword we just have to check the number of 1’s in the non-zero codewords.

In general a code with minimum distance d_{min} corrects $T = \frac{d_{min}-1}{2}$ errors. This is obvious since an error pattern has to bring us at least halfway to another codeword to be not-correctable.

What if there is more than one error ?

If there are two errors, the syndrome is the sum of two rows in H. This is still non-zero but equal to the syndrome for a single error and since one error is more likely than two we will usually chose to correct this position thereby making one more error.

If there is more than two errors the syndrome could also be zero and we will not be able to correct anything – but there will still be errors. In general a syndrome can be generated by a number of different error vectors and we chose the most likely, in this case no errors or a single error.

Finally we note, as you might have guessed, that Hamming (3,1) refers to a Hamming code with codewords of length 3 and 1 information bit and Hamming (7,4) refers to a Hamming code with codewords of length 7 and 4 information bit.

In general, we can construct a Hamming code with **k** information symbols and codewords of length **n** based on a parameter m as

$$(n,k) =(2^m-1,\ 2^m-1-m) \qquad m=2,3,4\ldots\ldots$$

1.3 Error Probabilities

If the bit error probability is p, the probability of receiving a k-bit vector without errors is $(1-p)^k$. The probability of receiving it with errors is $1-(1-p)^k$.

If we correct 1 error in n bits the probability of decoding a n-bit codeword correctly is $(1-p)^n+n\cdot p\cdot(1-p)^{n-1}$. The probability that it is not correct is $1-((1-p)^n+n\cdot p\cdot(1-p)^{n-1})$.

We can now evaluate the Hamming codes in terms of "block error probability". I.e. the probability that the codeword of length n is correct after decoding. This can be compared to the probability of receiving a block of k bits correctly without encoding.

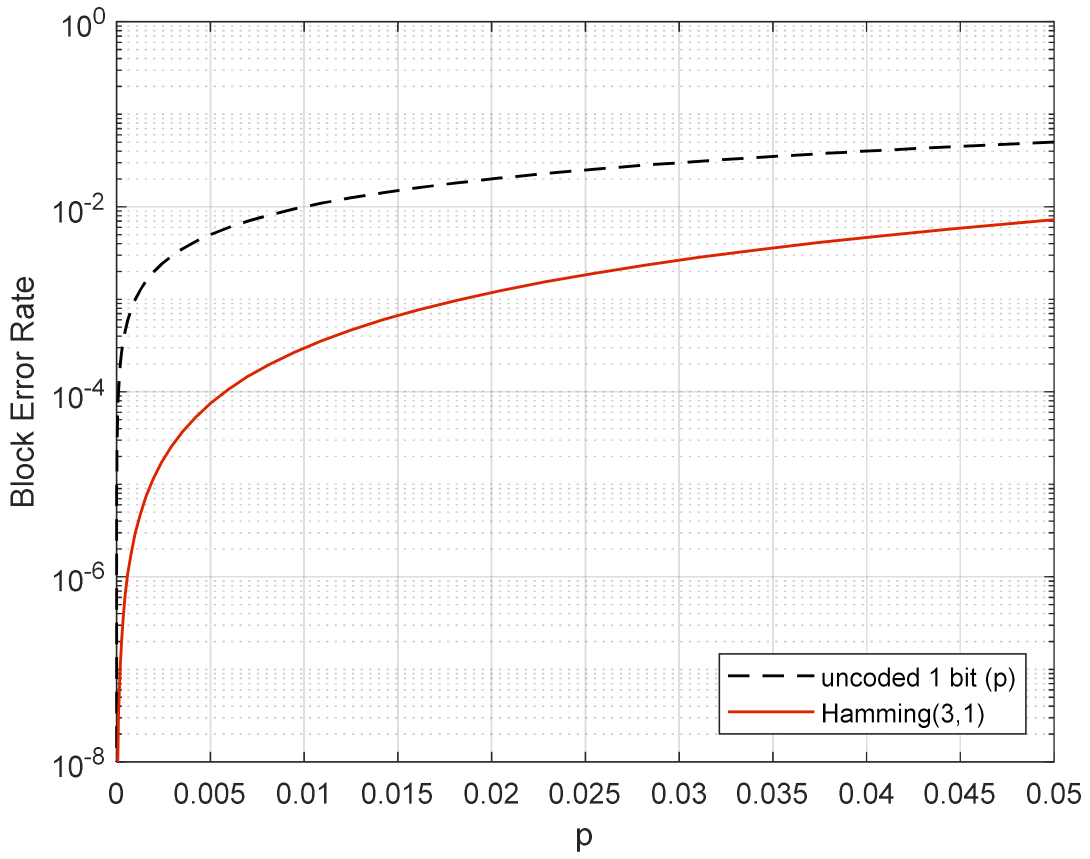

Figure 1 Block Error Rate for Hamming (3,1) compared to channel error probability p

As seen from Figure 1 there is a huge improvement from the little Hamming (3,1) code when we see it as function of the channel error probability. But we need to transmit three times as many bits !

An effective way to combat noise is of course to turn up the power of the transmitter, but that is exactly what we don't want to do. In some cases because it is not possible or very difficult and in most cases just to save power.

1.4 Signal-to-noise Ratio

We usually denote the energy per channel bit as E_s and measure the signal-to-noise ratio E_s/N_0, in decibel (dB),where N_0 is a measure of the noise energy. Be aware that we use the term E_s/N_0 both for the value in dB an as a real number. So

$$E_s/N_0\ [\text{dB}] = 10\cdot\log_{10}(E_s/N_0\ [\text{real}])$$

For a "normal" Gaussian channel the relation between the error probability p and the signal-to-noise ratio E_s/N_0 is

$$p = Q\left(\sqrt{\frac{2E_s}{N_0}}\right) = \frac{1}{2}erfc\left(\sqrt{\frac{E_s}{N_0}}\right)$$

In Matlab

```
p=qfunc(sqrt(2*10.^(EsN0/10)))
```

or

```
p=0.5*erfc(sqrt(10.^(EsN0/10)))
```

Notice that we use E_s/N_0 as a real number in the formulas.

We can also specify the signal-to-noise ratio for the energy per information bit E_b where $E_b= E_s/R$. $R=k/n$ is the rate of the code i.e. number of information bits k divided by number of coded bits n.

We have

$$E_b/N_0\ [\text{dB}] = 10\cdot\log_{10}(E_b/N_0\ [\text{real}]) = 10\cdot\log_{10}(E_s/N_0\ [\text{real}]/R) = E_s/N_0\ [\text{dB}]-10\cdot\log_{10}(R)$$

When using the Hamming (3,1) there is only 1/3 of the original energy per transmitted bit, so it is really not fair to make a comparison like the one in Figure 1. Instead we should measure the error probability after decoding as function of the signal-to-noise ratio per information bit E_b/N_0.This is what we have in Figure 2 where we see that the repetition code is not good at all. It would be much better just to turn up the signal.

The situation is better for the Hamming (7,4) code. We can calculate the block error probabilities as mentioned above but what we really want is the bit error probability after decoding (for Hamming(3,1) block errors was the same as bit errors). This is not possible (or at least hard) to calculate so we have instead made a simulation in Matlab. The results for Hamming (3,1), Hamming (7,4) and Hamming (15,11) are shown in Figure 3, here we see that the (7,4) codes actually give an improvement (at least for $E_b/N_0>$ 6.5 dB).

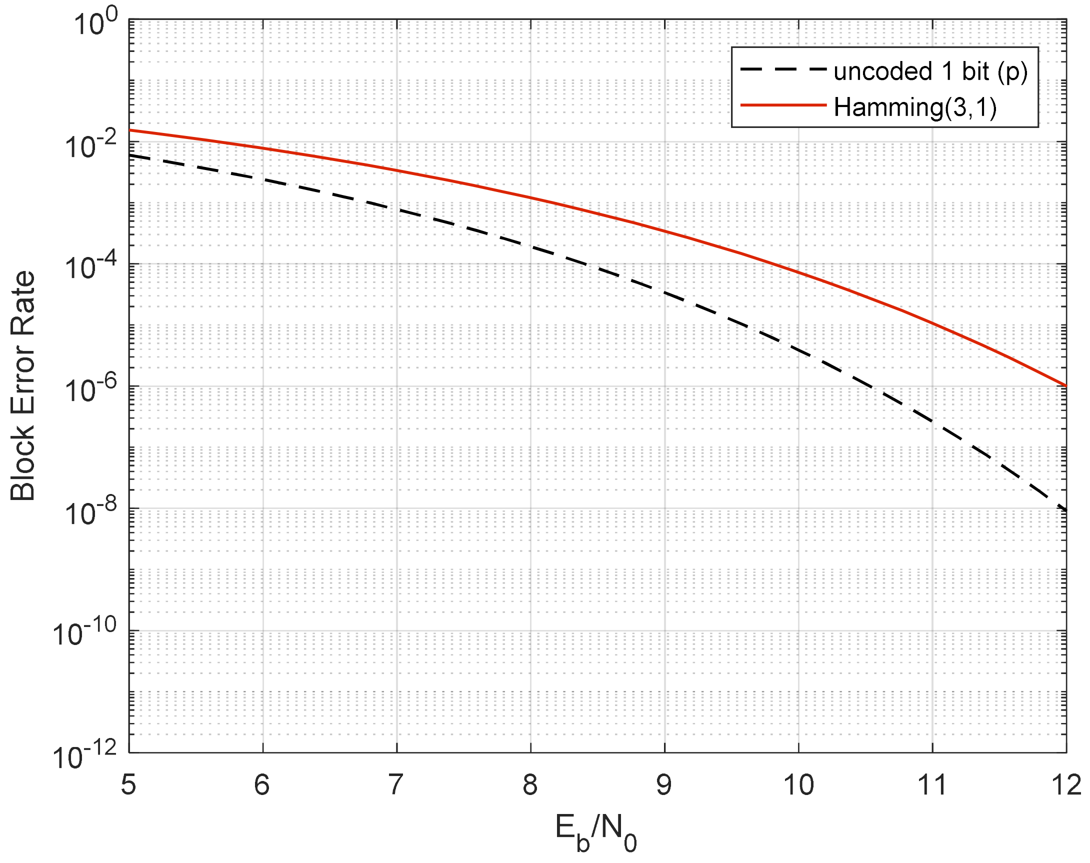

Figure 2 Block Error Rate for Hamming (3,1) compared to uncoded signal as function of signal-to-noise level per information bit E_b/N_0

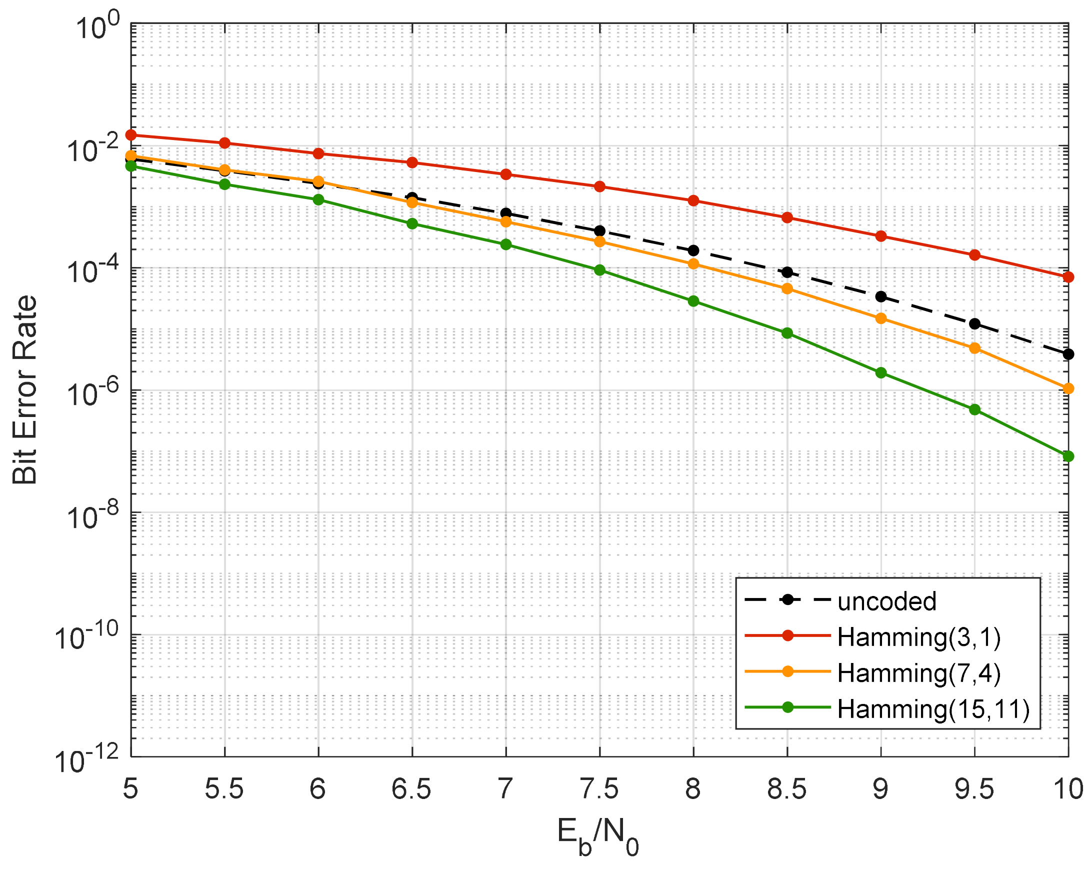

Figure 3 Simulated Bit Error Rate for Hamming codes compared to uncoded signal as function of signal-to-noise level per information bit E_b/N_0

In general it is good to have a long code (large n). This is intuitively clear since we can average out the errors on a bigger chunk of the data. But of course the complexity of the decoder increases with increasing n. Also, we can not continue just to correct single errors. We will have to construct multi-error correcting codes.

An example of codes correcting more than one error per block is BCH codes which is a naturel extension of Hamming codes. But first we have to learn about Finite Fields/Galois Fields.

Summary

A (n,k) code encode/map k information bits to a n-bit codeword. Only 2^k of the 2^n possible n-bit vectors are therefore legal codewords and some transmission errors can be detected/corrected since the received word is not a legal codeword.

We measure the distance between codewords as the number bits where the value differ. If the minimum distance between the codewords is d_{min} then we can detect all error patterns where the number of errors is less than d_{min}. Alternatively, we can correct all error patterns where the number of errors is less or equal $(d_{min}-1)/2$. When we correct errors it is always a guess. We chose the most likely error pattern to produce the received word – or in other words, we chose the codeword that is most likely to have been transmitted. If the decoder is optimal it is a maximum likelihood decision but often we are not able to make an optimal decoding and only correct a set of the most likely error patterns.

We denote the rate as R=k/n . Transmission of the n-k parity bits require energy/time so to compare codes with different rates we add a “penalty” of -10·log10(R) dB on the signal-to-noise ratio per information bit E_b/N_0.

2. BCH and Reed-Solomon Codes

2.1 Galois Fields

Galois fields or finite fields (på dansk endelige legemer) represent a finite number of elements where we can use two operations, called addition and multiplication, and still have results in the finite set. We also require the existence of a "zero" and a "one" element. When pronouncing Galois, remember that he was French.

From the two basic operations follows of course that we also have subtraction, division and powers, i.e. we can have polynomials.

The most simple Galois fields are obtained by calculating modulo some prime, here the result is obviously inside the set. One of the features of a GF is that there exist a primitive element α so every element – except 0 – can be expressed as a power of α.

element	0	1	2	3	4
Power of α	-	α^0	α^1	α^3	α^2

Table 5 Galois Field obtained by calulations modulo 5, primitive element α chosen to 2.

It is easy to see that this only works for prime numbers. If we try to calculate modulo 4 we will see that 2×1=2 and 2×3=2. So what is the result of 2/2, 1 or 3 ?

Doing digital electronics we would prefer a field with 2^m elements. This can be constructed by seeing the elements as polynomials of degree m-1 with binary coefficients. We then define additions as addition of the polynomials modulo 2, i.e. the coefficients are added modulo 2 and remain binary. Multiplication are performed modulo a specific polynomial of degree m.

Notice that since 1+1=0 modulo 2 there are no minus in GF(2^m) – or rather minus is equal to plus.

As example we use GF(2^3) with the primitive polynomial z^3+z+1.

Elements (z^2+z) and (z^2+1) are added to (z+1), since $z^2+z^2=0$, and multiplied to ($z^4+z^3+z^2+z$) modulo (z^3+z+1) = (z+1).

In Table 6 we show the full field.

Polynomial	Power of α	Binary representation	log
0	-	0 = [0 0 0]	-
1	α^0	1 = [0 0 1]	0
z	α^1	2 = [0 1 0]	1
z^2	α^2	4 = [1 0 0]	2
z+1	α^3	3 = [0 1 1]	3
z^2+z	α^4	6 = [1 1 0]	4
z^2+z+1	α^5	7 = [1 1 1]	5
z^2+1	α^6	5 = [1 0 1]	6
1	α^7	1 = [0 0 1]	0

Table 6 GF(2^3) generated from z^3+z+1.

In general we use the binary representation for addition and the logarithmic representation for multiplication. So elements (z^2+z) and (z^2+1) are added by xor of their binary representations 6 and 5 giving (110) xor (101) = (011) i.e. 3 or (z+1). They are multiplied by adding their logarithms 4 and 6 getting 10. This is taken modulo 7 since we have 2^3 elements minus the “zero element” and, as indicated in Table 6, the powers of α just continue cyclically. The final result of the multiplication is then 3, i.e. (z+1).

2.2 BCH codes

We saw that Hamming codes was able to correct one error because all the rows in the parity matrix H was distinct.

One might get the bright idea that we would be able to correct two errors if we construct a parity matrix where all combinations of one or two rows give different syndromes (and not the all-zero syndrome). Of cause, the matrix might need to have twice as many columns since the syndromes should specify two positions. To construct such a matrix we will use the theory of Galois Fields.

If we rearrange the order of the rows in H and see them as elements in a Galois Field, the (7,4) Hamming code becomes

$$H_3 = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \alpha^6 \\ \alpha^5 \\ \alpha^4 \\ \alpha^3 \\ \alpha^2 \\ \alpha^1 \\ \alpha^0 \end{bmatrix}$$

Where α is a primitive element in GF(2^3) specified by the polynomium z^3+z+1.

A two-error correcting BCH code over GF(2^3) is then specified by

$$H_{BCH} = \begin{bmatrix} \alpha^6 & (\alpha^6)^3 \\ \alpha^5 & (\alpha^5)^3 \\ \alpha^4 & (\alpha^4)^3 \\ \alpha^3 & (\alpha^3)^3 \\ \alpha^2 & (\alpha^2)^3 \\ \alpha^1 & (\alpha^1)^3 \\ \alpha^0 & (\alpha^0)^3 \end{bmatrix} = \begin{bmatrix} \alpha^6 & \alpha^4 \\ \alpha^5 & \alpha^1 \\ \alpha^4 & \alpha^5 \\ \alpha^3 & \alpha^2 \\ \alpha^2 & \alpha^6 \\ \alpha^1 & \alpha^3 \\ \alpha^0 & \alpha^0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

This (7,1) BCH code only have the codewords [0 0 0 0 0 0 0] and [1 1 1 1 1 1 1] but it illustrates the principle.

To make a BCH code correcting more errors we just continue in the same way having $(\alpha^i)^5$, $(\alpha^i)^7$ etc.

It is quite difficult to prove that the BCH codes actually are able to correct the specified number of errors, but we are engineers – we prove the concept by writing a program that actually decodes the errors as expected.

A BCH code constructed like this is often able to correct more than the specified number of errors. In our example there is 63 non-zero syndromes but only 7+7×6/2=28 cases with one or two errors. However, the decoding method used is often “Bounded Distance Decoding” meaning that we only correct the specified number of errors – here two. The extra capacity give some error detecting capability. Since only 28 of the 63 non-zero syndromes are used in our example the remaining 35 indicates a non-correctable error. If we take a totally random 7-bit vector the chance are 35/63 that we will hit a syndrome indicating non-correctable errors.

2.3 Generator Polynomials and Systematic Encoding

We can see the codewords and received vectors for the (7,4) Hamming code as binary polynomials.

$$C=[c_6, c_5, ,c_4, c_3, c_2, c_1, c_0] \Rightarrow C(z)= c_6z^6+ c_5z^5+ c_4z^4+ c_3z^3+ c_2z^7+ c_1z^1+c_0$$

$$R=[r_6, r_5, ,r_4, r_3, r_2, r_1, r_0] \Rightarrow R(z)= r_6z^6+ r_5z^5+ r_4z^4+ r_3z^3+ r_2z^7+ r_1z^1+r_0$$

When we specify the parity matrix H as elements in a Galois Field the syndrome is now just R(α) and the requirement $C\cdot H = \underline{0}$ now means that α must be a root in C(z). The **binary** polynomial of lowest degree where α is a root (the minimal polynomial for α) is $G(z)=z^3+z+1$ so for C(z) to be a codeword G must divide C. Note that α is of course root in the non-binary polynomial $(z-\alpha)=(z+\alpha)$.

If we also see the 4-bit information as a polynomial

$$X=[x_3, x_2, x_1, x_0] \Rightarrow X(z)= x_4z^4+ x_3z^3+ x_2z^7+ x_1z^1+x_0$$

The encoding becomes

$$C(z)=X(z)\cdot G(z)$$

For the BCH code example we find that the minimal polynomial for α^3 is z^3+z^2+1 so the generator polynomial becomes $G(z)=(z^3+z+1)\cdot (z^3+z^2+1)= z^6+z^5+ z^4+z^3+z^2+z^1+1$ which fits with the two codewords [0 0 0 0 0 0 0] and [1 1 1 1 1 1 1].

The encoding above is non-systematic, i.e. the information can not be seen directly in the codeword, but we can quite easily make the encoding systematic.

If we look at $z^{(n-k)}\cdot X(z)$ the polynomial has (n-k) zero’s at the end – exactly what we need for a remainder with division by G(z). The encoding is then performed by finding the remainder of $z^{(n-k)}\cdot X(z)/G(z)$ and adding this to $z^{(n-k)}\cdot X(z)$.

A fun fact about generator polynomials are that if G(z) divides z^n+1 then the code becomes cyclic, meaning that if $[c_6, c_5,,c_4, c_3, c_2, c_1, c_0]$ is a codeword so is $[c_5,,c_4, c_3, c_2, c_1, c_0 ,c_6]$. This is the case for both the Hamming code and the BCH code above.

An example

We will encode (1 0 0 1) = (z^3+1) by the Hamming (7,4) as specified by H_3 above.

We therefore divide $z^3\cdot(z^3+1) = (z^6+z^3)$ by (z^3+z+1) and get the remainder (z^2+z).

The codeword is therefore $(z^6+z^3+z^2+z)$ = (1 0 0 1 1 1 0).

If we multiply by H_3 we get (101)+(011)+(100)+(010) = 0. Also, $C(\alpha) = \alpha^6+\alpha^3+\alpha^2+\alpha = 0$.

2.4 Decoding the Two-error Correcting BCH code

When we see the binary vectors as binary polynomials we can calculate the syndrome $r \cdot H$ of the two-error correcting BCH as the two halves

$$S_1 = R(\alpha) \quad \text{and} \quad S_3 = R(\alpha^3)$$

where R is the received vector seen as a binary polynomial and α is a primitive element in the Galois Field.

If R is a codeword $S_1 = S_3 = 0$.

If there is one error at position e_1, $S_3=S_1^3$ and $S_1= \alpha^{e1}$.

If there is two errors at positions e_1 and e_2, $S_1 = \alpha^{e1} + \alpha^{e2}$ and $S_3 = \alpha^{3\cdot e1} + \alpha^{3\cdot e2}$,

The error locator is defined as

$$\sigma(x) = (x+ \alpha^{e1})\,(x+ \alpha^{e2}) = x^2+ \sigma_1 x+ \sigma_2$$

where

$$\sigma_1 = S_1$$

$$\sigma_2 = (S_1^3+S_3)/S_1$$

We define $y = x/\sigma_1$ and rewrite $\sigma(x)$ as

$$\sigma(y) = y^2+y+ \sigma_2/\sigma_1^2$$

Now the roots are defined by the constant $A= \sigma_2/\sigma_1^2$. Since we are in a finite field $GF(2^m)$ there is only 2^m possible values for this constant and we can make a look-up table of the solutions. Note that not all values of A have solutions.

Finally the roots found in the table look-up is multiplied by σ_1 to get the results α^{e1} and α^{e2}.

A three-error correcting BCH code can be decoded after the same principles – although somewhat more complicated – but for decoding of BCH codes correcting more than 3 errors a more complicated decoding algorithm is needed, usually involving Euclids algorithm.

Example using $GF(2^3)$ from Table 6

We transmit [1 1 1 1 1 1 1], but receive [0 1 1 1 1 0 1].

$S_1 = R(\alpha) = \alpha^5+ \alpha^4+ \alpha^3+ \alpha^2+ 1$ = [1 1 1] xor [1 1 0] xor [0 1 1] xor [1 0 0] xor [0 0 1] = [1 1 1] = α^5

$S_3 = R(\alpha^3) = \alpha^{15}+ \alpha^{12}+ \alpha^9+ \alpha^6+ 1 = \alpha+ \alpha^5+ \alpha^2+ \alpha^6+ 1$ = [0 1 0] xor [1 1 1] xor [1 0 0] xor [1 0 1] xor [0 0 1] = [1 0 1] = α^6

$\sigma_1 = S_1= \alpha^5$

$\sigma_2 = (S_1^3+S_3)/S_1 = (\alpha^{15}+ \alpha^6)/ \alpha^5 = (\alpha+ \alpha^6)/\alpha^5 = [1\ 1\ 1]/ \alpha^5 = \alpha^5/\alpha^5 = 1$

$A = \sigma_2/\sigma_1^2 = \alpha^0/\alpha^{10} = \alpha^{-10} = \alpha^4$

$\sigma(y) = y^2 + y + \alpha^4$ has roots α and α^3 (pre-calculated)

The errors are therefore at position $\log(\alpha\cdot\alpha^5) = 6$ and position $\log(\alpha^3\cdot\alpha^5) = \log(\alpha) = 1$

The error pattern is [1 0 0 0 0 1 0] and the decoded word [1 1 1 1 1 1 1].

2.4.1 Calculating the Syndrome

For many implementations the calculation of the syndromes represent a significant part of the decoder complexity.

One approach is calculating the syndrome as a recursion

```
S = $r_{n-1}$
for i = n-2:-1:0
    S = $\alpha\cdot S + r_i$
end
```

The approach can be used for more than one bit at the time. For two bits the recursion become

$$S = \alpha^2\cdot S + \alpha\cdot r_i + r_{i-1}$$

Since the received values r are just 0 or 1, multiplying by α just means moving the bit one position to the left.

In general we can see the multiplication by α as the multiplication by a binary matrix. For $GF(2^3)$ illustrated in Table 6 the matrix is

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

Notice that the first row of A is basically the primitive polynomial and the rest is mainly an identity matrix.

Multiplying by α^n can be performed by multiplying by A^n.

2.5 Reed-Solomon Codes

Now we have introduced the Galois Field elements and codewords as polynomials we may get the idea that the codewords could be a vector of GF elements not just of bits. Again we will extend the (7,4) Hamming code. If the codeword is not just seven bits but seven Galois elements, i.e. 21 bits, and we still want to correct one “symbol” error, i.e. one of the Galois elements, we need to augment the parity matrix H so we not only get the position of the error but also the size. This time we use

$$H_{RS} = \begin{bmatrix} \alpha^6 & (\alpha^6)^2 \\ \alpha^5 & (\alpha^5)^2 \\ \alpha^4 & (\alpha^4)^2 \\ \alpha^3 & (\alpha^3)^2 \\ \alpha^2 & (\alpha^2)^2 \\ \alpha^1 & (\alpha^1)^2 \\ \alpha^0 & (\alpha^0)^2 \end{bmatrix} = \begin{bmatrix} \alpha^6 & \alpha^5 \\ \alpha^5 & \alpha^3 \\ \alpha^4 & \alpha^1 \\ \alpha^3 & \alpha^6 \\ \alpha^2 & \alpha^4 \\ \alpha^1 & \alpha^2 \\ \alpha^0 & \alpha^0 \end{bmatrix}$$

When we receive a vector R we see it as a polynomial with coefficients in GF(2^3). If we calculate the syndrome for the two columns of H separately, we therefore get

$$S_1 = R(\alpha) \quad \text{and} \quad S_2 = R(\alpha^2)$$

Since the syndromes are 0 for the codeword parts of R a single error with value v at position p will result in

$$S_1 = v \cdot \alpha^p \quad \text{and} \quad S_1 = v \cdot \alpha^{2p}$$

So, $S_2/S_1 = \alpha^p$ or $p = \log(S_2/S_1)$ and $v = S_1/\alpha^p$ - and we are actually able to correct one symbol error.

The small example is a (7,5) Reed-Solomon code over GF(2^3) with $G = (z+\alpha)\cdot(z+\alpha^2)$, able to correct one symbol error.

A t error correcting Reed-Solomon code can be specified by the generator polynomial

$$G = \prod_{i=1}^{2t} (z + \alpha^i)$$

It is time for an example.

We have the information [0 1 1 1 1 1 0 0 0 0 0 1 1 0 1], that is the GF(2^3) elements [α^3 α^5 0 1 α^6].

The generator polynomial is $(z+\alpha)\cdot(z+\alpha^2) = z^2+(\alpha+\alpha^2)\cdot z+\alpha\cdot\alpha^2= z^2+\alpha^4\cdot z+\alpha^3$

We divide $(\alpha^3\cdot z^6+\alpha^5\cdot z^5+z^3+\alpha^6\cdot z^2)$ by $(z^2+\alpha^4\cdot z+\alpha^3)$ and get the remainder (α^4)

The codeword is C = [α^3 α^5 0 1 α^6 0 α^4]

We add the noise [0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0] = [0 0 0 0 α^4 0 0] and get R = [α^3 α^5 0 1 α^3 0 α^4].

$S_1 = R(\alpha) = \alpha^3\cdot \alpha^6+ \alpha^5\cdot \alpha^5+ \alpha^0\cdot \alpha^3+ \alpha^3\cdot \alpha^2+ \alpha^4= \alpha^9+ \alpha^{10}+ \alpha^3+ \alpha^5+ \alpha^4= \alpha^2+ \alpha^3+ \alpha^3+ \alpha^5+ \alpha^4$ = 4 xor 6 xor 7 = 5 = α^6

and $S_2= R(\alpha^2) = \alpha^8$, so the error position is $\log(\alpha^8/\alpha^6) = \log(\alpha^2) = 2$ (counting 0, 1, 2.. from right) and the error value is $(\alpha^6/\alpha^2) = (\alpha^4)$ = [1 1 0].

When the Reed-Solomon codes correct more than one error the decoding soon get very complicated. A method often used in practice is based on the Euclid algorithm. This could be the case for some of the most used RS codes the (255,223) 16 error correcting code and the (255,239) 8 error correcting code both based on GF(2^8), i.e. the symbols are bytes.

Summary

With the introduction of Galois fields the simple one-error correcting Hamming code can be augmented to a t-error correcting BCH code or a t-symbol correcting Reed-Solomon code. We can also replace the generator matrix by a polynomial, which is the normal way to specify BCH or RS codes.

For two or three-error correcting BCH codes or one error correcting Reed-Solomon the decoding can be performed by fairly simple calculations, but very soon much more complicated decoders based on the Euclid algorithm are required.

3. Product Codes (how to swallow an elephant)

In general it is good to have a long code (large n). This is intuitively clear since we can average out the errors on a larger block of data. But of course the complexity of the decoder increases with increasing length.

A way to handle this is to use the old saying: “If you want to swallow an elephant you just have to take one bite at the time”. So we construct the long codes with a number of smaller codes, which we decoded one by one. The final decoding may not be optimal in respect to what we could have done if we had unlimited decoding resources – but often it is fairly good.

A simple construction is product codes where we make a two-dimensional structure where all the rows and all the columns are codewords in some block code. This is illustrated in Figure 4 for a (256,239) “constituent” code.

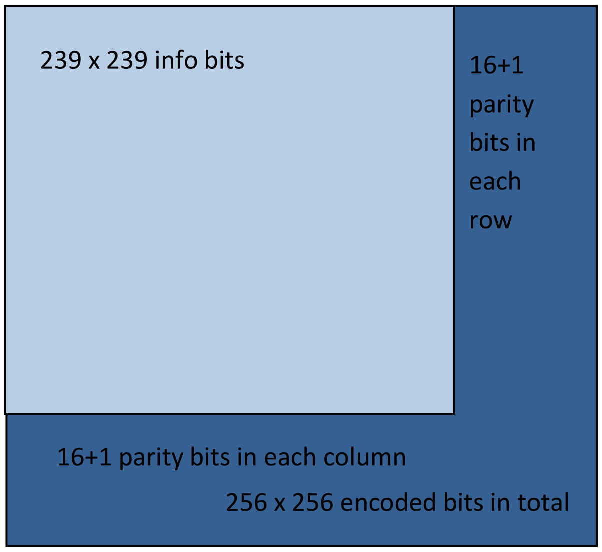
239 x 239 info bits
16+1 parity bits in each row
16+1 parity bits in each column
256 x 256 encoded bits in total

Figure 4 Product code

Encoding is easy. The data is set up in a two-dimensional k x k matrix, here 239x239. Each row is encoded to get a k x n matrix, here 239x256. Finally each column is encoded to get the n x n matrix, here 256x256. The last n-k (17) rows are automatically codewords since the code is linear.

For the decoding all the rows are decoded – then all the columns. This decoding may be repeated in a number of iterations.

In Figure 5 we show the results for an (256,239) extended BCH code. By extended we mean that the normal two-error correcting (255,239) BCH code over GF(2^8) is extended by an additional parity bit calculated as the sum of the first 255 bits. The “frame” is therefore 57121 information bits and 65536 encoded bits. The rate is 0.87.

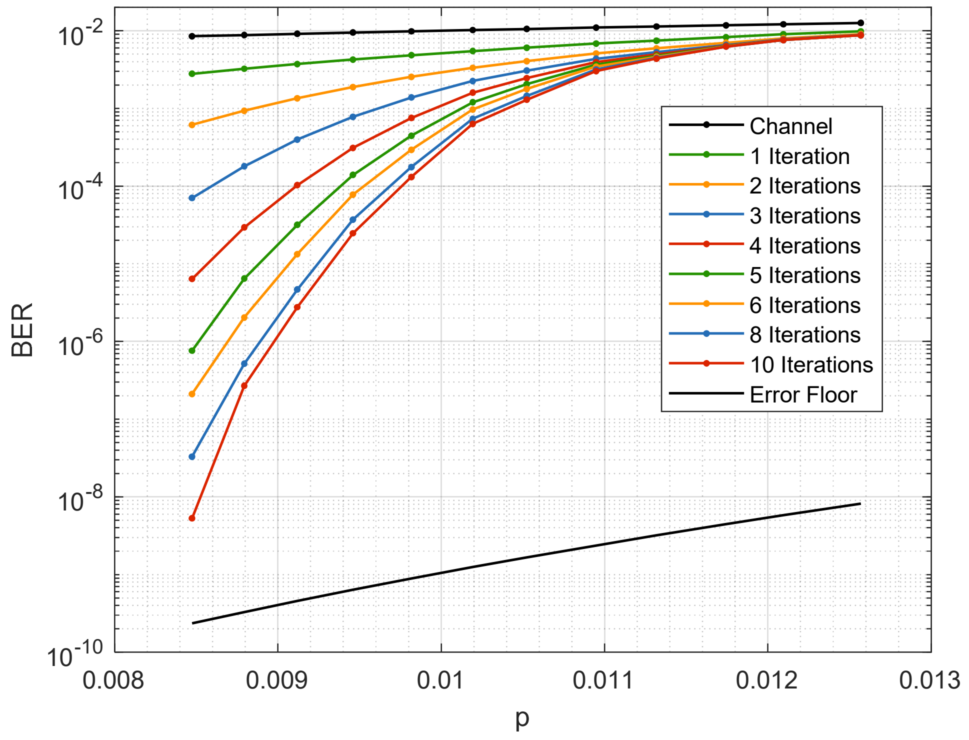

Figure 5 Performance for (256,239) extended BCH code, hard-decision.

The results in Figure 5 is obtained with hard-decision only, if we can have a soft-decision input and soft-decision between the iterations the performance can be much better – but the complexity of the decoder will also increase substantially.

The extended BCH code in this example corrects all single and double errors, and detects all triple errors. This means that if we have three rows with three errors and the errors appear in the same three columns, i.e. we have a 3 x 3 square of errors, this errors pattern will not be corrected. The three column decoders will detect three errors and not correct anything – and likewise for the three columns.

It is a general drawback of product codes that we have these “uncorrectable” error patterns with relatively few errors. We call it an “Error-floor”. In Figure 5 we have included an estimate of the error floor for this specific product code.

The error floor problem arise because with use the suboptimal decoding strategy of decoding the rows and columns one by one. If we see the product code as one big code the minimum distance is the square of the minimum distance of the constituent code, and we should in principle be able to correct these error floor patterns, however with a much more complicated decoding algorithm.

Summary

Very long codes can be constructed by concatenation of smaller codes. Product codes use a two-dimensional structure.

The decoding is performed by decoding the constituent codes one by one – possibly with iterations. This decoding strategy does not give the optimal performance but it is the best we can do, and often very effective.

4. Convolutional Codes

4.1 Encoding

An alternative to the (n,k) block codes described above is convolutional codes. Here we see the information as a bit stream and make a number of output streams as functions of the last m+1 bits. The functions are just an xor of a fixed selection of the m+1 bits and they are specified as a binary vector or traditionally in octal notation.

The most simple convolutional code is specified as g0= [1 1 1], g1=[1 0 1], or [7,5] in octal. This means that there is two output bits for every input bit, i.e. rate ½. One output is (x_t xor x_{t-1} xor x_{t-2}) and the other one is (x_t xor x_{t-2}).

The encoder is illustrated in Figure 6, where we use a kind of hardware notation. The boxes marked D are delay elements, or actually a shift-register.

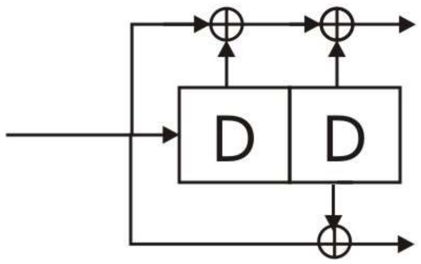

Figure 6 Encoder for convolutional code [7,5]

One of the most used convolutional codes is [171,133], i.e. g0=[1 1 1 1 0 0 1] and g1= [1 0 1 1 0 1 1]. The output functions are (x_t xor x_{t-1} xor x_{t-2} xor x_{t-3} xor x_{t-6}) and (x_t xor x_{t-2} xor x_{t-3} xor x_{t-5} xor x_{t-6}), but for now we will continue with the little [7,5] code as example.

We say that the convolutional code has memory m, corresponding to the number of delay elements used in the implementation, and the encoding can be illustrated by a state machine as shown in Figure 7. Here we see that if we start in state [00] and encode [1 1 0 0] we will get the output [11 01 01 11] and be back in state [00].

4.2 Viterbi Decoding

To understand the decoding we draw a so-called “trellis” where we have all states at every time step as illustrated in Figure 8. Now every possible output sequence from the encoder is represented by a path through the trellis and we can make some kind of measurement of the likelihood of each sequence and find the best one. At first we will just measure the number of errors.

We will label each branch with the number of errors, i.e. the number of positions where the branch output deviates from the received sequence at this step. We will call this “Branch-Metrics”. Then each state at time t can be labelled with the number of errors on the path leading to this state. We will call this “State-

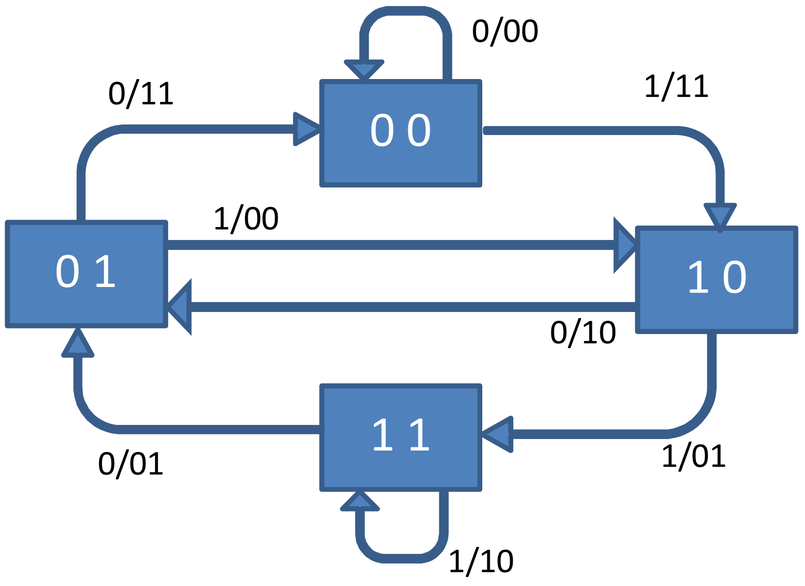

Figure 7 Convolutional code [7,5] as state machine. Transitions labelled by input/output.

Metrics". However, as we see two paths lead to every state. This means that from there they will be identical and since we just want to find the overall best one at the end, we can make a decision between the two incoming paths right away, i.e. chose the best one. The full decoding is therefore performed as a number of "Add-Compare-Select" operations and a "Trace-back" as we shall see.

The decoding is illustrated in Figure 9. We assume that the encoder is started from the [00] state and initialize the first state metrics (t=0) to 0 for state [00] and some large number for the rest. The remaining state metrics are now calculated by the "Add-Compare-Select" process. For each state we look at the two predecessors and add the state-metric to the corresponding branch-metrics. We now have two possible state metrics of which we chose the best, in this case the one with the least number of errors. In case of a tie we just make a random choice (like always selecting 0).The result of our selection is not just a new state metric but also as a "survivor" pointing at the select path. The survivor is the bit that is just shifted out of

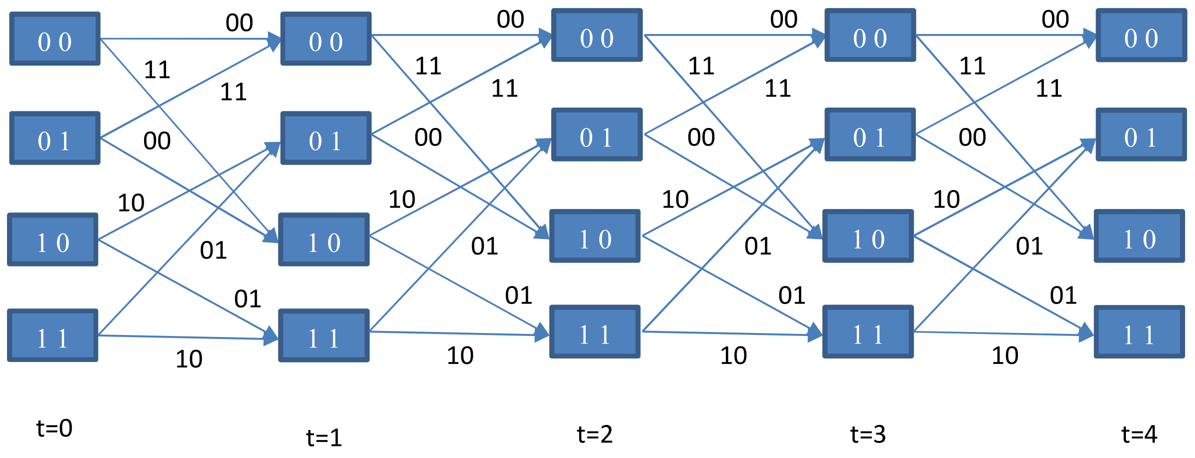

Figure 8 Trellis for decoding of the [7,5] convolutional code

the state register and this should be save in a large array.

We assume that the encoder is brought back to the all-zero state after the actual information sequence. This is done by sending m 0's. We call this termination. In our example we assume this is done already after two information bits so the encoder is back at [00] at t=4. We will therefore start the "trace-back" at state [00] for t=4.

Since the code is terminated the last two information bits are [0 0]. The survivor for state [00] at time t=4 is 1, so most likely we have shifted a 1 out of the state register and came from state [01] at t=3. We can now add one more bit to our decoded sequence and get [1 0 0] – notice that the bits are decoded backwards. We continue from state [01] at t=3 where the survivor is 1. The decoded sequence is now [1 1 0 0] and we continue from state [11] at t=2.

The final decoded sequence ends up as [0 0 1 1 0 0], where the first two 0's are the initial all-zero state and the last two 0's are the termination bits. The actual information is therefore just [1 1]. There is one error since the encoded sequence for [1 1 0 0] is [11 01 01 11] (and the state metric for state [00] at t=4 is 1).

Although this decoding , known as Viterbi decoding, is in fact quite simple, the above describtion may appear somewhat confusing. We will therefore present the decoding as a simple Matlab program. The initial tables "ouput_0" and "output_1" are the branches leading to state s when the preceding state has a 0 or 1 respectively as last bit – i.e. the bit that is shifted out. These can be seen from Figure 8. Remember that Matlab index all arrays from 1 which is why there is a "+1" on all the state indexes.

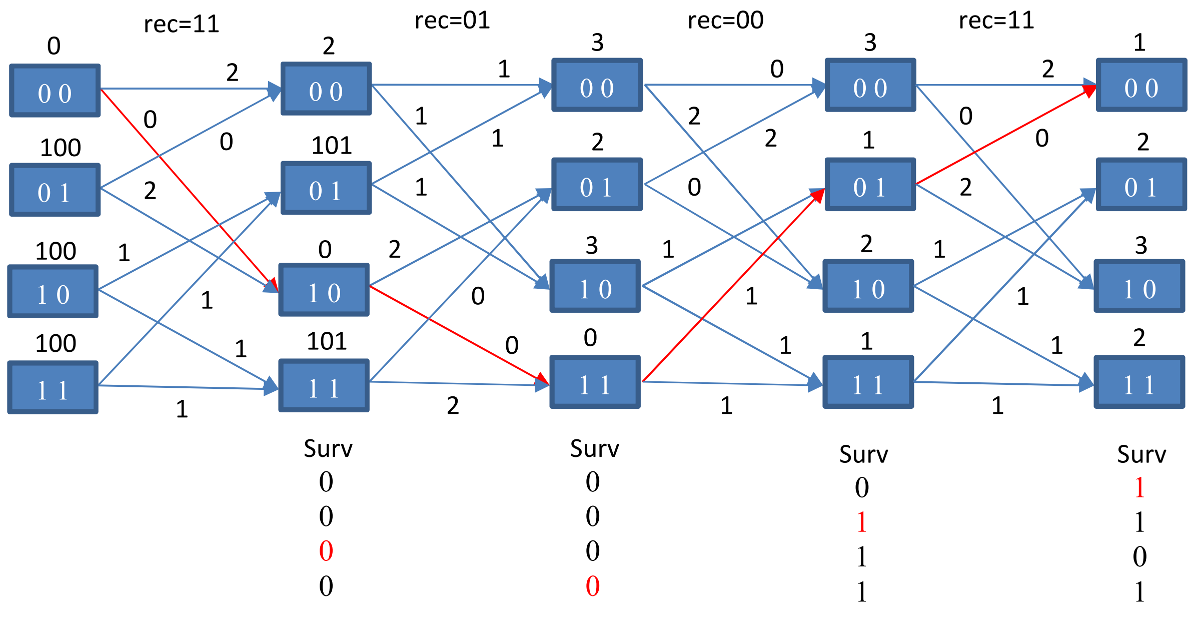

Figure 9 Decoding example

```
rec=[1 1; 0 1; 0 0; 1 1];

% g0 = 1 1 1
% g1 = 1 0 1
output_0 = [0 0;
            1 0;
            1 1;
            0 1];

output_1 = [1 1;
            0 1;
            0 0;
            1 0];

% Forward
state_metric=[0 100 100 100];
new_state_metric=[0 0 0 0];
survivors=zeros(4,length(rec(:,1)));
for t=1:length(rec(:,1))
    for s=0:3
       branch_metric_0 = sum(xor(output_0(s+1,:),rec(t,:)));
       branch_metric_1 = sum(xor(output_1(s+1,:),rec(t,:)));
       state_metric_0 = state_metric(2*mod(s,2)+1)+branch_metric_0;
       state_metric_1 = state_metric(2*mod(s,2)+1+1)+branch_metric_1;

       if state_metric_0 < state_metric_1
           new_state_metric(s+1)=state_metric_0;
           survivors(s+1,t)=0;
       else
           new_state_metric(s+1)=state_metric_1;
           survivors(s+1,t)=1;
       end
    end
    state_metric=new_state_metric;
end

% Backward
output=[0 0];
s=0;
for t=length(rec(:,1)):-1:3
    output=[survivors(s+1,t) output];
    s=2*mod(s,2)+survivors(s+1,t);
end
output
```

Figure 10 Simple Matlab program for Viterbi decoding

4.3 Decoding without Termination

The concept of starting and ending the encoder in the zero-state may be inconvenient in many cases. Sometimes we miss the start of the transmission or we may not want to wait for the end. Also, the termination bits decrease the coding rate.

Viterbi decoders are therefore often designed to cope with a continuous stream of encoded data. In this case the trace-back part of the decoding is just started in a random state after 2·L steps (in practice the all-zero state is usually chosen). In this case, the output will not be correct, but after the first L steps it is reasonable to assume that we are on the correct path and the trace-back for the last L steps are a valid decoded output. The process is repeated when a new set of L steps are received so we actually perform the trace-back twice, first as a kind of training and then for the final release.

The trace-back is illustrated in Figure 11. As seen, we can actually have an initial trace-back and a final trace-back running simultaneously.

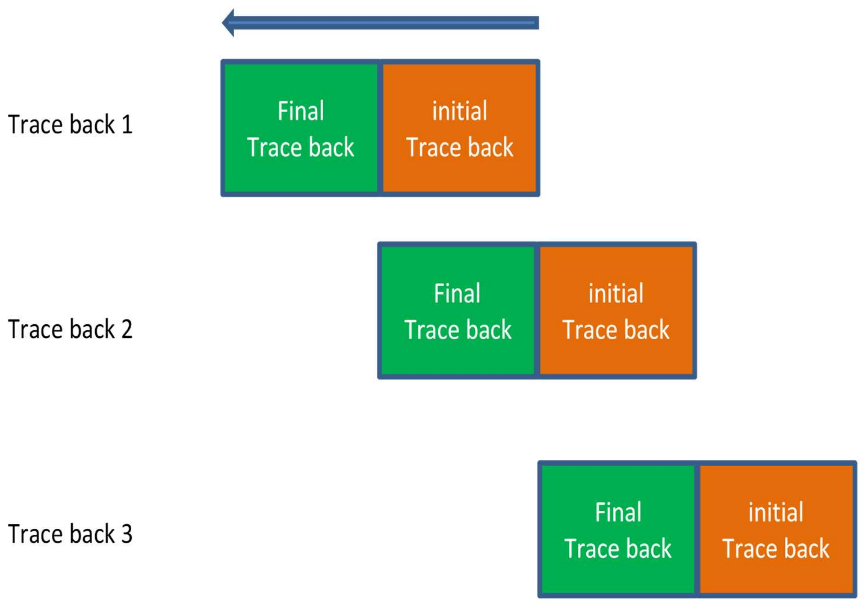

Figure 11 Continuous Trace-back

The trace-back length L should be at least 10 times the memory, but should be adjusted for the specific implementation. If L is too small, there will be additional decoding errors. If L is too large we use unnecessary memory to store the survivors.

If the starting state is unknown all states are initialized to the same metric (0). This will give a higher probability of decoding error in the start, but it is the best we can do.

When the decoding is performed on a long stream of data the state metrics will just keep increasing. This may be inconvenient and since we are just interested in the relative difference we can subtract a constant from all state metrics at a given step, i.e. subtracting the minimum of the state metrics. This is called rescaling.

4.4 Soft-Decision

The real strength of Viterbi decoding is the ability to use soft-decision input. Soft-decision means that the received value is not just one bit but ideally a real number, or, in practice, a number of bits giving a reliability of the received signal. This can give a significant improvement in the performance – provided of course that the channel is able to deliver the soft-decision values.

We now have to use real probabilities in the decoder instead of just counting the errors as we did above. For every path in the trellis we have a transmitted sequence X and want to estimate the probability of receiving the received sequence Y.

For BPSK modulation the probability of receiving a symbol y when x was sent is

$$P(y|x) = \frac{1}{\sqrt{2\pi\sigma}} \cdot e^{-\frac{(y-x)^2}{2\sigma^2}}$$

For a vector of independent symbols this becomes

$$P(\mathbf{Y}|\mathbf{X}) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma}} \cdot e^{-\frac{(y_i-x_i)^2}{2\sigma^2}}$$

Since it is much more easy to add things together than to multiply them we will used the logarithm of the probability to get a metric. In fact we will take –log which means that a low value represent a high probability and a high value a low probability. Further we can skip all constants and scaling factors since we just want to compare vectors of the same length. Maximizing P(**Y**|**X**$_t$) therefore ends up as minimizing

$$\sum_{i=1}^{n} (y_i - x_i)^2$$

i.e. the distance, which is intuitively clear.

For modulation we usually assume that the signal is transmitted as +1 and -1 (BPSK) and not 0 and 1. The received value y likewise come as a signed value. We then get

x = +1	$(y-x)^2 = (y^2+1-2y) \approx -2y \approx -y$
x = -1	$(y-x)^2 = (y^2+1+2y) \approx 2y \approx y$

since the constant 1 can be omitted and also y^2 since that they will be the same on all branches at the given step. The scaling factor 2 are also omitted.

This can be further refined by adding the absolute value of y. Again we omit the scaling factor 2.

If y is positive : metric for x = +1 => -y+abs(y) = 0

metric for x = -1 => y+abs(y) = 2·abs(y) ≈ abs(y)

If y is negative : metric for x = +1 => -y+abs(y) = 2·abs(y) ≈ abs(y)

metric for x = -1 => y+abs(y) = 0

We therefore end up with something that is almost as simple as counting the errors – we just have to count the size of the error instead of just 1 for each error.

4.5 Puncturing

Convolutional codes are born with relatively low coding rates, i.e. 1/2, 1/3… etc. In some contexts a higher coding rate is desired. This can be accomplished by puncturing.

The idea is simply that we omit some of the bits from the encoder. These are not transmitted but put back in the stream before the decoding as "erasures". An erasure is a symbol indicating no preferences for 0 or 1, with soft-decision it is simply the value y = 0.

A puncturing pattern could be specified like shown in Table 7, meaning that every other bit from output 1 is omitted. The rate is therefore 2/3. Another example is shown in Table 8 where the output sequence is

Out1(0), Out2(0),Out2(1),Out1(2),Out1(3),Out2(3),Out2(4),Out1(5)…..

Output 1	1 0
Output 2	1 1

Table 7 Puncturing pattern to rate 2/3

Output 1	1 0 1
Output 2	1 1 0

Table 8 Puncturing pattern to rate 3/4

Puncturing like this of course requires synchronization of encoder of decoder, but this can often be accomplished by trial-and-error in the decoder since nothing will be corrected if the puncturing pattern is synchronized wrongly. Actually this is also the case for the branches themselves which consist of a number (here 2) of bits and also require synchronization.

Summary

A convolutional code encode a stream of data by looking at the last m+1 bits. The outputs are an xor of a selection of theses bits. These selections are often specified in octal notation as e.g. [171,133], meaning that the output functions are (x_t xor x_{t-1} xor x_{t-2} xor x_{t-3} xor x_{t-6}) and (x_t xor x_{t-2} xor x_{t-3} xor x_{t-5} xor x_{t-6}).

Convolutional codes are decoded by Viterbi decoding. Here there is a forward path where the basic operation is an "Add-Compare-Select" process. Each encoder state at each time step is given a "state metric" reflecting the received sequence up to this point. Since there is two paths leading to every state a selection is made by comparing the metrics for these. The choice is saved as a survivor to be used in the "Trace-back" process. In this, the final decoded path is found by a backwards search through the stored survivors.

The Viterbi decoder usually work on a continuous stream by performing the trace-back in two steps. First a training sequence and then a final decoding.

The real strength of convolutional codes are the ability to use soft-decision. Here the errors are simply weighted by the size of the received symbol.

5. Chase Decoding

Many transmission channel are able to deliver a reliability information along with the actual received bit – soft decision. While convolutional codes are good at using this information it is not so easy for block codes. A rather primitive method – but effective in some cases - are Chase decoding (where Chase is the name of the inventor).

The basic idea of Chase decoding is to find multiple codewords close to the received word by a combination of bit-flipping and hard decision decoding. A number of input bits is flipped, i.e. changed from 0 to 1 or vice versa, and a hard decision decoder are then used to find a number of additional errors making it a valid codeword. For each of the possible codewords the distance to the actual received word is calculated based on the soft input values. Finally, the codeword with the best distance is chosen as the most likely.

There exist many variations of the algorithm, but basically there are two parameters: How many positions are candidates for flipping and how many bits can be flipped simultaneously. The candidates for flipping are a number of positions with the most unreliable received values but the main drawback by including too many patterns are extended decoder complexity since the number of hard-decision decodings increase rapidly. For the more efficient solutions the decoding approaches true maximum likelihood decoding.

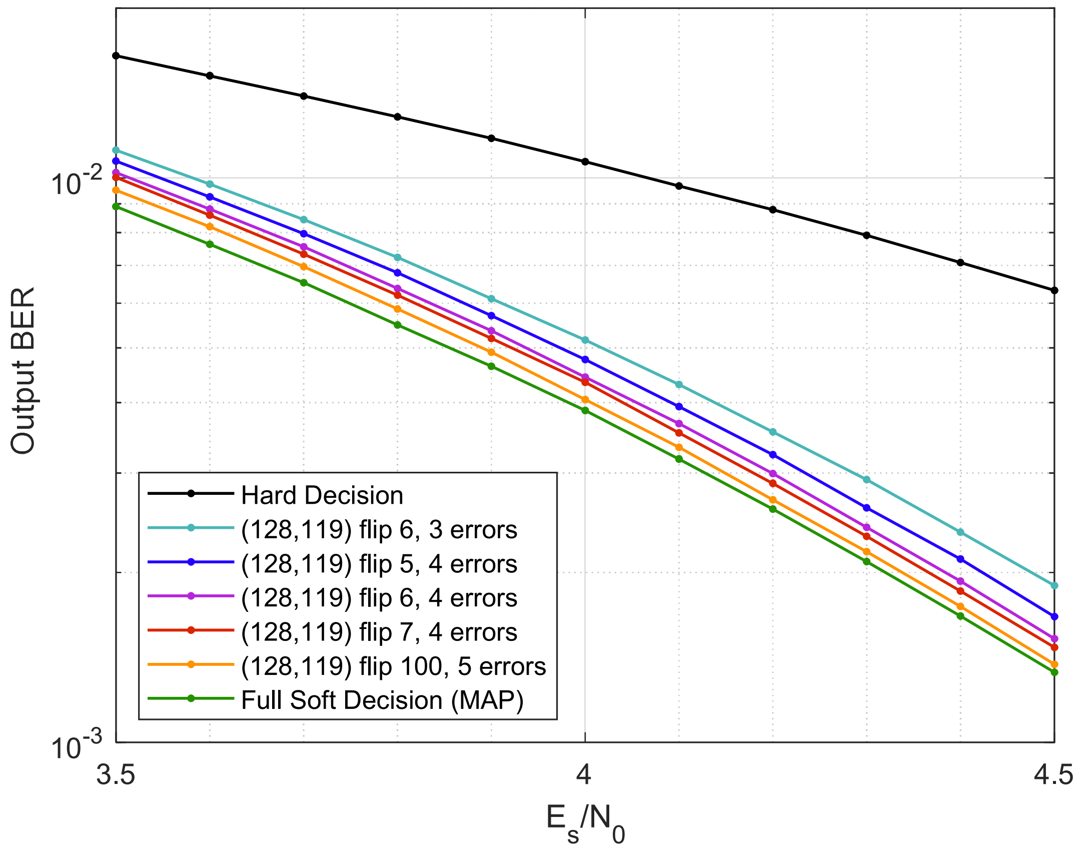

Figure 12 Chase decoding of (128,119) expurgated and extended Hamming code from OIF ZR standard

In Figure 12 we show the results for Chase decoding of the (128,119) expurgated and extended Hamming code from the OIF ZR standard with a number of different parameters. By “flip 6, 4 errors” we mean that we find the 6 most unreliable positions and flip up to 3 at the time - i. e. correct 4 errors including the one from the Hamming code. The results are compared to hard decision and full soft decision.

5.1 Calculating the Distance

In general we want to calculate the probability for receiving the actual received vector when the various suggested codewords where transmitted. Here we only consider BPSK modulation, the situation may be somewhat more complicated for more advanced modulation formats.

For BPSK modulation the probability of receiving a symbol y when x was sent is

$$P(y|x) = \frac{1}{\sqrt{2\pi}\sigma} \cdot e^{-\frac{(y-x)^2}{2\sigma^2}}$$

For a vector of independent symbols this becomes

$$P(\mathbf{Y}|\mathbf{X}) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \cdot e^{-\frac{(y_i-x_i)^2}{2\sigma^2}}$$

Since we just want to compare vectors of the same length we can take the logarithm to the probability and in general skip all constants and scaling factors. Maximizing P($\mathbf{Y}$|$\mathbf{X}_t$) therefore ends up as minimizing

$$\sum_{i=1}^{n} (y_i - x_i)^2$$

i.e. the distance, which is intuitively clear. Also, the probability for the vector is just the sum of the individual bits and we can restrict ourselves to the positions where the codewords disagree.

In general we assume that x is transmitted as +1 and -1 (BPSK) and y likewise come as a signed value. The values of interest may be referred to as a log-likelihood (LL) of -1 and +1, where higher probabilities give larger LL values.

LL(x=+1)	$-(y-x)^2 = -(y^2+1-2y) \approx y$
LL(x=-1)	$-(y-x)^2 = -(y^2+1+2y) \approx -y$

The LL’s are often combined to the log-likelihood-ratio LLR = LL(x=+1) - LL(X=-1) = 2y. Again the scaling factor can be omitted and we say LLR = y.

We take the hard decision vector as reference and may define the log-likelihood for this to 0. Any changes from the hard decision vector should decrease the LL value for the vector, i.e. the hard decision vector is the most likely without regard to the coding.

Changing the assumption about a specific transmitted bit from -1 to +1 - by bit-flipping or decoding - means adjusting the log-likelihood for the vector by LL(x=+1) - LL(x=-1) ≈ LLR = y. Since the received value y is negative in this case (we received -1) this is the same as subtracting abs(y). Likewise when changing from +1 to -1 the log-likelihood should be adjusted by LL(x=-1) - LL(x=+1) ≈ -LLR = -y. Again subtracting abs(y).

The “penalty” of changing a bit away from the hard decision value is always abs(y).

The above description means that all codeword weights are negative, i.e. less than the hard decision set to 0. For an implementation it might be an advantage to change the sign to get positive values and chose the minimum value instead of the maximum.

Summary

A block code can be decoded using the soft input information by a combination of bit-flipping and hard decision decoding. This may give a good improvement of the performance – but the cost is a significant increase in the decoding time, since the hard decision decoder must be used many times for each codeword - or a significant increase in the size of the implementation if we implement many parallel decoders.