A Survey on Turbo Codes and Recent Developments

by

Halvor Utby

halvor.utby@student.uib.no

Thesis submitted in partial fulfilment of the requirements for the degree of *Master of Science*.

University of Bergen
Department of Informatics
December 4, 2006

Preface

This thesis is the result of my studies as a Master student at the Department of Informatics at the University of Bergen.

I would first of all like to thank my advisor Matthew G. Parker for his guidance and help on my thesis. His excellent guidance has been a significant asset to this thesis.

I would also like to thank my fellow students and friends for their help, support and understanding through the process of this thesis; Geir Kjetil Nilsen, Joakim Grahl Knudsen, Olaf Garnaas, Bjørn Harald Fotland, Stian K. Reime, Øystein Nyheim and Sondre Rønjom.

Finally, I would like to thank my girlfriend Marianne, and my family for being supportive and encouraging me to keep going.

Bergen, December 4, 2006,

Halvor Utby

Contents

List of Figures

Chapter 1

Introduction

In today's world communication is one of the most important technologies present. One problem with communication is that wrong interpretations may occur. Fortunately human language is constructed in a way that if a letter is wrong the reader will discover the error and correct it, depending on the context, or ask for a retransmission, with a "Pardon?". In the world of computers however, the situation is a bit different. Words are often constructed by two different signs, one and zero, compared to the English language, which has 26 letters. The zeros and ones are then put in an unique order which makes the word "unmistakable". If one sign is wrong it can be much harder to translate, or in this context, decode the received message to the correct message which was transmitted compared to the English language, which should have much fewer difficulties with correcting errors. This is because English is much more redundant than the unprotected binary message, and this is why many different coding schemes are constructed and used over the entire world. Turbo codes are one of the new ones, first presented in 1993 by Claude Berrou, Alain Glavieux and Punya Thitimajshima[10]. What surprised the coding community was that they came astonishingly close to the Shannon-limit[59]. The Shannon-limit will be explained later in this thesis.

The purpose of most coding schemes is to improve the error correcting capabilities, but there are coding schemes, like kids SMS[1] language, whose purpose is to be short and quick to write. For example "Hello" might be something like "lo", however the disadvantage of this kind of scheme is that each codeword might decode to several likely decoded words. It is often seen when kids communicate with SMS to their parents that wrong interpretations occur. Par-

[1] Short Message Service

ents do not know all their child's "codewords" and misunderstand the meaning of an abbreviation.

Turbo-codes however, like many coding schemes, are made to improve the error correcting capabilities of the message. Turbo-codes are a class of high-performance linear error correcting code which have found use in, for example, satellite communications in deep-space and in other areas like cellphone communication [12, 39], for example in 3G[2] [3, 17].

The difference between coding theory and cryptography will now be defined.

Coding theory is the science of encoding data so that when it is sent over a channel it is most probably decoded to the same data that was sent. Chapter 3 will give a short introduction on coding theory.

Cryptography is the science of encoding data so that nobody except the receiver can decode and read the data that was sent over a channel. So if the data was intercepted by a third party, the third party should not be able to read the data.

This thesis will explain some of the basics of turbo codes, and some of the new developments over the last five years. The reader should be aware that the amount of research is quite large, so this thesis will only discuss some selected topics. In particular, the thesis emphasises recent research into interleaving schemes.

The structure of this thesis is as follows. **Chapter 2** explains briefly some of the channels encountered in this thesis, and the different ways of understanding the binary digits that are used through this thesis. A brief presentation of signal modulation is also given. **Chapter 3** gives a short introduction to some of the coding theory that is the building blocks of turbo codes. **Chapter 4** explains the different components of turbo codes, that is; encoding, interleaving, puncturing, and decoding. **Chapter 5** presents some of the recent developments on turbo codes over the last five years, mainly focusing on interleaving. For a simple product code, four different interleavers have been presented in **Chapter 6**, where the results of

[2] 3G (3rd Generation mobile communicatoin system), is a relatively new mobile phone standard with a high rate data transmission, that can, for example, be used to transfer voice and data simultaneously.

the simulation of these are presented in **Chapter 7**, together with an explanation of the simulations. The conclusion of this thesis is finally given in **Chapter 8**. **Appendix A** explains the use of the programs given in the enclosed CD.

Chapter 2

Data structures and Channels

2.1 Data structures

The alphabet of a digital computer is $\{0, 1\}$ together with the binary operators XOR and AND, ie. $\mathbf{GF(2)}$[1]. The XOR and AND operations are defined as shown in figure 2.1. When data bits are sent over a channel it is sometimes more appropriate to use $\{-1, 1\}$. The turbo decoders of Chapter 6 map +1 to "1" and -1 to "0". So they use the alphabet $\{0, 1\}$ and the switch to a $\{-1, 1\}$ happens when one considers log-likelihoods. If not stated otherwise, the term

	0	1
0	0	1
1	1	0

(a) The XOR operation

	0	1
0	0	0
1	0	1

(b) The AND operation

Figure 2.1: The truth table for the binary operations XOR and AND.

"binary vectors" will for the rest of this thesis mean the elements $\{0, 1\}$ along with the binary operators XOR and AND.

2.2 Channels

A channel is the medium used to transport information from a sender to a receiver. Since none of the codes in this thesis are actually sent in practice, a simulated channel has to be constructed so that the "sent" message will encounter some kind of noise.

[1] Galois Field

2.2.1 Additive White Gaussian Noise (AWGN)

A common channel model used is the Additive White Gaussian Noise, or AWGN, channel, which is a good model for many satellite and deep space communication links. On the other hand, this is a not such a good model for most terrestrial links since they are also concerned with multipath, terrain blocking and interference.

If a transmitted signal $s(t)$ at the time t is sent over an AWGN channel the received signal $r(t)$ is given by

$$r(t) = s(t) + n(t), \tag{2.1}$$

where $n(t)$ is Gaussian noise [40]. This Gaussian noise is a random, normally distributed variable, that can be expressed as the function

$$n(t) = \frac{1}{\sigma\sqrt{2\pi}} \exp^{-\frac{(t-\mu)^2}{2\sigma^2}}. \tag{2.2}$$

σ is the noise variable, and μ is the mean to the function of the normal distribution [40]. Since the signals sent over the channel are electric voltages the mean will be $\mu = \pm 1$. In figure 2.2 the curve

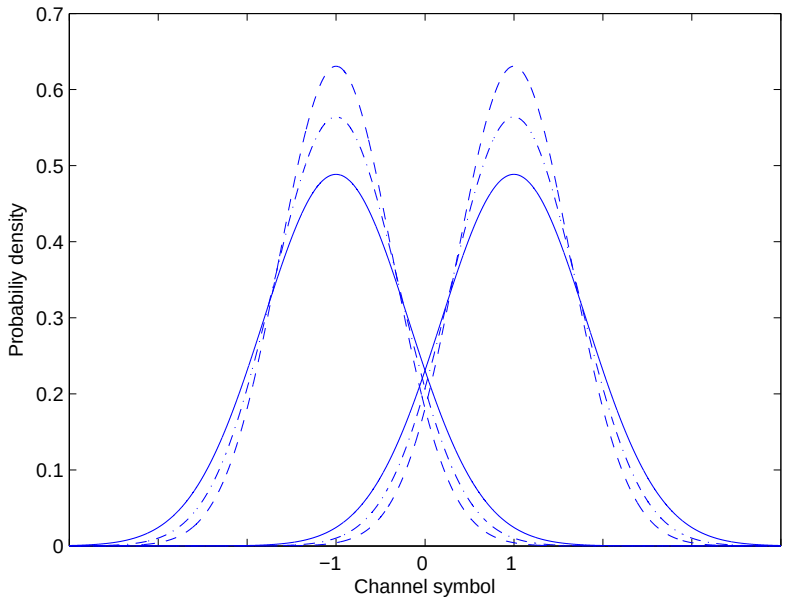

Figure 2.2: The probability density functions for Signal-to-Noise Ratios (SNRs) = 1.5dB, 2.0dB, 2.5dB from bottom to top.

for the probability function is given for a code with rate $R = 1/2$. Equation (3.5) has been used to compute σ.

2.2.2 Discrete Memoryless Channel (DMC)

In a Discrete Memoryless Channel model, the received signal at any given time is only dependent on the transmitted signal at a given time, and therefore the channel is called memoryless. Transition probabilities describe the different "transformations" that may occur during transmission of an information sequence, and are normally denoted $P(j|i), 0 \leq i \leq M-1, 0 \leq j \leq Q-1$, where i is the transmitted symbol from the sender, and j is the received symbol at the receiver. M is the number of possible inputs in the channel model, and Q is the number of possible outputs from the channel model. For example, if the input is binary, which means that $M = 2$, and the output is quaternary, that is $Q = 4$, the channel model can be represented by figure 2.3 [40].

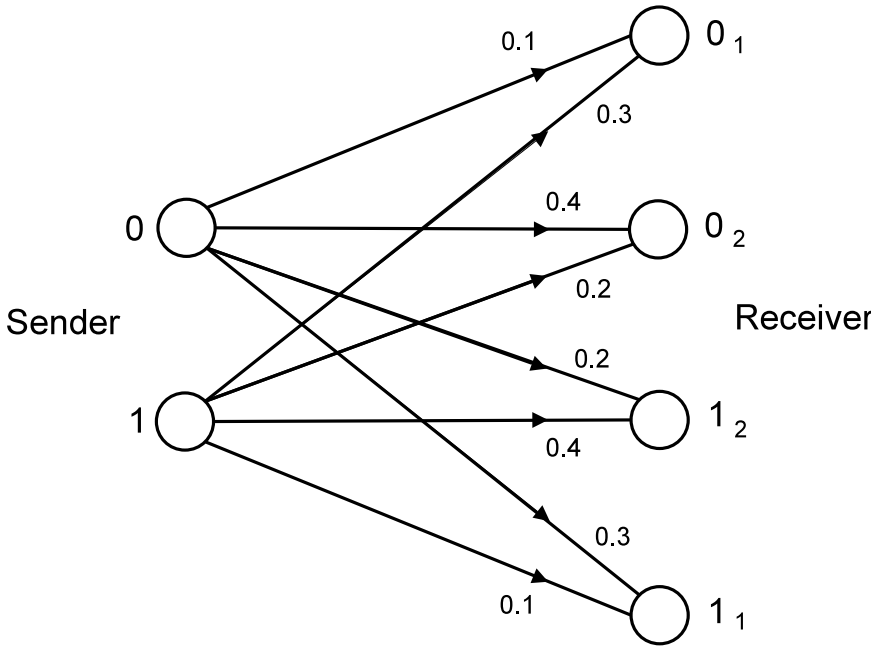

Figure 2.3: An example of a Discrete Memoryless Channel with $M = 2$ and $Q = 4$.

Each arrow in the figure shows different possible transformations, where the decimal number connected to an arrow denotes the transition probability of this transformation. A similar example can be found in [40]. Later, in chapter 3.4.3, the Viterbi algoritm is introduced. If the Viterbi algorithm is used over a DMC it is normally helpful to represent the different transition probabilities in a metric table. Computing a metric table is simply done by using the logarithm base 10 on the transition probabilities [40], resulting in figure

2.4.

$v_l \setminus r_l$	0_1	0_2	1_1	1_2
0	-1.0	-0.4	-0.7	-0.52
1	-0.52	-0.7	-0.4	-1.0

Figure 2.4: An example of a metric table. The numbers are the logarithm base 10 to the transition probabilities shown in figure 2.3. Where v_l are the sent signals and r_l are the received signals.

These metrics are used to weight the different paths in a trellis. This will be explained later in chapter 3.4.3.

2.2.3 Binary Symmetric Channel (BSC)

In a Binary symmetric channel, or BSC, which is a special case of a DMC where $M = 2$ and $Q = 2$, the probability of a "1" becoming a "0", and "0" becoming "1" the same. This is an idealized communications channel model, and probably not very likely to be experienced often in real life. A BSC is illustrated in figure 2.5, which shows that the probability that the correct bit is received is (1-p). Every bit is independent of all the other bits. If the probability of an error is known to be large, $p > 1/2$, the decoder could simply swap the bits, namely "1" to "0", and "0" to "1".

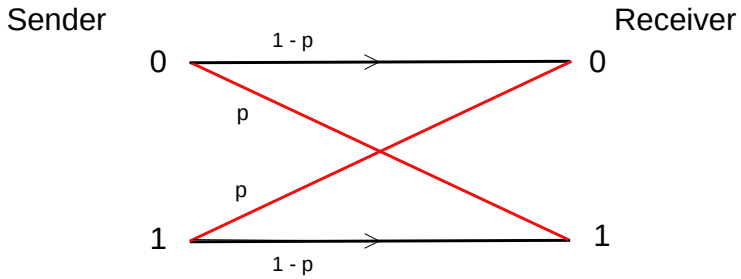

Figure 2.5: A Binary Symmetric Channel

2.2.4 Binary Erasure Channel (BEC)

A Binary Erasure Channel (BEC), shown in figure 2.6, has two inputs and three outputs [53]. Two of the outputs are the same as the input, namely, $\{0, 1\}$. The third output is $\{?\}$ that can be read as somewhere in between $\{0, 1\}$. This state can be explained in that the demodulator of the receiver does not know how to interpret the signal.

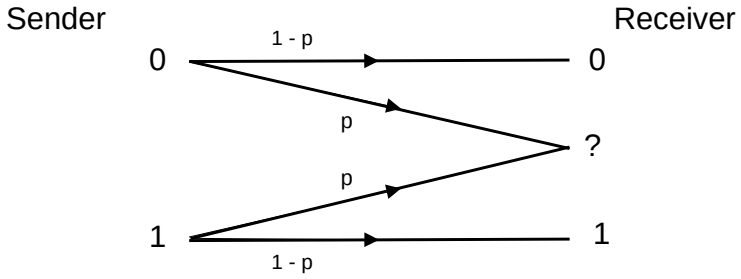

Figure 2.6: A Binary Erasure Channel

2.3 Signal Modulations

When data is sent over a channel, the signal has to be modulated by some kind of waveform. This waveform has a duration of T seconds and is able to generate the two signals $s_1(t)$ and $s_2(t)$ for the encoded "1" and "0" respectively. The simplest form of modulation is the Binary Phase Shift Keying (BPSK) [67], where the optimum choice of signal is

$$s_1(t) = \sqrt{\frac{2E_s}{T}} \cos 2\pi f_0 t, 0 \leq t \leq T, \tag{2.3a}$$

$$s_2(t) = -\sqrt{\frac{2E_s}{T}} \cos 2\pi f_0 t, 0 \leq t \leq T, \tag{2.3b}$$

where the frequency f_0 is a multiple of $1/T$ and E_s is the energy of each signal [40].

To increase the bit rate, other modulations can be used [67, 40]. All these modulations have $M = 2^l$ channel signals for some l. The signals can therefore be given by

$$s_i(t) = \sqrt{\frac{2E_s}{T}} \cos(2\pi f_0 t + \phi_i), 0 \leq t \leq T, \tag{2.4}$$

where $\phi_i = 2\pi(i-1)/M$ for $1 \leq i \leq M$. One modulation technique that is widely used is 4-PSK[2], which has $M = 4$. A 4-PSK signal constellation is shown in figure 2.7.

[2]also known as Quadrature Phase Shift Keying (QPSK).

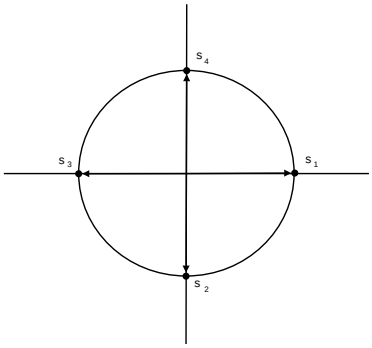

Figure 2.7: A QPSK or 4-PSK signal constellation.

Chapter 3

Coding theory

As there are plenty of good books on coding theory this chapter will not enter deep into the science of coding theory. It will only scratch the surface and give a short introduction in some of the areas, that are closely related to Turbo-codes.

3.1 Hamming Distance and Hamming Weight

The Hamming distance between two binary vectors of the same length, is the number of positions in which the symbols in the two vectors differ. So if the first vector is subtracted from the second vector (modulo-2), then the number of bits which equal "1" sum up to the Hamming distance between the two vectors. The Hamming distance between two vectors, r and v, is often denoted $d(r, v)$.

The free distance (d) is a minimal Hamming distance between different encoded sequences. Convolutional codes produces a continuous bitstream when encoding, therefore free distance can be understood as a minimal length of an erroneous burst error at the output of a convolutional decoder [37, 11, 24, 23]. Where a burst error is several consecutive bits disturbed by noise.

If not stated otherwise the Hamming distance is for the rest of the thesis referred to as distance.

The Hamming weight is the Hamming distance of a vector from the zero vector[45].

If not stated otherwise the Hamming weight is for the rest of the thesis referred to as weight.

3.1.1 Minimum distance

An important parameter of many codes, for example, block codes is the minimum distance [40]. Let $x = (x_0, x_1, \cdots, x_{n-1})$ be a binary vector of length n. Then the weight of x is denoted $w(x)$. Let $y = (y_0, y_1, \cdots, y_{n-1})$ also be a binary vector of length n. The distance between x and y is defined as the number of places they differ and is denoted $d(x, y) = w(x + y)$. If a code has the set of codewords C, the minimum distance of C is defined by

$$d_{\min} \triangleq \min \{d(x, y) : x, y \in C, x \neq y\} . \tag{3.1}$$

3.1.2 Error Vector

Interference can occur when a codeword is sent over a channel, as shown in figure 3.1. This interference can be expressed as an error vector. For example, if a sent vector v, and a received vector r, which are both binary, are subtracted from the other, then the result, E, will also be binary vectors [13]. This error vector represents the noise on the channel, where $E = r - v$. It is preferable that the Hamming Weight of the error vector is as small as possible.

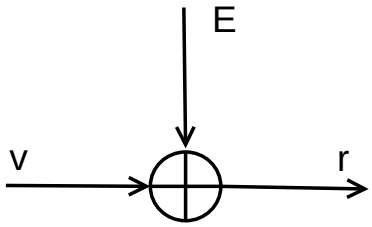

Figure 3.1: The codeword sent, v ,is added with noise vector E resulting in the received vector r.

3.2 Bit-Error Rate and Signal-to-Noise Ratio

Bit Error Rate, or BER, is defined as the probability that a decoded infomation bit from the output of the decoder is in error[40]. This error probability should be as low as possible given the constraints of bandwidth and power. If a code has code rate $R = k/n$, it means that the n symbols of the output codeword depend on only k bits of the input information. The energy per transmitted symbol, normally denoted E_s, will then be a factor in the energy-per-information bit

$$E_b = \frac{E_s}{R}. \tag{3.2}$$

Coded communication systems is often measured by their error probability. This error probability can be expressed in the terms of the energy-per-information bit E_b to the to the one sided Power Spectral Density (PSD) N_0. At the receiver the received signal can be denoted E_b/N_0, that expresses the Signal-to-Noise Ratio (SNR). The SNR is normally expressed in decibel (dB), that means that $10 \log E_b/N_0$ is the norm of expressing SNR [67]. It follows from equation (3.2) that the SNR can be rewritten as

$$\frac{E_b}{N_0} = \frac{E_s}{R \cdot N_0}. \tag{3.3}$$

For example, in an AWGN channel, discussed in section 2.2.1, is the noise variance $2\sigma^2$ in equation (2.2) the PSD, that means

$$N_0 = 2\sigma^2. \tag{3.4}$$

Hence, can the the noise variance for a selected SNR over an AWGN channel be computed by [40]

$$\sigma^2 = \frac{1}{2R\frac{E_b}{N_0}}. \tag{3.5}$$

Equation (3.5) shows that the noise variance is not only dependent of the SNR but also the rate of the code. In chapter 6 this calculations are used to find the noise variance for some selected SNR, where the results are presented in chapter 7.2.

3.3 Log-likelihood algebra and probability

Most of the different iterative decoding algorithms used on turbo codes uses log-likelihood algebra [30] in the decoding process. Therefore, is a brief introduction on the subject required.

3.3.1 Bayes' theorem

Bayes' theorem is one of the foundations of mathematical hypothesis testing. The theorem shows how to update variables in light of new information, also called a posteriori information. The decoding process of turbo-codes is iterative, which means that it is done in several steps. This iterative process uses the likelihood information to reconcile the difference between the two decoders. This will be explained in more detail in chapter 4.4 on page 46. Bayes' theorem is of central importance to the understanding of turbo-codes.

The theorem says:

$$\mathrm{P(A|B)} = \frac{\mathrm{P(B|A)\ P(A)}}{\mathrm{P(B)}}. \tag{3.6}$$

Equation (3.6) is easily derived from the definition of conditional probability, the probability of event A, given event B, is as follows:

$$\mathrm{P(A|B)} = \frac{\mathrm{P(A \cap B)}}{\mathrm{P(B)}}. \tag{3.7}$$

Similarly:

$$\mathrm{P(B|A)} = \frac{\mathrm{P(B \cap A)}}{\mathrm{P(A)}}. \tag{3.8}$$

(3.7) and (3.8) can then be arranged together and result in:

$$\mathrm{P(A|B)\ P(B) = P(A \cap B) = P(B|A)\ P(A)}. \tag{3.9}$$

Dividing both sides by Pr(B), results in Bayes' theorem.

$$\mathrm{P(A|B)} = \frac{\mathrm{P(B|A)\ P(A)}}{\mathrm{P(B)}}. \tag{3.10}$$

The result of Bayes' theorem can be expressed in alternative forms if preferable, interested readers are referred to [40].

3.3.2 Log-likelihood algebra

Turbo codes use a couple of different algorithms to decode the received message. As stated earlier, a central part of these algorithms is log-likelihood algebra [30, 63, 40]. Let a set D have the elements $\{+1, -1\}$. For simplicity, let -1 be the null element under mod 2 addition of the exponent of (-1), denoted $\oplus$. The probability that a random variable D takes the value d is denoted by $P_D(d)$. The log-likelihood ratio (LLR) $L_D(d)$, is defined by

$$L_D(d) = \log \frac{P_D(d = +1)}{P_D(d = -1)}. \tag{3.11}$$

This LLR is often called the "soft" value of the variable D. If a random variable X influences the random variable D, then the log-likelihood ratio is conditioned. A conditioned LLR is denoted $L_{D|X}(d|x)$ which in [30] is defined by

$$\begin{aligned} L_{D|X}(d|x) &= \log \frac{P_D(d=+1|x)}{P_D(d=-1|x)} \\ &= \log \frac{P_D(d=+1)}{P_D(d=-1)} + \log \frac{P_D(x|d=+1)}{P_D(x|d=-1)} \qquad (3.12) \\ &= L_D(d) + L_{X|D}(x|d). \end{aligned}$$

$L_{X|D}(x|d)$ is the LLR obtained by measuring the output x under the condition that $d = +1$ or $d = -1$. This measurement is normally done when the receiver receives a message that is sent over some kind of channel. Examples of channels are given in chapter 2.2.

If two random variables d_1 and d_2 are statistically independent, then the sum of their log-likelihood ratios (LLRs) will then be defined as [64][1]:

$$L(d_1) \boxplus L(d_2) \triangleq L(d_1 \oplus d_2) = \ln \left[\frac{e^{L(d_1)} + e^{L(d_2)}}{1 + e^{L(d_1)} + e^{L(d_2)}} \right]. \qquad (3.13)$$

This can be approximated by

$$\approx (-1) \times sgn\left[L(d_1)\right] \times sgn\left[L(d_2)\right] \times \min(|L(d_1)|, |L(d_2)|). \quad (3.14)$$

The $\oplus$ sign denotes modulo-2 addition, the $+$ sign is ordinary addition and the $\boxplus$ denotes log-likelihood addition, which is the mathematical operation defined in equation (3.13).
sgn is the sign of the number, ie +1 or -1, and
min is the minimum of two elements. Following, the sum of two LLRs, where one is very large or very small is then defined by

$$\begin{aligned} L(d) \boxplus \infty &= L(d) \\ L(d) \boxplus -\infty &= -L(d), \end{aligned}$$

respectively, and

$$L(d) \boxplus 0 = 0.$$

In section 6.1 is a decoding example based on equation (3.14) given.

3.3.3 Maximum likelihood decoding (MLD)

When a decoder receives a codeword r, its task is to produce an estimate u' of the sent message u. Since every message u has one

[1] The calculations of the result of equation (3.13) can be found in Appendix a in [64]

unique codeword v connected to it, the task of the decoder is to find an estimate v' of the sent codeword v. Having section 3.1.2 in mind, it should be clear that $u' = u$ only if $v' = v$ [40]. However, if $v' \neq v$ an error has clearly occurred. Using conditional probability, described in chapter 3.3.1, to find the conditional error probability of the decoder is then defined as

$$P(E|r) \triangleq P(v' \neq v|r). \tag{3.15}$$

One then uses the probability of the event E, that in [52] is defined as the sum of the probabilities of the outcomes in E. The probability of the received codeword r is given by $P(r)$, and the error probability of the decoder is therefore

$$P(E) = \sum_r P(E|r)P(r). \tag{3.16}$$

In section 3.1.2 it was stated that the Hamming weight of the error vector E should be as small as possible, equivalently to minimising $P(E)$. However r is produced before decoding, and is therefore independent of the decoding rule used. So minimising $P(E)$, will also minimise $P(E|r)$, given in equation (3.15) [40]. Minimising (3.15) is equivalent to maximising $P(v' = v|r)$. For a given r, $P(E|r)$ is minimised by choosing the vector v' as the codeword v that maximises

$$P(v|r) = \frac{P(r|v)P(v)}{P(r)}. \tag{3.17}$$

In a memoryless channel each received symbol depends only on the responding sent symbol, therefore, in a DMC[2], described in chapter 2.2.2, maximising (3.17) is equivalent to maximising $P(r|v)$, if $P(v)$, which means that every codeword is equally likely to occur

$$P(r|v) = \prod_i P(r_i|v_i). \tag{3.18}$$

A maximum likelihood decoder (MLD) chooses its estimates to maximise equation (3.18). From [2, 40] we know that log x is a monotone increasing function of x, so maximising (3.18) is the same as maximising the log-likelihood function

$$\log P(r|v) = \sum_i \log P(r_i|v_i). \tag{3.19}$$

Later in section 3.4.3 equation (3.28) defines a specialised decoding rule for the MLD in a BSC[3] model, which in that case replaces equation (3.19).

[2]Digital Memoryless Channel
[3]Binary Symmetric Channel

3.4 Types of Codes

Turbo codes can use two quite different types of error-control code. This section will briefly describe the two.

3.4.1 Block codes

A block code encoder divides the information bits into blocks of k bits. Each block then consists of a message, $u = (u_0, u_1, \cdots, u_{k-1})$. Accordingly, the message is written in binary, $\{0, 1\}$, and the total number of possible messages is 2^k. Each message u, will then independently be transformed by the encoder into codewords, denoted $v = (v_0, v_1, \cdots, v_{n-1})$. The encoder can use many different techniques. Interested readers can find more information about linear block codes in[40].

Hamming code

A Hamming code is a linear error-correcting code, which can detect and correct single-bit errors. The code can also detect double-bit errors, but not correct them.

Let V(r, 2) denote the set of all ordered r-tuples over $\mathbf{GF}(2)$. One vector in such a set is denoted x = $(x_1, x2, \ldots, x_r)$. In [49] the Ham(16,11) extended Hamming code is presented as a block code which can be used in Turbo Product codes. An algebraic introduction to Hamming codes can be found in [13].

Definition Let r be a positive integer and let H be an $r \times (2^r - 1)$ matrix whose columns are the distinct non-zero vectors of V(r, 2). The code having H as its parity-check matrix is called a binary *Hamming code* and is denoted by Ham(r, 2)[31].

3.4.2 Convolutional codes

Convolutional codes are a type of error-correcting code that are often used to improve the performance of wireless digital communication links, like radio or satellite links. They were first introduced in P. Elias book "Coding for Noisy Channels" in 1955[40, 19]. One important difference between convolutional codes and block codes is that the encoder contains memory. Encoders of convolutional codes can

also be divided into two categories, namely feedforward and feedback. In both of these categories the encoder can be systematic or nonsystematic.

Encoding

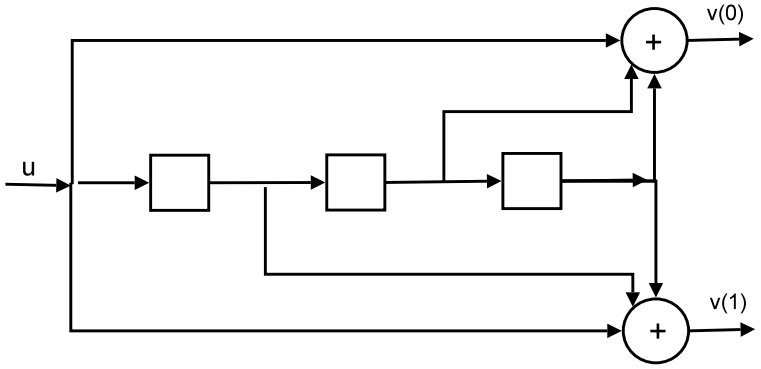

Figure 3.2: A binary nonsystematic feedforward convolutional encoder

Figure 3.2 is based on a similar figure found in [40]. This example show how a simple convolutional encoder with rate R = 1/2 might work. The figure can be viewed as a shift register with u as input, and $v^{(0)}$ and $v^{(1)}$ as output. The information sequence $u = (u_0, u_1, u_2, \ldots)$ enters the encoder, one bit at the time. From the figure the reader can see that the encoder holds memory. This encoder actually contains memory of order m = 3, which is shown by the boxes in the figure. The circle with a plus inside XORs the bits from the boxes. It is often easier to represent the encoder figure as generator sequences instead. From the figure the generator sequences, $g^{(0)}$ and $g^{(1)}$, will be:

$$g^{(0)} = (1\ 0\ 1\ 1), \tag{3.20}$$

$$g^{(1)} = (1\ 1\ 0\ 1), \tag{3.21}$$

$g^{(0)}$ and $g^{(1)}$ are prescribed by the connections shown in the figure. Computing the outputs, $v^{(0)}$ and $v^{(1)}$, is done by performing discrete convolution, denoted $\circledast$. All the operations are modulo-2. The two output sequences will then be denoted by the following encoding equations:

$$v^{(0)} = u \circledast g^{(0)}, \tag{3.22}$$

$$v^{(1)} = u \circledast g^{(1)}. \tag{3.23}$$

Discrete convolution can be written as a sum of products for all $l \geq 0$, namely:

$$v_l^{(j)} = \sum_{i=0}^{m} u_{l-i} g_i^{(j)} = u_l g_0^{(j)} + u_{l-1} g_1^{(j)} + \cdots + u_{l-m} g_m^{(j)}, j = 0, 1. \quad (3.24)$$

Let us pick a random message, namely $u = (1\ 0\ 1\ 0\ 1)$. Thus encoding this message can be expressed:

$$v^{(0)} = (1\ 0\ 1\ 0\ 1) \circledast (1\ 0\ 1\ 1) = (1\ 0\ 0\ 1\ 0\ 1\ 1\ 1), \quad (3.25)$$

$$v^{(1)} = (1\ 0\ 1\ 0\ 1) \circledast (1\ 1\ 0\ 1) = (1\ 1\ 1\ 0\ 1\ 0\ 0\ 1). \quad (3.26)$$

Before transmission of the two output sequences, (3.25) and (3.26), can be concatenated to form a codeword of the form:

$$v = (1\ 1,\ 0\ 1,\ 0\ 1,\ 1\ 0,\ 0\ 1,\ 1\ 0,\ 1\ 0,\ 1\ 1). \quad (3.27)$$

It is sometimes convenient to represent the two generator sequences by a matrix. This matrix, denoted $\mathbf{G}$, is constructed by interlacing the generator sequences, in this case $g^{(0)}$ and $g^{(1)}$. Every interleaved row in $\mathbf{G}$ will be exactly the same as the first row of $\mathbf{G}$, the only difference being a shift of length l, where l is the number of generator sequences. So in this example each shift will be 2. The number of rows of $\mathbf{G}$ will be the same as the length of the information sequence u.

$$\mathbf{G} = \begin{bmatrix} g_0^{(0)} g_0^{(1)} & g_1^{(0)} g_1^{(1)} & g_2^{(0)} g_2^{(1)} & \cdots & g_m^{(0)} g_m^{(1)} & & \\ & g_0^{(0)} g_0^{(1)} & g_1^{(0)} g_1^{(1)} & \cdots & g_{m-1}^{(0)} g_{m-1}^{(1)} & g_m^{(0)} g_m^{(1)} & \\ & & g_0^{(0)} g_0^{(1)} & \cdots & g_{m-2}^{(0)} g_{m-2}^{(1)} & g_{m-1}^{(0)} g_{m-1}^{(1)} & g_m^{(0)} g_m^{(1)} \\ & & \ddots & & & & \ddots \end{bmatrix}.$$

The codeword can now be defined as a matrix-vector multiplication, $v = u\mathbf{G}$.

$$= (1\ 0\ 1\ 0\ 1) \begin{bmatrix} 1\,1 & 0\,1 & 1\,0 & 1\,1 & & & & \\ & 1\,1 & 0\,1 & 1\,0 & 1\,1 & & & \\ & & 1\,1 & 0\,1 & 1\,0 & 1\,1 & & \\ & & & 1\,1 & 0\,1 & 1\,0 & 1\,1 & \\ & & & & 1\,1 & 0\,1 & 1\,0 & 1\,1 \end{bmatrix}$$

$. = (1\ 1,\ 0\ 1,\ 0\ 1,\ 1\ 0,\ 0\ 1,\ 1\ 0,\ 1\ 0,\ 1\ 1)$ which is exactly the same as found in (3.27).

Convolutional encoders can be constructed in many different ways. The previous example was a code with rate R = 1/2. Other encoders with rate 2/3 might have for instance two input sequences and three output sequences, examples of which can be found in[40].

Systematic encoders have one output, $v_{(i)}$, that is systematic, meaning that the output at this point is exactly the same as the input data. Compared to figure 3.2 this means that, for example, output $v^{(0)}$ is an exact copy of u unlike the case for $v_{(0)}$ in 3.2. The other output $v^{(1)}$ should not also be a copy of u, otherwise we will simply generate a repetition code. If, on the other hand, there were two distinct input sequences, then for the encoder to be systematic, one would require two systematic output sequences.

A Systematic Feedback encoder can generate the same code as a corresponding nonsystematic feedforward encoder. However the mapping between information sequences and codewords is different. Figure 3.3 shows the recursive data handling in a systematic feedback encoder, where the arrow at the bottom feeds back the data to the beginning of the encoder. The arrow at the top, which ends in $v_{(0)}$, is a systematic sequence, and therefore this encoder is systematic.

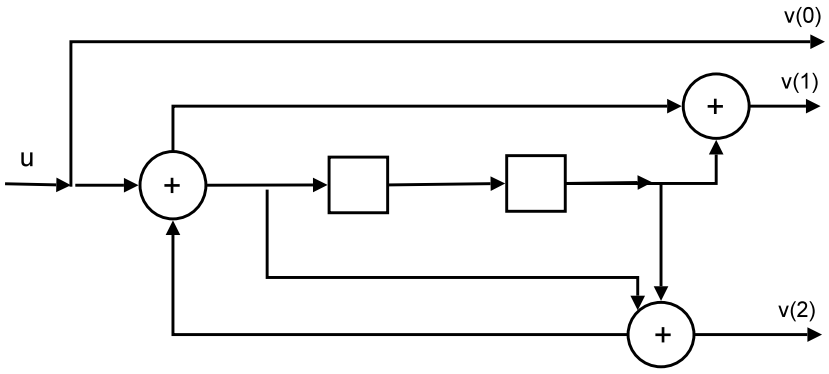

Figure 3.3: A systematic feedback convolutional encoder

3.4.3 Decoding of Convolutional codes

A convolutional decoder can be understood as a finite state machine [52], and can therefore be represented by a trellis structure. This trellis is constructed by the different states of the encoder. The decoder's task is to estimate the sequence which was really sent. Figure 3.4 shows the trellis structure used to decode codewords sent via a channel from the encoder shown in figure 3.2.

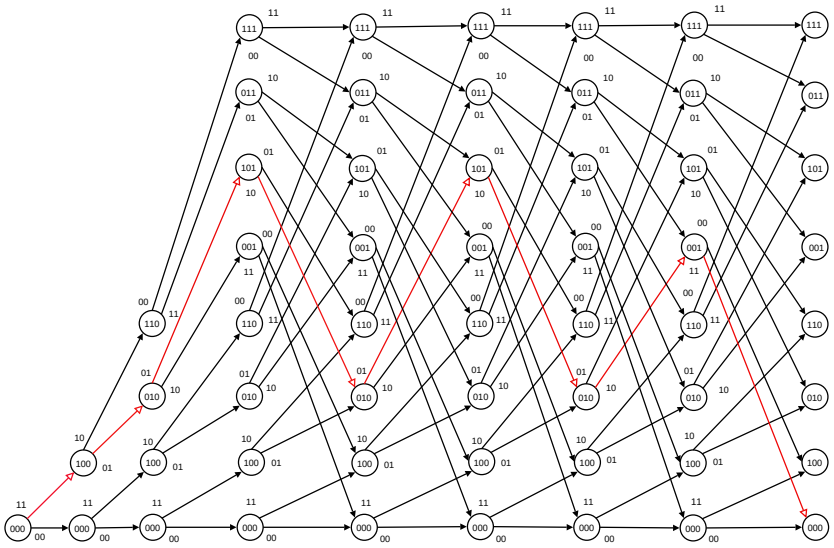

Figure 3.4: Decoding trellis for the example code in figure 3.2. The arrows with unfilled head shows the path to the sequence $u = (1\ 0\ 1\ 0\ 1)$. The numbers inside each node tell the state of the encoder.

Viterbi algorithm

The Viterbi algorithm, was introduced in 1967 by Andrew J. Viterbi in "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm"[18, 40]. It is a decoding algorithm for convolutional codes which exploits the decoding trellis. A received word r might differ from the sent codeword v. This difference is called the error vector, denoted E. Each path through the trellis represents a received codeword, and the log-likelihood function, $\log(r|v)$, is called the metric associated with the path of the sent codeword, which often is denoted as $\mathrm{M}(r|v)$.

The algorithm is fairly simple and contains 3 steps[40, 48]:

1. Start at the left-most state of the trellis and set time t = i. Compute and save the states survivor path and metric.

2. For each state at time, t+1, compute the metric of all incoming paths by adding the metric of the former states survivor with the metric for the connection path between the two nodes. For each state compare all the metrics entering the state and select the path with the largest metric. Store this survivor path together with its metric, and delete all other paths.

3. If t is less than the length of the information sequence, repeat

2. If not, stop and return the survivor path.

This algorithm is in fact a maximum likelihood decoder. Using this algorithm on the code produced by the encoder shown in 3.2 and sent over a channel, for example a BSC[4], where the transition probability $p < 1/2$ is just for simplicity. Section 3.3 says that the log-likelihood function, first defined in (3.19) will then be:

$$\log P(r|v) = \mathrm{d(r,\ v)} \log \frac{p}{1-p} + N \log(1-p). \tag{3.28}$$

N is the length of the codeword, and d(r, v) is the Hamming distance between **r** and **v**. Since $\log \frac{p}{1-p} < 0$ and $N \log(1-p)$ is a constant for all codewords, and the channel is a BSC, the MLD, algorithm will choose the codeword, v, that has the smallest Hamming distance from the codeword r, which is the final survivor. In other words the path with the smallest Hamming distance, will be the output of the decoder. The reader should notice that if another channel had been used, which does not have the simple structure of the BSC, then the survivor path would not necessarily be the one with the smallest Hamming distance. It would rather be the path with the largest metric.

When the codeword (1 0 1 0 1) is entered in figure 3.2 the result is given in (3.27) to be v = (1 1, 0 1, 0 1, 1 0, 0 1, 1 0, 1 0, 1 1). However, when sent over a channel the received codeword might be modified. For example, let the received codeword be
r = (1 0, 0 1, 0 0, 1 0, 0 1, 1 0, 1 0, 1 1). Then the error vector is
E = (0 1, 0 0, 0 1, 0 0, 0 0, 0 0, 0 0, 0 0). The decoding trellis for codeword v' is shown in figure 3.5. In this example the error vector E has Hamming weight 2, which means that the received codeword has 2 errors. When receiving this message, the receiver normally knows nothing, or little, about the sent codeword or the number of errors. Using the Viterbi algorithm, given above, and the decision rule given in equation (3.28), a trellis, shown in figure 3.5, can be constructed. This trellis illustrates all possible codewords, and the minimised Hamming distance from the received codeword r to the path, is shown in each node. On the right hand side are the final states of the trellis. The node with the smallest Hamming Weight is chosen as the surviving path, hence the selected "correct" codeword. The "00" state in the trellis, shown as the bottom line, is a state where the encoder registers are all zero. When encoding a message an encoder is normally empty, and therefore starts and ends the decoder always in the "00" state. The "correct" codeword should also

[4]BSC was described in chapter 2.2.3

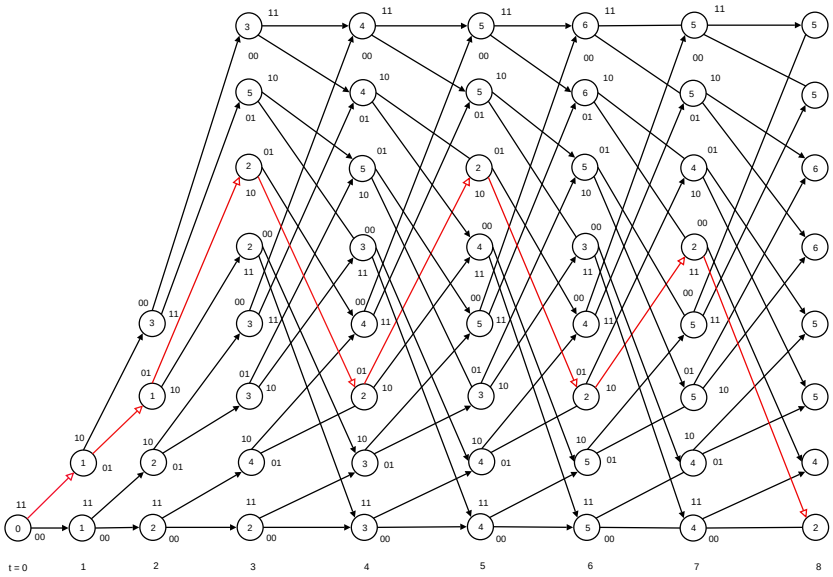

Figure 3.5: Decoding trellis with the Hamming weights inside each node. Paths that survive has arrows with unfilled heads. The final surviving path is marked in red.

end in the ”00” state. However, a long codeword can enter the ”00” state through its way to the final node in its path.

The Viterbi algorithm can not directly be used in turbo codes since the algorithm output is hard decisions. However, after some adjustments the algorithm uses soft input and returns soft output, this algorithm is called SOVA and are discussed in section 4.5.2.

BCJR algorithm

The BCJR algorithm was discovered by Bahl, Cocke, Jelinek and Raviv and first presented in [5] in 1974. This algorithm is more complex than the previous Viterbi algorithm. However, in Turbo Codes the BCJR algorithm, also called Maximum A Posteriori Probability (MAP), is quite important, because this algorithm is able to yield the A Posteriori Probability (APP) for each decoded bit. The difference between the Viterbi algorithm and the BCJR algorithm is that the Viterbi algorithm finds the most probable information sequence that was sent, whilst the BCJR algorithm finds the most probable information bit to have been sent given the encoded sequence. When Turbo codes were first discovered in [10] in 1993, the decoding algorithm presented was a modified Bahl algorithm, which is exactly the same as a modified BCJR algorithm. This modified version had a

recursive character to make it practical for Turbo Codes, which will be explained in chapter 4.5.1. However, now the original version will be presented. The BCJR decoder receives a sequence with transmitted data r and the a priori log-likelihood ratios of the information bits $L_a(u_l)$, $l = 0, 1, \cdots, h-1$. h is the block length of the sequence u. For example, figure 2.4 shows a table of transition probabilities $P(r_l|v_l)$, where v_l denotes the transmitted symbol at index l and r_l is the received symbol. Using this data the algorithm calculates the a posteriori LLRs (APP LLRs) for each information bit

$$L(u_l) \equiv \ln \left[\frac{P(u_l = +1|r)}{P(u_l = -1|r)} \right]. \tag{3.29}$$

The decoder output is given by hard decisions of the resulting estimated values $\hat{u}$, where

$$\hat{u}_l = \begin{cases} +1 \text{ if } L(u_l) > 0 \\ -1 \text{ if } L(u_l) < 0 \end{cases} , l = 0, 1, \cdots, h-1. \tag{3.30}$$

Equation (3.29) can be reformulated after several steps[5] to

$$L(u_l) = \ln \left\{ \frac{\sum_{(s',s)\in S_l^+} p(s_l = s', s_{l+1} = s, r)}{\sum_{(s',s)\in S_l^-} p(s_l = s', s_{l+1} = s, r)} \right\}, \tag{3.31}$$

where p is the probability density function and s is some state in the trellis. The set S_l^+ denotes the set of all state pairs $s_l = s', s_{l+1} = s$ in the trellis, that correspond to the input bit $u_l = +1$ at time l, and the set S_l will be all the pairs corresponding with $u_l = -1$. Further the probability density function can be evaluated recursively by

$$p(s', s, r) = p(s', s, r_{t<l}, r_l, r_{t>l}), \tag{3.32}$$

where r_l represents the portion of the received sequence at time l, and $r_{t<l}$ and $r_{t>l}$ represents the portion received, before and after time l, respectively. Then after using Bayes' rule given in 3.3.1

$$\begin{aligned} p(s', s, r) &= p(r_{t>l}|s', s, r_{t<l}, r_l)p(s', s, r_{t<l}, r_l) \\ &= p(r_{t>l}|s', s, r_{t<l}, r_l)p(s, r_l|s', r_{t<l})p(s', r_{t<l}) \\ &= p(r_{t>l}|s)p(s, r_l|s')p(s', r_{t<l}). \end{aligned} \tag{3.33}$$

The last equality is then split in to three parts, as follows

$$\alpha_l(s') \equiv p(s', r_{t<l}) \tag{3.34}$$

$$\gamma_l(s', s) \equiv p(s, r_l|s') \tag{3.35}$$

$$\beta_{l+1}(s) \equiv p(r_{t>l}|s). \tag{3.36}$$

[5] Not reproduced here. Interested readers should read [5, 40]

Equation (3.33) can then be rewritten as

$$p(s', s, r) = \beta_{l+1}(s)\gamma_l(s', s)\alpha_l(s'). \tag{3.37}$$

The forward metric $\alpha_{l+1}(s)$ can be derived from equation (3.34), and this forward recursion is defined as

$$\alpha_{l+1}(s) = \sum_{s' \in \sigma_l} \gamma_l(s', s)\alpha_l(s'), \tag{3.38}$$

where σ_l is the set of all states at time l, and the backward metric for $\beta_l(s')$ can in a similar way be defined as

$$\beta_l(s') = \sum_{s \in \sigma_{l+1}} \gamma_l(s', s)\beta_{l+s}(s). \tag{3.39}$$

Then the branch metric $\gamma_l(s', s)$ be defined as

$$\gamma_l(s', s) = P(u_l)p(r_l|v_l). \tag{3.40}$$

which after some modification steps can be written as

$$\gamma_l(s', s) = P(u_l)e^{-E_s/N_0 \| r_l - v_l \|^2}, \tag{3.41}$$

where $\| r_l - v_l \|^2$ is the Euclidean distance between the received r_l and the transmitted v_l. E_s/N_0 is the Signal to Noise Ratio (SNR). The initial conditions for $\alpha_0(s)$ and $\beta_K(s)$, where $K = h + m$[6] is the length of the input sequence u, is defined by

$$\alpha_0(s) = \begin{cases} 1, s = 0 \\ 0, s \neq 0 \end{cases}, \tag{3.42}$$

and

$$\beta_K(s) = \begin{cases} 1, s = 0 \\ 0, s \neq 0 \end{cases}. \tag{3.43}$$

Finally the algorithm be given

BCJR Algorithm

1. Set the forward and backward metrics according to equation (3.42) and (3.43), respectively.
2. Use equation (3.40) to compute the branch metrics $\gamma_l^*(s', s)$, $l = 0, 1, \cdots, K - 1$.

[6] m was defined in section 3.4.2 as the memory of the encoder.

3. Use equation (3.38) to compute the forward metrics $\alpha^*_{l+1}(s)$, $l = 0, 1, \cdots, K-1$.

4. Use equation (3.39) to compute the backward metrics $\beta^*_l(s')$, $l = K-1, K-2 \cdots, 0$.

5. By using equation (3.31) compute the APP log-likelihood ratios $L(u_l)$ for $l = 0, 1, \cdots, h-1$.

6. Finally use equation (3.30) to compute the hard decisions $\hat{u}_l$, $l = 0, 1, \cdots, h-1$.

Now a codeword encoded with the encoder shown in figure 3.7 is sent over a channel, in this example a DMC[7], with the transition probabilities given in figure 3.6.

$v_l \setminus r_l$	0_1	0_2	1_1	1_2
0	0.1	0.4	0.2	0.3
1	0.3	0.2	0.4	0.1

Figure 3.6: The transition probabilities for the DMC illustrated in figure 2.3

Apply the BCJR algorithm on the received codeword

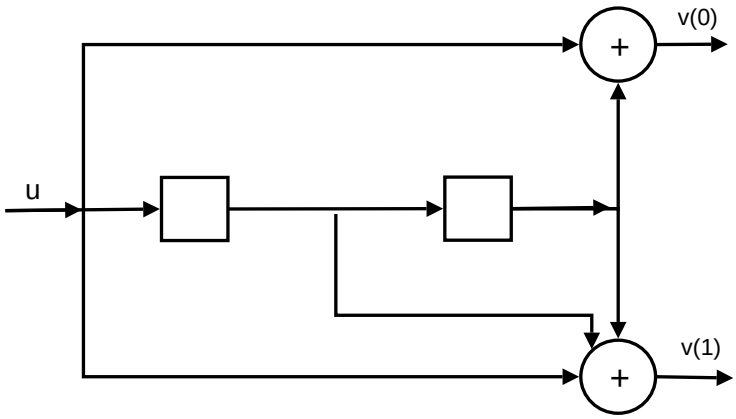

Figure 3.7: Convolutional encoder with memory m = 2.

$$r = (1_2 1_1, 1_1 0_1, 1_2 0_1, 1_2 1_1), \tag{3.44}$$

where the information bits have the probabilities

$$P(u_l = 0) = \begin{cases} 3/4, & l = 0, 1, \cdots, 3 \\ 1, & l = 4. \end{cases} \tag{3.45}$$

[7]Discrete Memoryless Channel. Explained in section 2.2.2

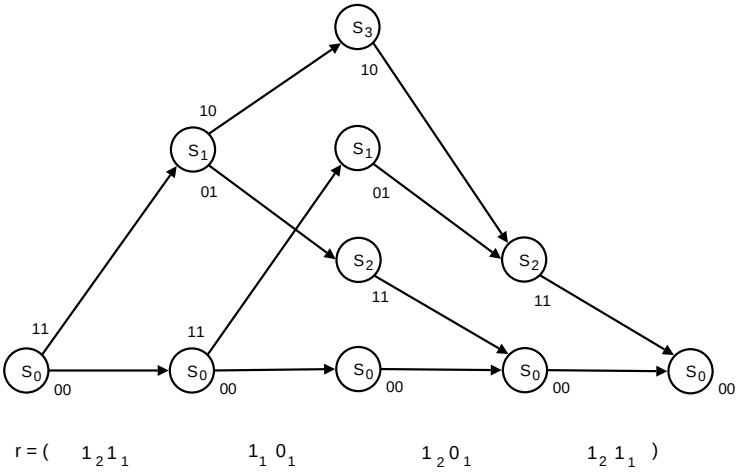

Figure 3.8: BCJR trellis with length K = 4.

Start at step 2. in the algorithm, since step 1. is an implementation step. Firstly we compute the branch metrics. Since the channel is a DMC, the branch metrics can be computed by using equation (3.40)

$$\begin{aligned}\gamma_0(S_0, S_0) &= P(u_0 = 0)P(1_2 1_1|00) = (3/4)P(1_2|0)P(1_1|0) \\ &= (3/4)(0.3)(0.2) = 0.045 \\ \gamma_0(S_0, S_1) &= P(u_0 = 1)P(1_2 1_1|11) = (3/4)P(1_2|1)P(1_1|1) \\ &= (1/4)(0.1)(0.4) = 0.01.\end{aligned} \tag{3.46}$$

Branch metrics for γ_1

$$\begin{aligned}\gamma_1(S_0, S_0) &= P(u_0 = 0)P(1_1 0_1|00) = (3/4)(0.2)(0.1) = 0.015 \\ \gamma_1(S_0, S_1) &= P(u_0 = 1)P(1_1 0_1|11) = (1/4)(0.4)(0.3) = 0.03 \\ \gamma_1(S_1, S_2) &= P(u_0 = 0)P(1_1 0_1|01) = (3/4)(0.2)(0.3) = 0.045 \\ \gamma_1(S_1, S_3) &= P(u_0 = 1)P(1_1 0_1|10) = (1/4)(0.4)(0.1) = 0.01.\end{aligned} \tag{3.47}$$

Branch metrics for γ_2

$$\begin{aligned}
\gamma_2(S_0, S_0) &= P(u_0 = 0)P(1_2 0_1|00) = (1)(0.3)(0.1) = 0.03 \\
\gamma_2(S_2, S_0) &= P(u_0 = 0)P(1_2 0_1|11) = (1)(0.1)(0.3) = 0.03 \\
\gamma_2(S_1, S_2) &= P(u_0 = 0)P(1_2 0_1|01) = (1)(0.3)(0.3) = 0.09 \\
\gamma_2(S_3, S_2) &= P(u_0 = 0)P(1_2 0_1|10) = (1)(0.1)(0.1) = 0.01.
\end{aligned} \tag{3.48}$$

Branch metrics for γ_3

$$\begin{aligned}
\gamma_3(S_0, S_0) &= P(u_0 = 0)P(1_2 1_1|00) = (1)(0.3)(0.2) = 0.06 \\
\gamma_3(S_2, S_0) &= P(u_0 = 0)P(1_2 1_1|11) = (1)(0.1)(0.4) = 0.04.
\end{aligned} \tag{3.49}$$

Further, the α_1 will be

$$\begin{aligned}
\alpha_1(S_0) &= \gamma_0(S_0, S_0)\alpha_0(S_0) = (0.045)(1) = 0.045 \\
\alpha_1(S_1) &= \gamma_0(S_0, S_1)\alpha_0(S_0) = (0.01)(1) = 0.01.
\end{aligned} \tag{3.50}$$

One can normalise the forward metrics in the following way to avoid numerical precision problems, since the metric results are relatively small

$$\begin{aligned}
A_1(S_0) &= \alpha_1(S_0)/(\alpha_1(S_0) + \alpha_1(S_1)) \\
&= (0.045)/((0.045) + (0.01)) = 0.8181 \\
A_1(S_1) &= \alpha_1(S_1)/(\alpha_1(S_0) + \alpha_1(S_1)) \\
&= (0.01)/((0.045) + (0.01)) = 0.1818.
\end{aligned} \tag{3.51}$$

The backward metrics

$$\begin{aligned}
\beta_3(S_0) &= \gamma_3(S_0, S_0)\beta_4(S_0) = (0.06)(1) = 0.06 \\
\beta_3(S_2) &= \gamma_3(S_2, S_0)\beta_4(S_0) = (0.04)(1) = 0.04,
\end{aligned} \tag{3.52}$$

can also be normalised

$$\begin{aligned}
B_3(S_0) &= \beta_3(S_0)/(\beta_3(S_0) + \beta_3(S_0)) \\
&= (0.06)/((0.06) + (0.04)) = 0.6 \\
B_3(S_2) &= \beta_3(S_2)/(\beta_3(S_2) + \beta_3(S_0)) \\
&= (0.04)/((0.06) + (0.04)) = 0.4.
\end{aligned} \tag{3.53}$$

The rest of the normalised metrics are shown in figure 3.8 The final

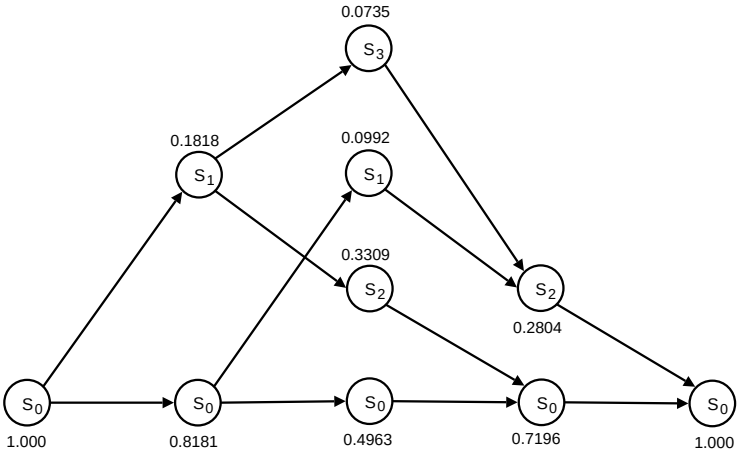

Figure 3.9: Trellis with the normalised forward metrics.

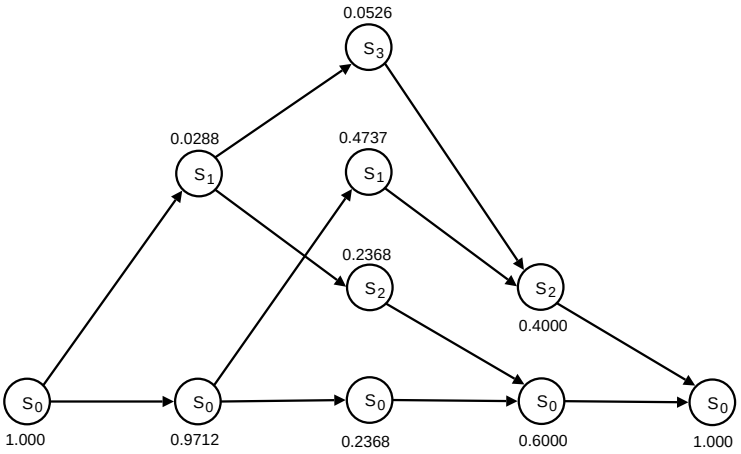

Figure 3.10: Trellis with the normalised backward metrics.

APP log-likelihood ratios will then be calculated by using equation (3.31), where the result is mapped according to equation (3.30)

$$\begin{aligned} L(u_0) &= \ln\left\{\frac{P(s_0 = S_0, s_1 = S_1, r)}{P(s_0 = S_0, s_1 = S_0, r)}\right\} \\ &= \ln\left\{\frac{B_1(S_1)\gamma_0(S_0, S_1)A_0(S_0)}{B_1(S_0)\gamma_0(S_0, S_0)A_0(S_0)}\right\} \\ &= \ln\left\{\frac{(0.0288)(0.01)(1)}{(0.9712)(0.045)(1)}\right\} = -5.022, \end{aligned} \tag{3.54}$$

$$
\begin{aligned}
L(u_1) &= \ln\left\{\frac{P(s_1 = S_0, s_2 = S_1, r) + P(s_1 = S_1, s_2 = S_3, r)}{P(s_1 = S_0, s_2 = S_0, r) + P(s_1 = S_1, s_2 = S_2, r)}\right\} \\
&= \ln\left\{\frac{B_2(S_1)\gamma_1(S_0, S_1)A_1(S_0) + B_2(S_3)\gamma_1(S_1, S_3)A_1(S_1)}{B_2(S_0)\gamma_1(S_0, S_0)A_1(S_0) + B_2(S_2)\gamma_1(S_1, S_2)A_1(S_1)}\right\} \\
&= \ln\left\{\frac{(0.4737)(0.03)(0.8181) + (0.0526)(0.01)(0.1818)}{(0.2368)(0.015)(0.8181) + (0.2368)(0.045)(0.1818)}\right\} \\
&= 0.884.
\end{aligned}
\tag{3.55}
$$

From figure 3.8 and the fact that the encoder is feedforward, the APP log-likelihood values to $L(u_2)$ and $L(u_3)$ can be ignored, since they are termination bits which are known to be zero. This results in the decoded message $\hat{u} = (\hat{u}_0, \hat{u}_1) = (0, 1)$.

Like the Viterbi algorithm the BCJR algorithm can not be used for turbo decoding, but some modifications makes it capable of decoding turbo codes. On of the methods of modifying this algorithm is explained in section 4.5.1.

Chapter 4

Turbo-codes

Coding a message can be done in several ways. Most methods add some sort of parity to the original codeword. The parity bits then serve to strengthen every bit in the codeword against bit errors. For example the codeword 1010 could be coded in the following simple way: the first and the second bits are added together mod two, and the third and fourth bits are added together. Then the resulting codeword will be: 101011. If a bit error occurred so that the receiver receives 101111 instead of the correct codeword, then the receiver can easily verify that an error has occurred.

However, this code is not able to correct the error. Therefore the receiver has to ask for retransmission which also has to be verified and this might be very time consuming, especially in a noisy environment. Thus, in particular for long distance transmissions like deep-space communication[1], error correcting codes are preferred.

4.1 Shannon limit

For years the Shannon limit was regarded as a theoretical limit that was impossible to reach. Nevertheless, when Turbo-codes together with LDPC[2] were discovered, they both came close in approaching the Shannon limit.

The Shannon limit was derived by Claude Shannon in 1948[12]. He discovered that there is a theoretical limit of maximum possible information transfer rate over a noisy channel with interference and data corruption. Thus, Shannon proved that there is a maximum

[1] Or in other time dependent data communication
[2] Low-Density Parity-Check codes [22, 40, 38]

amount of information that can be transmitted in a fixed number of bits. The theory does not say anything about how to construct these codes, it only tells us how good the best possible code can be. Therefore, given a noisy channel with information capacity, C, and information transmitted at a rate R, then as long $R < C$ there exists a coding method that allows the error at the receiver to be made arbitrarily small. Theoretically it is therefore possible to transmit information almost without error up to a limit of C bits per second. If the sender tries to send more information beyond the channel capacity, then the receiver will not receive any extra useful information.

The channel capacity, C, is expressed by the following equation:

$$C = B \log_2(1 + \frac{S}{N}) \tag{4.1}$$

B is the bandwidth of the channel, S is the total power over the bandwidth and N is the total noise power over the bandwidth.

4.2 Encoding

Turbo-codes can be coded in different ways, but can be summarised by figure 4.1. Here the scheme starts with the information, u, or message, that is to be sent. U is sent through encoder 1 and u is also interleaved[3], which often is denoted by π, and the result, u', is sent through encoder 2. Encoder 1 and encoder 2 are normally identical, the reason their output is different is totally dependent on the interleaving. The information, u or v_0, is together with the result from encoder 1, v_1, and the result from encoder 2, v_2, concatenated. Accordingly some encoders puncture[4] the result and the result or codeword can then be sent over a channel.

As mentioned earlier, turbo-codes can be coded in different ways. One of the methods is called convolutional coding and this method was described in the early days of turbo-codes[10].

4.2.1 Product-code

Turbo product codes (TPC) are normally built from smaller code word blocks. In section 3.4.1 is was mentioned that an example of such a code was the (16, 11) extended Hamming code given in [49].

[3] will be explained in more detail in 4.3
[4] will be explained in more detail in 4.4

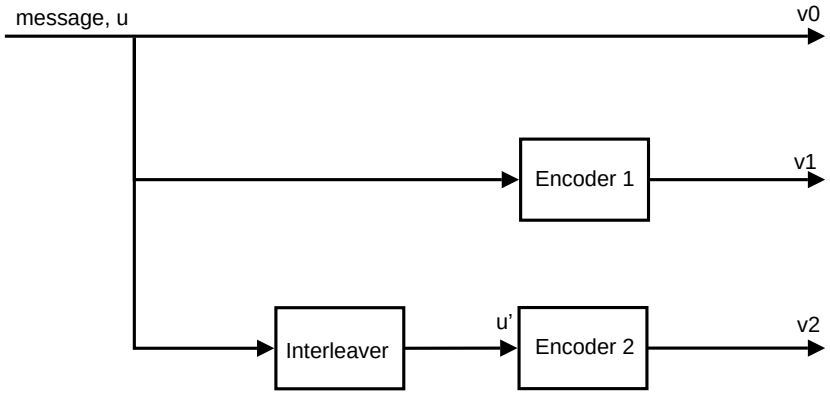

Figure 4.1: The encoding scheme of turbo-codes

I	I	P_H
I	I	P_H
P_V	P_V	P_{VH}

Figure 4.2: A turbo product code (TPC).

The (16, 11) denotes that the code takes 11 information bits and encodes this information by computing 5 parity bits, which are appended to the information bits. A TPC will encode a block of codewords by first computing the parity bits row by row and by then appending the parity bits to each row, also called horizontal encoding. Then the parity bits are computed column by column and are then appended to the column, which is called vertical encoding. These two blocks of parity bits are then used as input for the parity on the parity, that is parity bits depending on both vertical and horizontal parity bits. A small example is given in figure 4.2, where I denotes information bits and P denotes parity bits. This encoding scheme which is in two dimensions can also be extended to three dimensions [49]. Results in [49] state that a product code uses less energy and gives higher data rates than a rate R=1/2 convolutional code. Chapter 6 presents an example of a simple product-code, where the parity on parity bits are not computed.

4.2.2 Convolutional codes

Normally in Turbo Codes, two Recursive Systematic Convolutional (RSC) codes are used [10]. These two codes are concatenated and then sent over a channel. Figure 4.1 shows a code with rate $R = 1/3$. This can be constructed by concatenating two RSC codes with rate

$R = 1/2$, where the last encoder, which is interleaved, ignores the systematic bits. The two encoders can, for example, be like the one shown in figure 3.3, which is a rate $R = 1/3$ encoder. They are normally, for simplicity, the same encoder. However, this is not a necessity.

One important remark on RSC are that since they are recursive, the encoder will not return to the all zero state by adding a tail of all zero. Instead the current context of the encoder has to be appended in the tail. Figure 5.6 illustrates how this tail is appended by the dotted lines.

Later, in section 5.2 we will look into Nonsystematic (NSC) Turbo codes.

4.3 Interleaving

Interleaving is merely just a rearranging of the data, or to be more precise, a rearranging of the order in which the data is read. For example, as presented in 6.1, the four bits of data are arranged in a 2 x 2 matrix before the data is first encoded rowwise, and then encoded columnwise. This type of swapping is a simple form of interleaving and is called block interleaving.

4.3.1 Why interleave?

The main point of interleaving is to protect the data from burst errors. This can be explained by seeing the interleaving step as a temporal permutation of bits. If n errors occurs on n consecutive bits in an uninterleaved code segment, then the errors will be spread over the entire block on the interleaved code segment as illustrated in figure 4.3. So what the interleaver really does is to increase the free distance to the concatenated code [43]. The free distance in Turbo codes can be understood as the minimum Hamming distance of a code. Spectral thinning is a process that has been shown to reduce the number of low-weight codewords and has therefore an impact on the minimum distance of the code [47]. Different approaches to increase and find the free distance of Turbo codes are given in [11, 24, 23]. Different methods of interleaving will be the subject of section 5.1.

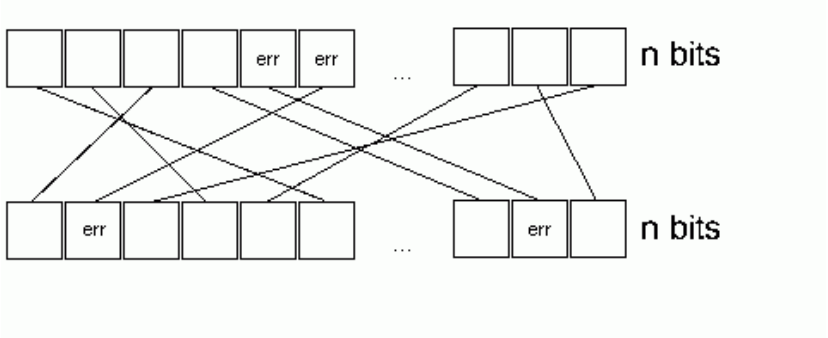

Figure 4.3: How interleaving disperses the errors burst.

4.4 Puncturing

Puncturing systematically removes some of the parity bits after encoding. This is, for example, applied to constituent codes. Both encoders might want to send the information. This repetition is inefficient so one encoder may puncture its information bits. The information bits in say, the second encoder are therefore ignored, and the rate of the code is increased.

For example, if the output v_0 from the encoder, illustrated in figure 4.1, is "abc", v_1 is "abcde" and v_2 is "cbafg". Here "abc" is sent three times, which is a waste of energy and time. Puncturing should therefore be used and the result might then be "abcdefg".

A example of puncturing patterns is given in table 5.1.

4.5 Decoding

The decoding schemes of Turbo-codes can, in short, be described by figure 4.4, and the reader should recall the coding scheme shown in figure 4.1. In the figure the data is derived in different parts, v_0' is the message part, v_1' is the data from the first encoder and v_2' is the data from the second encoder. Variables are marked with a ' since the data sent from the encoder is not necessarily the same as the received data due to introduction of channel errors. The two decoders are normally similar, however since the encoded data has been interleaved, the data between the two encoders has to be repeatedly deinterleaved and interleaved to match its associated encoder. Accordingly the output from the first decoder is sent to the second decoder, which then uses this data in it's estimate of the

codeword. This data is then sent back to the first encoder. This iterative method normally runs at least six times before the estimate is so good that the decoder is certain to have found the correct codeword, or in other words, the result converges to a codeword. However the method can also experience divergence, and will then not terminate if there is no limit to the maximum number of iterations allowed. Results that are sent between the decoders are called soft decisions, and are real number estimates between, for example, $\{-7, +7\}$. Depending on the implementation these numbers can be $\{-\infty, +\infty\}$. Returning to the example, a soft decision of -7 will be interpreted as a certain 0, and a soft decision of +7 will be a certain 1. A soft decision of -1 will most probably be a 0, but this is uncertain and more iterations will be preferable. If the soft decision is 0, the number can be either 0 or 1 and further iterations are required. When the final decision is made the result is called a hard decision. It should also be noticed that if the data sent from the encoder was punctured, then the receiver has to take this into account. This is normally done in DEMUX operations before the decoding takes place. Since puncturing systematically removes bits, the DEMUX systematically inserts bits.

There are several ways to decode a turbo code. [70] describes four different algorithm, namely the MAP, Log-MAP, Max-Log-MAP and the SOVA algorithm. A modified SOVA algorithm described in [20] is proposed to be equivalent to the Max-Log-MAP algorithm. In this thesis only the Log-MAP and the SOVA algorithm are briefly described.

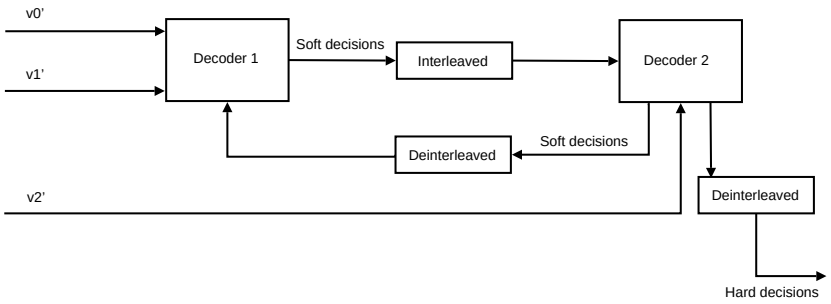

Figure 4.4: The decoding scheme of turbo-codes

4.5.1 Log-MAP Algorithm

Logarithmic Maximum A-posteriori Probability (Log-MAP), is a modified version of the BCJR algorithm described in section 3.4.3. Log-MAP decoding is usually used with systematic encoders since the first estimate of the real value u_l is the received systematic data v_0.

The Log-MAP algorithm was first introduced in [51] as a modified version of the Max-Log-MAP algorithm. Further, the Max-Log-MAP algorithm is a simplified version of the MAP algorithm [70]. In the MAP algorithm the a posteriori LLRs are computed by using equation (3.31), where the probabilities are replaced with equation (3.37) together with the equations (3.38), (3.39) and (3.40).

Simplifying these equations is then done by transferring them into the log arithmetic domain and using the approximation

$$\ln\left(\sum_i e^{x_i}\right) = \max_i(x_i) \tag{4.2}$$

$\max_i(x_i)$ is the maximum value of x_i. Then, $A_l(s)$, $B_l(s)$ and $\Gamma_l(s', s)$ are, after using the definitions given in equations (3.38), (3.39) and (3.40), defined by

$$A_{l+1}(s) \triangleq \ln(\alpha_{l+1}(s)) \approx \max_{s'}(A_l(s') + \Gamma_l(s', s)) \tag{4.3}$$

$$B_l(s') \triangleq \ln(\beta_l(s')) \approx \max_{s}(B_{l+1}(s) + \Gamma_l(s', s)) \tag{4.4}$$

and

$$\Gamma_l(s', s) \triangleq \ln(\gamma_l(s', s)) = \frac{u_l L_a(u_l)}{2} + \frac{L_c r_l \cdot v_l}{2}, \tag{4.5}$$

where L_a is the a priori log-likelihood values to the sent information bits. v_l is the sent bit, r_l is the received bit and L_c is the channel reliability factor. The computation of these equations can be found in [70, 51, 58, 40].

Since the approximation in equation (4.2) yields an inferior soft-output compared to the MAP algorithm, the Log-MAP algorithm uses the Jacobian logarithm to fix this problem [70, 51].

$$\ln(e^{\delta_1} + e^{\delta_2}) = \max(\delta_1, \delta_2) + \ln(1 + e^{-|\delta_2 - \delta_1|}) \tag{4.6}$$

When equation (4.3) and (4.4) use equation (4.6) instead of (4.2) the algorithm is called Log-MAP.

4.5.2 Soft-Output Viterbi Algorithm (SOVA)

Soft Decision Viterbi Decoding, or SOVA for short, was first presented in [29] and is a decoding technique that is similar to the Viterbi algorithm given in section 3.4.3. There are two differences between the Viterbi algorithm and the SOVA [70]. First, the path metrics are modified to use a priori information when deciding the path through the trellis that is most likely. Second, the soft output has reliability information about the decoded output [40]. This reliability information is the a posteriori log-likelihood ratios. These kind of decoders are also called Soft-In Soft-Out (SISO) decoders.

Consider a trellis, similar to the one shown in figure 3.4. As said earlier, every path through this trellis represents different codewords. All these paths can be divided into states that are connected by an edge to one or two other states in the same path. Each of these states has a metric that denotes the "probability" of the surviving path going through their state. This metric $M(s_l)$ depends on the metric to the previous state $M(s'_{l-1})$ in the path and the metric to the edge, $\gamma_l(s', s)$[5] between them, like in the original Viterbi algorithm. The metric is now defined by

$$M(s_l) \triangleq M(s'_{l-1}) + \ln(\gamma_l(s', s)) \tag{4.7}$$

In a binary trellis all states will have two incoming edges. Since the trellis starts and ends in the all zero state the starting states will have one, or none incoming edges, because the rest of the states are unreachable and therefore ignored. Anyway, when two paths enter a state, their metrics are computed, and are then compared, and the largest metric $M(s_l)$ is selected, while the other $M(\hat{s}_l)$ is discarded. The difference between these two metrics is

$$\Delta_l = M(s_l) - M(\hat{s}_l) \geq 0 \tag{4.8}$$

which is the log-likelihood ratio of the selected metric being the correct decision [70]. It is shown in [27] that the log-likelihood ratio of the information bit u_l given the received bit r_l can be approximated by

$$L(u_l|r_l) \approx u_l \cdot \min_{i=l,\cdots,l+\delta}(\Delta_i) \tag{4.9}$$

where δ is a number of states after l. The SOVA algorithm follows the same steps as the algorithm given in section 3.4.3. However, the soft input and output is computed by the equations above to find the most likely path. When the most likely path has been found the

[5] Defined in equation (3.35)

hard decisions are computed by using equation (3.30).

[51] showed that the SOVA algorithm is half as complex as the Max-Log-MAP algorithm, but it is also not nearly as accurate.

Chapter 5

New Research in Turbo Codes

Turbo codes were first presented in [10] in 1993. Since then research into Turbo codes has grown drastically. The modified BAHL[1] algorithm was the first decoding algorithm presented for Turbo codes. Later it was discovered that the SOVA algorithm, see chapter 4.5.2, which is a modified version of the Viterbi algorithm, could be used in the decoding of Turbo codes. The chapter will briefly present some of the research done on turbo codes the last five years. It should be noted that some sections do not contain new research but explain some of the foundations to the newer research.

The most important current applications of Turbo codes are for space communication and mobile communication. Turbo codes are also used in a standard for Digital Video Broadcasting (DVB) [16].

The last section in this chapter discusses the use of turbo codes in a new type of correlation attacks. This knowledge is older than five years, but is included to show that turbo codes are used in different contexts.

5.1 Interleaving

As stated earlier, interleaving has a large influence on the free distance[2] in a turbo code [66, 43, 71], which lowers the error floor of the code [21]. When the BER to a turbo code is plotted depending on the noise, the resulting curve is shaped like a waterfall that

[1] The BAHL algorithm, or the BCJR algorithm was explained in section 3.4.3
[2] Explained in section 3.1.

suddenly flattens. This part of the curve is called the error floor. A BER curve for a turbo code is shown in figure 5.1. The error floor phenomenom is due to the weight distribution of turbo codes [40], or more precisely, since it is a fact that turbo codes normally do not have large minimum distances the performance curve flattens out at BERs below 10^{-5}. Because of this it follows that lowering the error floor results in better codes, which in some cases may result in faster convergence when decoding [21]. One effective way of

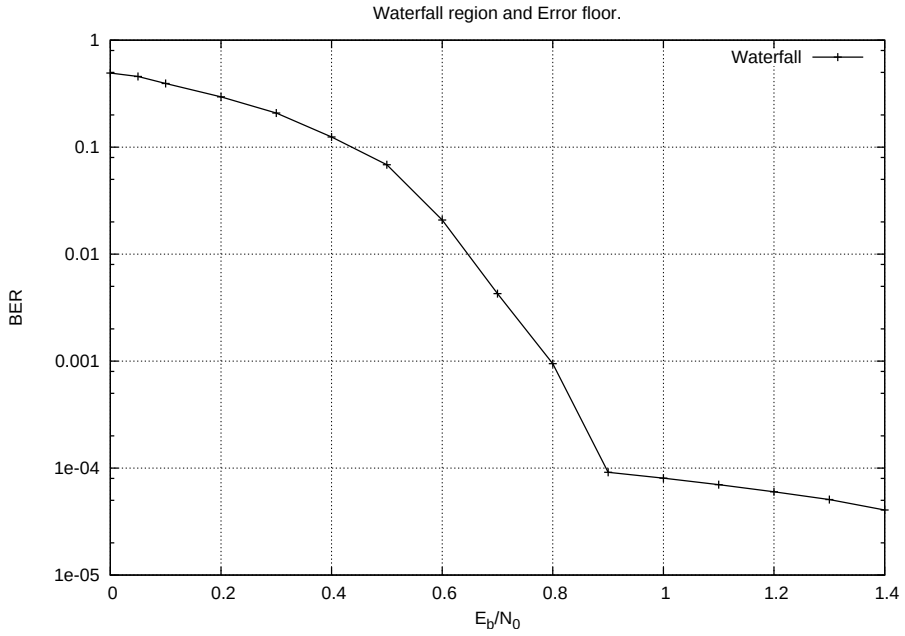

Figure 5.1: A BER curve showing the waterfall region and the error floor.

lowering the error floor is to use appropriate interleavers. This is of course the reason for the large amount of research done in this area. Interleavers can be divided into two main classes, namely, random interleavers and deterministic interleavers [65]. Random interleavers often permute the information bits pseudorandomly, which means that they are not random, they just appear to be. However, they may contain some elements of randomness, and are therefore called random interleavers. An example of an improved random interleaver is the S-random interleaver discussed in [21, 14]. Deterministic interleavers on the other hand permute the information bits in an arranged manner. These interleavers can actually perform better than random ones when the sequence length is short [65]. Block interleavers are examples of deterministic interleavers.

5.1.1 S-Random interleavers

Semi-random interleavers, or S-random interleavers proposed in [15] in 1995, are pseudorandom interleavers with a restriction on the randomness. This restriction says that no two input positions within a distance S, can be permuted to two output positions within a distance S [65]. In this way the code is better protected against short burst errors [3]. Since these successive bits in the original sequence are spread over a larger output sequence by the interleaver, they will not be mapped to short burst errors in the interleaved sequence.

Further improvements of the S-random interleaver were proposed in [21]. These improvements were the two-step S-random interleavers which when used with short sequence size perform better than the S-random interleaver. The two-step S-random interleaver has the same constraint as the original S-random interleaver, namely that, for some randomly selected position, i cannot be mapped to $f(i)$ if there exists a j such that the following conditions are met

$$0 < i - j \leq S_1, \quad |f(i) - f(j)| \leq S_2. \tag{5.1}$$

In addition the two-step interleaver has more constraints. Some randomly selected position i cannot be mapped to $f(i)$ if there exists $j, k, l < i$ such that the following conditions are met

$$\begin{aligned} 0 < i - j \leq T_1, \quad & |f(i) - f(k)| \leq T_2, \\ 0 < |k - l| \leq T_1, \quad & |f(j) - f(l)| \leq T_2. \end{aligned} \tag{5.2}$$

This constraint protects against two independent burst errors. Further, this method can be used for three independent burst errors and so on, however, this leads to increased complexity [21]. It should also be noted that the given method can not guarantee the existence of an interleaver with the given constraints.

5.1.2 Quadratic interleavers

Quadratic interleavers are a type of deterministic interleaver [65, 57] proposed in 2000 in [66]. They have a very simple representation based on quadratic congruence such that it can be shown for $N = 2^n$ that

$$c_i = \frac{k \cdot i(i+1)}{2} (\text{mod } N) \tag{5.3}$$

is a permutation, where $i \in \{0, 1 \ldots, N-1\}$ and k an odd constant. The interleaver mapping is in [66] given as an algorithm

[3] Defined in section 3.1.

1. Compute $c_0 = 0$
2. Compute $c_i \equiv c_{m-1} + k \cdot i (\text{mod } N)$, for $i \in \{0, 1 \ldots, N-1\}$, and k is an odd constant.

For example, an interleaver with $N = 8$ and $k = 1$ will have an unique 8-cycle that can be computed by the given algorithm, or by using equation (5.3). This cycle will in either case be $(0, 1, 3, 6, 2, 7, 5, 4)$. Then this cycle can be used as input in the index mapping function

$$D_{N:CN} : c_i \mapsto c_{i+1} (\text{mod } N), i \in \{0, 1 \ldots, N-1\}\,. \tag{5.4}$$

The interleaver, or permutation vector then becomes $[1, 3, 7, 6, 0, 4, 2, 5]$.

An extension of this scheme was also proposed in [66], where the result of equation (5.4) is shifted cyclically by h units, then each element is added by a constant $v (\text{mod } N)$.

The performance of these deterministic interleavers is better than random interleavers in the error floor region [66], and the performance in the waterfall region is similar to the performance of the random interleavers.

5.1.3 Permutation polynomials

Another type of deterministic interleaver is based on permutation polynomials over the ring of integers modulo N. A ring of integers modulo N is denoted $\mathbb{Z}_N$ [65]. The permutation polynomial has to be of the form

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m \tag{5.5}$$

where m is a small positive integer, and $a_i > 0$, for all $i \in \{0, 1, \ldots, m\}$. For now we only consider the case $N = 2^n$. This polynomial must satisfy the following three conditions to be a permutation polynomial over the integer ring $\mathbb{Z}_{2^n}$

1. a_1 is odd,
2. $a_2 + a_4 + a_6 + \cdots$ is even,
3. $a_3 + a_5 + a_7 + \cdots$ is even.

For example, the chosen polynomial $P(x) = 2x^3 + 2x^2 + x + 1$ satisfies the conditions given above, and is therefore a permutation polynomial. The derivative of the polynomial $P(x)$ is defined as

$$P'(x) = a_1 + 2a_2 x + 3a_3 x^2 + \cdots + m a_m x^{m-1} \tag{5.6}$$

For $n = 3$ we get $N = 2^3 = 8$, that means that a sequence $\{0, 1, 2, \ldots, 7\}$ should be permuted by the chosen polynomial $P(x)$. The permutation will then become as shown in equation (5.7)

$$\begin{array}{lll} P(0) = & 1 & \mod 8 = 1 \\ P(1) = & 6 & \mod 8 = 6 \\ P(2) = & 27 & \mod 8 = 3 \\ P(3) = & 76 & \mod 8 = 4 \\ P(4) = & 165 & \mod 8 = 5 \\ P(5) = & 306 & \mod 8 = 2 \\ P(6) = & 511 & \mod 8 = 7 \\ P(7) = & 792 & \mod 8 = 0 \end{array} \tag{5.7}$$

This permutation can, for example, be used as an interleaver function in a small turbo code. In [65] this method is, with some additional constraints, extended for $N = p^n$, where p is any prime number.

Finding good polynomials

Different permutation polynomials lead to different performance of the interleaver. An interesting task is to find the best permutation polynomial for a given code. In this task there are aspects that have to be considered. First of all the interleaver functions constructed by permutation polynomials can be seen as an arithmetic operation rather than a lookup table, which normally is the case in a random interleaver. Taking this into account the permutation polynomial should have the lowest possible complexity [65]. Different permutation polynomials also have different free distance. As mentioned earlier, the free distance has a large impact on lowering the error floor. Finding this free distance, however, can be difficult when using interleavers constructed by permutation polynomials. Therefore, the effective free distance, d_{ef}, which is defined as the minimum distance associated with an input error event of weight 2, used as one of the design criteria to find good interleavers [65].

Interleavers with small effective free distance are usually associated with bad performance, so they can be ruled out when searching for good interleavers. This does not mean that interleavers with large effective free distance can guarantee good performance [65]. A short error event of weight 2 should not be mapped to another short error event of weight 2 in the interleaved code. Therefore, similar to the constraint in the S-random interleaver there should be restrictions on the mapping in the interleaver to avoid this.

Given two bit positions $(x, x+t)$, for some t, the two will be interleaved by the permutation polynomial to $\pi(x)$ and $\pi(x+t)$ respectively. The distance between them is given by t. After interleaving, however, the distance is denoted by

$$\Delta(x,t) = \pi(x+t) - \pi(x) \mod N. \tag{5.8}$$

For simplicity we restrict our search of good permutation polynomials to polynomials of second degree[4]. A polynomial of the form $P(x) = bx^2 + ax$ is a permutation polynomial over $\mathbb{Z}_{p^n}$ if and only if $a \neq 0$ and $b = 0 \mod p$, for some prime number p, and satisfies the condition that $P'(x) \neq 0 \mod p$ for all integers $x \in \mathbb{Z}_{p^n}$. When using a systematic recursive convolutional encoder, a cycle of length τ is defined as the cycle of the output of the encoded input sequence $[1,0,0,0,\ldots]$. If this output then, for example, is $[1,1,0,1,1,0,1,1,0,1,\ldots]$, the cycle $[1,0,1]$ has a cycle length of $\tau = 3$ [65].

Let $t+1$ be the length of the error event with weight 2 in the first encoder, where t is a small multiple of the cycle with length τ. The order of t is denoted o_t. The length of the error event in the second encoder is then given by

$$\Delta(x,t) = P(x+t) - P(x) = 2btx + bt^2 + at, \tag{5.9}$$

where the coefficient of x is $c_1 = 2bt$. The property that if $z = xy$ then the order is $o_z = o_x + o_y$ gives that the order of the coefficient is given by $o_{c_1} = o_2 + o_b + o_t$. The order of N, where $N = \prod_{i=1}^{m} p_i^{n_{p_i}}$ is in [65] defined to be the vector $o_N = [n_{p_1}, n_{p_2}, \ldots, n_{p_m}]$. The distance of $\Delta(x,t)$ to zero is expressed by

$$\begin{aligned} s &= \pm\Delta(x,t) \mod p_N^{o_{c_1}} \\ &= \pm(bt^2 + at) \mod p_N^{o_{c_1}}. \end{aligned} \tag{5.10}$$

For a code with a, b, and τ the effective free distance can be computed from [65]

$$L_{a,b,\tau} = \min(|t| + |s|). \tag{5.11}$$

When selecting a and b the best approach described in [65] is for a given τ, to fix o_b and calculate equation (5.11), and then select good parameters.

[4]The constant term given as a_0 in equation (5.5) can be ignored since it only causes a cyclic shift to the permuted values [57].

Quadratic Inverses

Decoding turbo codes that have been interleaved with permutation polynomials can be done in the same way that other turbo codes are decoded. However, the procedure of deinterleaving can be a demanding task, and therefore the idea of using quadratic permutation polynomials with an "inverse" has been proposed [57].

A permutation polynomial $H(x) = h_1 x + h_2 x^2 \mod p$, where $p = 2$ and $h_1 + h_2$ is odd, is a quadratic permutation polynomial [57]. If a quadratic permutation polynomial is of the form $F(x) = f_1 x + f_2 x^2 \mod N$ there exists, according to [57], at least one quadratic polynomial $G(x) = g_1 x + g_2 x^2 \mod N$ that inverts $F(x)$ at $x = 0, 1, 2$. This polynomial is found if N is odd, by solving the linear congruences

$$g_2(f_1 + f_2)(f_1 + 2f_2)(f_1 + 3f_2) \equiv -f_2 \mod N. \tag{5.12a}$$

$$g_1(f_1 + f_2) + g_2(f_1 + f_2)^2 \equiv 1 \mod N. \tag{5.12b}$$

If, on the other hand, N is even there are two quadratic polynomials $G_1(x) = g_{1,1}x + g_{1,2}x^2 \mod N, G_2(x) = g_{2,1}x0g_{2,2}x^2 \mod N$, which can be found by solving the linear congruences

$$g_{1,2}(f_1 + f_2)(f_1 + 2f_2)(f_1 + 3f_2) \equiv -f_2 \mod \frac{N}{2}. \tag{5.13a}$$

$$g_{1,1}(f_1 + f_2) + g_{1,2}(f_1 + f_2)^2 \equiv 1 \mod N. \tag{5.13b}$$

When $(g_{1,1}, g_{1,2})$ is found, $(g_{2,1}, g_{2,2})$ can be computed by $g_{2,1} \equiv g_{1,1} + \frac{N}{2} \mod N$ and $g_{2,2} \equiv g_{1,2} + \frac{N}{2} \mod N$ [57]. An important remark is that $G(x)$ is not necessarily an inverse polynomial of $F(x)$. It is only guaranteed that $G(x)$ inverts $F(x)$ at the points $x = 0, 1, 2$.

Comparing results

In [65] the results of comparing S-random interleavers, quadratic interleavers and permutation polynomial-based interleavers is presented. Permutation polynomial-based interleavers performed better than quadratic interleavers and S-random interleavers for both bit-error rate and frame-error rate. However, for very long frame sizes the S-random interleavers perform better than the permutation polynomial-based interleavers.

5.1.4 Hamiltonian graphs

A Hamiltonian graph is a set of vertices connected in a graph, where there is a path which visits each vertex exactly once. For example,

a circle, a square, and a triangle will all be Hamiltonian graphs. A Hamiltonian cycle is the path of the Hamiltonian graph. Finding Hamiltonian cycles from a given graph is a NP-complete problem. However interleavers for a given a Hamiltonian cycle can, according to [43], be constructed from 3-regular Hamiltonian graphs, like the one shown in figure 5.2. That a graph is 3-regular means that every vertex has three undirected edges. In the case of a 3-regular Hamiltonian graph, two of the edges will naturally be part of the Hamiltonian cycle. The third edge, that is not in the Hamiltonian cycle, is the interesting one.

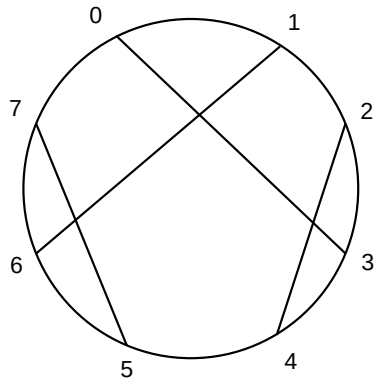

Figure 5.2: An example of a 3-regular Hamiltonian Graph with eight vertices.

If two vertices, i and j, have an edge between them, it is given that $j \neq i \pm 1$, since this edge $e(i, i \pm 1)$ is already on the Hamiltonian graph. Since the graph is 3-regular Hamiltonian, neither of i or j has any other non-Hamiltonian edges. That means that if these third edges are used to construct an interleaver π they will be one-to-one edges, $\pi(i) = j$ and $\pi(j) = i$. It should be noted that in a 3-regular graph the number of vertices is always even, so that only interleavers with even size can be constructed by using 3-regular Hamiltonian graphs.

The construction of these third edges, can be explained by seeing the cycle of vertices as a wheel [43, 50]. A spoke in this wheel represents the third edge to the vertex i, where i is some vertex on the cycle with N vertices. All the spokes can be represented by a vector $(c_0, \ldots, c_{s-1})$. The i vertex is then connected with vertex $j = (i + c_{i \mod s})$. Hence, the graph will have s spokes, and therefore for a spoke vector to be valid [50] the following equations have to be satisfied

$$N \equiv 0 \mod s \tag{5.14}$$

and

$$c_i = N - c_{(i+c_i) \mod s} \quad \forall \quad 0 \leq i \leq s-1 \tag{5.15}$$

The interleaver function will therefore be given by

$$\pi(i) = (i + c_{i \mod s}) \mod N, \quad \text{for } 0 \leq i \leq N-1 \tag{5.16}$$

According to [43], these interleavers have better BER performance than quadratic [57] and S-random [1, 14] interleavers. However, the construction of these interleavers can involve a search for valid spoke vectors in a space of $O(N^{\frac{s}{2}})$ vectors.

5.1.5 Bit-Interleaved Turbo-Coded Modulation

Bit-Interleaved Turbo-Coded Modulation (BITCM) is a bandwidth and power efficient coding technique. It is based on serial concatenation of binary turbo coding, bit interleaving, and high order modulation [26, 56]. This technique is capable of achieving BER performance close to the capacity limit [25]. The idea of BITCM is basically to map the encoded bits of a standard turbo code to a certain signal constellation[5] [25, 26]. A QPSK signal constellation is shown in figure 2.7. Most of the research on BITCM systems has been on Quadrature Amplitude Modulation (QAM) signal constellations [26]. Figure 5.3 shows the constellation diagram of a 16-QAM [67]. To optimise BITCM performance, it is necessary to employ a

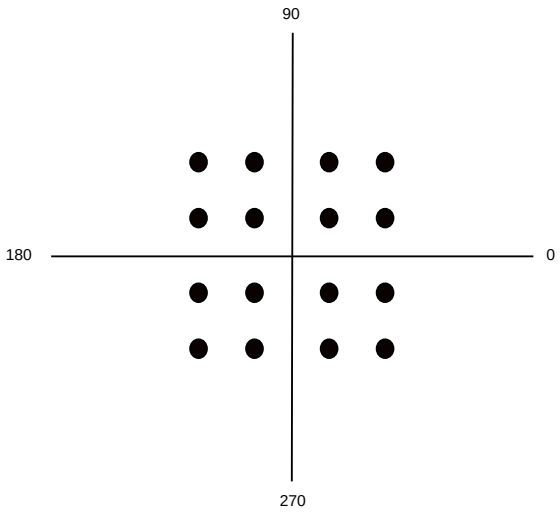

Figure 5.3: A 16-QAM signal constellation.

modulation scheme that has the smallest BER at low SNR. The

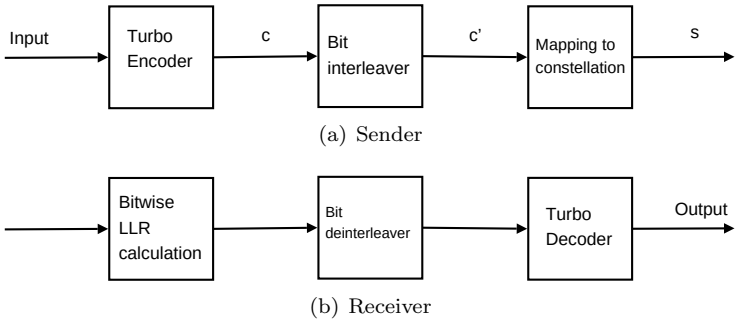

Figure 5.4: Diagram of the transmission scheme for a BITCM.

transmission scheme for a BITCM is shown in figure 5.4 [56].

In [56] a greedy algorithm is introduced to lower the error floor of BITCM on the AWGN channel. Given a set L of all codewords of weight less than a given threshold τ computed by the improved GPB algorithm given in [54]. Then, for any codeword $c = (c_0, \cdots, c_{n-1}) \in L$, let $X(c)$ denote the support, that is, the index set of nonzero coordinates. The support of L is then given by $X(L)$. A subset H of X of minimum cardinality, such that $|X(c) \cap H| \geq l(c), 0 \leq l(c) \leq |X(c)|$, for all $c \in L$ is then called a minimum hitting set [56]. The hitting distribution is then prescribed by the values $l(c)$, where $c \in L$. Now, choose $l(c) = l'(w(c))$, which means that the value only depends on the Hamming weight of c. Then, let $N(p) = |\{c \in L : p \in X(c)\}|$ for all $p \in X$.

The greedy algorithm given below will then construct a hitting set H. This hitting set has a target hitting distribution $\{l(c) : c \in L\}$.

Algorithm 1 Greedy Hitting Set $(L, \{l(c) : c \in L\})$ [56]

Compute $X(L)$ and $N(p)$ for all $p \in X$, and the set $H = \emptyset$.
while $|X(c) \cap H| < l(c)$ for all $c \in L$ and $X \neq \emptyset$ **do**
  Set $p_{\max} = \arg\max_{p \in X} N(p)$.
  if $\exists c \in L : p_{\max} \in X(c)$ and $|X(c) \cap H| < l(c)$ **then**
    Set $H = H \cup \{p_{\max}\}$.
  end if
  Remove $p_{\max}$ from X.
end while

[5]The reader should recall section 2.3.

Performance results

The results of using this algorithm compared to randomly generated bit interleavers has shown equal or better performance in the waterfall region, and a large improvement of the performance in the error floor region. By using maximum likelihood decoding the given algorithm was able to design BITCM schemes with a frame error rate of 10^{-12} and 10^{-17} at 2.6 and 3.8 dB from the unconstrained channel capacity [56].

Further work

The study proposed in [56] has mainly considered an approach with respect to the lower dimensions of X. It is therefore stated in [56] that further research should investigate a higher dimensional approach. Most research on BITCM has mainly concentrated on different QAM signal constellations. Further research should consider using more energy efficient signal constellations [56].

5.1.6 Remarks on interleavers

There are discovered many different methods of interleaving. This section has discussed a few of them, which all have had the goal of improving the turbo code. Many other interleaving schemes should have been discussed, however, because of limited time these schemes will not be mentioned in this thesis. Another interesting open problem is comparing the performance of these interleaver described in this section.

5.2 Nonsystematic Turbo Codes

Turbo codes were originally constructed by two Recursive Systematic Convolutional (RSC) encoders. However, other construction methods have also been used. In [6] turbo codes are constructed by two recursive NonSystematic Convolutional (NSC) encoders, which have some interesting results. In [9] it was already mentioned that NSC turbo codes have the same free distance as RSC, and that the RSC codes exhibit better performance at low Signal-to-Noise Ratio (SNR). The construction of a NSC is done by two nonsystematic feedback convolutional encoders. The two encoders, shown in figure 5.5 as G_0 and G_1, code the information data u, and simultaneously G_2 codes the interleaved information data u'. If G_2 is either the same

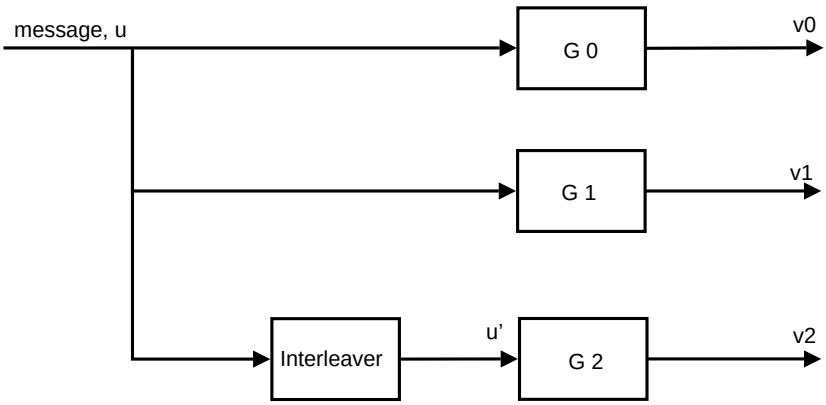

Figure 5.5: A NonSystematic Convolutional (NSC) encoder.

encoder as G_0 or G_1, the NSC is symmetric and if G_2 is neither, the NSC is asymmetric.

5.2.1 Decoding

Since the information bits, which in figure 4.1 were denoted by v_0, are not part of a NSC, the receiver has no measurements of these information bits, or systematic bits. Therefore, the decoder assumes that the APP LLRs to the information bits are equal to zero. Then some SISO decoding scheme, for example Log-MAP or SOVA which was described briefly in section 4.5, decodes the code in the same way as a systematic code.

5.2.2 Good Nonsystematic Turbo Codes

For a convolutional encoder with rate $R = k/n$, the minimum weight of the ith parity output sequence will be in the range $(1 \leq i \leq n-k)$ for a systematic encoder, and in $(1 \leq i \leq n)$ for a nonsystematic encoder when the input sequence is of weight 2 ($d_2(i)$) [6]. The minimum distances of the systematic (d_2^s) and nonsystematic (d_2^{ns}) encoder for a weight 2 input can then be written as

$$d_2^s = 2 + \sum_{i=1}^{n-k} d_2(i), \tag{5.17}$$

and

$$d_2^{\text{ns}} = \sum_{i=1}^{n} d_2(i), \tag{5.18}$$

respectively. Then by assuming the worst case[6] the effective free distance for a systematic and a nonsystematic turbo code is then defined as

$$d_{\text{eff}}^{s} = d_2^{s} + d_2^{(2)}, \tag{5.19}$$

and

$$d_{\text{eff}}^{\text{ns}} = d_2^{\text{ns}} + d_2^{(2)}, \tag{5.20}$$

respectively. Finding good nonsystematic turbo codes involves, according to [6], finding constituent encoders with large d_2^{ns}, $d_2^{(2)}$ and low convergence thresholds. Some results of different encoders are presented in [6], that are not reproduced here.

Some nonsystematic encoders are catastrophic, since they do not provide any a posteriori extrinsic information. However, some of these catastrophic encoders can, after using doping, which is replacing some of the nonsystematic bits with systematic bits, converge at low SNRs. But doped encoders seem to have a worse performance in the error floor region [6].

5.2.3 Properties

Nonsystematic turbo codes have, according to [6], larger values of effective free distance compared to systematic turbo codes. This is the reason for their good performance in the error floor region of the BER curve. So the major benefit of nonsystematic turbo codes is their improved performance in the error floor region.

5.3 Turbo codes in 3G

Turbo codes were proposed in [39] as the error-correction scheme for the third generation mobile technology, more commonly known as 3G. 3G is divided into two standards [17], where one of them is the Universal Terrestrial Radio Access (UTRA), which is based on the Wideband Code-Division Multiple Access (WCDMA). The other standard is called CDMA2000 [39]. The coding scheme used in the CDMA2000 standard is shown in figure 5.6, which has a data service above 14.4 kbps [39]. This coding scheme can with different puncturing patterns achieve different coding rates, namely, 1/2, 1/3 and 1/4. The different patterns are shown in table 5.1 [3, 39]. In both CDMA2000 and UTRA/WCDMA the interleaver is divided

[6] When a input of weight 2 is encoded to give the output of weight d_2^{s} or d_2^{ns} is interleaved to an input of weight 2 for the second encoder which gives the an output of weight $d_2^{(2)}$.

RATE	1/2	1/3	1/4
X(t)	11	11	11
$Y_0(t)$	10	11	11
$Y_1(t)$	00	00	10
X'(t)	00	00	00
$Y'_0(t)$	01	11	01
$Y'_1(t)$	00	00	11

Table 5.1: Puncturing patterns for different code rates [39].

into a small number of "mother interleavers" that use pruning to skip unnecessary indexes. For example, if the mother interleaver is the permutation $(4, 6, 1, 7, 3, 0, 2, 5)$, that has length eight, then a "child interleaver" of length five will then, by pruning the indexes 6, 7 and 5, be $(4, 1, 3, 0, 2)$. The CDMA2000 mother interleaver is a two-dimensional matrix 32×2^n, where $n \in \{4, 5, \cdots 10\}$. The information data is entered into this matrix row by row, where they are permuted to a linear congruence sequence given by $x(i+1) = (x(i)+c) \mod 2^n$, where $x(0) = c$ and c is a row-specific value given in a lookup table [39]. The rows are then reordered by reading the rows in a different order. The matrix is then read column by column, where the output is given to the second encoder.

5.3.1 Performance of turbo codes in 3G

In [39] turbo codes are compared to convolutional codes[7] under realistic 3G conditions. The performance of turbo codes are in most cases better than the performance of the convolutional codes. Convolutional codes only seem preferable when the amount of data to be transmitted is small. However, when the amount of data is large, turbo codes outperform the convolutional codes. This is because when the data size increases, the spectral thinning in the turbo interleaver, makes the number of "neighbour" codewords smaller, hence, the turbo codes becomes more effective. The standard turbo code used in CDMA2000 also has automatic repeat request (ARQ) mechanism, which improves the Frame Error Rate (FER). Power control is also an important feature, since 3G is wireless communication where the environment can change quickly. Fast power control effects the performance greatly [3, 17, 39]. Turbo codes have an efficient way of adjusting the transmission power, and especially in

[7] Discussed in section 3.4.2.

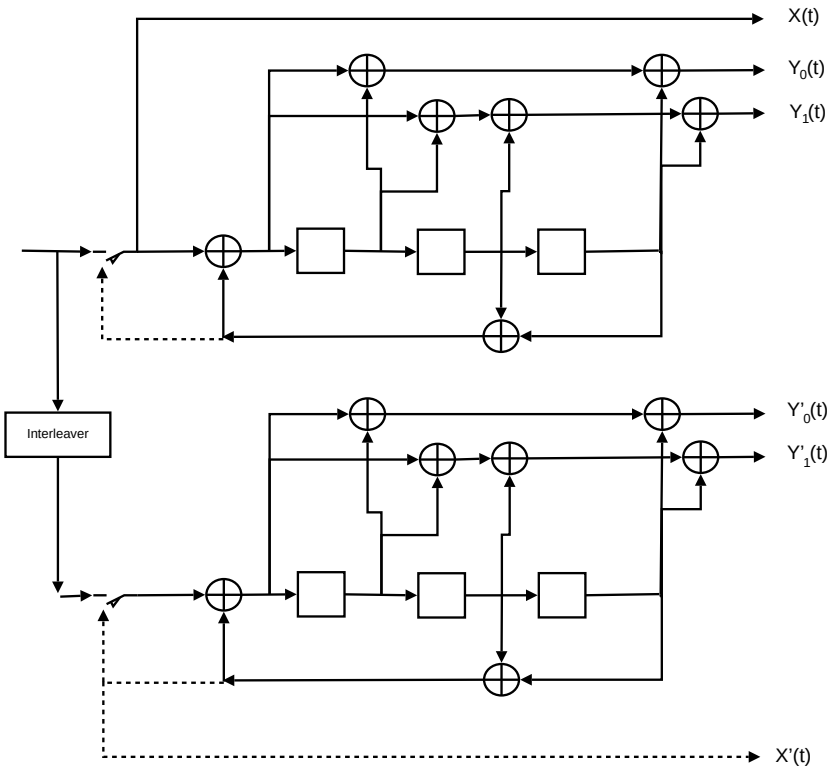

Figure 5.6: The standardized encoding scheme of turbo-codes in 3G.

mobile devices with limited power resources this is very important.

5.3.2 Further work

3G is one of the newest technologies in the area of mobile communications. However, the fourth generation (4G) is already being developed. One promised improvement of the 4G is higher data transmission rate. Though information is limited on this topic, turbo codes might be a component of the next generation of mobile communications.

5.4 Fast correlation attacks

In [35] the techniques of turbo codes have been used in new algorithms for fast correlation attacks. A fast correlation attack [44] is one of the most important class of attacks on a Linear Feedback Shift Register (LFSR) based stream cipher [35, 45]. Figure 5.7 shows a small LFSR with two taps. Figure 5.8 shows the concept of how an information sequence u is encoded with a stream cipher. The

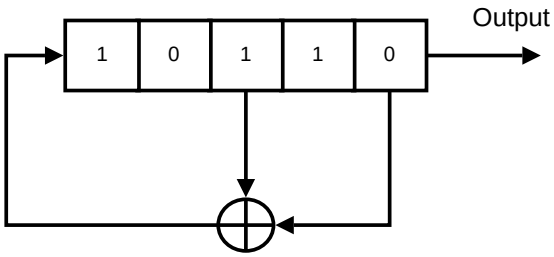

Figure 5.7: Linear Feedback Shift Register (LFSR) with two taps.

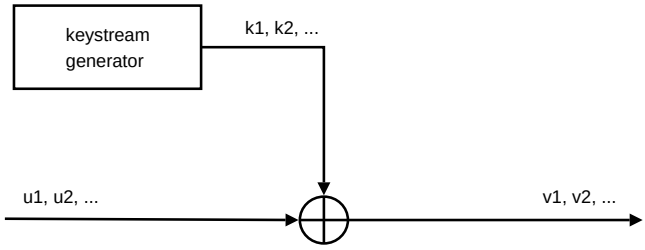

Figure 5.8: Binary additive stream cipher.

keystream k can be generated by only one, or many LFSRs that are added together. This keystream is then added bitwise to the information sequence, which results in the output sequence v.

A correlation between a known keystream sequence and the output l of one of the LFSRs is always present [44, 61, 60, 36]. When this correlation is of the form $P(l_i = k_i) \neq 0.5$ [44, 36], in other words, the correlation is different than a half, and further, if the number of taps in the LFSR is small, then the two algorithms proposed in [44] for fast correlation attack can be used.

These algorithms have been known since the late eighties, without substantial improvements. In 1999 an attack using convolutional codes was proposed in [36]. The same year the same authors proposed a similar attack using turbo codes [35]. These new kinds of correlation attacks are based on combining the iterative decoding technique proposed in [44], with the framework of convolutional codes [36, 35].

The first object of the correlation attacks proposed in [44] is to find a set of suitable parity check equations in the code C. Second, one uses these parity check equations in a fast decoding algorithm to recover the transmitted codeword. This is then used to find the

initial state of the LFSR.

5.4.1 Improved fast correlation algorithm

The following section assumes the reader is familiar with the basics of fast correlation attacks [44]. Given a $l \times N$ generator matrix G_{LFSR} for a code generated by a LFSR, a received sequence z, the probability of an error p, the number of iterations I and the number of constituent codes M, the fast correlation attack using turbo code techniques can be used. First, let $\pi_2, \ldots, \pi_M$ be M-1 random permutations permuting indices $B+1, \ldots, J$. The remaining indices should be left fixed. Then define the generator matrices for the M different codes that are permuted versions of the G_{LFSR}, i.e. $G_1 = G_{\text{LFSR}}, G_2 = \pi_2(G_{\text{LFSR}}), \ldots, G_M = \pi_M(G_{\text{LFSR}})$. For G_i, where $2 \leq i \leq M$, $\pi_i(z)$ denotes the received sequence. Find all the parity checks of the form given in equation 5.21 for every G_i, where $1 \leq i \leq M$.

$$\begin{aligned} u_n + \textstyle\sum_{i=1}^{B} c_{i1} u_{n-i} + u_{i_{n1}} + u_{j_{n1}} &= 0, \\ u_n + \textstyle\sum_{i=1}^{B} c_{i2} u_{n-i} + u_{i_{n2}} + u_{j_{n2}} &= 0, \\ &\vdots \\ u_n + \textstyle\sum_{i=1}^{B} c_{im(n)} u_{n-i} + u_{i_{nm(n)}} + u_{j_{nm(n)}} &= 0, \end{aligned} \tag{5.21}$$

where $m(n)$ denotes the found parity checks for position n. These steps are the precomputing part of the algorithm [35].

By using the given error probability p and $P(u_n = z_n) = 1 - p$, construct the a priori probability vector $(P(u_{B+1}), P(u_{B+2}), \ldots, P(u_J))$. Then, for each G_i, construct the received sequence r by

$$r_n^{(0)} = z_n, \quad r_n^{(k)} = z_{i_{nk}} + z_{j_{nk}}, \quad j \leq k \leq m. \tag{5.22}$$

This sequence can be understood as the "received" sequence for the turbo code. Compute the corresponding a priori probabilities for the convolutional codeword vector v_n, $B+1 \leq k \leq J$, by using

$$P(v_n^{(k)} = r_n^{(k)}) = (1-p)^2 + p^2, i \leq k \leq m. \tag{5.23}$$

Now, update for each G_i the probability

$$P(v_n^{(0)}) = P(u_n), \quad B+1 \leq n \leq J. \tag{5.24}$$

The MAP algorithm, mentioned in section 4.5 and described in [70], can now be used on G_i with starting state distribution $P(s_s) =$

$P(v_{B+1}^{(0)}|r), P(v_{B+2}^{(0)}|r), \ldots, P(v_J^{(0)}|r)$, where $1 \leq i \leq M$. When $i = M + 1$ restart at $i = 1$. Compute the probabilities

$$P(u_{B+1}) = P(v_{B+1}^{(0)}|r), P(u_{B+2}) = P(v_{B+2}^{(0)}|r), \ldots, P(u_J) = P(v_J^{(0)}|r)$$

. As long as the number of iterations is less than $I \times M$, the algorithm should go back to equation 5.24 and run more iterations. If the number of iterations are sufficient the most probable value for each symbol $u_{5B+1}, u_{5B+2}, \ldots, u_{5B+l}$ is selected, and from them one can calculate the initial state u_0[8]. This initial state vector should be verified by checking its encoding result[9].

5.4.2 Performance

This improved form of correlation attack has, according to the results presented in [35], an improved performance as M grows and B is fixed. Further, these results have shown that the improved form is more efficient than the correlation attacks proposed in [44]. [35] also proposes a parallel version of the attack, however this will not be discussed here.

5.4.3 Comments

Using turbo techniques in fast correlation attacks, shows that the turbo technique can be adapted to other areas. This indicates that turbo codes will, in the future, be used in many different contexts. An open question for the future is to find other areas where turbo codes can be useful.

[8] A pseudocode version of this algorithm can be found in [35].
[9] This is the usual procedure of checking the results correctness.

Chapter 6

Examples of Turbo product codes

The following sections describe different codes constructed in this thesis. They are all programmed in c++ and will be explained later in Appendix A.

6.1 Turbo-code no. 1

This encoder was based on the examples given in [63, 64]. It is a product code.

6.1.1 Encoder

This example uses a product code to encode a four bit message. The parity check will be computed in the following way.

a	b	ab
c	d	cd
ac	bd	

Figure 6.1: Encoder. The double letters are added. $ab = a + b \pmod 2$

After encoding, the codeword contains four bits of information and four bits of parity check. In total this is eight bits. The interleaving here just swaps b and c. The result returned will then be ac and bd, where ac = a + c (mod 2) = a $\oplus$ b, and bd = b + d (mod 2) = b $\oplus$ d. This is illustrated in figure 6.2.

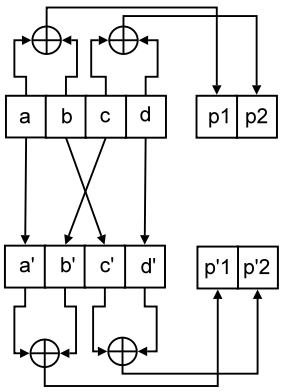

Figure 6.2: The original interleaver

6.1.2 Decoding

The decoding algorithm is based on a similar algorithm found on page 8 in [63].

Algorithm

In section 3.3.2 log-likelihood ratios (LLR) were defined in equation (3.11). Now equation (3.12), which defined the conditioned log-likelihood ratio, will be rewritten. $L_{X|D}(x|d)$ was defined as LLR obtained by measuring the output x under the conditions that $d = +1$ or $d = -1$. To simplify notation the $L_{X|D}(x|d)$ is replaced with $L_c(x)$, since $L_c(x)$ really is a LLR found by measuring the channel output at the receiver. The equations $L_D(d)$ will from now on be written as $L(d)$, and $L_{D|X}(d|x)$ will be written as $L(d|x)$. Hence,

$$L(d|x) = L(d) + L_c(x) \tag{6.1}$$

Remember that $L(d|x)$ is a real number that represents a soft decision made from the received signal. Turbo codes are iterative, and each decoding iteration receives the soft output $L(\hat{d})$, or LLR result, from the previous decoding iteration. Expressing this soft output for a systematic code, as shown in [10, 30] is done by equation

$$L(\hat{d}) = L(d|x) + L_e(\hat{d}) \tag{6.2}$$

where $L_e(\hat{d})$ is called extrinsic LLR, and which holds extra knowledge about the decoding process divided in horizontal and vertical directions. In the following equations $L_{eh}(\hat{d})$ will represent the extrinsic LLR from the horizontal decoding process and $L_{ev}(\hat{d})$ will represent the extrinsic LLR from the vertical decoding process. By

replacing $L(d|x)$ we obtain

$$L(\hat{d}) = L_c(x) + L(d) + L_e(\hat{d}) \tag{6.3}$$

1. If the a priori probabilities of the data bits are equally likely, then set the a priori Log-Likelihood Ratio (LLR) $L(d) = 0$.
2. Decode horizontally, and using equation (6.3) obtain the horizontal extrinsic LLR as shown below:
$$L_{eh}(\hat{d}) = L(\hat{d}) - L_c(c) - L(d)$$
3. Set $L(d) = L_{eh}(\hat{d})$ for the vertical decoding of step 4.
4. Decode vertically, and using equation (6.3) obtain the vertical extrinsic LRR as shown below:
$$L_{ev}(\hat{d}) = L(\hat{d}) - L_c(x) - L(d)$$
5. Set $L(d) = L_{ev}(\hat{d})$ and repeat steps 2 through 5.
6. When enough iterations, at least five, have been done, one is able to yield a reliable decision, and go to step 7. This means that for each iteration the soft decision must be compared with the soft decision from the preceding iteration. If the difference between their estimates is small, the soft decision is assumed to have converged and a reliable decision can be made.
7. The soft output is:
$$L(\hat{d}) = L_c(x) + L_{eh}(\hat{d}) + L_{ev}(\hat{d}) \tag{6.4}$$

Consequently one could say that one starts with the measurements of the received codeword. One then sets the a priori LLR, noted $L(d)$, equal to zero, and then starts to iterate. One first decodes from left to right, then from top to bottom. If the result is a reliable decision, measured by soft decision convergence from preceding iterations, terminate the algorithm and return the soft output. If the result is not reliable further iterations are required. Normally at least 5 or 6 iterations are needed before a reliable decision can be returned, however more iterations might be needed.

6.1.3 Decoding example

This example is taken from Fundamentals of Turbo Codes by Bernard Sklar [63]. Suppose the message intended to be sent is 1001. For the coding scheme given above, the encoding will return the output shown in figure 6.3.

1	0	1
0	1	1
1	1	

Figure 6.3: Encoded 1001

This resultant codeword is 10011111. After passing through a channel, errors are introduced to the codeword. For instance, suppose the codeword received is:

$$\{x_i\}, \{x_{ij}\} = 0.75, 0.05, 0.10, 0.15, 1.25, 1.0, 3.0, 0.5 \tag{6.5}$$

In this example an AWGN interference model is used for the channel, discussed in section 2.2.1. Therefore the channel measurement of a signal x_k at time k is:

$$L_c(x_k) = log_e \left[\frac{p(x_k|d_k = +1)}{p(x_k|d_k = -1)} \right] \tag{6.6}$$

$$= log_e \left(\frac{\frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2} \left(\frac{x_k - 1}{\sigma} \right)^2 \right]}{\frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2} \left(\frac{x_k + 1}{\sigma} \right)^2 \right]} \right) \tag{6.7}$$

$$= -\frac{1}{2} \left(\frac{x_k - 1}{\sigma} \right)^2 + \frac{1}{2} \left(\frac{x_k + 1}{\sigma} \right)^2 = \frac{2}{\sigma^2} x_k \tag{6.8}$$

If one assumes that the noise variance σ^2 is unity, one can make a simplifying assumption:

$$L_c(x_k) = 2x_k \tag{6.9}$$

Then get the LLR values from equation (6.5):

$$\{L_c(x_i)\}, \{L_c(x_{ij})\} = 1.5, 0.1, 0.2, 0.3, 2.5, 2.0, 6.0, 1.0 \tag{6.10}$$

$d_1 = 1$	$d_2 = 0$	$d_{12} = 1$
$d_3 = 0$	$d_4 = 1$	$d_{34} = 1$
$d_{13} = 1$	$d_{24} = 1$	

Figure 6.4: Encoder output binary digits

$L_c(x_1) = 1.5$	$L_c(x_2) = 0.1$	$L_c(x_{12}) = 2.5$
$L_c(x_3) = 0.2$	$L_c(x_4) = 0.3$	$L_c(x_{34}) = 2.0$
$L_c(x_{13}) = 6.0$	$L_c(x_{24}) = 1.0$	

Figure 6.5: Decoder input log-likelihood ratios $L_c(x)$

To express the soft output $L(\hat{d}_1)$ for the received signal corresponding to data d_1, one uses equation 6.3, and then gets:

$$L(\hat{d}_1) = L_c(x_1) + L(d_1) + \{[L_c(x_2) + L(d_2)] \boxplus L_c(x_{12})\} \tag{6.11}$$

In general the soft output $L(\hat{d}_i)$ for the received signal corresponding to data d_i is

$$L(\hat{d}_i) = L_c(x_i) + L(d_i) + \{[L_c(x_j) + L(d_j)] \boxplus L_c(x_{ij})\} \tag{6.12}$$

Here $L_c(x_i)$, $L_c(x_j)$ and $L_c(x_{ij})$ are the channel LLR measurements of the received data to d_i, d_j and d_{ij} and $\{[L_c(x_j) + L(d_j)] \boxplus L_c(x_{ij})\}$ is the extrinsic LLR given by the code.

Computing the extrinsic likelihoods

The decoder is iterative and consists of two independent decoders that feed soft output to each other. The first decoder works on parity bits from the first encoder, and the second one works on the parity bits given by the second decoder. For the given example, the horizontal, or first decoder, is denoted as $L_{eh}(\hat{d})$ and the vertical, or second decoder, is denoted as $L_{ev}(\hat{d})$. Then the eight different extrinsic likelihoods are computed by the following equations:

$$L_{eh}(\hat{d}_1) = \left[L_c(x_2) + L(\hat{d}_2)\right] \boxplus L_c(x_{12}) \tag{6.13a}$$

$$L_{ev}(\hat{d}_1) = \left[L_c(x_3) + L(\hat{d}_3)\right] \boxplus L_c(x_{13}) \tag{6.13b}$$

$$L_{eh}(\hat{d}_2) = \left[L_c(x_1) + L(\hat{d}_1)\right] \boxplus L_c(x_{12}) \tag{6.13c}$$

$$L_{ev}(\hat{d}_2) = \left[L_c(x_4) + L(\hat{d}_4)\right] \boxplus L_c(x_{24}) \tag{6.13d}$$

$$L_{eh}(\hat{d}_3) = \left[L_c(x_4) + L(\hat{d}_4)\right] \boxplus L_c(x_{34}) \tag{6.13e}$$

$$L_{ev}(\hat{d}_3) = \left[L_c(x_1) + L(\hat{d}_1)\right] \boxplus L_c(x_{13}) \tag{6.13f}$$

$$L_{eh}(\hat{d}_4) = \left[L_c(x_3) + L(\hat{d}_3)\right] \boxplus L_c(x_{34}) \tag{6.13g}$$

$$L_{ev}(\hat{d}_4) = \left[L_c(x_2) + L(\hat{d}_2)\right] \boxplus L_c(x_{24}) \tag{6.13h}$$

If the data given by figure 6.5 is used, and the $L(d)$ values reset to zero, the result will be:

$$L_{eh}(\hat{d}_1) = (0.1 + 0) \boxplus 2.5 \approx -0.1 = \text{ new}L(d_1) \tag{6.14a}$$

$$L_{eh}(\hat{d}_2) = (1.5 + 0) \boxplus 2.5 \approx -1.5 = \text{ new}L(d_2) \tag{6.14b}$$

$$L_{eh}(\hat{d}_3) = (0.3 + 0) \boxplus 2.0 \approx -0.3 = \text{ new}L(d_3) \tag{6.14c}$$

$$L_{eh}(\hat{d}_4) = (0.2 + 0) \boxplus 2.0 \approx -0.2 = \text{ new}L(d_4) \tag{6.14d}$$

To compute the log-likelihood addition, equation (3.14) is used. Then the second decoder, that uses the new $L(d)$ values computed by the first decoder is used to compute the following results:

$$L_{ev}(\hat{d}_1) = (0.2 - 0.3) \boxplus 6.0 \approx 0.1 = \text{ new}L(d_1) \tag{6.15a}$$

$$L_{ev}(\hat{d}_2) = (0.3 - 0.2) \boxplus 1.0 \approx -0.1 = \text{ new}L(d_2) \tag{6.15b}$$

$$L_{ev}(\hat{d}_3) = (1.5 - 0.1) \boxplus 6.0 \approx -1.4 = \text{ new}L(d_3) \tag{6.15c}$$

$$L_{ev}(\hat{d}_4) = (0.1 - 1.5) \boxplus 1.0 \approx 1.0 = \text{ new}L(d_4) \tag{6.15d}$$

The first two decoding steps of the first iteration are now stored in $L_{ed}(\hat{d})$ and $L_{ev}(\hat{d})$. Now the soft output has to be computed. The

1.5	0.1
0.2	0.3

(a) Original $L_c(x_k)$

-0.1	-1.5
-0.3	-0.2

(b) Horizontal $L_{eh}(\hat{d})$

0.1	-0.1
-1.4	1.0

(c) Vertical $L_{ev}(\hat{d})$

Figure 6.6: Original $L_c(x_k)$, the $L_{eh}(\hat{d})$ and $L_{ev}(\hat{d})$

improved LLR is computed by $L_c(x_k) + L_{eh}(\hat{d}) + L_{ev}(\hat{d}) =$. From figure 6.6 it is seen that the improved LLR is as shown in figure 6.7:

The first LLR values are then computed. In this case one can see that the correct codeword has already been found. However this can

1.5	-1.5
-1.5	1.1

Figure 6.7: Improved LLR

not be known for sure. Therefore more iterations will be needed to obtain a more confident decision. After four iterations the result is as shown in figure 6.8

2.6	-2.5
-2.6	2.5

Figure 6.8: $L(\hat{d})$ after four iterations

The confidence about these decisions is high, since the soft decisions have converged to the same result for two iterations. Therefore no further iterations are needed and the correct message is: 1001.

6.2 Turbo-code no. 2

It is now time to explore changes in the interleaver. Code number 2, shown if figure 6.9, works much in the same way as code number 1 shown in figure 6.2. The only difference between them is the interleaver.

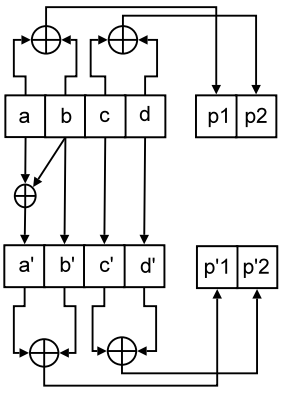

Figure 6.9: The interleaver is replaced by a permutation over $\mathbb{Z}_{2^4}$.

The interleaver is now replaced by a more general permutation that maps $\mathbb{Z}_{2^4}$ to $\mathbb{Z}_{2^4}$ as shown in figure 6.9. The permuted parity bits

are then computed by:

$$P_1' = a \oplus b \oplus b = a \tag{6.16}$$

$$P_2' = c \oplus d = P_2 \tag{6.17}$$

This permutaion is by equation (6.16) and (6.17) justified to be in $\mathbb{Z}_{2^4}$, since the result of $P_1' = a$ and $P_2' = P_2$ is already known to be in $\mathbb{Z}_{2^4}$. Earlier, in the section about interleaving, it was stated that the point of interleaving was to protect the code against burst errors. This was done by spreading the data out between the parity bits. As one can see from equation (6.16) and (6.17), this is not the case here. Accordingly the choice of permutation function in figure 6.9 is not a good one.

6.2.1 Decoding

Decoding this code is very similar to that of section 6.1. Equation (6.12) is still valid, however the new permutation model has now to be taken into consideration. Therefore some of the equations from (6.13a) to (6.13h) will differ in this decoder. To be precise all the $L_{ev}(\hat{d}_i)$ will differ:

Equation (6.13b) will become:

$$L_{ev}(\hat{d}_1) = \left[L_c(x_2) + L(\hat{d}_2)\right] \boxplus L_c(x_7) \tag{6.18}$$

Equation (6.13d) will become:

$$L_{ev}(\hat{d}_2) = \left[L_c(x_1) + L(\hat{d}_1)\right] \boxplus 0 \tag{6.19}$$

Equation (6.13f) will become:

$$L_{ev}(\hat{d}_3) = \left[L_c(x_4) + L(\hat{d}_4)\right] \boxplus L_c(x_8) \tag{6.20}$$

Equation (6.13h) will become:

$$L_{ev}(\hat{d}_4) = \left[L_c(x_3) + L(\hat{d}_3)\right] \boxplus L_c(x_8) \tag{6.21}$$

It is now easily seen that the b in figure 6.2 on page 70 does not have as much parity information as the other bits in the message. Accoringly the assumption that the choice of interleaver function could have been better seems to be correct. Nevertheless, if one tries to decode the same message given in the example, one gets the correct result, but it takes more iterations.

6.3 Turbo-code no. 3

Since the errors in our choice of permutation function were quite obvious, an example with a better permutation function should be given, as shown in figure 6.10.

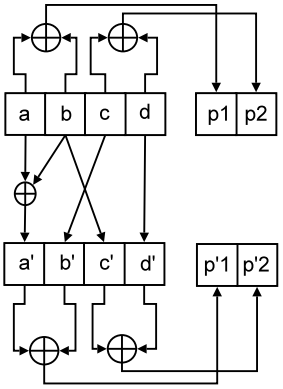

Figure 6.10: The interleaver replaced by another permutation.

Figure 6.10 shows a function where each parity bit of the permuted code is a product of several bits. The parity bits can be expressed as:

$$P_1' = a \oplus b \oplus c \tag{6.22}$$

$$P_2' = b \oplus d \tag{6.23}$$

Since P_1' and P_2' both are results of XORed values from $\mathbb{Z}_{2^4}$, according to section 2.1 and [13, 52], the result will also be in $\mathbb{Z}_{2^4}$. In this case the $L_{eh}(d_i)$ equations will be the same, however the $L_{ev}(d_i)$ equations will differ from the ones given in (6.13b) to (6.13h), that result in:

$$L_{ev}(\hat{d}_1) = \Big[\Big[L_c(x_3) + L(\hat{d}_3)\Big] \boxplus L_c(x_7)\Big] \boxplus \Big[\Big[L_c(x_2) + L(\hat{d}_2)\Big] \boxplus L_c(x_7)\Big] \tag{6.24}$$

$$L_{ev}(\hat{d}_2) = \Big[L_c(x_4) + L(\hat{d}_4)\Big] \boxplus L_c(x_8) \tag{6.25}$$

$$L_{ev}(\hat{d}_3) = \Big[\Big[L_c(x_1) + L(\hat{d}_1)\Big] \boxplus L_c(x_7)\Big] \boxplus \Big[\Big[L_c(x_2) + L(\hat{d}_2)\Big] \boxplus L_c(x_7)\Big] \tag{6.26}$$

$$L_{ev}(\hat{d}_4) = \Big[L_c(x_2) + L(\hat{d}_2)\Big] \boxplus L_c(x_8) \tag{6.27}$$

Decoding the given example with this decoder uses 15 iterations to find the right result. This is much slower than the two other decoders.

6.4 Turbo-code no. 4

This time the encoding in both the original and the interleaved part was different from its predecessors. Acctually the encoding is the same as multiplying the message u with a matrix defined as

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \tag{6.28}$$

If the selected message is $abcd$, the four parity bits will be computed as

$$A \cdot \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} bcd \\ acd \\ abd \\ abc \end{bmatrix}. \tag{6.29}$$

Put in context with figure 6.11 the outputs will be $p_1 = bcd$, $p_2 = acd$, $p_1' = abd$ and $p_2' = abc$, where the two first parity bits are the horizontal parity bits, and the last two are the vertical parity bits. Therefore we exchange c with a for the first parity bit in encoder two, and d with b for the second parity bit for encoder two, as shown in figure 6.11

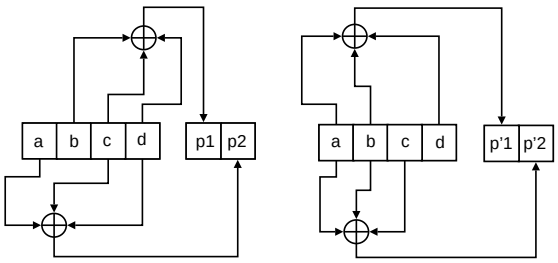

Figure 6.11: The third coding scheme. The figure to the left shows the encoding before "interleaving", and the right one shows the encoding after.

Decoding this code involved modifying the equations (6.13) and equation (3.14) to fit the chosen encoder scheme. Since the parity bits are products of several bits the resulting equations have to compute the log-likelihood ratio between all the different bits, resulting in

$$\begin{aligned} L_{eh}(\hat{d}_1) = \; & \Big[\Big[L_c(x_3) + L(\hat{d}_3)\Big] \boxplus L_c(x_{34})\Big] \\ & \boxplus \Big[L_c(x_4) + L(\hat{d}_4)\Big] \boxplus L_c(x_{34}) \end{aligned} \tag{6.30a}$$

$$\begin{aligned} L_{ev}(\hat{d}_1) = \; & \Big[\Big[L_c(x_2) + L(\hat{d}_2)\Big] \boxplus L_c(x_{13})\Big] \\ & \boxplus \Big[\Big[L_c(x_4) + L(\hat{d}_4)\Big] \boxplus L_c(x_{13})\Big] \\ & \boxplus \Big[\Big[L_c(x_2) + L(\hat{d}_2)\Big] \boxplus L_c(x_{24})\Big] \\ & \boxplus \Big[\Big[L_c(x_3) + L(\hat{d}_3)\Big] \boxplus L_c(x_{24})\Big] \end{aligned} \tag{6.30b}$$

$$\begin{aligned} L_{eh}(\hat{d}_2) = \; & \Big[\Big[L_c(x_3) + L(\hat{d}_3)\Big] \boxplus L_c(x_{12})\Big] \\ & \boxplus \Big[L_c(x_4) + L(\hat{d}_4)\Big] \boxplus L_c(x_{12}) \end{aligned} \tag{6.30c}$$

$$\begin{aligned} L_{ev}(\hat{d}_2) = \; & \Big[\Big[L_c(x_1) + L(\hat{d}_1)\Big] \boxplus L_c(x_{13})\Big] \\ & \boxplus \Big[\Big[L_c(x_4) + L(\hat{d}_4)\Big] \boxplus L_c(x_{13})\Big] \\ & \boxplus \Big[\Big[L_c(x_1) + L(\hat{d}_1)\Big] \boxplus L_c(x_{24})\Big] \\ & \boxplus \Big[\Big[L_c(x_3) + L(\hat{d}_3)\Big] \boxplus L_c(x_{24})\Big] \end{aligned} \tag{6.30d}$$

$$\begin{aligned} L_{eh}(\hat{d}_3) = \; & \Big[\Big[L_c(x_2) + L(\hat{d}_2)\Big] \boxplus L_c(x_{12})\Big] \\ & \boxplus \Big[\Big[L_c(x_4) + L(\hat{d}_4)\Big] \boxplus L_c(x_{12})\Big] \\ & \boxplus \Big[\Big[L_c(x_1) + L(\hat{d}_1)\Big] \boxplus L_c(x_{34})\Big] \\ & \boxplus \Big[\Big[L_c(x_4) + L(\hat{d}_4)\Big] \boxplus L_c(x_{34})\Big] \end{aligned} \tag{6.30e}$$

$$\begin{aligned} L_{ev}(\hat{d}_3) = \; & \Big[\Big[L_c(x_1) + L(\hat{d}_1)\Big] \boxplus L_c(x_{24})\Big] \\ & \boxplus \Big[L_c(x_2) + L(\hat{d}_2)\Big] \boxplus L_c(x_{24}) \end{aligned} \tag{6.30f}$$

$$
\begin{aligned}
L_{eh}(\hat{d}_4) = \; & \Big[\Big[L_c(x_2) + L(\hat{d}_2)\Big] \boxplus L_c(x_{12})\Big] \\
& \boxplus \Big[\Big[L_c(x_3) + L(\hat{d}_3)\Big] \boxplus L_c(x_{12})\Big] \\
& \boxplus \Big[\Big[L_c(x_1) + L(\hat{d}_1)\Big] \boxplus L_c(x_{34})\Big] \\
& \boxplus \Big[\Big[L_c(x_3) + L(\hat{d}_3)\Big] \boxplus L_c(x_{34})\Big]
\end{aligned} \tag{6.30g}
$$

$$
\begin{aligned}
L_{ev}(\hat{d}_4) = \; & \Big[\Big[L_c(x_1) + L(\hat{d}_1)\Big] \boxplus L_c(x_{13})\Big] \\
& \boxplus \Big[L_c(x_2) + L(\hat{d}_2)\Big] \boxplus L_c(x_{13})
\end{aligned} \tag{6.30h}
$$

The equations look quite complex, but the same approximation introduced from equation (3.13) to equation (3.14) is used, which implies that all values are compared in pairs, and then the smallest ratio is selected as the new extrinsic LLR. However, it should be mentioned that this approximation is quite rough, and this will be the topic of section 6.5.

6.5 Approximation difference

The approximation of equation (3.13) is given in equation (3.14) and this approximation has been used in the codes described in this chapter. As said earlier, this approximation is quite rough. Let the example given in section 6.1.3 be computed without using this approximation. Denoting this unapproximated summation operation, defined by equation (3.13), with $\boxplus'$, equation (6.14a) to equation (6.14d), which are the horizontal extrinsic values, become

$$L_{eh}(\hat{d}_1) = (0.1 + 0) \boxplus' 2.5 = -0.0848 = \text{ new}L(d_1) \tag{6.31a}$$

$$L_{eh}(\hat{d}_2) = (1.5 + 0) \boxplus' 2.5 = -1.2049 = \text{ new}L(d_2) \tag{6.31b}$$

$$L_{eh}(\hat{d}_3) = (0.3 + 0) \boxplus' 2.0 = -0.2278 = \text{ new}L(d_3) \tag{6.31c}$$

$$L_{eh}(\hat{d}_4) = (0.2 + 0) \boxplus' 2.0 = -0.1521 = \text{ new}L(d_4) \tag{6.31d}$$

Further, the vertical values become

$$L_{ev}(\hat{d}_1) = (0.2 - 0.3) \boxplus' 6.0 = 0.0277 = \text{ new}L(d_1) \tag{6.32a}$$

$$L_{ev}(\hat{d}_2) = (0.3 - 0.2) \boxplus' 1.0 = -0.0682 = \text{ new}L(d_2) \tag{6.32b}$$

$$L_{ev}(\hat{d}_3) = (1.5 - 0.1) \boxplus' 6.0 = -1.4056 = \text{ new}L(d_3) \quad (6.32c)$$

$$L_{ev}(\hat{d}_4) = (0.1 - 1.5) \boxplus' 1.0 = 0.4729 = \text{ new}L(d_4) \quad (6.32d)$$

When comparing these results, shown in figure 6.12, with the approximated results given in figure 6.6 all numbers have the same sign as the approximated version. However, they seem to converge slower than the approximated results. The next iterations will then

-0.0848	-1.2049
-0.2278	-0.1521

(a) Horizontal $L_{eh}(\hat{d})$

0.0277	-0.0682
-1.4056	0.4729

(b) Vertical $L_{ev}(\hat{d})$

1.4428	-1.1731
-1.4334	0.6208

(c) Improved LLRs

Figure 6.12: The $L_{eh}(\hat{d})$, $L_{ev}(\hat{d})$ and Improved LLRs

give

$$L_{eh}(\hat{d}_1) = (0.1 + 0.0277) \boxplus' 2.5 = -0.1083 = \text{ new}L(d_1) \quad (6.33a)$$

$$L_{eh}(\hat{d}_2) = (1.5 - 0.0682) \boxplus' 2.5 = -1.1558 = \text{ new}L(d_2) \quad (6.33b)$$

$$L_{eh}(\hat{d}_3) = (0.3 - 1.4056) \boxplus' 2.0 = 0.8066 = \text{ new}L(d_3) \quad (6.33c)$$

$$L_{eh}(\hat{d}_4) = (0.2 + 0.4729) \boxplus' 2.0 = -0.5044 = \text{ new}L(d_4) \quad (6.33d)$$

and the vertical values

$$L_{ev}(\hat{d}_1) = (0.2 + 0.8066) \boxplus' 6.0 = -1.0007 = \text{ new}L(d_1) \quad (6.34a)$$

$$L_{ev}(\hat{d}_2) = (0.3 - 0.5044) \boxplus' 1.0 = 0.0942 = \text{ new}L(d_2) \quad (6.34b)$$

$$L_{ev}(\hat{d}_3) = (1.5 - 0.1083) \boxplus' 6.0 = -1.3824 = \text{ new}L(d_3) \quad (6.34c)$$

$$L_{ev}(\hat{d}_4) = (0.1 - 1.1558) \boxplus' 1.0 = 0.4548 = \text{ new}L(d_4), \quad (6.34d)$$

which can be represented in a figure 6.13

After four iterations the LLRs are as shown in figure 6.14 The LLRs are at this point converging to the same result as the approximated result. However, as pointed out earlier, the LLRs seem to converge slower.

-0.1083	-1.1558
0.8066	-0.5044

(a) Horizontal $L_{eh}(\hat{d})$

-1.0007	0.0942
-1.3824	0.4548

(b) Vertical $L_{ev}(\hat{d})$

1.8476	-1.6218
-1.8393	1.6526

(c) Improved LLRs

Figure 6.13: The $L_{eh}(\hat{d})$, $L_{ev}(\hat{d})$ and improved LLRs after 2 iterations

2.2829	-1.9437
-2.2700	1.9873

Figure 6.14: LLRs after 4 iterations

Chapter 7

Simulations and Results

All the simulations were run on an Intel Pentium Centrino processor 2.00GHz, with a total of 1024 MB memory running SuSE 10.1.

7.1 Simulations

All the results presented in this chapter were produced by the $fort.out$ program described in section A. This program simultaneously simulates the four codes presented in chapter 6. The drawback of doing the simulations simultaneously is that the memory usage gets quite large, and at the point when the number of iterations exceeds approximately 74700, no result is returned. This is unfortunate when finding the Bit-Error Rate (BER), since the technique used required 100 error blocks to compute the average BER of the simulations. When the number of simulations gave over 100 error block the BER could be computed by

$$\text{BER} = \frac{\text{Number of bit errors}}{\text{Number of sent bits}}. \tag{7.1}$$

However, when the E_b/N_0 grew the number of errors went down, as expected, and therefore more iterations where needed to find the correct BER. When the number of simulations came to the limit of approximately 74700, the correctness of the BER went down as the E_b/N_0 grew. The error floor, which usually appears in the region where the BER curve of the turbo codes is below 10^{-5} [66], is therefore not present. The reader should be aware of this when studying the graphs.

7.1.1 Channel simulation

All the simulations used an AWGN channel. This was emulated by using equation (2.2), which is a normally distributed function, where the function takes the Signal to Noise Ratio (SNR)[1] as in data and returns a random normally distributed noise variable. This noise is then added to the "transmitted" binary vector that is received by the receiver.

7.2 Results

Figure 7.1 shows the BER curves of the four codes discussed in chapter 6. The figure shows that code no. 2 and no. 3 have worse performance than the original code no. 1. This result is not very suprising since some of the parity bits in code no. 2 and 3 have less parity information about the information bits they are a "product" of. For example, equation (6.16) says that the parity bit P_1' has no information about the bit called b, since the bits are XORed. This will normally result in slower convergence, which figure 7.2 shows is the case here. It should be remarked that the program has been limited to a number of iterations i, where $i \in (6, 7, \cdots, 30)$.

Code no. 4 is, as figure 7.1 shows, catastrophic[2]. The code does not seem to decode correctly for any E_b/N_0, and has therefore a high BER curve. This code should therefore not be used.

Code no. 1 has very good BER performance, however the results beyond $E_b/N_0 = 1.0$ have large uncertainty because of the reasons described in section 7.1. Knowledge of the performance in the error floor region is not given in the figure, so further research is needed to give a complete description of the code.

7.2.1 Remarks

The graphs make it clear that code no. 1 is the best code of the four. From this one may conclude that replacing interleavers with more general permutations may be a bad idea. However, it should be remarked that the length of the information bits is four, which limits the permutation possibilities. Therefore, further research should look into codes with larger information bit length. One of the factors that

[1] Not given as decibel.

[2] Catastrophic in this context should not be confused with the catastrophic encoders discussed in section 5.2.

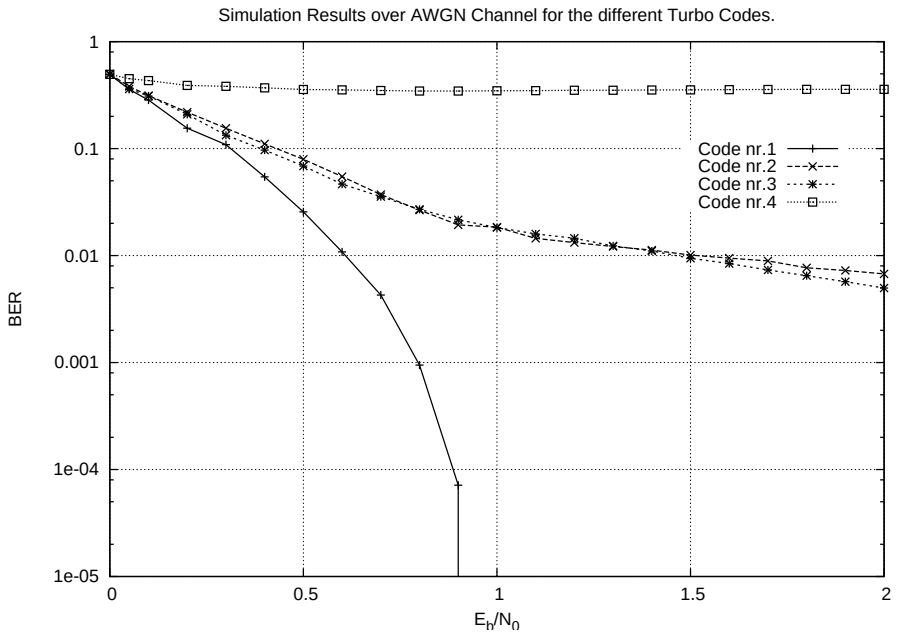

Figure 7.1: The BER curves for the simulations of the turbo codes under different signal-to-noise ratios.

motivated this research was to investigate the possibility of using the interleaver or more general permutation as a cryptographic key for the turbo code. Even if the results in the graphs were unpromising, several different angles should be investigated.

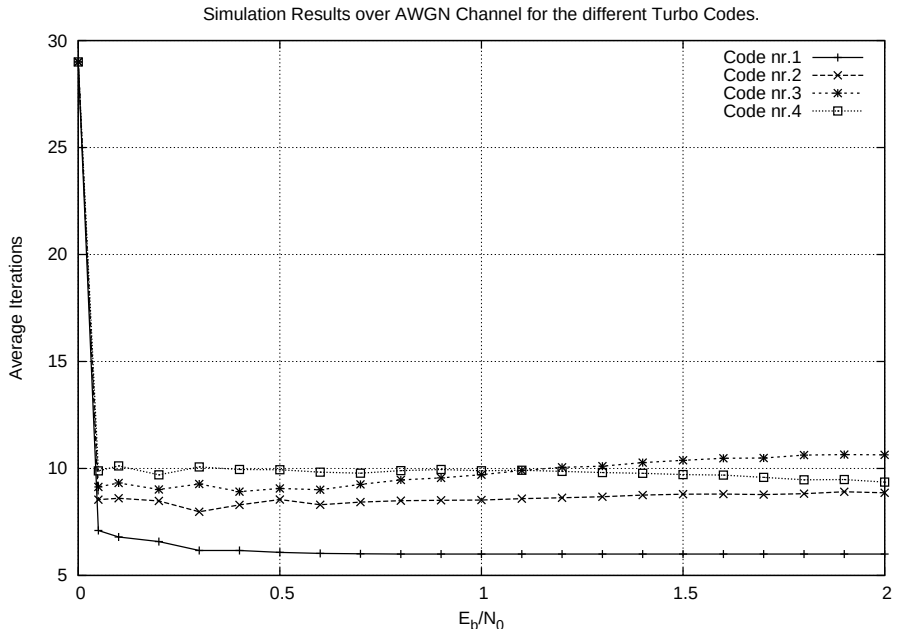

Figure 7.2: Number of iterations under different signal-to-noise ratios.

Chapter 8

Conclusion

In this thesis some of the building blocks of turbo codes, and the basics of turbo codes have been described. Some of the recent developments of turbo codes have been introduced. Especially different methods of interleaving have been discussed. The reason for the focus on interleaving is that interleavers have a large influence on the free distance. When the free distance of a turbo code increases, the error floor performance improves.

Interleavers can be divided into "random" and deterministic interleavers. Mainly, deterministic interleavers have been discussed, but a brief introduction on S-random interleavers has been given. A deterministic interleaver is a permutation of an arranged manner. This can make the analyse of the interleaver simpler.

When turbo code were introduced for the first time in [10], the proposed codes used two Recursive Systematic Convolutional (RSC) encoders. [6] describes codes constructed by NonSystematic Convolutional (NSC) encoders. These codes show a better performance at low Signal-to-Noise Ratios (SNRs). The construction of good NSC codes has been discussed, where some rules of obtaining good codes has been introduced. Mainly the benefit of using NSC codes is their improved performance in the error floor region compared to RSC codes.

Some of the current applications of turbo codes have also been described. The main current applications are 3G mobile communications and space communications, but turbo codes are also used in, for example, the standard for Digital Video Broadcasting (DVB) [16]. The two standards of 3G have both selected turbo codes as their error correcting code. The turbo coding in the 3G standard

CDMA2000 has briefly been described and compared to other candidates for this standard of the third generations of mobile communications [39].

A improved fast correlation attack using turbo code techniques has been discussed. This attack has shown improved efficiency compared to the previous known fast correlation attacks.

The result of a small turbo product code has been given. The interleaver was replaced by a more general permutation and the results investigated. These codes were simulated in an AWGN environment under different signal to noise ratios. The graphs of these experiments showed that the original interleaver had the best BER performance and used the fewest iterations to estimate the correct codeword. These experiments were only performed on small codes, and therefore the permutation possibilities are limited. Further work should examine similar permutations on longer codes.

8.1 Further work

[56] stated that further research should investigate a higher dimensional approach to construct better BITCMs. Other methods of interleaving is an interesting subject that can improve the turbo code performance. Another interesting field would be developing turbo codes for the fourth generation of mobile communications, where high data transmission rates are preferable.

The remarks given in chapter 7 said that replacing the interleaver with more general permutations should be investigated for longer codes. The possibility of using the interleaver or more general permutation as a cryptographic key leaves several open questions that should be investigated.

Appendix A

Programs

The program *turbo.m* is written in Matlab, all the other programs are written in c++. The following command is used when compiling the c++ programs in Linux:
c++ -o $< filename.out > < filename.cpp >$
The .out filename does not have to match the .cpp filename, but it is recommended for simplicity. More information about compiling can be found by typing "man c++" in a terminal window. Running the programs is done by the command:
$./ < filename.out > < argument(s) >$

A.1 Description of programs

The programs should be placed in context with this thesis, therefore a short description is approproate.

Encoders:
*encoder*3 is the program discussed in section 6.1.
*encoder*4 is the program discussed in section 6.2.
*encoder*5 is the program discussed in section 6.3.
*encoder*6 is the program discussed in section 6.4.

Channel:
channel is a AWGN channel, that needs two arguments. First a codeword, on the form given by *encoder*3 - *encoder*6, second the E_b/N_0. For example: "./channel.out 11100111 0.1"

Decoding:
*decoder*3 is the decoder to encoder3.
*decoder*4 is the decoder to encoder4.

*decoder*5 is the decoder to encoder5.
*decoder*6 is the decoder to encoder6.

All of the decoding programs need a list of measured channel values given by the program channel. For example: "./decoder3.out 0.7092 0.8872 1.231 -1.0494 -0.9826 1.2212 0.7456 0.6886". All decoders check if the soft output converges. If the improvement from last iteration is less than 0.01 for all information bits, and the number of iterations is bigger than 6, the decoding terminates and returns the decoded message. If the number of iterations reaches 30 and the log-likelihood ratios do not converge, the decoding algorithm terminates and returns the closest decoded message. In this case retransmission should be called.

These programs have been combined to format the programs:
codeword uses all the programs given above with a user friendly output. This program take the E_b/N_0 as an argument.
$fort$ is the same program as codeword, except that it returns gnuplot friendly output. In addition to the E_b/N_0 argument, this program also needs a positive integer. This integer decides how many times the program is run. When making graphs it can be preferable to run the program several times, to see if the results repeat. For each run a new message is selected. Instead of giving the decoded message, like *codeword*, this program returns the BER to the different codes, and the average number of iterations to decode the message. However, if preferable, the program can easily be changed to give other output. The simulation results in section 7.2 are all produced by this program.

The final program *turbo.m* is a matlab program used in section 6.5 to calculate the LLR values without using the approximation given in equation (3.13).

A.2 Data structures

All the c++ programs use a self written data structure which at first was used because it was spacesaving. After a while it was discovered that this was not the case. However, it was still easier to work with this data structure instead of an integer structure.
TODO

Bibliography

[1] Aliazam Abbasfar and Kung Yao. Interleaver design for high speed turbo decoders. *IEEE Wireless Communications and Networking Conference, Vol. 3*, pages 1611–1615, March 2004.

[2] Robert A. Adams. *Calculus : A Complete Course*. Addison-Wesley, 4. edition, 1999.

[3] Paul Ampadu and Kevin Kornegay. An efficient hardware interleaver for 3G turbo decoding. *IEEE Radio and Wireless Conference, 2003. RAWCON '03*, pages 199–201, 10-13 Aug. 2003.

[4] Jakob Dahl Andersen. A turbo tutorial. *TELE (issn: 1396-1535), Issue: 15*, October 1999.

[5] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv. Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Transactions on Information Theory, Vol. IT-20*, pages 284–287, March 1974.

[6] Adrish Banerjee, Francesca Vatta, Bartolo Scanavino, and Daniel J. Costello Jr. Nonsystematic turbo codes. *IEEE Transactions on Communications. Vol. 53, No. 11*, pages 1841–1849, November 2005.

[7] N. Benvenuto, P. Bisaglia, and A. E. Jones. Performance of turbo detection in a variable rate wireless system using linear block codes and interleaving. *IEEE Transactions on vehicular technology, Vol. 49, No. 6*, pages 2189–2197, November 2000.

[8] Claude Berrou, Patrick Adde, Ettiboua Angui, and Stéphane Faudeil. A low complexity soft-output viterbi decoder architecture. *IEEE Communications, ICC 1993, Vol. 2*, pages 736–740, May 1993.

[9] Claude Berrou and Alain Glavieox. Near optimum error correcting coding and decoding: Turbo-codes. *IEEE Transactions*

on Communications. Vol. 44, No. 10., pages 1261–1271, October 1996.

[10] Claude Berrou, Alain Glavieux, and Punya Thitimajshima. Near shannon limit error-correcting coding and decoding: turbo-codes. *ICC 1993, Geneva, Switzerland*, pages 1064–1070, May 1993.

[11] W. J. Blackert, E. K. Hall, and S. G Wilson. An upper bound on turbo code free distance. *IEEE International Conference on Communications, Vol. 2*, pages 957–961, June 1996.

[12] Alister Burr. Turbo-codes: The ultimate error control codes? *Electronics and Communications Engineering Journal, Vol. 13, Issue 4*, pages 155–165, 2001.

[13] Lindsay N. Childs. *A Concrete Introduction to Higher Algebra*. Springer-Verlag, 2. edition, 2000.

[14] Libero Dinoi and Sergio Benedetto. Design of fast-prunable s-random interleavers. *IEEE Transactions on Wireless Communications, Vol. 4m No. 5*, pages 2540–2548, September 2005.

[15] D. Divsalar and F. Pollara. On the design of turbo codes. *TDA Progress Report 42-123*, November 1995.

[16] C. Douillard, M. Jézéquel, C. Berrou, N. Brengarth, J. Tousch, and N. Pham. The turbo code standard for DVB-RCS. *in Proc. 2nd International Symposium on Turbo Codes and Related Topics, Brest, France*, pages 535–538, September 2000.

[17] Dejan Drajic. FER and channel interleavers in 3G (WCDMA) systems using turbo-coding. *IEEE Telecommunications in Modern Satellite, Cable and Broadcasting Services, Vol. 2*, pages 407–410, September 2005.

[18] G. David Forney Jr. The viterbi algorithm. *Proceedings of the IEEE, Vol. 61, No. 3*, pages 268–278, March 1973.

[19] G. David Forney Jr. The viterbi algorithm: A personal history. *arXiv.org:cs/0504020*, March 2005.

[20] Marc P. C. Fossorier, Frank Burkert, Shu Lin, and Joachim Hagenauer. On the equivalence between SOVA and the Max-Log-MAP decodings. *IEEE Communications Letters, Vol. 2, Nr. 5*, pages 137–139, May 1998.

[21] Christine Fragouli and Richard D. Wesel. Semi-random interleaver design criteria. *IEEE Global Telecommunications Conference, Vol. 5*, pages 2352–2356, 1999.

[22] Robert G. Gallager. *Low-Density Parity-Check Codes*. Cambridge : M.I.T. Press, 1963.

[23] Roberto Garello, Franco Chiarakuce, Paola Pierleoni, Marco Scaloni, and Sergio Benedetto. On error floor and free distance of turbo codes. *IEEE International Conference on Communications, Vol. 1*, pages 45–49, June 2001.

[24] Roberto Garello, Paola Pierleoni, and Segio Benedetto. Computing the free distance of turbo codes and serially concatenated codes with interleacers: Algorithm and applications. *IEEE Journal on Selected Areas in Communications, Vol. 19, No. 5*, pages 800–812, May 2001.

[25] Stèphane Y. Le Goff and Faisal Obaid Al-Ayyan. Design of bit-interleaved turbo-coded modulations. *IEEE Electronics Letters, Vol. 37, No. 16*, pages 1030–1031, August 2001.

[26] Stèphane Y. Le Goff and Faisal Obaid Al-Ayyan. On the design of bit-interleaved turbo-coded modulation. *IEEE Proceedings Information Theory Workshop*, pages 73 – 75, September 2001.

[27] Joachim Hagenauer. Source-controlled channel decoding. *IEEE Transactions on Communications, Vol. 43, No. 9*, pages 2449–2457, September 1995.

[28] Joachim Hagenauer. The EXIT chart - introduction to extrinsic information transfer in iterative processing. *Proceedings of the 12th European Signal Proceeding Conference (EUSIPCO)*, September 2004.

[29] Joachim Hagenauer and Peter Hoeher. A Viterbi decoding algorithm with soft-decision output and its applications. *IEEE Global Conference on Communications*, pages 1680–1686, November 1989.

[30] Joachim Hagenauer, Elke Offer, and Lutz Papke. Iterative decoding of binary block and convolutional codes. *IEEE Transactions on Information Theory. Vol. 42, No. 2*, pages 429–445, March 1996.

[31] Raymond Hill. *A First Course in Coding Theory*. Clarendon press, Oxford, 2002.

[32] Sheryl L. Howard and Christian Schlegel. Differential turbo-coded modulation with APP channel estimation. *IEEE Transactions on Communications, Vol. 54, Nr. 8*, pages 1397–1406, August 2006.

[33] Axel Huebner, Kamil Sh. Zigangirov, and Daniel J. Costello Jr. A new cycle-based joint permutor design for multiple turbo codes. *IEEE Transactions on Communications, Vol. 54, No. 6*, pages 961–965, June 2006.

[34] Ewald Hueffmeier, Janak Sodha, and Stephen Wicker. Termination bits in the BCJR algorithm. *Proceedings of Third International Symposium on Communication Systems Networks and Digital Signal Processing*, pages 101–104, July 2002.

[35] Thomas Johansson and Fredrik Jönsson. Fast correlation attacks based on turbo code techniques. *Lecture Notes in Computer Science*, 1666:181–197, 1999.

[36] Thomas Johansson and Fredrik Jönsson. Improved fast correlation attacks on stream ciphers via convolutional codes. *Lecture Notes in Computer Science*, 1592:347+, 1999.

[37] Daniel J. Costello Jr. Free distance bounds for convolutional codes. *IEEE Transactions on Information Theory, Vol. It-20, No. 3*, pages 356–365, May 1974.

[38] Joakim Grahl Knudsen. Master thesis: Randomised construction and dynamic decoding of LDPC codes, Department of Informatics, Bergen, December 2005.

[39] Lie-Nan Lee, A. Roger Hammons Jr., Feng-En Sun, and Mustafa Eroz. Application and standardization of turbo codes in third-generation high-speed wireless data services. *IEEE Transactions on Vehicular Technology. Vol. 49, No. 6.*, pages 2198–2207, November 2000.

[40] Shu Lin and Daniel J. Costello Jr. *Error Control Coding*. Pearson Higher Education, 2. edition, 2003.

[41] Renato R. Lopes and John R. Barry. The soft-feedback equalizer for turbo equalization of highly dispersive channels. *IEE Transactions on Communications, Vol. 54, Nr. 5*, pages 783–788, May 2006.

[42] Kouraichi M, Ben Belghith O., Kachouri A., and Kamiun L. Evaluation of SOVA algorithm in turbo code. *IEEE Control, Communication and Signal processing*, pages 659–663, 2004.

[43] Arya Mazumdar, A. K. Chaturvedi, and Adrish Banerjee. Construction of turbo code interleavers from 3-regular hamiltonian graphs. *arXiv:cs.IT/0512093 v1*, December 2005.

[44] W. Meier and O. Staffelbach. Fast correlation attacks on stream ciphers. In *Lecture Notes in Computer Science on Advances in Cryptology-EUROCRYPT'88*, pages 301–314, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[45] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[46] Risto Nordman. Application of the Berrou SOVA algorithm in decoding of a turbo code. *European Transactions on telecommunications, Vol. 14*, pages 245–254, 2003.

[47] Lance C. Perez, Jan Seghers, and Daniel J. Costello Jr. A distance spectrum interpretation of turbo codes. *IEEE Transactions of Information Theory, Vol. 42, No. 6*, pages 1698–1709, November 1996.

[48] Phillipe Piret. *Convolutional Codes : An Algebraic Approach*. Cambridge : M.I.T. Press, 1988.

[49] Jessica Pursley. Turbo product codes and channel capacity. *IEEE Southeast Conference (SECON) student paper competition*, 2001.

[50] Carey Radebaugh and Ralf Koetter. Wheel codes: Turbo-like codes on graphs of small order. *Proceedings. IEEE Information Theory Workshop*, pages 78–81, March - April 2003.

[51] Patrick Robertson, Emmanuelle Villebrun, and Peter Hoeher. A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain. *IEEE Communications Conference, Vol. 2*, pages 1009–1013, 1995.

[52] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. McCraw - Hill, 5. edition, 2003.

[53] Eirik Rosnes and Øyvind Ytrehus. Turbo decoding on the binary erasure channel: Finite-length analysis and turbo stopping sets. *arXiv:cs.IT/0602072 v1 20 Feb 2006*, February 2006.

[54] Eirik Rosnes and Øyvind Ytrehus. Improved algorithms for the determination of turbo-code weight distributions. *IEEE Transactions on Communications, Vol. 53, No. 1*, pages 20–26, January 2005.

[55] Eirik Rosnes and Øyvind Ytrehus. Turbo stopping sets: the uniform interleaver and efficient enumeration. *Proceedings. International Symposium on Information Theory*, pages 1251–1255, September 2005.

[56] Eirik Rosnes and Øyvind Ytrehus. On the design of bit-interleaved turbo-coded modulation with low error floors. *IEEE Transactions on Communications, Vol. 54, No. 9*, pages 1563–1573, September 2006.

[57] Jonghoon Ryo and Oscar Y. Takeshita. On quadratic inverses for quadratic permutation polynomials over integer rings. *IEEE Transactions on Information Theory, Vol. 52, No. 3*, pages 1254–1260, March 2006.

[58] H. R. Sadjadpour. Maximum a posteriori decoding algorithms for turbo codes. In R. M. Rao, S. A. Dianat, and M. D. Zoltowski, editors, *Proc. SPIE Vol. 4045, p. 73-83, Digital Wireless Communication II, Raghuveer M. Rao; Soheil A. Dianat; Michael D. Zoltowski; Eds.*, pages 73–83, July 2000.

[59] C. E. Shannon. *A mathematical theory of communication*. Urbana : University of Illinois Press, 1948.

[60] T. Siegenthaler. Decrypting a class of stream ciphers using ciphertext only. *IEEE Transactions on Computers, Vol. 34, No. 1*, pages 81–85, January 1985.

[61] T. Siegenthaler. Correlation-immunity of nonlinear combining functions for cryptographic applications. *IEEE Transactions on Information Theory, Vol. 30, No. 5*, pages 776–780, September 1984.

[62] Marcin Sikora and Daniel J. Costello Jr. A new SISO algorithm with application to turbo equalization. *arXiv:cs.IT/0504017 v1 6. April 2005*, April 2005.

[63] Bernard Sklar. Fundamentals of turbo codes. *http://www.informit.com/articles*, 2002.

[64] Bernard Sklar. A primer on turbo code concepts. *IEEE Communications Magazine, Vol. 35, no. 12*, pages 94–102, December 1997.

[65] Jing Sun and Oscar Y. Takeshita. Interleavers for turbo codes using permutation polynomials over integer rings. *IEEE Transactions on Information Theory, Vol. 51, No. 1*, pages 101–119, January 2005.

[66] Oscar Y. Takeshita and Daniel J. Costello Jr. New deterministic interleaver designs for turbo codes. *IEEE Transactions on Information Theory, Vol. 46, Issue 6*, pages 1988–2006, Sept. 2000.

[67] Andrew S. Tanenbaum. *Computer Networks*. Pearson Education, Inc., Prentice Hall, 4. edition, 2003.

[68] Stephan ten Brink. Convergence behavior of iteratively decoded parallel concatenated codes. *IEEE Transactions on Communications, Vol. 49, No. 10*, pages 1727 – 1737, October 2001.

[69] Christian Skauge Knudtsen.
master thesis: Rotasjonsinvariante turbokoder (norwegian).
www.ub.uib.no/elpub/2003/h/413002/Hovedoppgave.pdf, Department of Informatics, Bergen, June 2003.

[70] Jason P. Woodard and Lajos Hanzo. Comparative study of turbo decoding techniques: An overview. *IEEE Transactions on vehicular Technology, Vol. 49, No. 6*, pages 2208–2232, November 2000.

[71] Yan-Ziu Zheng and Yu T. Su. Inter-block permuted turbo codes. *arXiv:cs.IT/0602020 v1*, February 2006.

[72] Yan-Ziu Zheng and Yu T. Su. A turbo coding system for high speed communications. *arXiv:cs.IT/060395 v1*, March 2006.