Master's Thesis

# Error performance of turbo codes

**Per Ola Ingvarsson**
**and**
**Henrik Svenell**

Dec 18, 1998

**Abstract**

In recent years iterative decoding has regained popularity, with the remarkable results presented in a paper by a group of French researchers. They introduced a new family of convolutional codes, nicknamed "Turbo codes" after the resemblance with the turbo engine. A turbo code is built from a parallel concatenation of two recursive systematic codes linked together by nonuniform interleaving. Decoding is done iteratively by two separate a posteriori probability decoders, each using the decoding results from the other one. For sufficiently large interleaver sizes, the error correction performance seems to be close to Shannon's theoretical limit.

In this Master's Thesis we examine the performance of turbo-codes on the additive white Gaussian noise channel. The influence of the size of the encoder memory, different types and sizes of interleavers are examined together with two different decoding algorithms, the one-way algorithm and the two-way algorithm. We show that the two algorithms have the same performance and that the choice of interleaver and encoder is important.

# Contents

# Chapter 1

# Introduction

When transmitting digital information over a noisy channel, it is demanded that the user can retrieve data with high fidelity or even with no errors. The simplest way to protect data from corruption is to increase the transmitting power or the so-called signal-to-noise ratio (SNR). However, this is expensive and in some way impractical. For example, the price for increasing SNR with one decibel is to increase the transmission power with about 25%. In a satellite communication system this could cost millions of dollars as the higher effect probably means higher weight of the satellite. An alternative and more efficient way to solve the problem is to use error-control coding, which increases the reliability of the communication by adding redundancy to the information. The redundancy can then be used to detect or correct errors.

In 1993 a new coding scheme, called turbo codes by its discoverers, a group of French researchers, was introduced. It was one of the most important developments in coding theory for many years. The main advantages of this method, when used together with an iterative decoding scheme, is low complexity in spite of high performance, which makes it suitable for mobile communication. It is therefore part of the standard for the third-generation mobile telecommunications systems.

The main purpose of this Masters Thesis is to study the bit-error performance of turbo codes on the additive white Gaussian noise channel. The influence of the encoder memory size (2,4 and 6), different interleavers (block and random interleavers) and decoding algorithms (two-way and one-way) are examined.

The report is organized as follows: The second section, Information transmission, is an introduction to digital communication systems and to coding theory. Section 3 describes convolutional encoders and in particular the recursive systematic encoders. In Section 4 we look at different interleavers and Section 5 describes the turbo encoder. In Section 6, the iterative decoder and the a posteriori decoder, are described. Section 7 describes our implementation of the different algorithms. Section 8 contains the simulation results for various combinations of turbo codes and the last section consists of a summary and the conclusions. In Appendix A tables of our simulations are shown.

# Chapter 2

# Information transmission

## 2.1 Racing towards the Shannon limit

The history of error-control coding and *information theory* began in 1948 when Claude E. Shannon published his famous paper "A Mathematical Theory of Communication" [Sha48]. In his paper Shannon showed that every communication channel has a parameter $C$ (measured in bits per second), called the *channel capacity*. If the desired data transmission rate $R_t$ (also measured in bits per second) of the communication system is less than $C$, it is possible to design a communication system, using *error-control coding*, whose probability of errors in the output is as small as desired. If the data transmission rate is larger than $C$ it is not possible to make the error probability tend towards zero with any code. In other words: channel noise establishes a limit for the transmission rate, but not for transmission reliability. Shannon's theory also tells us that it is more economical to make use of a good code than trying to build a good channel, e.g., increasing the signalling power. We must note that Shannon did not tell us how to find suitable codes, his achievement was to prove that they exist.

## 2.2 A digital communication system

Figure 2.1 shows the functional diagram of a digital communication system. It consists of an *encoder*, a *channel* and a *decoder*. The source information, $\mathbf{x}$, can be either an analog signal, such as an audio or video signal, or a digital signal, e.g., computer communication. Without loss of generality we can assume that the source is discrete in time, since, according to the sampling theorem, any time continuous signal can be completely reconstructed if the original continuous signal was sampled with a sampling frequency twice the highest frequency in the signal.


Figure 2.1: A communication system.

In his paper Shannon showed that the problem of sending information from a source to a destination over a channel always can be divided into two subproblems. The first is to represent

the output from the information source as a sequence of binary digits. This is called *source coding*. The other subproblem is to map the information sequence into a binary sequence suitable for sending over the channel. This is called *channel coding*. The main advantage of this *separation principle* is that it is possible to use the same channel for different sources without reconstructing the channel coding.

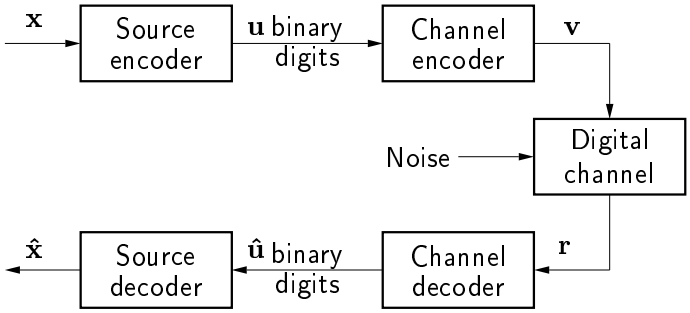



Figure 2.2: A communication system with the encoder and decoder divided into two subcoders.

Figure 2.2 shows our communication system divided according to Shannon's separation principle. The discrete signal from the signal source is fed into the *source encoder* whose purpose is to represent the source output using as few binary digits as possible. In other words, it removes as much redundancy as possible from the source output. The output from the source encoder is called the *information sequence*, $\mathbf{u}$. It is fed into the *channel encoder* whose purpose is to add, in a controlled manner, some redundancy to the binary information sequence. The resulting sequence is called the *code sequence*, $\mathbf{v}$. The code sequence travels from the encoder to the decoder through the channel. This can be from one place to another, e.g., between a mobile phone and a base station, or from one time to another, e.g., in a tape recorder or a CD player. On the channel the signal always will be subjected to distortion, noise or fading. The purpose of the *channel decoder* is to correct the errors in the *received sequence*, $\mathbf{r}$, that was introduced by the channel, using the redundancy that was introduced by the channel encoder. The output of the channel decoder is the estimated code sequence, $\hat{\mathbf{u}}$. From this sequence the *source decoder* reconstructs the original source sequence.

## 2.3 The discrete time channel

In the previous section we indicated that the channel provides the connection between the transmitter and the receiver. The physical channel can be a pair of wires that carry an electrical signal or an optical fiber that carries the signal on a modulated light beam. It can be free space over which the signal is radiated by an antenna or another media such as magnetic tapes or disks. In either case we cannot directly send digital information on the channel, thus we need a way to transform the digital signal into a useful analog waveform, see Figure 2.3. The transformation from the digital signal to the analog waveform is called modulation. The modulator transforms every code word to an analog waveform of duration $T_s$. Some of the most common modulation methods are *pulse amplitude modulation* (PAM), *phase shift keying* (PSK) and *frequency shift keying* (FSK) [Lin97]. Besides these modulation methods there are a number of more advanced modulation methods, e.g., the *Gaussian minimum shift keying*

(GMSK) used in GSM. In this thesis we will only consider transmission using binary phase shift keying (BPSK) modulation. This means that the modulator generates the waveform

$$s_0(t) = \begin{cases} \sqrt{\frac{2E_s}{T_s}} \cos \omega t & , 0 \leq t < T_s \\ 0 & , otherwise \end{cases} \tag{2.1}$$

for the input 0 and $s_1(t) = -s_0(t)$ for the input 1. Here, $E_s$ is the signal energy and $T_s$ is the duration of the signal. The modulated signal then enters the channel.

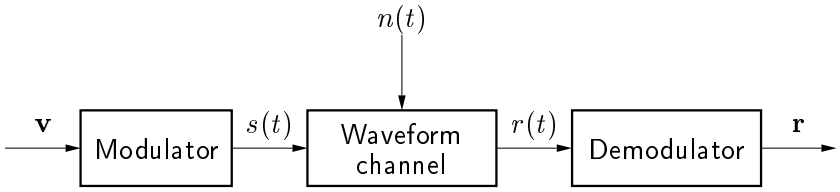



Figure 2.3: A decomposition of a digital communication channel.

Since the channel introduces noise, the modulated code words, transmitted through the channel, are corrupted. In this thesis we will only consider the *additive white Gaussian noise* (AWGN) channel. The AWGN noise, $n(t)$, introduced by the channel is Gaussian, with zero-mean and a two-sided spectral density of $N_0/2$, i.e.,

$$\begin{aligned} E[n(t)] &= 0 \\ V[n(t)] &= \frac{N_0}{2} \end{aligned} \tag{2.2}$$

and it is added to the modulated signal. The received signal is $r(t) = s(t) + n(t)$. The *demodulator* processes the channel-corrupted transmitted waveform and produces a sequence of numbers, $\mathbf{r} = r_1 r_2 \ldots$, that represent estimates of the transmitted data symbols. This is done using a *matched filter*, i.e., it calculates the convolution between the received symbol and the matched filter

$$r_i = \sqrt{\frac{2}{T_s}} \int_0^{T_s} r(t) \cos(\omega t) dt = \pm\sqrt{E_s} + n_i,$$

where $n_i$ is the additive noise disturbance. Thus we have

$$r_i \in N(\pm\sqrt{E_s}, \sqrt{N_0/2}). \tag{2.3}$$

The sequence $\mathbf{r}$ is then fed into the channel decoder, see Figure 2.2.

The *signal-to-noise ratio* (SNR) is a measure of the quality of the channel and is defined as the average ratio of the energy of the desired signal to the energy of the noise signal, $E_b/N_0$. It is often measured in dB. If we are sending an uncoded BPSK signal, the signal energy $E_s$ is equal to the bit energy $E_b$. By using a sign decision rule we get, for the uncoded case, the bit-error probability

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right),$$

where

$$Q(x) = \int_x^\infty \frac{1}{\sqrt{2\pi}} \exp^{-y^2/2}\, dy$$

is the complementary error function of Gaussian statistics [Lin97].

## 2.4 A short introduction to coding theory

Error-control coding has taken two major directions: *block codes* and *convolutional codes*. As the rest of the report deals with convolutional codes we here give a short introduction to block codes.

### 2.4.1 Block codes

In the block coding case, the sequence of information bits coming from the source encoder are divided into blocks of length $K$. These blocks are called *messages*. There are $2^K$ distinct messages at the input of the encoder. The block *encoder* maps each distinct $K$-tuple of information into an $N$-tuple of code, i.e., the *code word*. A binary (N,K) block code $\mathcal{B}$ is a set of $M = 2^K$ binary N-tuples. N is called the *block length* and the ratio

$$R = \frac{K}{N} \tag{2.4}$$

is called the *code rate*. In order to be able to correct or detect errors $K$ has to be less than $N$. The $N - K$ extra added symbols are called the *parity-check symbols*.

Other important properties of a block code are the *Hamming weight*, the *Hamming distance* and the *minimum distance*. The Hamming weight, $w_H(\mathbf{x})$, is defined as the number of non-zero elements in the codeword $\mathbf{x}$ and the Hamming distance, $d_H(\mathbf{x}, \mathbf{y}) = w_H(\mathbf{x} - \mathbf{y})$, is the number of positions where $\mathbf{x}$ differs from $\mathbf{y}$. The minimum distance, $d_{min}$, is defined as the minimum Hamming distance between any pair of code words. It determines the *error correction capability* $t$ of the code as

$$t = \lfloor \frac{d_{min} - 1}{2} \rfloor.$$

The block code guarantees correction of the errors if the received sequence differs in $t$ or less positions from the correct code word. In some cases it might also be able to correct errors that differs in more than $t$ positions.

Another commonly used property for a code is *linearity*. A code is linear if a bitwise modulo-2 addition of two code words results in another code word. The all-zero word is always a code word in a linear code.

**Example 2.1:** Table 2.1 shows the famous binary (7,4) Hamming block code with one of the possible mappings from information sequence to code sequence. The rate of the code is $R = 4/7$.

The Hamming weight of the code word 1000011 is 3 and of 0001111 it is 4. The Hamming

| Message | Code word |
|---|---|
| 0000 | 0000000 |
| 0001 | 0001111 |
| 0010 | 0010110 |
| 0011 | 0011001 |
| 0100 | 0100101 |
| 0101 | 0101010 |
| 0110 | 0110011 |
| 0111 | 0111100 |
| 1000 | 1000011 |
| 1001 | 1001100 |
| 1010 | 1010101 |
| 1011 | 1011010 |
| 1100 | 1100110 |
| 1101 | 1101001 |
| 1110 | 1110000 |
| 1111 | 1111111 |

Table 2.1: The binary (7,4) Hamming block code.

distance between the code words 1000011 and 0001111 is 3. The minimum distance $d_{min}$ of the code $\mathcal{B}$ is 3 as the minimum Hamming distance for each pair of code words is 3. The number of errors that can be corrected is $t = 1$, which means that all errors with Hamming weight 1 can be corrected. For example if we receive the code word 1110110 we know that it probably is the code word 1100110 that has been sent and we decode it as $\hat{u} = 1100$.

■

# Chapter 3

# Convolutional coding

## 3.1 The general convolutional encoder

In a convolutional encoder the information bits $\mathbf{u} = u_0 u_1 \ldots$ are not separated into blocks, as in the case of block codes, but instead they form a semi-infinite sequence that is shifted symbol-wise into a shift register, see Figure 3.1. The encoder consists of shift registers and modulo-2 adders. The memory, $m$, of the encoder is the number of delay (D) elements in the shift registers. For the encoder in Figure 3.1 the memory $m$ equals 2. The output of the encoder is the modulo-2 sum of the values in different elements. The output symbols are then fed into a parallel to serial converter, the serializer.

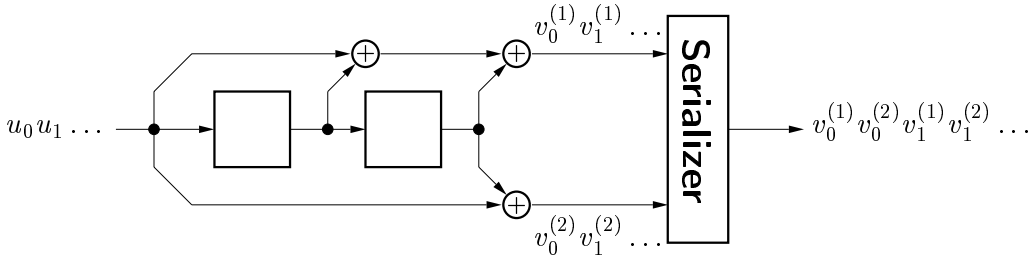



Figure 3.1: An encoder for a binary rate $R = 1/2$ convolutional code.

The number of input bits to the encoder at each time unit is equal to $b$. The input bits form the information sequence

$$\mathbf{u} = \mathbf{u_0 u_1} \ldots = u_0^{(1)} u_0^{(2)} \ldots u_0^{(b)} u_1^{(1)} u_1^{(2)} \ldots u_1^{(b)} \ldots . \tag{3.1}$$

In a similar way the number of output bits from the encoder at a time unit is equal to $c$ and they form, after serialization, the code sequence

$$\mathbf{v} = \mathbf{v_0 v_1} \ldots = v_0^{(1)} v_0^{(2)} \ldots v_0^{(c)} v_1^{(1)} v_1^{(2)} \ldots v_1^{(c)} \ldots . \tag{3.2}$$

The rate $R$ of the encoder is defined as $R = b/c$ and it describes how much redundancy that has been added. The encoder in Figure 3.1 has one input ($b = 1$) and two output bits ($c = 2$), which means that the rate is $R = 1/2$.

The content of the shift registers is called the state, $\sigma$, of the encoder and the number of states equals $2^m$. The state describes the past history of the encoder. For the encoder in Figure 3.1, at time $t$, we have $\sigma_t = u_{t-1}u_{t-2}$, i.e., the state is the input bits which entered the memory at the two previous time units. The current state together with the input symbols are sufficient to determine the next state and the output symbols. A so-called *state-transition diagram* can be drawn to illustrate this. It shows what the next state and the output will be given a certain state and input. The state-transition diagram for the encoder in Figure 3.1 is shown in Figure 3.2. As seen in Figure 3.2 there is a one-to-one correspondence between the input symbol and the next state, given the previous state.

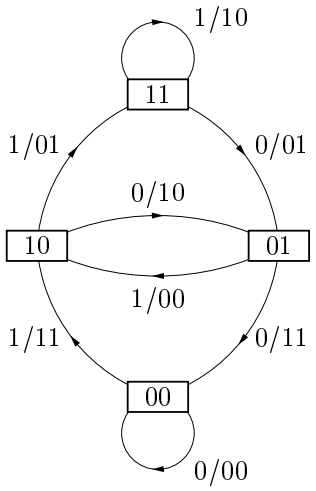



Figure 3.2: The state-transition diagram for the encoder in figure 3.1.

Some of the definitions for block codes are also applicable for convolutional codes. All convolutional codes are linear. The Hamming weight and the Hamming distance are defined in the same way as for block codes. The distance measure between two code sequences of a convolutional code is called the *free distance*, defined as the minimum Hamming distance between any two differing code sequences,

$$d_{free} = \min_{\mathbf{v} \neq \mathbf{v}'} d_H(\mathbf{v}, \mathbf{v}'). \tag{3.3}$$

**Example 3.1:** Consider the encoder in Figure 3.1 and let the input sequence be $\mathbf{u} = 1100$. If the initial state is $\sigma_0 = 00$ then it follows from the state-transition diagram in Figure 3.2 that the state sequence is

$$\sigma_{00} \xrightarrow{1/11} \sigma_{10} \xrightarrow{1/01} \sigma_{11} \xrightarrow{0/01} \sigma_{01} \xrightarrow{0/11} \sigma_{00}$$

and that the code sequence is

$$\mathbf{v} = 11010111$$

■

The state-transition diagram can be extended to a *trellis diagram* by adding a time axis to the state-transition diagram, see Figure 3.3. The trellis diagram can for example be used to

find the minimum distance or free distance of a code. Some decoding algorithms use a trellis representation to decode convolutional codes [VO79].

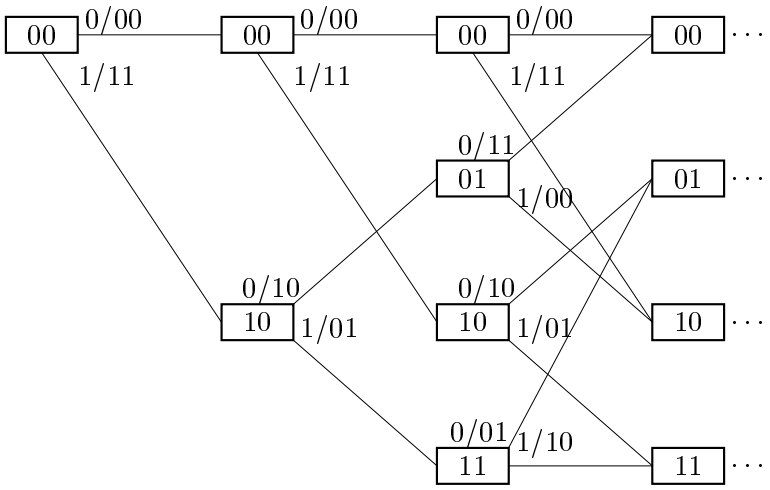



Figure 3.3: A binary rate $R = 1/2$ trellis structure for the encoder in Figure 3.1.

Now we introduce the *delay operator D*. Multiplying a sequence with D is equivalent to delaying the sequence one time unit. This gives us another way of representing the information and the code sequence, i.e.,

$$\begin{aligned} \mathbf{u}(D) =& \ldots + \mathbf{u_{-1}} D^{-1} + \mathbf{u_0} + \mathbf{u_1} D^1 + \mathbf{u_2} D^2 + \ldots \\ \mathbf{v}(D) =& \ldots + \mathbf{v_{-1}} D^{-1} + \mathbf{v_0} + \mathbf{v_1} D^1 + \mathbf{v_2} D^2 + \ldots \end{aligned} \tag{3.4}$$

where $\mathbf{u}_i = u_i^{(1)} u_i^{(2)} \ldots u_i^{(b)}$ and $\mathbf{v}_i = v_i^{(1)} v_i^{(2)} \ldots v_i^{(c)}$.

We know that the there exists a linear operator which transforms the information sequence into the code sequence, i.e., the encoder, which can be represented in matrix form as

$$\mathbf{v}(D) = \mathbf{u}(D) G(D) \tag{3.5}$$

where $G(D)$ is called the *generator matrix*. Obviously we need to be able to reconstruct the information sequence, i.e., $G(D)$ must have a right inverse. If the right inverse exists, the generator matrix is called an *encoder matrix*.

**Example 3.2:** Consider the encoder in Figure 3.1. The generator matrix for the encoder is

$$G(D) = \begin{pmatrix} 1 + D + D^2 & 1 + D^2 \end{pmatrix}.$$

The output sequence $\mathbf{v}(D) = \left(\mathbf{v}^{(1)}(D) \mathbf{v}^{(2)}(D)\right)$ of the encoder with generator matrix $G(D)$ can be written as

$$\begin{aligned} \mathbf{v}^{(1)}(D) =& \mathbf{u}(D)(1 + D + D^2), \\ \mathbf{v}^{(2)}(D) =& \mathbf{u}(D)(1 + D^2). \end{aligned} \tag{3.6}$$

■

A generator matrix $G(D)$ is *equivalent* to another generator matrix $G'(D)$ if the same code sequence can be generated by rearranging the information symbols, i.e., $G(D) = f(D)G'(D)$.

## 3.2 The recursive systematic encoder

In the previous section we discussed the convolutional encoder in general. In this section we will describe a special convolutional encoder that is used in the turbo coding scheme. This convolutional encoder is called *recursive systematic* encoder, see Figure 3.4.

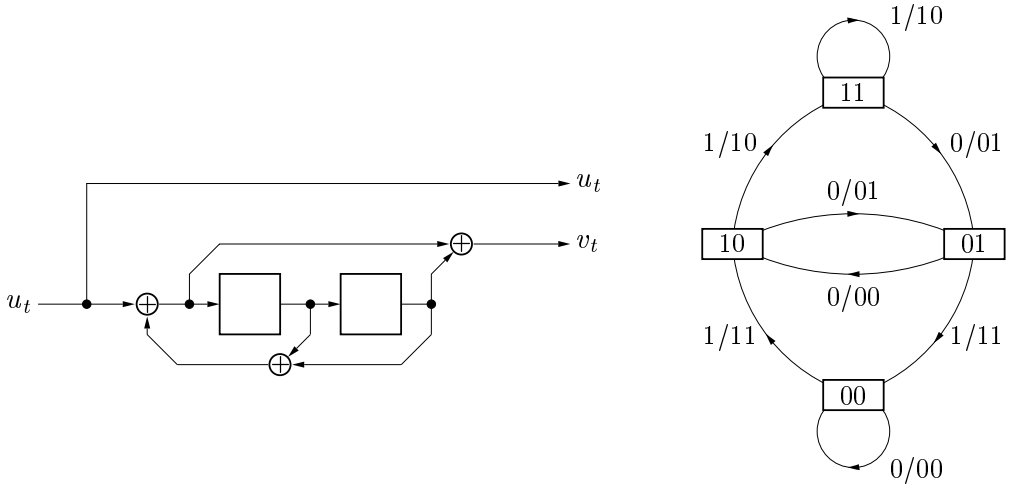



Figure 3.4: A rate $R = 1/2$ recursive systematic convolutional encoder and its state-transition diagram.

A *systematic* encoder has the property that the $b$ information symbols appear unchanged among the $c$ code symbols along with $c - b$ parity-check symbols. In Figure 3.4 the first symbol of $\mathbf{v}_t$ is the information symbol, i.e., $v_t^{(1)} = u_t$, and the second, $v_t^{(2)}$, is the parity-check symbol. Since the $b$ information symbols appear unchanged in the code sequence and we can always permute the columns in $G(D)$, i.e., rearrange the order of the code sequence and still obtain an *equivalent* convolutional code. Hence, we can write the recursive systematic encoder as

$$G(D) = \begin{pmatrix} I_b & R(D) \end{pmatrix} \tag{3.7}$$

where $I_b$ is the $b \times b$ identity matrix and $R(D)$ is a $(c - b) \times b$ rational matrix.

Since the shift register has a feedback loop, it is called *recursive*. The feedback loop corresponds to the denominator in the $R(D)$ matrix.

**Example 3.3:** The encoder in Figure 3.4 has the generator matrix

$$G(D) = \begin{pmatrix} 1 & \frac{1+D^2}{1+D+D^2} \end{pmatrix}$$

and it is equivalent to the encoder in Figure 3.1 since

$$G'(D) = f(D)G(D) = (1+D+D^2)\begin{pmatrix} 1 & \frac{1+D^2}{1+D+D^2} \end{pmatrix} = \begin{pmatrix} 1+D+D^2 & 1+D^2 \end{pmatrix}.$$

■

Another way of representing the encoder is to use the *parity check matrix*, $H$. This method is suitable for describing turbo encoders and will be used in Chapter 5.1. A code sequence satisfies the condition $\mathbf{v}H^T = 0$, where we call $H^T$ *the syndrome former matrix.* Since the convolutional code word satisfies $\mathbf{v}(D) = \mathbf{u}(D)G(D)$, then

$$\mathbf{v}(D)H(D)^T = \mathbf{u}(D)G(D)H(D)^T = 0 \tag{3.8}$$

and since $\mathbf{u}(D) \neq 0$ in general we have

$$G(D)H(D)^T = 0 \tag{3.9}$$

**Example 3.4:** To find the parity-check matrix $H(D)$ for the encoder in Figure 3.4 or for the equivalent encoder in Figure 3.1 we need to satisfy the condition $G(D)H(D)^T = 0$

$$G(D)H(D)^T = \begin{pmatrix} 1 & \frac{1+D^2}{1+D+D^2} \end{pmatrix}\begin{pmatrix} 1+D^2 \\ 1+D+D^2 \end{pmatrix} = 0.$$

This gives us

$$H(D) = \begin{pmatrix} 1+D^2 & 1+D+D^2 \end{pmatrix}$$

■

The syndrome former matrix $H(D)^T$ can be expanded as

$$H(D)^T = H_0^T + H_1^T D + \ldots + H_{m_s}^T D^{m_s}, \tag{3.10}$$

where $H_i^T, 0 \leq i \leq m_s$, is a $c \times (c-b)$ matrix and $m_s$ is the memory of the syndrome former, which in general is not the same as the encoder memory $m$.

Combining (3.10) with the equality $\mathbf{v}H^T = 0$ we get

$$\mathbf{v}_t H_0^T + \mathbf{v}_{t-1} H_1^T + \cdots + \mathbf{v}_{t-m_s} H_{m_s}^T = 0. \tag{3.11}$$

For causal code sequences we have

$$H^T = \begin{pmatrix} H_0^T & H_1^T & \ldots & H_{m_s}^T & & \\ & H_0^T & H_1^T & \ldots & H_{m_s}^T & \\ & & \ddots & \ddots & & \ddots \end{pmatrix} \tag{3.12}$$

which is the semi-infinite syndrome former matrix.

**Example 3.4 (cont.):** The memory of the syndrome former is $m_s = 2$ and

$$\begin{aligned} H_0 &= (\ 1\ \ 1\ ) \\ H_1 &= (\ 0\ \ 1\ ) \\ H_2 &= (\ 1\ \ 1\ ) \end{aligned}$$

The corresponding semi-infinite syndrome former matrix is

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

■

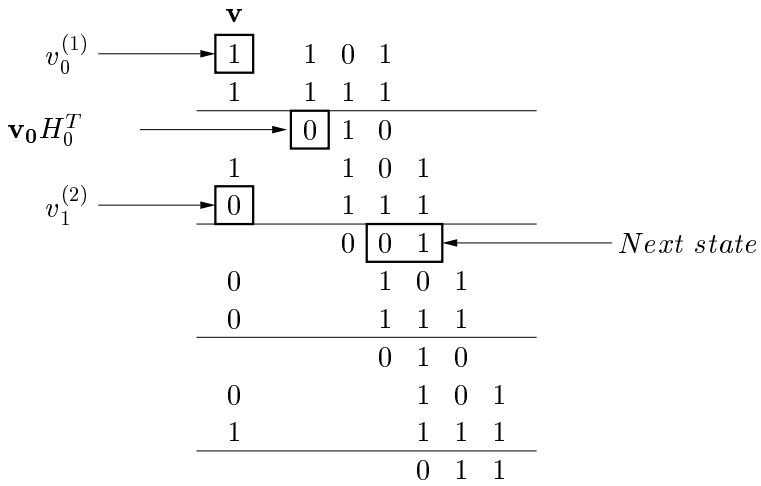
Using (3.11) we get the following semi-infinite equation systems

$$\begin{aligned} &\mathbf{v}_0 H_0^T = 0 \\ &\mathbf{v}_0 H_1^T + \mathbf{v}_1 H_0^T = 0 \\ &\mathbf{v}_0 H_2^T + \mathbf{v}_1 H_1^T + \mathbf{v}_2 H_0^T = 0 \\ &\vdots \\ &\mathbf{v}_0 H_{m_s}^T + \ldots + \mathbf{v}_{m_s-1} H_1^T + \mathbf{v}_{m_s} H_0^T = 0 \\ &\mathbf{v}_1 H_{m_s}^T + \ldots + \mathbf{v}_{m_s} H_1^T + \mathbf{v}_{m_s+1} H_0^T = 0 \\ &\vdots \end{aligned} \tag{3.13}$$

Since the encoder is systematic we know that the code word $\mathbf{v}_i$ consists of $b$ information symbols and $c - b$ parity-check symbols. The first vector equation in (3.13) gives us $c - b$ scalar equations which can be solved by substitution, i.e., the $c - b$ parity-check symbols in $\mathbf{v}_0$ can be calculated. By using the next part of (3.13) we again have $c - b$ unknown parity-check symbols in $\mathbf{v}_1$ and $c - b$ equations which can be solved, etc.

The following example describes how the parity-check symbols are calculated and a way of representing the state of the encoder.

**Example 3.4 (cont.):** Consider the same encoder again and the information sequence $\mathbf{u} = 1100$. The first parity-check symbol is equal to the first information symbol, i.e., $v_0^{(1)} = u_0 = 1$. The second parity-check symbol $v_0^{(2)}$ is calculated by solving $\mathbf{v}_0 H_0^T = 1 \cdot 1 \oplus 1 \cdot v_0^{(2)} = 0 \Rightarrow v_t^{(0)} = 1$. Next we calculate the state of the encoder which represents the past equations. The state is $\mathbf{v}_0 H_1^T = 1$ and $\mathbf{v}_0 H_2^T = 0$. The next parity-check symbol, $v_1^{(2)}$, is calculated using the state $\sigma = 10$, and the information bit $u_1 = v_1^{(1)} = 1$, etc.




From the matrix above we see that the state sequence is

$$\sigma_{00} \xrightarrow{1/11} \sigma_{10} \xrightarrow{1/10} \sigma_{10} \xrightarrow{0/00} \sigma_{10} \xrightarrow{0/01} \sigma_{11}$$

and that the code sequence is $\mathbf{v} = 11100001$. This is the same result as we would get if we used the state-transition diagram in Figure 3.4

■

# Chapter 4

# The interleaver

The interleaver is a device that rearranges, or permutes, the input bits in a predefined manner. Ideally, two symbols that are close to each other in time in the input should be far apart in the output. Normally an interleaver is used to transform burst errors, that can occur in the coded signal, into single errors. A burst error is when several consecutive bit-errors occur. The turbo coding scheme uses the interleaver to design a longer and more complex encoder. One of the goals when designing interleavers for turbo codes is to re-map the information sequence so that at least one of parity-check sequences always has a high Hamming weight. The design of the interleaver is a very important part in the effectiveness of the turbo coding system.

Interleavers can be divided into two general classes, block interleavers, which we mainly will discuss here, and convolutional interleavers. The difference between them can be described as that a block interleaver takes a block of symbols and then rearranges them while the convolutional interleaver continuously rearranges the symbols.

The opposite of the interleaver is the *deinterleaver* which takes interleaved sequence as its input and produces the original sequence as its output.

## 4.1 Block interleavers

### 4.1.1 The "classical" block interleaver

The simplest interleaver is the *"classical" block interleaver*. It uses two memories with $I$ rows and $J$ columns each. The input symbols are written row by row and then read out column by column. The reason for using two memories is that we want to be able to read from one memory while writing into the other. The delay of the interleaver is $I \times J$.

**Example 4.1:** Consider the interleaver in Figure 4.1 and the input sequence $\mathbf{x} = x_0 x_1 \dots x_7$. First the symbols $x_0 \dots x_3$ are written into the first memory of the encoder. When $x_4$ is written, $x_0$ is read and when $x_5$ is written $x_2$ is read, etc. The output will be $\mathbf{x}' = x_0 x_2 x_1 x_3 x_4 x_6 x_5 x_7$.

■

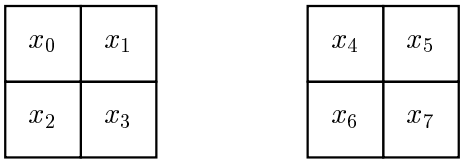



Figure 4.1: Two memories of a $2 \times 2$ block interleaver with the symbols $\mathbf{x} = x_0 x_1 \ldots x_7$ written into it.

### 4.1.2 The pseudo-random block interleaver

In the *pseudo-random* block interleaver data is written into a memory in a pseudo-random order and read out column by column. Here we also use two memories so that we can read from one memory while we write to the other one.

**Example 4.2:** Consider the pseudo-random interleaver in Figure 4.2 and the input se-

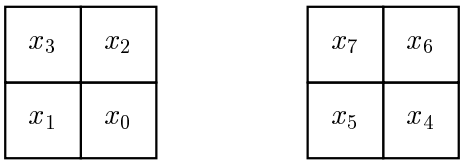



Figure 4.2: Two memories of a $2 \times 2$ random interleaver with the symbols $\mathbf{x} = x_0 x_1 \ldots x_7$ written into it.

quence $\mathbf{x} = x_0 x_1 \ldots x_7$. In the figure the symbols are written into the interleaver in a pseudo-random way. Then they are read out column by column and get the output sequence $x_3 x_1 x_2 x_0 x_7 x_5 x_6 x_4$.

■

### 4.1.3 The multi stage interleaver (MIL)

The *multi-stage interleaver* (MIL), is a method used in the ARIB standard proposal [ARI98] to describe a pseudo-random interleaver using three rules. By combining the rules complex pseudo-random interleavers can be described.

The first rule is

$$L[N \times M].$$

It describes an $N$ rows by $M$ columns block interleaver with the size $L$. If $L < MN$ the last $MN - L$ positions of the interleaver will not be used.

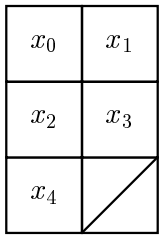



Figure 4.3: The input symbols interleaved in a $5[3 \times 2]$ interleaver.

**Example 4.3:** Consider the input symbols $\mathbf{x} = x_0x_1x_2x_3x_4$ and the interleaver defined by the rule $5[3 \times 2]$, see Figure 4.3. Since $5 < 3 \times 2$ the last position will not be used. The symbols are written row by row into the interleaver. They are then read out column by column and the output is $x_0x_2x_4x_1x_3$.

■

The second rule,

$$R\{A\}$$

describes an interleaver that reverses the order of $A$ input symbols.

**Example 4.4:** The input symbols $\mathbf{x} = x_0x_1x_2x_3x_4$ are going to be interleaved with the interleaver described by the rule $R\{5\}$. The output is $x_4x_3x_2x_1x_0$.

■

The third rule used in MIL is

$$L[N1 \times M1, N2 \times M2, \dots],$$

which means that the first $L$ input symbols should be permuted using an $L[N1 \times M1]$ interleaver, the second $L$ input symbols should be interleaved using an $L[N2 \times M2]$ interleaver, etc.

**Example 4.5:**

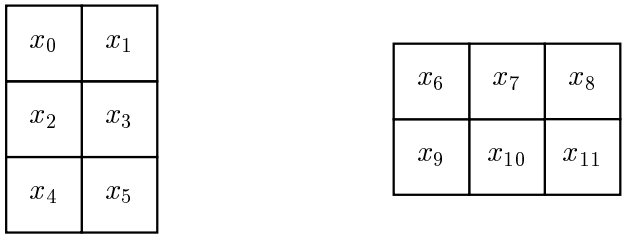



Figure 4.4: The first 6 input symbols written into a $6[3 \times 2]$ interleaver and the next 6 symbols written into a $6[2 \times 3]$ interleaver.

Consider the input symbols $\mathbf{x} = x_0x_1x_2 \ldots x_{11}$ and the rule $6[3 \times 2, 2 \times 3]$. The first 6 symbols will be interleaved in a $3 \times 2$ block interleaver and the next 6 symbols will be interleaved using a $2 \times 3$ block interleaver. The output is $x_0x_2x_4x_1x_3x_5x_6x_9x_7x_{10}x_8x_{11}$.

■

Now we can combine the three rules and get a multi-stage interleaver.

$$L[R1 \times R2]$$

is a block interleaver with the size $L$ where the row indices are interleaved using the interleaver defined by $R1$ and the column indices are interleaved using the interleaver defined by $R2$. $R1$ and $R2$ can be a combination of any of the three rules above.

**Example 4.6:** The input sequence $\mathbf{x} = x_0x_1x_2 \ldots x_6$ is to be permuted with $7[R\{3\} \times 3[2 \times 2]]$.

We start by interleaving the row indices, $j_0j_1j_2$, with $R\{3\}$. The interleaved result is $j_2j_1j_0$. Then we expand the rule $3[2 \times 2]$ and interleave the column indices, $i_0i_1i_2$, which permutes into $i_0i_2i_1$.

With $\mathbf{i}$ as column indices and $\mathbf{j}$ as row indices we write the sequence into a $3 \times 3$ interleaver, see Figure 4.5. The symbols are then read out column by column and the output is $x_6x_3x_0x_5x_2x_4x_1$.

■

| | $i_0$ | $i_2$ | $i_1$ |
|---|---|---|---|
| $j_2$ | $x_6$ | | |
| $j_1$ | $x_3$ | $x_5$ | $x_4$ |
| $j_0$ | $x_0$ | $x_2$ | $x_1$ |

Figure 4.5: The interleaved indexes $\mathbf{i}$ used as column indexes in a $3 \times 3$ block interleaver.

## 4.2 Convolutional interleavers

The difference between a *convolutional interleaver* and a block interleaver is that the block interleaver takes a block of symbols and permutes them while the convolutional interleaver rearranges the symbols continuously (cf. convolutional vs. block encoders). In Figure 4.6 a convolutional interleaver-deinterleaver pair is depicted. The interleaver consists of shift registers of different lengths on the interleaver and deinterleaver side and multiplexors that choose where the symbols go. All the multiplexors change position synchronously after each

symbol so that successive encoder outputs enter different rows of the interleaver memory. Since the different rows have different lengths, different symbols will get different delays. The first encoder output enters the top interleaver row and is transmitted over the channel immediately. Then it enters the top row of the deinterleaver memory where it is delayed $(I-1)j$ time units. The second encoder output symbol enters the second row of the interleaver and is delayed $j$ time units on the interleaver side. Thus adjacent encoder outputs are transmitted $j$ time units apart and not affected by the same channel error burst. After passing through both the interleaver and the deinterleaver, all the symbols have the same delay.

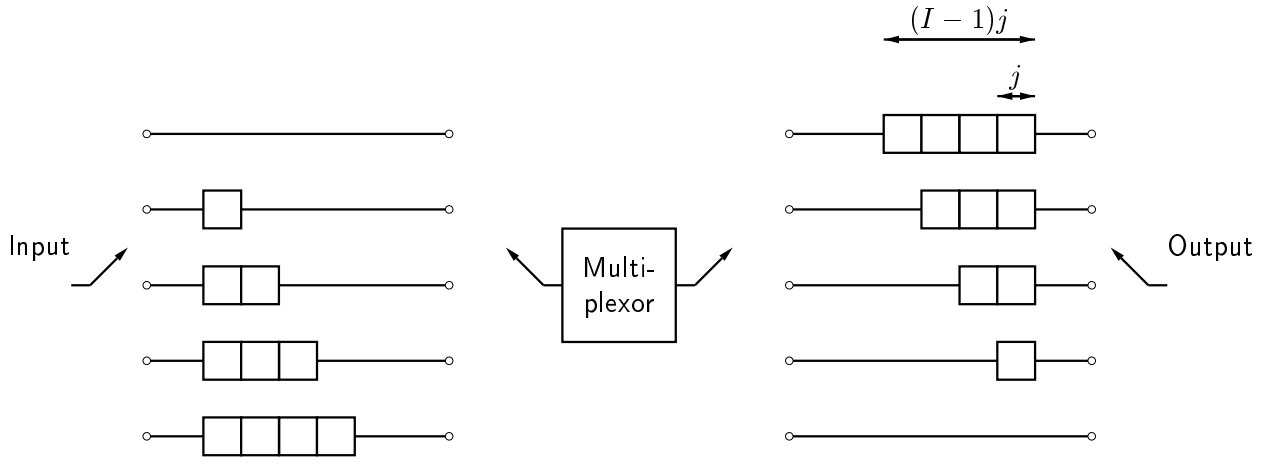



Figure 4.6: A convolutional interleaver and deinterleaver.

## 4.3 Matrix representation of the interleaver

The interleaver can be expressed as a matrix called a *scrambling matrix* or *scrambler*. If $\mathbf{x}$ is the bi-infinite binary input sequence of an interleaver then the output $\mathbf{y}$ can be expressed as $\mathbf{y} = \mathbf{x}S$, where $S$ is the scrambler.

A bi-infinite matrix $S = (s_{ij}),\ i, j \in \mathbb{Z}$, that has one 1 in each row and one 1 in each column and satisfies the causality condition

$$s_{ij} = 0, i < j \tag{4.1}$$

is called a *convolutional scrambler*.

The *identity scrambler* has ones only along the diagonal. The output symbols will not be permuted.

**Example 4.7:** Consider the block interleaver in Figure 4.7. Both memories of the interleaver are shown in the figure. The input symbols are written into the interleaver row by row in one of the memories and read out column by column from the other. When the fifth symbol

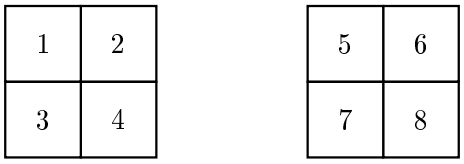



Figure 4.7: A $2 \times 2$ block interleaver.

is written into the second memory, the first symbol is read from the first memory and when the sixth symbol is written, the third is read, etc. The scrambling matrix representation for the interleaver is

$$S = \begin{pmatrix}
\ddots & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & \\
1 & & & & 1 & & & & & & & & & \\
2 & & & & & & 1 & & & & & & & \\
3 & & & & & 1 & & & & & & & & \\
4 & & & & & & & 1 & & & & & & \\
5 & & & & & & & & 1 & & & & & \\
6 & & & & & & & & & & 1 & & & \\
7 & & & & & & & & & 1 & & & & \\
8 & & & & & & & & & & & 1 & & \\
9 & & & & & & & & & & & & 1 & \\
 & & & & & & & & & & & & & \ddots
\end{pmatrix}.$$

The one at $i = 1$, $j = 5$ corresponds to the reading of the first symbol when the fifth is written and the one at $i = 3$, $j = 6$ corresponds to that the third symbol is read when the sixth is written, and so on. The empty positions denote zeros.

■

A random interleaver can be constructed, using a method by Jimenez and Zigangirov [JZ97], by taking an $n \times n$ diagonal matrix and permuting the columns in a random fashion. This way we will still have exactly one 1 in each column and one 1 in each row. Then we unwrap the submatrix, which is below the diagonal, as shown in Figure 4.8. The unwrapped matrix is then repeated indefinitely to form a scrambling matrix.

In Chapter 5.1 we will use the scrambler together with the syndrome former to describe the turbo codes. For this we need some definitions.

A *multiple convolutional* scrambler is a bi-infinite matrix $S = (s_{ij})$, $i, j \in \mathbb{Z}$, that has *at least* one 1 in each row and one 1 in each column and that satisfies the causality condition (4.1). Since there is more than one 1 in each row, the multiple convolutional scrambler not only permutes the symbols, but it also makes copies of them. If all the rows have the same number of ones, the scrambler is *homogeneous*.

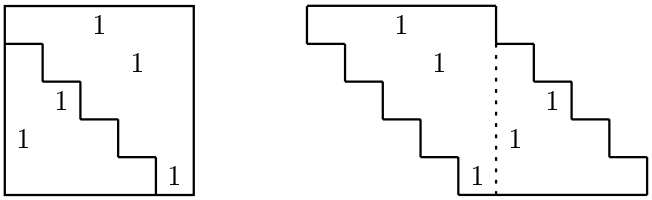


Figure 4.8: A $5 \times 5$ matrix before and after unwrapping.

Consider the non-multiple scrambling matrices $S^{(1)} = (s_{ij}^{(1)})$ and $S^{(2)} = (s_{ij}^{(2)})$. The matrix

$$S = S^{(1)} \boxplus S^{(2)} = (s_{ij}), \quad i, j \in \mathbb{Z}$$

is called-column interleaved if

$$\begin{cases} s_{i(2j)} & = s_{ij}^{(1)} \\ s_{i(2_j+1)} & = s_{ij}^{(2)} \end{cases}$$

for all $i, j \in \mathbb{Z}$. This means that every other column in the matrix $S$ comes from $S^{(1)}$ and $S^{(2)}$ respectively.

**Example 4.8:** Consider the scramblers $S^{(1)}$ and $S^{(2)}$, where $S^{(1)}$ is the identity scrambler.

$$S^{(1)} = \begin{pmatrix} \ddots & 0 & 1 & 2 & 3 & 4 & \\ 0 & 1 & & & & & \\ 1 & & 1 & & & & \\ 2 & & & 1 & & & \\ 3 & & & & 1 & & \\ 4 & & & & & 1 & \\ & & & & & & \ddots \end{pmatrix} \quad \text{and} \quad S^{(2)} = \begin{pmatrix} \ddots & 0 & 1 & 2 & 3 & 4 & 5 & \\ 0 & & & & & 1 & & \\ 1 & & & 1 & & & & \\ 2 & & & & & & 1 & \\ 3 & & & & 1 & & & \\ 4 & & & & & & & \\ & & & & & & & \ddots \end{pmatrix}$$

To make $S = S^{(1)} \boxplus S^{(2)}$ we take column 0 from $S^{(1)}$ and insert into column $2 \cdot 0 = 0$ in $S$. Next we take column 0 from $S^{(2)}$ and insert it into column $2 \cdot 0 + 1 = 1$ of $S$ and so on. The column-interleaved matrix $S = S^{(1)} \boxplus S^{(2)}$ is

$$S = \begin{pmatrix} & \mathbf{0_1} & \mathbf{0_2} & \mathbf{1_1} & \mathbf{1_2} & \mathbf{2_1} & \mathbf{2_2} & \mathbf{3_1} & \mathbf{3_2} & \mathbf{4_1} & \mathbf{4_2} & \mathbf{5_1} & \mathbf{5_2} & \\ \ddots & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & \\ 0 & 1 & & & & & & & & & 1 & & & \\ 1 & & & 1 & & & 1 & & & & & & & \\ 2 & & & & & 1 & & & & & & & 1 & \\ 3 & & & & & & & 1 & 1 & & & & & \\ 4 & & & & & & & & & 1 & & & & \\ & & & & & & & & & & & & & \ddots \end{pmatrix}. \tag{4.2}$$

The bold indices in the top row indicate which column of the matrices $S^{(1)}$ and $S^{(2)}$ the columns in $S$ come from.

■

If the input sequence of the convolutional scrambler consists of subblocks of $c$ binary symbols and the output sequence consists of subblocks of $d$ binary symbols (see Chapter 5.1), it is convenient to divide the scrambler matrix into $c \times d, d \geq c$ submatrices $S_{ij},\ i, j \in \mathbb{Z}$ so that $S = (S_{ij})$. Since each column of $S$ has one 1, the rows will have in average $d/c$ ones. The ratio $d/c$ is called the rate of the scrambler.

**Example 4.8 (cont.):** The column-interleaved matrix $S$ can be divided into submatrices of size $1 \times 2$, where one column of the submatrix comes from $S^{(1)}$ and the other from $S^{(2)}$. Thus the rate of $S$ is $2/1$.

■

We need one more definition to be able to describe the turbo code.

Consider the two matrices $S^{(1)} = \left(S_{ij}^{(1)}\right)$ and $S^{(2)} = \left(S_{ij}^{(2)}\right)$ whose submatrices are of sizes $c_1 \times d_1$ and $c_2 \times d_2$, respectively. The matrix

$$S = S^{(1)} \boxplus S^{(2)} = (S_{ij}),\ i, j \in \mathbb{Z} \tag{4.3}$$

is called *row-column* interleaved if

$$\begin{cases} S_{(2i)(2j)} & = S_{ij}^{(1)} \\ S_{(2i)(2j+1)} & = 0 \\ S_{(2i+1)(2j)} & = 0 \\ S_{(2i+1)(2j+1)} & = S_{ij}^{(2)} \end{cases} \tag{4.4}$$

for all $i, j \in \mathbb{Z}$.

**Example 4.9:** Consider the column-interleaved scrambler (4.2) and call it $S^{(1)}$. By row-column interleaving it with two identity scramblers, $S^{(2)}$ and $S^{(3)}$ we get the rate $4/3$ scrambler

$$S = \left(\begin{array}{cc|cccccccccccccccccccc}
 & & \mathbf{0_1} & \mathbf{1_1} & \mathbf{0_2} & \mathbf{0_3} & \mathbf{2_1} & \mathbf{3_1} & \mathbf{1_2} & \mathbf{1_3} & \mathbf{4_1} & \mathbf{5_1} & \mathbf{2_2} & \mathbf{2_3} & \mathbf{6_1} & \mathbf{7_1} & \mathbf{3_2} & \mathbf{3_3} & \mathbf{8_1} & \mathbf{9_1} & \mathbf{4_2} & \mathbf{4_3} \\
 & \ddots & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \\
\hline
\mathbf{0_1} & 0 & 1 & & & & & & & & & & & & & & & & & 1 & & \\
\mathbf{0_2} & 1 & & & 1 & & & & & & & & & & & & & & & & & \\
\mathbf{0_3} & 2 & & & & 1 & & & & & & & & & & & & & & & & \\
\mathbf{1_1} & 3 & & & & & 1 & & & & & 1 & & & & & & & & & & \\
\mathbf{1_2} & 4 & & & & & & & 1 & & & & & & & & & & & & & \\
\mathbf{1_3} & 5 & & & & & & & & 1 & & & & & & & & & & & & \\
\mathbf{2_1} & 6 & & & & & & & & & 1 & & & & & & & & & & & \\
\mathbf{2_2} & 7 & & & & & & & & & & & 1 & & & & & & & & & \\
\mathbf{2_3} & 8 & & & & & & & & & & & & 1 & & & & & & & & \\
\mathbf{3_1} & 9 & & & & & & & & & & & & & 1 & 1 & & & & & & \\
 & & & & & & & & & & & & & & & & & & & & & \ddots
\end{array}\right). \tag{4.5}$$

The bold row indices show which matrices the rows come from and analogously, the bold column indices show which matrices the columns come from.

■

# Chapter 5

# The turbo encoder

## 5.1 The turco encoder

In order to achieve high coding gains with moderate decoder complexity, concatenation has proven to be an attractive scheme. A concatenated code consists of two separate codes which are combined to form a large code. Concatenation of error control codes was first studied by David G. Forney in 1966 [For66]. Classically, concatenation consisted in cascading a block encoder with a convolutional encoder in a serial structure with an interleaver (see Chapter 4) separating them. Typically the block encoder was used as outer encoder, to protect against burst errors, and the convolutional encoder as inner encoder, to reduce the bit error.

A new scheme was introduced by a group of French researchers, Berrou, Glavieux and Thitimajshima in 1993 [BGT93]. They used a parallel concatenated scheme which got the nickname turbo coding after the resemblance with the turbo engine. When the French researchers decoded a rate $R = 1/3$ turbo code with a so-called *iterative decoding* algorithm they claimed that a bit error of $10^{-5}$ could be achieved at a SNR of 0.7dB. This was initially met by the coding community with much skepticism, but when the result was reproduced by other researchers the community was convinced.

The turbo encoder is built up by two recursive systematic encoders and an interleaver, in a parallel concatenated manner, see Figure 5.1. The first encoder takes the information sequence and produces the first parity-check sequence, $\mathbf{v}^{(1)}$. The second encoder takes the interleaved version of the information sequence and produces the second parity-check sequence, $\mathbf{v}^{(2)}$. The two parity-check sequences together with the information sequence, $\mathbf{v}^{(0)} = \mathbf{u}$, form the output of the turbo encoder, i.e.,

$$\mathbf{v} = \mathbf{v_0 v_1} \ldots = v_0^{(0)} v_0^{(1)} v_0^{(2)} v_1^{(0)} v_1^{(1)} v_1^{(2)} \ldots .$$

## 5.2 Parity-check matrix representation of the turbo encoder

A more general way of representing the turbo encoder is by using a parity-check matrix representation. This representation can also be used to describe low-density parity-check codes as described in [JZ98], in fact they show that turbo codes are special cases of low-density

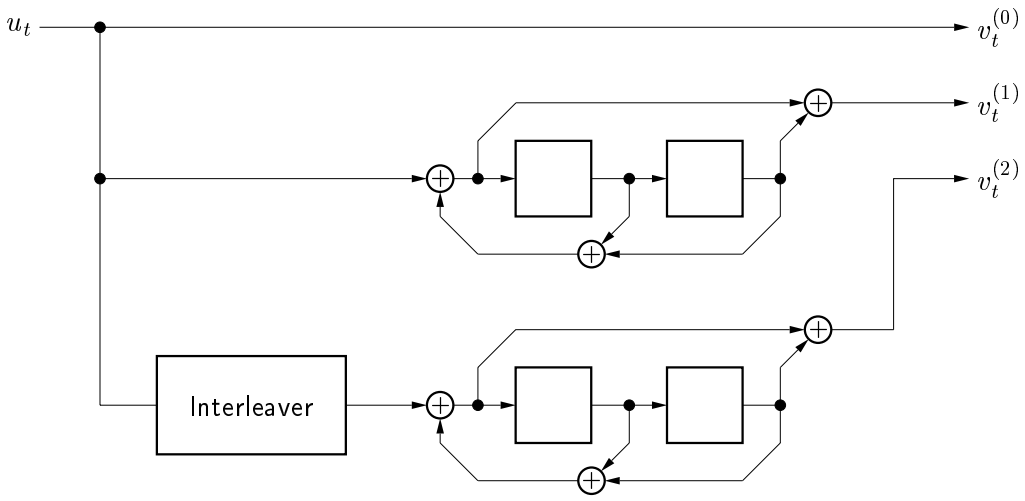



Figure 5.1: A rate $R = 1/3$, systematic, parallell concatenated convolutional encoder.

parity-check codes.

Since each of the recursive systematic encoders can be described with a parity-check matrix as described in Chapter 3.2 and the interleaver can be described with a scrambling matrix, described in Chapter 4.3 we can combine these and construct a representation of the turbo encoder.

We use the rate $R = 4/3$ row-column-interleaved convolutional scrambler in (4.5), designed from two identity scrambling matrix $S^{(2)}$ and $S^{(3)}$ together with another scrambling matrix $S^{(1)}$, corresponding to the interleaver in the turbo coder. The scrambling matrix $S$ transforms the input sequence $(u_t v_t^{(1)} v_t^{(2)})$ to $(u_t u'_t v_t^{(1)} v_t^{(2)})$ where $u'_t$ is the interleaved version of $u_t$.

We use a combination of two row-column interleaved parity-check matrices, with submatrices of size $2 \times 1$ to design the combined parity-check matrix, $H_{cc}^T$.

$$H_{cc}^T = \begin{pmatrix} \ddots & & & & & & & \\ & 1 & 0 & 1 & & & & \\ & 1 & 1 & 1 & & & & \\ & & 1 & 0 & 1 & & & \\ & & 1 & 1 & 1 & & & \\ & & & 1 & 0 & 1 & & \\ & & & 1 & 1 & 1 & & \\ & & & & 1 & 0 & 1 & \\ & & & & 1 & 1 & 1 & \\ & & & & & & & \ddots \end{pmatrix} \tag{5.1}$$

which corresponds to a parity-check matrix with inputs $(u_t, v_t^{(1)}, u'_t, v_t^{(2)})$. To make it correspond to $(u_t, u'_t, v_t^{(1)}, v_t^{(2)})$ we swap row 2 and 3, 6 and 7 and so on. We obtain

$$H_{cc}^{'T} = \begin{pmatrix} \ddots & & & & & & & & \\ & 1 & & 0 & & 1 & & & \\ & & 1 & & 0 & & 1 & & \\ & 1 & & 1 & & 1 & & & \\ & & 1 & & 1 & & 1 & & \\ & & & 1 & & 0 & & 1 & \\ & & & & 1 & & 0 & & 1 \\ & & & 1 & & 1 & & 1 & \\ & & & & 1 & & 1 & & 1 \\ & & & & & & & & \ddots \end{pmatrix} \tag{5.2}$$

We can now design a parity-check matrix which describes the complete turbo encoder by multiplying $S$ with the combined parity-check matrix $H_{cc}^{'T}$ to get

$$\mathbf{H}_{tu}^{T} = S\mathbf{H}_{cc}^{'T} \tag{5.3}$$

Where $H_{tu}^{T}$ is the total parity-check matrix of the turbo encoder.

# Chapter 6

# Iterative decoding

## 6.1 General principles

This section describes a decoding method called *iterative decoding*. Iterative decoding is the preferred decoding method for the turbo coding scheme as simulations show that a remarkably good performance can be achieved, close to the Shannon limit, despite the relatively low complexity of the iterative algorithm.

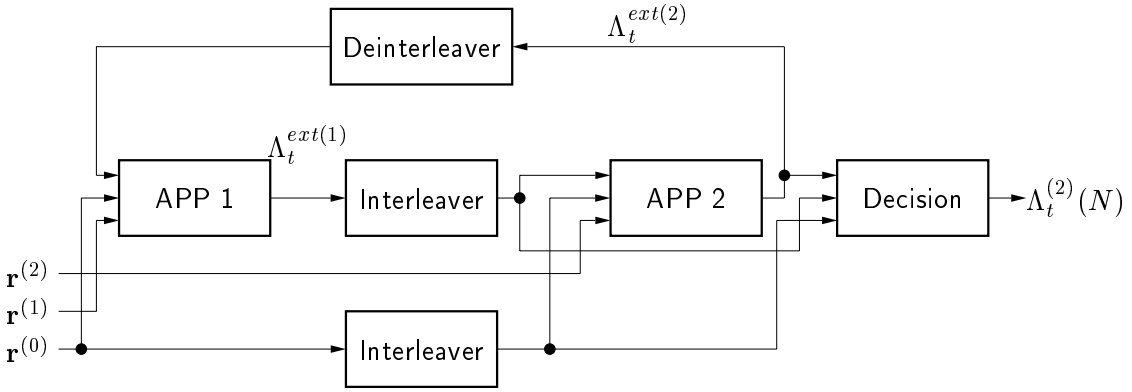



Figure 6.1: Iterative decoding procedure.

The fundamental idea of iterative decoding is that two or more a posteriori probability (APP) decoders exchange soft information, see Figure 6.1. One of the decoders calculates the a posteriori probability distribution of the information sequence and passes that information to the next decoder. The new decoder makes use of the information and computes its own version of the probability distribution. This exchange of information is called an iteration. After a certain number of iterations, $N$, a decision is made at the second decoder. For each iteration the probability that we decode in favour of the correct decision will improve.

Let

$$\mathbf{r} = \mathbf{r}_0\mathbf{r}_1 \ldots = r_0^{(0)} r_0^{(1)} r_0^{(2)} r_1^{(0)} \ldots \tag{6.1}$$

denote the received sequence corresponding to the code sequence generated by the turbo encoder in Figure 5.1 and let $\mathbf{r}_{\not l}^{(0)}$ denote the sequence of received symbols corresponding to

the information sequence $\mathbf{u}$ *except* the received symbol $r_t^{(0)}$, i.e.,

$$\mathbf{r}_{\not t}^{(0)} \stackrel{def}{=} r_0^{(0)} r_1^{(0)} r_2^{(0)} \dots r_{t-1}^{(0)} r_{t+1}^{(0)} \dots . \tag{6.2}$$

Now let

$$\pi_t(0) \stackrel{def}{=} P(u_t = 0) \tag{6.3}$$

denote the a priori probability that the information symbol $u_t = 0$.

Each iteration of the iterative decoding algorithm is executed in two phases. Let $\pi_t^{(l)}(i)$, $l = 1, 2$, $i = 1 \dots N$ denote the a posteriori probability, $P(u_t = 0 | \mathbf{r}_{[0,t)})$, which is obtained in the $l$th phase of the $i$th iteration.

In the first phase of the first iteration, the first a posteriori probability decoder uses the a priori probability $\pi_t(0)$ and the received sequences $\mathbf{r}^{(0)}$ and $\mathbf{r}^{(1)}$ to calculate the a posteriori probability, $\pi_t^{(1)}(1) = P(u_t = 0 | \mathbf{r}^{(0)} \mathbf{r}^{(1)})$. This can be rewritten as

$$\pi_t^{(1)}(1) \stackrel{def}{=} P(u_t = 0 | r_t^{(0)} \mathbf{r}_{\not t}^{(0)} \mathbf{r}^{(1)}) = \frac{P(r_t^{(0)} | u_t = 0) \pi_t(0) P(\mathbf{r}_{\not t}^{(0)} \mathbf{r}^{(1)} | u_t = 0)}{P(r_t^{(0)} \mathbf{r}_{\not t}^{(0)} \mathbf{r}^{(1)})} \tag{6.4}$$

where we have separated the dependence on $r_t^{(0)}$. Since the channel is memoryless by assumption $P(r_t^{(0)} | u_t = 0)$ does not depend on the a priori information and $P(\mathbf{r}_{\not t}^{(0)} \mathbf{r}^{(1)} | u_t = 0)$ only depends on a priori probabilities $P(u_j = 0)$ for $j \neq t$.

Analogously we define the a posteriori probability that $u_t = 1$ as

$$P(u_t = 1 | \mathbf{r}^{(0)} \mathbf{r}^{(1)}) = 1 - \pi_t^{(1)}(1) = \frac{P(r_t^{(0)} | u_t = 1)(1 - \pi_t(0)) P(\mathbf{r}_{\not t}^{(0)} \mathbf{r}^{(1)} | u_t = 1)}{P(r_t^{(0)} \mathbf{r}_{\not t}^{(0)} \mathbf{r}^{(1)})}. \tag{6.5}$$

Let $\Lambda_t^{(1)}(1)$ denote the ratio of the a posteriori probabilities for the information symbols after the first phase of the first iteration.

$$\Lambda_t^{(1)}(1) = \frac{\pi_t^{(1)}(1)}{1 - \pi_t^{(1)}(1)}. \tag{6.6}$$

Combining this with (6.4) and (6.5) we get

$$\Lambda_t^{(1)}(1) = \frac{\pi_t(0)}{1 - \pi_t(0)} \frac{P(r_t^{(0)} | u_t = 0)}{P(r_t^{(0)} | u_t = 1)} \frac{P(\mathbf{r}_{\not t}^{(0)} \mathbf{r}^{(1)} | u_t = 0)}{P(\mathbf{r}_{\not t}^{(0)} \mathbf{r}^{(1)} | u_t = 1)}. \tag{6.7}$$

The ratio

$$\Lambda_t(0) = \frac{\pi_t(0)}{1 - \pi_t(0)} \tag{6.8}$$

is the likelihood ratio of the a priori probabilities. In practice we often have $\pi_t(0) = 1/2$ which gives $\Lambda_t(0) = 1$. The ratio

$$\Lambda_t^{int} = \frac{P(r_t^{(0)}|u_t = 0)}{P(r_t^{(0)}|u_t = 1)} \tag{6.9}$$

is called the *intrinsic likelihood ratio* for the information symbol $u_t$. Since we use BPSK-modulation over the AWGN channel we can calculate this as

$$\Lambda_t^{int} = \frac{P(r_t^{(0)}|u_t = 0)}{P(r_t^{(0)}|u_t = 1)} = \frac{\frac{1}{\sqrt{N_0\pi}} \cdot e^{\frac{-(r_t^{(0)} - \sqrt{Es})^2}{N_0}}}{\frac{1}{\sqrt{N_0\pi}} \cdot e^{\frac{-(r_t^{(0)} + \sqrt{Es})^2}{N_0}}} = e^{\frac{4r_t^{(0)}\sqrt{Es}}{N_0}}. \tag{6.10}$$

The intrinsic likelihood ratio does not change for a certain symbol during the iterations.

The last part of (6.7) is called the *extrinsic likelihood ratio* for the first phase of the first iteration, i.e.,

$$\Lambda_t^{ext(1)}(1) = \frac{P(\mathbf{r}_{\not}^{(0)}\mathbf{r}^{(1)}|u_t = 0)}{P(\mathbf{r}_{\not}^{(0)}\mathbf{r}^{(1)}|u_t = 1)}. \tag{6.11}$$

Thus we can rewrite (6.7)

$$\Lambda_t^{(1)}(1) = \Lambda_t(0)\Lambda_t^{int}\Lambda_t^{ext(1)}(1), t = 0, 1 \dots , \tag{6.12}$$

which is the outcome of the first phase of the first iteration.

During the second phase the a posteriori decoder calculates the likelihood ratios for the information sequence based on the interleaved version of $\mathbf{r}_t^{(0)}$ and on the received sequence $\mathbf{r}_t^{(2)}$ corresponding to the second parity-check sequence. If the interleaver is sufficiently large, the interleaved version of $\mathbf{r}_{\not}^{(0)}$ will be independent of the non-interleaved one used in (6.11). The decoder also exploits its knowledge of the likelihood ratios obtained from the first phase. Since the a priori likelihood ratio $\Lambda_t(0) = 1$ and the second decoder adds the intrinsic likelihood ratio to its output we only use the extrinsic likelihood ratio, $\Lambda^{ext(1)}(1)$, from the first phase as a priori input to the second phase.

The output from the second decoder is

$$\Lambda_t^{(2)}(1) = \Lambda_t(0)\Lambda_t^{int}\Lambda_t^{ext(1)}(1)\Lambda_t^{ext(2)}(1), t = 0, 1, \dots , \tag{6.13}$$

where

$$\Lambda_t^{ext(2)}(1) = \frac{P(\mathbf{r}_{\not}^{(0)}\mathbf{r}^{(2)}|u_t = 0)}{P(\mathbf{r}_{\not}^{(0)}\mathbf{r}^{(2)}|u_t = 1)}. \tag{6.14}$$

Then, in the same manner, we use the extrinsic information, $\Lambda_t^{ext(2)}(1)$, from the second phase of the first iteration as the a priori information to the first phase of the second iteration. As ouput we have

$$\Lambda_t^{(1)}(2) = \Lambda_t(0)\Lambda_t^{int}\Lambda_t^{ext(2)}(1)\Lambda_t^{ext(1)}(2). \tag{6.15}$$

After N iterations we make a decision based on the output from the extrinsic information from both the decoders together with the intrinsic information and the a priori information.

$$\Lambda_t^{(2)}(N) = \Lambda_t(0)\Lambda_t^{int}\Lambda_t^{ext(1)}(N)\Lambda_t^{ext(2)}(N), t = 0, 1, \dots , \tag{6.16}$$

If $\Lambda_t^{(2)}(N) > 1.0$ then we decode 0 and otherwise 1.

In the following sections we consider two different algorithms for a posteriori probability decoding, i.e., the two-way and the one-way algorithms. In contrast to the popular Viterbi and sequential decoding algorithms [JZ98] the APP decoding algorithms has not been used much in practical application because of their high complexity and long decoding delay. The Viterbi algorithm is a *maximum-likelihood* (ML) decoding algorithm for convolutional codes and its output is the most probable transmitted code path. It cannot provide us with information on individual bit error probabilities for the different information bits. The APP decoder is a *maximum a posteriori probability* (MAP) decoding algorithm which minimizes the bit error probability.

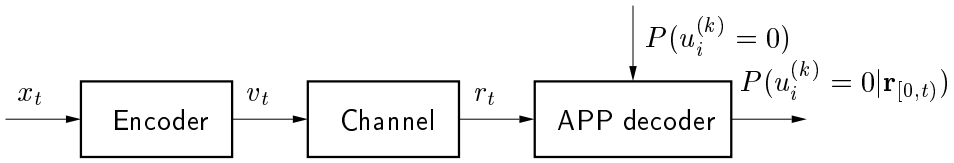



Figure 6.2: A digital communication system with APP decoding.

The purpose of the APP decoder is to find the *a posteriori* probability for each information bit given the *a priori* probability and the received symbols $\mathbf{r}_{[0,t)}$, see Figure 6.2. In the following subsections we assume that the a priori probability $P(u_i^{(k)} = 0) = 1/2$, i.e., the binary encoded bits are equally likely.

## 6.2 The Two-way or BCJR-algorithm

The two-way algorithm is the most celebrated APP decoding algorithm for terminated convolutional codes and it is often called the BCJR algorithm in current literature after the inventors Bahl, Cocke, Jelinek and Raviv [BCJR74].

By *terminated* convolutional codes we mean that the information sequence is divided into blocks of length $n$ and these blocks are followed by $m$ dummy symbols which return the encoder to the zero-state. This means that we always start and end the decoding in the zero-state. The number of dummy symbols, $m$, is the size of the encoder memory.

**Example 6.1:** Consider the *recursive systematic* encoder in Figure 6.3. Assume that we want to encode a block of length $n = 4$ binary information bits, e.g., $\mathbf{u} = 1000$. To return the encoder to the zero-state we need to add two dummy bits $\mathbf{u}_{dummy} = 11$. We start in the zero-state and visit the following states:

$$\boldsymbol{\sigma} = \sigma_0 \overset{1}{\rightarrow} \sigma_2 \overset{0}{\rightarrow} \sigma_3 \overset{0}{\rightarrow} \sigma_1 \overset{0}{\rightarrow} \sigma_2 \overset{1}{\rightarrow} \sigma_1 \overset{1}{\rightarrow} \sigma_0$$

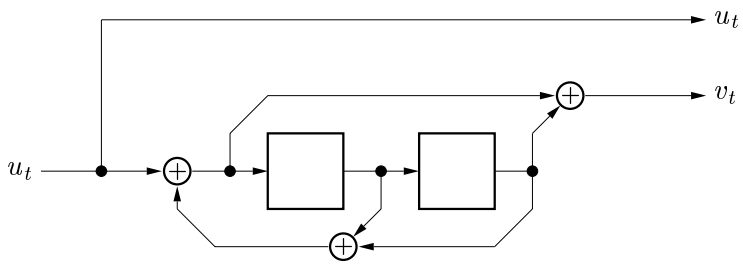



Figure 6.3: A recursive systematic convolutional encoder.

If the encoder was non-recursive we would just add zeros as dummy symbols since, in that case, the input symbols are shifted directly into the memory.

■

The two-way algorithm calculates the a posteriori probability $P(u_i = 0|\mathbf{r}_{[0,n+m)})$, which can be rewritten according to Bayes' rule in probability theory

$$P(u_i^{(k)} = 0|\mathbf{r}_{[0,n+m)}) = \frac{P(u_i^{(k)} = 0, \mathbf{r}_{[0,n+m)})}{P(\mathbf{r}_{[0,n+m)})} \tag{6.17}$$

The numerator is the joint probability that the sequence $\mathbf{r}_{[0,n+m)}$ is received and that the symbol $u_i^{(k)}$ is zero, given that the transmitted sequence is a code sequence. The denominator is the probability that we have received $\mathbf{r}_{[0,n+m)}$ given that the transmitted sequence is a code sequence. Equation (6.17) can be expressed by conditioning on the information sequence $\mathbf{u}_{[0,n)}$,

$$P(u_i^{(k)} = 0|\mathbf{r}_{[0,n+m)}) = \frac{\sum_{\mathbf{u}_{[0,n)} \in \mathcal{U}_{[0,n)i}^{(k)}} P(\mathbf{r}_{[0,n+m)}|\mathbf{u}_{[0,n)})P(\mathbf{u}_{[0,n)})}{\sum_{\mathbf{u}_{[0,n)} \in \mathcal{U}_{[0,n)}} P(\mathbf{r}_{[0,n+m)}|\mathbf{u}_{[0,n)})P(\mathbf{u}_{[0,n)})}, i = 0 \ldots n, k = 1 \ldots b \tag{6.18}$$

where $\mathcal{U}_{[0,n)i}^{(k)}$ is the set of information sequences that have $u_i^{(k)} = 0$ and $\mathcal{U}_{[0,n)}$ is the set of all information sequences. The information bits and the state-transition in the encoder has a one-to-one correspondence as discussed in Chapter 3.1. We define $\mathcal{S}_{[0,n+m)}$ as the set of state sequences such that we start and end in the zero-state

$$\mathcal{S}_{[0,n+m)} \stackrel{\text{def}}{=} \{\boldsymbol{\sigma}_{[0,n+m)} = \sigma_0\sigma_1 \ldots \sigma_{n+m} | \sigma_0 = \sigma_{n+m} = 0\} \tag{6.19}$$

and $\mathcal{S}_{[0,n+m)i}^{(k)}$ as

$$\mathcal{S}_{[0,n+m)i}^{(k)} \stackrel{\text{def}}{=} \{\boldsymbol{\sigma}_{[0,n+m)} = \sigma_0\sigma_1 \ldots \sigma_{n+m} | \sigma_0 = \sigma_{n+m} = 0, \sigma_i \to \sigma_{i+1} \Rightarrow u_i^{(k)} = 0\} \tag{6.20}$$

which is the set of state sequences that start and end in the zero state and where the state transition from state $\sigma_i$ to $\sigma_{i+1}$ corresponds to $u_i^{(k)} = 0$. We can now rewrite the a posteriori

probability once more

$$P(u_i^{(k)} = 0|\mathbf{r}_{[0,n+m)}) = \frac{\sum_{\boldsymbol{\sigma}_{[0,n+m)} \in \mathcal{S}_{[0,n+m)i}^{(k)}} P(\mathbf{r}_{[0,n+m)}|\boldsymbol{\sigma}_{[0,n+m)}) P(\boldsymbol{\sigma}_{[0,n+m)})}{\sum_{\boldsymbol{\sigma}_{[0,n+m)} \in \mathcal{S}_{[0,n+m)}} P(\mathbf{r}_{[0,n+m)}|\boldsymbol{\sigma}_{[0,n+m)}) P(\boldsymbol{\sigma}_{[0,n+m)})} \stackrel{\text{def}}{=} \frac{\gamma_i^{(k)}}{\gamma} \quad (6.21)$$

where $P(\mathbf{r}_{[0,n+m)}|\boldsymbol{\sigma}_{[0,n+m)})$ is the probability that $\mathbf{r}_{[0,n+m)}$ is received, given that the code corresponding to the state transitions $\boldsymbol{\sigma}_{[0,n+m)}$ was transmitted and $P(\boldsymbol{\sigma}_{[0,n+m)})$ is the a priori probability of the state sequence $\boldsymbol{\sigma}_{[0,n+m)}$. The numerator $\gamma_i^{(k)}$ is the sum of the probabilities that correspond to the state sequences $\mathcal{S}_{[0,n+m)i}^{(k)}$ and the denominator $\gamma$ is the sum of the probabilities corresponding to $\mathcal{S}_{[0,n+m)}$.

A more easy-to-grasp way of representing (6.21) is by using a matrix representation. Let $P_t$ denote the $2^m \times 2^m$ state transition matrix

$$P_t = (p_t(\sigma, \sigma'))_{\sigma,\sigma'} \quad (6.22)$$

where $2^m$ is the number of states and

$$p_t(\sigma, \sigma') = P(\mathbf{r}_t, \sigma_{t+1} = \sigma'|\sigma_t = \sigma) = P(\mathbf{r}_t|\sigma_{t+1} = \sigma', \sigma_t = \sigma) P(\sigma_{t+1} = \sigma'|\sigma_t = \sigma) \quad (6.23)$$

and $\sigma, \sigma' \in 0 \ldots 2^m - 1$. Since there is a one-to-one correspondence between the state transitions and the information sequence

$$P(\sigma_{t+1} = \sigma'|\sigma_t = \sigma) = \begin{cases} 1/2^b, & \text{if } \sigma \to \sigma' \text{ is possible} \\ 0, & \text{otherwise} \end{cases} \quad (6.24)$$

assuming $P(u_i^{(k)} = 0) = 1/2^b$ and $b$ is the number of input bits. This means that the state transition matrix $P_t$ is sparse as some of the matrix elements are zero.

**Example 6.2:** The recursive systematic encoder is Figure 6.3 with $m = 2$, has a sparse state-transition matrix with following appearance.

$$P_t = \begin{pmatrix} p_t(0,0) & 0 & p_t(0,2) & 0 \\ p_t(1,0) & 0 & p_t(1,2) & 0 \\ 0 & p_t(2,1) & 0 & p_t(3,2) \\ 0 & p_t(3,1) & 0 & p_t(3,3) \end{pmatrix}$$

For example the state transition $\sigma_0 \to \sigma_1$ does not exist $\Rightarrow p_t(0,1) = 0$.

■

Since BPSK-modulation is used over the AWGN channel we have

$$r_t^{(i)} \in N(\pm\sqrt{E_s}, \sqrt{N_0/2})$$

which results in the metric

$$P(\mathbf{r}_t|\sigma_t = \sigma, \sigma_{t+1} = \sigma') = \prod_{i=1}^{c} e^{-\frac{(r_t^{(i)} - w_t^{(i)})^2}{N_0}} \quad (6.25)$$

where $c$ corresponds to the number of code symbols, in our case $c = 2$, and $w_t^{(i)} \in \{\sqrt{E_s}, -\sqrt{E_s}\}$ corresponds to the noise-free version of the received signal, when the code symbol $v_t^{(i)} \in \{0, 1\}$ was generated by the state-transition $\sigma \to \sigma'$.

Let $\mathbf{e}_0$ be a $2^m$-dimensional row-vector with a 1 in the first position and zeros in all the other positions, i.e.,

$$\mathbf{e}_0 = (10 \dots 0). \tag{6.26}$$

This corresponds to $\sigma_0 = 0$. Consider the product

$$\mathbf{e}_0 P_0 P_1 \dots P_{n+m-1} = (\gamma 0 \dots 0) \tag{6.27}$$

where the zeros in the $2^m - 1$ positions comes from the fact that we terminate the sequence to the zero-state. The $\gamma$ value obtained in (6.27) is the same as that in (6.21). In order to calculate the denominator of (6.21), $\gamma_i^{(k)}$, we introduce, as a counterpart to $P_t$, the state transition matrix

$$P_t^{(k)} = (p_t^{(k)}(\sigma, \sigma'))_{\sigma,\sigma'} \tag{6.28}$$

where

$$\begin{aligned} &p_t^{(k)}(\sigma, \sigma') = P(r_t, \sigma_{t+1} = \sigma', u_t^{(k)} = 0 | \sigma_t = \sigma) = \\ &P(r_t | \sigma_{t+1} = \sigma', \sigma_t = \sigma, u_t^{(k)} = 0) P(\sigma_{t+1} = \sigma' | \sigma_t = \sigma, u_t^{(k)} = 0) P(u_t^{(k)} = 0). \end{aligned} \tag{6.29}$$

The matrix element $p_t^{(k)}(\sigma, \sigma')$ is the conditional probability that we at depth $t$ receive the $c$-tuple $\mathbf{r}_t$, that the encoder makes the state transition from $\sigma$ to $\sigma'$ and that the $k$th information symbol corresponds to $u_t^{(k)} = 0$. In a similar way as in (6.27) we have

$$\mathbf{e}_0 P_0 P_1 \dots P_{i-1} P_i^{(k)} P_{i+1} \dots P_{n+m-1} = (\gamma_i^{(k)} 0 \dots 0) \tag{6.30}$$

where $\gamma_i^{(k)}$ is the conditional probability that we receive $\mathbf{r}_{[0,n+m)}$ given that a code sequence corresponding to $u_i^{(k)} = 0$ was transmitted. The calculation of $\gamma_i^{(k)}$ is the most crucial part of the algorithm. To calculate $\gamma_i^{(k)}$, $i = 0 \dots n-1$ we need the following $n$ equations

$$\begin{aligned} \mathbf{e}_0 P_0^{(k)} P_1 P_2 \dots P_{n-1} P_n \dots P_{n+m-1} &= (\gamma_0^{(k)} 0 \dots 0) \\ \mathbf{e}_0 P_0 P_1^{(k)} P_2 \dots P_{n-1} P_n \dots P_{n+m-1} &= (\gamma_1^{(k)} 0 \dots 0) \\ \mathbf{e}_0 P_0 P_1 P_2^{(k)} \dots P_{n-1} P_n \dots P_{n+m-1} &= (\gamma_2^{(k)} 0 \dots 0) \\ \vdots \qquad\qquad \vdots \qquad\qquad &\vdots \\ \mathbf{e}_0 P_0 P_1 P_2 \dots P_{n-1}^{(k)} P_n \dots P_{n+m-1} &= (\gamma_{n-1}^{(k)} 0 \dots 0) \end{aligned} \tag{6.31}$$

To be able to calculate the $n$ equations efficiently we split the equations in two parts. The first part contains $\mathbf{e}_0 P_0 P_1 \dots P_{i-1}$, and the second part contains $P_i^{(k)} P_i + 1 \dots P_{n+m-1}$. The name two-way implies that decoding is done in two directions; first the first part of the equations are calculated in the forward direction and secondly the second part of the equations are calculated in the backward direction. The two parts are then combined to get the appropriate result.

In the forward direction we start at the root ($\mathbf{e}_0$) and define the forward or $\boldsymbol{\alpha}$-metric

$$\boldsymbol{\alpha}_i \stackrel{def}{=} (\alpha_i(0)\alpha_i(1)\dots\alpha_i(2^m-1)) = \mathbf{e}_0 P_0 P_1 \dots P_{i-1}, 1 \le i \le n-1 \tag{6.32}$$

By convention we let $\boldsymbol{\alpha}_0 = \mathbf{e}_0$. For each depth $i$ , $i = 1 \dots n-1$ the $\boldsymbol{\alpha}_i$ components are stored.

To be able to use a recursive scheme in the backward direction, the second part of the equations are calculated starting at the terminal node at depth $n+m$. Since the codes are terminated we know that we end in the zero-state at depth $n+m$, corresponding to $\mathbf{e}_0$. Since we go in the backward direction the $P$ matrix has to be transposed. We define the $\boldsymbol{\beta}^{(k)}$-metric

$$\begin{aligned}\boldsymbol{\beta}_i^{(k)} &\stackrel{def}{=} \left(\beta_i^{(k)}(0)\beta_i^{(k)}(1)\dots\beta_i^{(k)}(2^m-1)\right) \\ &= \mathbf{e}_0 P_{n+m-1}^T P_{n+m-2}^T \dots P_{i+1}^T (P_i^{(k)})^T,\ 0 \le i \le n, 1 \le k \le b\end{aligned} \tag{6.33}$$

Using the $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}^{(k)}$ metric we can calculate

$$\gamma_i^{(k)} = \sum_{\sigma=0}^{2^m-1} \alpha_i(\sigma)\beta_i^{(k)}(\sigma), 0 \le i \le n \tag{6.34}$$

and

$$\gamma = \alpha_{n+m}(0) \tag{6.35}$$

By combining (6.21) with (6.34) and (6.35) we obtain

$$P(u_i^{(k)} = 0|\mathbf{r}_{[n+m)}) = \frac{\sum_{\sigma=0}^{2^m-1} \alpha_i\beta_i^{(k)}(\sigma)}{\alpha_{n+m}(0)}, 0 \le i \le n, 0 \le k \le b \tag{6.36}$$

**Example 6.3:** Let us calculate the a posteriori probability, $P(u_i^{(k)} = 0|\mathbf{r}_{[0,n+m)})$, $0 \le i < n$, for the rate $R = 1/2$ binary encoder in Figure 6.3 in combination with BPSK-modulation and the AWGN channel. Let $SNR = 0$ dB and, for simplicity, we let the symbol energy be $E_b = 1$. The received sequence can be found in Table 6.1. Since the encoder only has one input $k = 1$.

| t | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $r_t^{(1)}$ | -3.5381 | 0.538998 | 0.396592 | 1.04663 | 0.132651 | 0.566748 |
| $r_t^{(2)}$ | 1.41079 | -0.0866733 | -1.11526 | 1.60251 | 3.56685 | -0.879046 |

Table 6.1: The received sequence, $\mathbf{r} = r_0^{(1)} r_0^{(2)} r_1^{(1)} \dots$ at $SNR = 0$ dB over the AWGN channel with BPSK modulation.

Combining $E_s = 1$ and $SNR = 0$ dB gives $N_0 = 1$ which can be used to calculate (6.25). $P(\mathbf{r}_t|\sigma_{t+1} = \sigma', \sigma_t = \sigma)$ can be found, for the different possible code symbols $\mathbf{v}_t$ at time $t$, in Table 6.2.

| | $t$ | | | | | |
|---|---|---|---|---|---|---|
| $\mathbf{v}_t$ | 0 | 1 | 2 | 3 | 4 | 5 |
| 00 | $9.87\cdot10^{-4}$ | 0.628 | 0.199 | 0.885 | 0.087 | 0.230 |
| 01 | $1.50\cdot10^{-3}$ | 0.705 | 0.882 | 0.105 | $7.45\cdot10^{-4}$ | 0.935 |
| 10 | 0.11 | 0.306 | 0.117 | 0.219 | 0.073 | 0.136 |
| 11 | 0.017 | 0.344 | 0.520 | 0.026 | $6.25\cdot10^{-4}$ | 0.439 |

Table 6.2: The possible values of $P(\mathbf{r}_t|\sigma_{t+1}=\sigma', \sigma_t=\sigma)$, depending on which sub-block, $\mathbf{v}_t$, the state-transition $\sigma \to \sigma'$ corresponds to.

Now we can design the state transition matrices by using the metric in Table 6.2.

$$P_t = \begin{pmatrix} m_t(00)P_0 & 0 & m_t(11)P_1 & 0 \\ m_t(11)P_1 & 0 & m_t(00)P_0 & 0 \\ 0 & m_t(10)P_1 & 0 & m_t(01)P_1 \\ 0 & m_t(01)P_0 & 0 & m_t(10)P_0 \end{pmatrix} \tag{6.37}$$

where $m_t(v_t)$ corresponds to the $P(\mathbf{r}_t|\sigma_{t+1}=\sigma', \sigma_t=\sigma)$ in Table 6.2. $P_0$ corresponds to the a priori probability $P(u_i=0)=1/2$ and $P_1$ corresponds to $P(u_i=1)=1/2$. We can now calculate the $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}_i^{(k)}$ metric as described in (6.32) and (6.33).

$$\begin{array}{lcccccc}
\alpha_0 & = ( & 1 & 0 & 0 & 0 & ) \\
\alpha_1 & = ( & 4.934\cdot10^{-4} & 0 & 8.415\cdot10^{-3} & 0 & ) \\
\alpha_2 & = ( & 1.550\cdot10^{-4} & 1.289\cdot10^{-3} & 8.483\cdot10^{-5} & 2.968\cdot10^{-3} & ) \\
\alpha_3 & = ( & 3.503\cdot10^{-4} & 1.314\cdot10^{-3} & 1.687\cdot10^{-4} & 2.117\cdot10^{-4} & ) \\
\alpha_4 & = ( & 1.721\cdot10^{-4} & 2.957\cdot10^{-5} & 5.861\cdot10^{-4} & 3.204\cdot10^{-5} & ) \\
\alpha_5 & = ( & 7.456\cdot10^{-6} & 2.126\cdot10^{-5} & 0 & 0 & ) \\
\alpha_6 & = ( & 5.747\cdot10^{-6} & 0 & 0 & 0 & )
\end{array}$$

$$\begin{array}{lcccccc}
\beta_0^{(1)} & = ( & 0.1447 & 0 & 0 & 0 & ) \\
\beta_1^{(1)} & = ( & 6.265\cdot10^{-3} & 0 & 0 & 8.172\cdot10^{-5} & ) \\
\beta_2^{(1)} & = ( & 2.773\cdot10^{-3} & 3.524\cdot10^{-3} & 4.270\cdot10^{-6} & 2.360\cdot10^{-6} & ) \\
\beta_3^{(1)} & = ( & 2.867\cdot10^{-4} & 9.191\cdot10^{-7} & 4.991\cdot10^{-6} & 1.589\cdot10^{-3} & ) \\
\beta_4^{(1)} & = ( & 9.084\cdot10^{-5} & 6.810\cdot10^{-5} & 5.609\cdot10^{-4} & 2.639\cdot10^{-4} & ) \\
\beta_5^{(1)} & = ( & 6.320\cdot10^{-8} & 3.333\cdot10^{-7} & 3.817\cdot10^{-8} & 8.860\cdot10^{-9} & )
\end{array}$$

From the $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}_i^{(k)}$ metric can now $\gamma$ and $\gamma_i^{(k)}$ be calculated from (6.27) and (6.30). We get the following result

$$\begin{aligned}
\gamma &= 5.747 \cdot 10^{-6} \\
\gamma_0^{(1)} &= 6.320 \cdot 10^{-8} \\
\gamma_1^{(1)} &= 4.765 \cdot 10^{-6} \\
\gamma_2^{(1)} &= 4.764 \cdot 10^{-6} \\
\gamma_3^{(1)} &= 5.602 \cdot 10^{-6} \\
\gamma_4^{(1)} &= 1.078 \cdot 10^{-6} \\
\gamma_5^{(1)} &= 1.079 \cdot 10^{-6}
\end{aligned} \tag{6.38}$$

which gives us

$$\begin{aligned}
P(u_0^{(1)} = 0) &= 0.0110 \\
P(u_1^{(1)} = 0) &= 0.8290 \\
P(u_2^{(1)} = 0) &= 0.8289 \\
P(u_3^{(1)} = 0) &= 0.9747 \\
P(u_4^{(1)} = 0) &= 0.1878 \\
P(u_5^{(1)} = 0) &= 0.1878
\end{aligned} \tag{6.39}$$

which is decoded to $\mathbf{u} = 1000$

■

We can use the fact that the $P_t$ matrix is sparse and describe the matrix multiplication in (6.27), in a more efficient way, using a trellis description. Each multiplication is equivalent to moving a time step in the trellis. To calculate both the $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ metric we need two different trellises, one in the forward direction and one in the backward direction. We introduce the trellis multiplicative metric $\mu_i(\sigma)$ in the forward direction and $\tilde{\mu}_i(\sigma)$ in the backward direction. In the forward direction we start in the zero-state, i.e.,

$$\mu_0(\sigma) = \begin{cases} 1, & \sigma = 0 \\ 0, & otherwise \end{cases} \tag{6.40}$$

which corresponds to $\mathbf{e}_0$ in (6.27). The forward metrics for the depth $t$ up to $n+m$ is then calculated as

$$\mu_t(\sigma') = \sum_{\sigma \to \sigma'} \mu_{t-1}(\sigma) p_{t-1}(\sigma, \sigma') \tag{6.41}$$

where $\sigma$ are the states that has a possible state-transition to $\sigma'$ in the trellis. This corresponds to the non-zero elements in the $P_t$ matrix. Thus we see that the trellis metric in the forward direction corresponds to the $\boldsymbol{\alpha}$-metric described earlier.

In the backward direction we start at depth $n+m$ in the trellis, in the zero-state, i.e.,

$$\tilde{\mu}_0(\sigma') = \begin{cases} 1, & \sigma' = 0 \\ 0, & otherwise \end{cases} . \tag{6.42}$$

We then move backward in the trellis until we have reached depth 0 in a similar way as in the forward direction.

$$\tilde{\mu}_i(\sigma) = \sum_{\sigma \in \sigma \to \sigma'} \tilde{\mu}_{t+1}(\sigma') p_t(\sigma, \sigma') \tag{6.43}$$

As the forward metric corresponds to the $\boldsymbol{\alpha}$-metric $\gamma$ can be found in the trellis as

$$\gamma = \mu_{n+m}(0). \tag{6.44}$$

$\gamma_i^{(k)}$ can be obtained by

$$\gamma_i^{(k)} = \sum_{\sigma=0}^{(2^m-1)} \sum_{\sigma'=0}^{(2^m-1)} \mu(\sigma) p_i^{(k)}(\sigma, \sigma') \tilde{\mu}_{i+1}(\sigma') \tag{6.45}$$

**Example 6.4:** Let us calculate the a priori probability for the same received sequence as in the previous example, shown in Table 6.1, by using trellis description. First the state-transition metric, $p_t(\sigma', \sigma)$, has to be calculated. This is done in the same manner as before and it is shown in Table 6.2. Now we can design the trellis in the forward direction by using (6.41). The resulting trellis is shown in Figure 6.4. The metric in the trellis is the same as the $\boldsymbol{\alpha}$ metric in the previous example. In the same way we can calculate the backward metric.

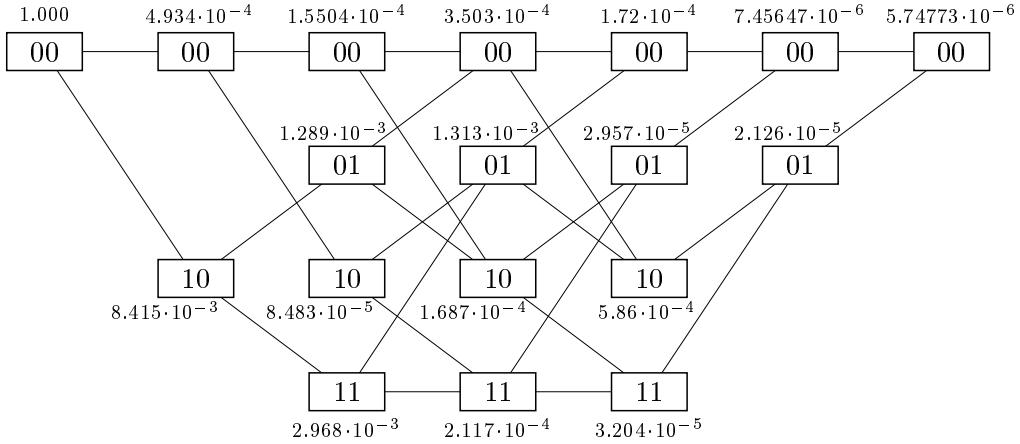



Figure 6.4: The forward metrics are written next to the corresponding states.

The resulting trellis is shown in Figure 6.5. The $\gamma$ and $\gamma_i^{(k)}$ can now be calculated and they are the same as before.

■

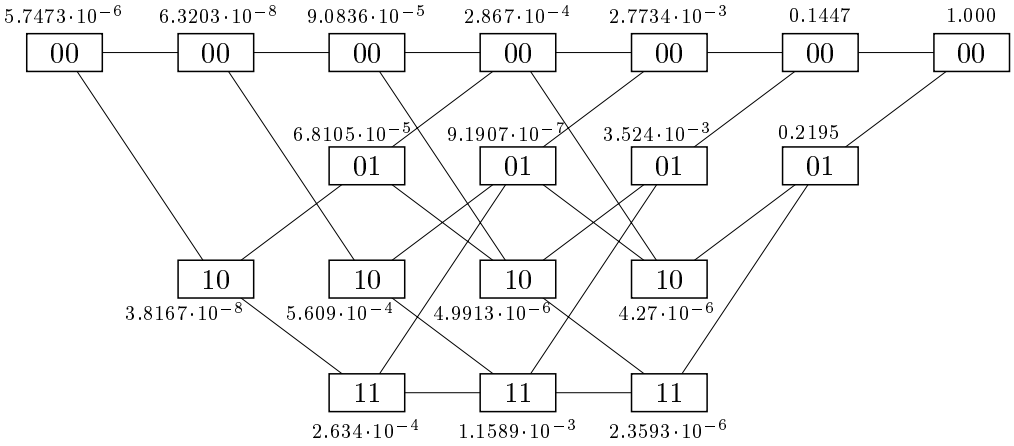



Figure 6.5: The backward metrics $\beta(\sigma)$ are written next to the corresponding states.

## 6.3 The One-way Algorithm for APP Decoding

The one-way algorithm was independently invented by Trofimov [Tro96] and Zigangirov [Zig98]. We use Zigangirov's version of the algorithm as described in [JZ98]. This algorithm is a forward-only algorithm and it can be used on non-terminated codes, as opposed to the two-way (BCJR) algorithm described in Section 6.2. The algorithm is recursive and it uses a sliding window of size $\tau$, i.e., in order to calculate the a posteriori probability for $u_i^{(k)} = 0$ the receiver has to reach the depth $i + \tau$. Analogously to (6.21) we have

$$\begin{aligned} P(u_i^{(k)} = 0|\mathbf{r}_{[0,i+\tau)}) &= \frac{P(\mathbf{r}_{[0,i+\tau)}, u_i^{(k)} = 0)}{P(\mathbf{r}_{[0,i+\tau)})} \\ &= \frac{\sum_{\sigma[0,i+\tau]\in\mathcal{S}^{(k)}_{[0,i+\tau]i}} P(\mathbf{r}_{[0,i+\tau)}|\boldsymbol{\sigma}_{[0,i+\tau]})P(\boldsymbol{\sigma}_{[0,i+\tau]})}{\sum_{\sigma[0,i+\tau]\in\mathcal{S}_{[0,i+\tau]}} P(\mathbf{r}_{[0,i+\tau)}|\boldsymbol{\sigma}_{[0,i+\tau]})P(\boldsymbol{\sigma}_{[0,i+\tau]})} \stackrel{\text{def}}{=} \frac{\gamma^{(k)}_{i+\tau}}{\gamma_{i+\tau}}, 1 \le k \le b, \end{aligned} \tag{6.46}$$

where $\mathcal{S}_{[0,i+\tau]}$ is the set of state-transition sequences $\boldsymbol{\sigma}_{[0,i+\tau]}$ such that $\sigma_0 = 0$, and $\mathcal{S}^{(k)}_{[0,i+\tau]i}$ is the set of state-transition sequences $\boldsymbol{\sigma}_{[0,i+\tau]}$ such that $\sigma_0 = 0$ and the transition from state $\sigma_i$ to state $\sigma_{i+1}$ is caused by $u_i^{(k)} = 0$, i.e.,

$$\mathcal{S}_{[0,i+\tau]} \stackrel{\text{def}}{=} \{\boldsymbol{\sigma}_{[0,i+\tau]} = \sigma_0\sigma_1\ldots\sigma_{i+\tau}|\sigma_0 = 0\} \tag{6.47}$$

and

$$\mathcal{S}^{(k)}_{[0,i+\tau]i} \stackrel{\text{def}}{=} \{\boldsymbol{\sigma}_{[0,i+\tau]} = \sigma_0\sigma_1\ldots\sigma_{i+\tau}|\sigma_0 = 0, \sigma_i \to \sigma_{i+1} \Rightarrow u_i^{(k)} = 0\}. \tag{6.48}$$

Note that we do not know in which state we end, as we do when we decode a terminated sequence with the two-way algorithm. $P(\mathbf{r}_{[0,i+\tau)}|\boldsymbol{\sigma}_{[0,i+\tau]})$ is the probability that $\mathbf{r}_{[0,i+\tau)}$ is received, conditioned on that the code sequence generated by the state sequence $\boldsymbol{\sigma}_{[0,i+\tau]}$ is transmitted and $P(\boldsymbol{\sigma}_{[0,i+\tau]})$ is the a priori probability of the state sequence $\boldsymbol{\sigma}_{[0,i+\tau]}$.

Now let

$$\gamma_{i+\tau} \stackrel{\text{def}}{=} \sum_{\sigma=0}^{2^m-1} \alpha_{i+\tau}(\sigma) \tag{6.49}$$

where $\alpha_{i+\tau}(\sigma)$ is given by (6.32). Let

$$\boldsymbol{\alpha}_{i+\tau}^{(k)} = \left(\alpha_{i+\tau}^{(k)}(0)\alpha_{i+\tau}^{(k)}(1)\dots\alpha_{i+\tau}^{(k)}(2^m-1)\right) \stackrel{\text{def}}{=} \mathbf{e_0}P_0P_1\dots P_{i-1}P_i^{(k)}P_{i+1}\dots P_{i+\tau-1}. \tag{6.50}$$

where $P_i$ and $P_i^{(k)}$ are given by (6.22) and (6.28). Let

$$\gamma_{i+\tau}^{(k)} \stackrel{\text{def}}{=} \sum_{\sigma=0}^{2^m-1} \alpha_{i+\tau}^{(k)}(\sigma). \tag{6.51}$$

Now we can write (6.46) as

$$P(u_i^{(k)} = 0|\mathbf{r}_{[0,i+\tau)}) = \frac{\gamma_{i+\tau}^{(k)}}{\gamma_{i+\tau}}. \tag{6.52}$$

In order to calculate $\gamma_{i+\tau}^{(k)}$ and $\gamma_{i+\tau}$ we use a recursive scheme. For $\gamma_{i+\tau}$ it follows from (6.32) that

$$\begin{cases} \boldsymbol{\alpha}_0 & = \mathbf{e}_0 \\ \boldsymbol{\alpha}_{i+\tau+1} & = \boldsymbol{\alpha}_{i+\tau}P_{i+\tau}. \end{cases} \tag{6.53}$$

This together with (6.49) makes it possible to calculate $\gamma_{i+\tau}$.

In order to calculate $\gamma_{i+\tau}^{(k)}$ we introduce

$$\begin{aligned} \boldsymbol{\alpha}_{ij}^{(k)} &= \left(\alpha_{ij}^{(k)}(0)\alpha_{ij}^{(k)}(1)\dots\alpha_{ij}^{(k)}(2^m-1)\right) \\ &\stackrel{\text{def}}{=} \mathbf{e_0}P_0P_1\dots P_{i-1}P_i^{(k)}P_{i+1}\dots P_{j-1},\ j-\tau < i \le j-1,\ 1 \le k \le b. \end{aligned} \tag{6.54}$$

Let

$$A_{ij} = \begin{pmatrix} \boldsymbol{\alpha}_{ij}^{(1)} \\ \boldsymbol{\alpha}_{ij}^{(2)} \\ \vdots \\ \boldsymbol{\alpha}_{ij}^{(b)} \end{pmatrix}, j-\tau < i \le j-1, \tag{6.55}$$

be a $b \times 2^m$ matrix and let $\mathbb{A}$ be a $b(\tau-1) \times 2^m$ matrix whose entries are the matrices $A_{it}$, $t-\tau < i < t$.

$$\mathbb{A}_t = \begin{pmatrix} A_{t-\tau+1,t} \\ A_{t-\tau+2,t} \\ \vdots \\ A_{t-1,t} \end{pmatrix}. \tag{6.56}$$

**Example 6.5:** Suppose that $m = 2$, $b = 1$, $c = 2$ and $\tau = 3$. Since $b = 1$ the $A_{ij}$ matrix will only contain $\boldsymbol{\alpha}_{ij}^{(1)}$. The $\mathbb{A}_t$ matrix will have the following appearance

$$\mathbb{A}_t = \begin{pmatrix} \boldsymbol{\alpha}_{t-2,t}^{(1)} \\ \boldsymbol{\alpha}_{t-1,t}^{(1)} \end{pmatrix} = \begin{pmatrix} \mathbf{e}_0 P_0 P_1 \dots P_{t-2}^{(1)} P_{t-1} \\ \mathbf{e}_0 P_0 P_1 \dots P_{t-2} P_{t-1}^{(1)} \end{pmatrix}.$$

The vector $\boldsymbol{\alpha}_t$ is

$$\boldsymbol{\alpha}_t = \mathbf{e}_0 P_0 P_1 \dots P_{t-2} P_{t-1}.$$

■

From the previous equations it follows that

$$\mathbb{A}_t P_t = \begin{pmatrix} A_{t-\tau+1,t+1} \\ A_{t-\tau+2,t+1} \\ \vdots \\ A_{t-1,t+1} \end{pmatrix}. \tag{6.57}$$

Now we calculate

$$A_{t,t+1} = \boldsymbol{\alpha}_t P_t^{(k)} = \begin{pmatrix} \boldsymbol{\alpha}_t P_t^{(1)} \\ \boldsymbol{\alpha}_t P_t^{(2)} \\ \vdots \\ \boldsymbol{\alpha}_t P_t^{(b)} \end{pmatrix} = \begin{pmatrix} \boldsymbol{\alpha}_{t,t+1}^{(1)} \\ \boldsymbol{\alpha}_{t,t+1}^{(2)} \\ \vdots \\ \boldsymbol{\alpha}_{t,t+1}^{(b)} \end{pmatrix}. \tag{6.58}$$

In order to get $\mathbb{A}_{t+1}$ we first remove the top matrix $A_{t-\tau+1,t+1}$ from $\mathbb{A}_t P_t$ and save it for later use. Then we shift all the other $A$-matrices up one position and insert $A_{t,t+1}$ from (6.58) in the empty position. We get

$$\mathbb{A}_{t+1} = \begin{pmatrix} A_{t-\tau+2,t+1} \\ A_{t-\tau+3,t+1} \\ \vdots \\ A_{t,t+1} \end{pmatrix}. \tag{6.59}$$

The rows of the matrix $A_{t-\tau+1,t+1}$, which we removed from $\mathbb{A}$, are the vectors $\boldsymbol{\alpha}_{t+1}^{(k)}$, $1 \le k \le b$, defined by (6.50). They are used together with $\boldsymbol{\alpha}_{t+1}$ to calculate the a posteriori probability

$$P(u_{t-\tau+1}^{(k)} = 0 | \mathbf{r}_{[0,t+1)}) = \gamma_{t+1}^{(k)} / \gamma_{t+1}$$

**Example 6.5 (cont.):** If we want to calculate $\mathbb{A}_{t+1}$ we start by multiplying the $\mathbb{A}_t$ matrix by $P_t$, getting

$$\mathbb{A}_t P_t = \begin{pmatrix} \mathbf{e}_0 P_0 P_1 \dots P_{t-2}^{(1)} P_{t-1} P_t \\ \mathbf{e}_0 P_0 P_1 \dots P_{t-2} P_{t-1}^{(1)} P_t \end{pmatrix}.$$

Then we remove the top row and shift the other up one position. We then calculate

$$A_{t,t+1} = \boldsymbol{\alpha}_t P_t^{(1)} = \mathbf{e}_0 P_0 P_1 \dots P_{t-2} P_{t-1} P_t^{(1)}$$

and insert it into the empty position in $\mathbb{A}$. This gives us the new matrix

$$\mathbb{A}_{t+1} = \begin{pmatrix} \mathbf{e}_0 P_0 P_1 \dots P_{t-2} P_{t-1}^{(1)} P_t \\ \mathbf{e}_0 P_0 P_1 \dots P_{t-2} P_{t-1} P_t^{(1)} \end{pmatrix}.$$

■

**Example 6.6:** Assume that we want to calculate the a posteriori probabilities, $P(u_i^{(k)} = 0|\mathbf{r}_{[0,i+\tau)})$ using the one-way algorithm for the same encoder and received sequence as in Example 6.3 and let $\tau = 2$. The received sequence can be found in Table 6.1. The matrices $P_t$ and $P_t^{(1)}$ can be calculated the same way as in the Example 6.3 with the metric in Table 6.2. To decode the first information symbol, $u_0$, we need $\boldsymbol{\alpha}_\tau$ and $\mathbb{A}_\tau$. First we initialize $\mathbb{A}$ to

$$\mathbb{A}_1 = \begin{pmatrix} \mathbf{e}_0 P_0^{(1)} \end{pmatrix} = \begin{pmatrix} 4.934{\cdot}10^{-4} & 0 & 0 & 0 \end{pmatrix}$$

and

$$\boldsymbol{\alpha}_1 = \mathbf{e}_0 P_0 = \begin{pmatrix} 4.934{\cdot}10^{-4} & 0 & 8.415{\cdot}10^{-3} & 0 \end{pmatrix}$$

Then we can start the decoding procedure by multiplying $\mathbb{A}_1$ and $\boldsymbol{\alpha}_1$ with $P_1$

$$\mathbb{A}_1 P_1 = \begin{pmatrix} \mathbf{e}_0 P_0^{(1)} P_1 \end{pmatrix} = \begin{pmatrix} 1.550{\cdot}10^{-4} & 0 & 8.483{\cdot}10^{-5} & 0 \end{pmatrix}$$

and

$$\boldsymbol{\alpha}_2 = \mathbf{e}_0 P_0 P_1 = \begin{pmatrix} 1.550{\cdot}10^{-4} & 1.289{\cdot}10^{-3} & 8.483{\cdot}10^{-5} & 2.968{\cdot}10^{-3} \end{pmatrix}$$

Then we calculate $A_{1,2}$ corresponding to (6.58)

$$A_{1.2} = \mathbf{e}_0 P_0 P_1^{(1)} = \begin{pmatrix} 1.550{\cdot}10^{-4} & 0 & 0 & 2.968{\cdot}10^{-3} \end{pmatrix}$$

and insert it into $\mathbb{A}_1 P_1$ corresponding to (6.47)

$$\mathbb{A}_2 = \begin{pmatrix} \mathbf{e}_0 P_0 P_1^{(1)} \end{pmatrix} = \begin{pmatrix} 1.550{\cdot}10^{-4} & 0 & 0 & 2.968{\cdot}10^{-3} \end{pmatrix}$$

Then we can calculate the a posteriori probability from $\boldsymbol{\alpha}_2$ and $\boldsymbol{\alpha}_2^{(k)}$, which we removed from $\mathbb{A}_1 P_1$, i.e.,

$$P(u_0 = 0|r_{[0,2)}) = \frac{\sum_{\sigma=0}^{2^m-1} \alpha_2^{(1)}(\sigma)}{\sum_{\sigma=0}^{2^m-1} \alpha_2(\sigma)} = 5.333{\cdot}10^{-2} \tag{6.60}$$

Then we calculate $\mathbb{A}_3$ in the same way and so on. We get the following a posteriori probabilities

$$\begin{aligned} P(u_1 = 0|r_{[0,3)}) &= 0.7526 \\ P(u_2 = 0|r_{[0,4)}) &= 0.7687 \\ P(u_3 = 0|r_{[0,5)}) &= 0.9184 \end{aligned}$$

which will be decoded to $\mathbf{u} = 1000$ which is the same as in Example 6.3.

■

# Chapter 7

# Implementation of the iterative decoding algorithm

This section describe how our simulation programs were designed. Since the two-way algorithm uses terminated codes and the one-way algorithm uses non-terminated codes and needs to be $\tau$ symbols ahead, the implementation of the two algorithms are quite different.

Important parameters when designing a mobile communication system are the delay of the decoder, the complexity of the algorithm and the memory consumption. These parameters will also be discussed.

## 7.1 Iterative decoding using the two-way algorithm

Since the code sequences decoded by the two-way algorithm are terminated, it is suitable to decode one block of $n$ information symbols at a time. Figure 7.1 shows a flowchart how each block of received symbols are decoded.

First we start with initializing the likelihood ratio, $\Lambda_i(0) = 1$ assuming that $P(u_i = 0) = 1/2$, for $i = 1 \ldots n$.

Then we use the two-way algorithm to decode a whole block of $n$ symbols, such that we get $n$ a posteriori likelihoods as output. We use a trellis implementation as described in Chapter 6.2. To avoid using the intrinsic likelihood ratio and the a priori likelihood ratio twice, we extract the extrinsic information by dividing the a posteriori likelihood by the intrinsic likelihood and the a priori likelihood ratio.

The extrinsic likelihood ratio is then interleaved together with the information sequence.

The extrinsic likelihood ratio from the first decoder is used as a priori information to the second decoder, which calculates its own version of the $n$ a posteriori likelihood ratios. These are divided by the a priori likelihood ratio and the intrinsic likelihood ratio to get the extrinsic likelihood ratio.

To use it in the first decoder we have to deinterleave the extrinsic likelihood ratio, which

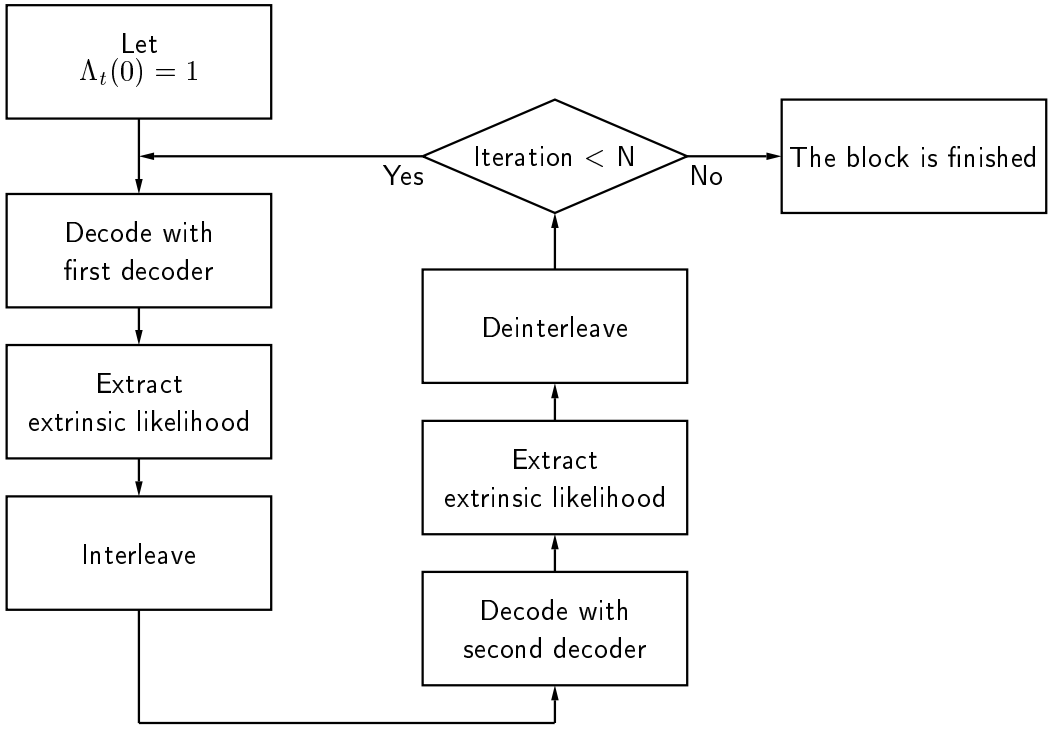



Figure 7.1: The flow of the iterative decoding of one block of symbols.

is then fed into the first decoder.

After $N$ iterations, a decision can be made based on (6.16). The process is then repeated for the next block of symbols.

To avoid numerical problems we had to limit the output likelihood ratios and normalize the $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}_i^{(k)}$ metric.

## 7.2 Iterative decoding using the one-way algorithm

In the one-way implementation of the iterative decoding algorithm we used a totally different scheme. Since the one-way algorithm is used on non-terminated codes, it is possible to use a pipeline structure of the decoder. Pipelining enables us to use several consecutive decoders to do parallel work on different data bits. A pipeline works much like an assembly line, i.e., when one decoder has finished decoding its data it passes the result on to the next decoder and continues to decode the next data input.

The one-way decoder needs the a priori likelihood ratio for symbol $i + \tau$ from the previous decoder to produce the a posteriori likelihood for symbol $i$, thus there is a delay between

the decoders. The $\tau$ likelihood ratios are interleaved and therefore the delay between the decoders depends on where those $\tau$ likelihood ratios are placed in the interleaver.

**Example 7.1:** If we use the $4 \times 4$ block interleaver in Figure 7.2, where the indices are in the order which they are written to the interleaver. With $\tau = 2$ we have to wait for $\Lambda_8$ before we can decode the first symbol of the interleaved sequence.

In the $4 \times 4$ random interleaver in the same figure we have to wait until $\Lambda_{15}$ is written into the interleaver before we decode the first symbol. Thus we have to wait for the whole interleaver to be full.

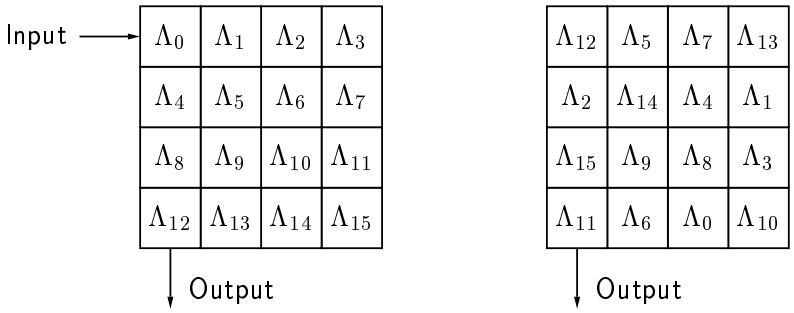



Figure 7.2: Likelihood ratios in a $4 \times 4$ block interleaver (left) and a random interleaver (right).

■

Since the delay can vary between $\tau$ and the size of the interleaver we decided to always use the maximum delay, i.e., we decode block-wise where one block has the size of the interleaver. This is not necessary since it is possible to calculate the actual delay of the interleaver and then decode symbol-wise.

The data flow in the decoding process is shown in Figure 7.3. The first decoder starts to process one block of data, with the a priori likelihood ratio $\Lambda_t(0) = 1$. The output a posteriori likelihood ratios are then divided by the a priori likelihood ratios and the intrinsic likelihood ratios the same way as with the two-way algorithm to produce the extrinsic likelihood ratios. Before the second decoder can start decoding its first block of data, the first decoder has to decode the next block to produce the extrinsic likelihood ratios needed. Then the second decoder can start to decode a block of data. In Figure 7.3 the numbers in the boxes correspond to the order in which the decoders can start decoding a block of data and the arrows show where the likelihood ratios come from.

## 7.3 A comparison between the one-way and two-way algorithm

Since most of the time is spent in the a posteriori decoders, it is essential to minimize the complexity of those. It is the same for both encoders.

The complexity calculation of the two-way and the one-way algorithms, for each block of $n$ symbols, can be divided into the following parts

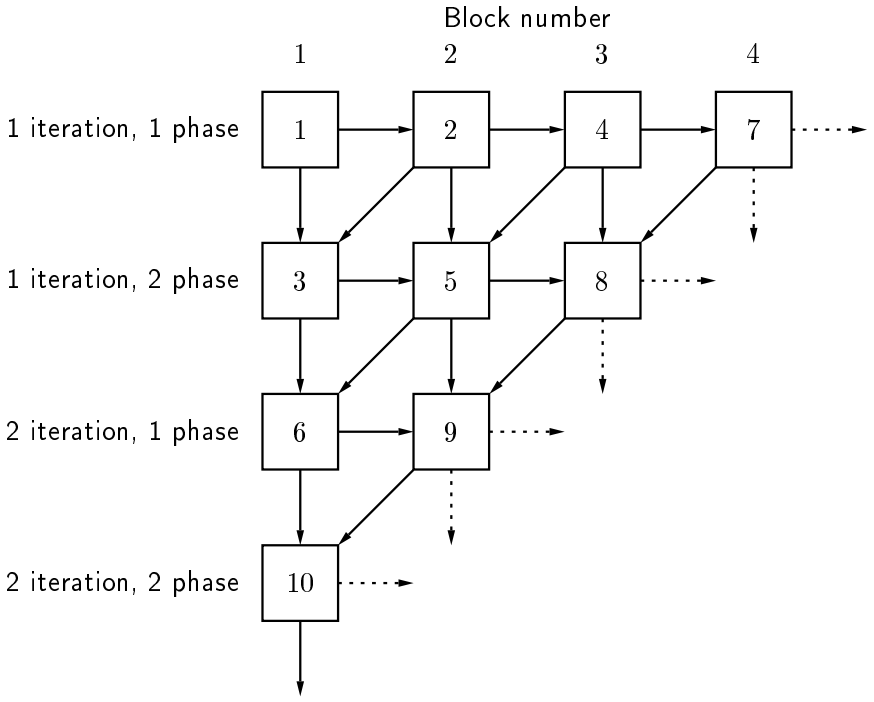



Figure 7.3: Datapath for the pipeline structure.

- Calculating the state-transition metric.
- Calculating the $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ metric or respectively the $\mathbb{A}$ matrix.
- Calculating the $\gamma$ and $\gamma_i^{(k)}$.

The complexity of calculating the state-transition metric is high, since it involves an $e^x$ operation.

When calculating the $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ metric or equivalently move through the trellis in the two-way algorithm we have to do 2 multiplications and 1 addition for each state and for each symbol, i.e., $2 \cdots n \cdot 2^m$ multiplications and $n \cdot 2^m$ additions.

The complexity of calculating the $\mathbb{A}$ matrix in the one-way implementation is somewhat higher. For each time step we have to do $\tau$ matrix multiplications. This gives us $\tau/2$ times the complexity of calculating the $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ metric.

Calculation of the extrinsic likelihood ratios can be done directly by a method described in [JZ98] and de/interleaving does not involve any arithmetic.

This is done $2N$ times as we have $2N$ decoders.

The memory consumption is relatively low for the two-way implementation compared to the one-way implementation.

Since we decode a block at a time with the two-way algorithm, we need to save all the received symbols for that block, i.e., $3n$ symbols. Between each iteration we need to save the likelihood ratios in the interleaver and the deinterleaver, i.e., $2n$ symbols. The two-way algorithm itself needs to temporarily save the $\boldsymbol{\alpha}$-metric as described in Section 6.2, i.e., $2^m n$ symbols. We do not need to save the $\boldsymbol{\beta}_i^{(k)}$ metric since we directly can calculate the a posteriori probability for each time step. The total memory usage is thus $6 \cdot 2^m n$ symbols.

The memory usage is for the one-way algorithm is very large, since we have to save the information symbols for all the iterations that the decoder processes, i.e., we have to save $2N \cdot 3n$ code symbols. Besides that we have the memory that the decoders are using which is $2N \cdot \tau 2^m$. The total memory usage is thus about $N$ times higher than for the two-way algorithm.

The delay of the two-way implementation is only one block as the decoder has to be finished when the next block of data arrives, i.e., the delay is $n$ symbols.

The delay of the one-way implementation is larger, since we have to wait for the pipeline to fill up before we can decode the symbols, i.e., the delay is $2N \cdot n$ which is $2N$ times higher than for the two-way implementation.

If we compare the iterative decoder, using the two-way a posteriori implementation, used on turbo codes with the Viterbi decoder used on a normal convolutional code with similar complexity we see that the iterative decoder has better performance but longer delay. For example a the Viterbi decoder used on a memory 7 convolutional decoder has approximately the same complexity as the memory 2 iterative decoder with 20 iterations. The performance is approximately 1.5 dB better for the iterative decoder at a bit error rate of $10^{-3}$.

# Chapter 8

# Results

In this section the results of our simulations are presented as comparisons between the bit-error rate for different interleaver sizes and interleaver types, different sizes of component encoders, different numbers of iterations and finally a comparison between the one-way and the two-way algorithms using different delays $\tau$ for the one-way algorithm.

We start by looking at the bit-error rates for different types and sizes of the interleaver. The interleavers used in the simulations are "classical" block interleavers with sizes $32 \times 32 = 1024$, $64 \times 64 = 4096$, $128 \times 128 = 16384$ and $256 \times 256 = 65536$ symbols, and pseudo-random interleavers of the same sizes. From this point on the "classical" block interleaver is called just block interleaver and the pseudo-random interleaver is called random interleaver.

The interleaver is, as we will see, a very important part of the performance of a turbo

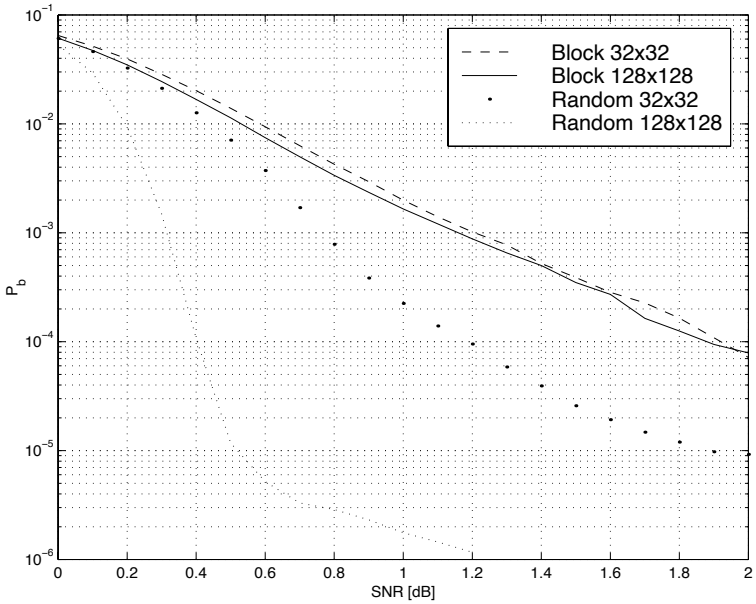



Figure 8.1: Diagram showing bit-error probabilities for different interleaver types and sizes for the two-way algorithm with 10 iterations and $m = 2$.

code system. If we compare the $32 \times 32$ block interleaver in Figure 8.1 with the $32 \times 32$ random interleaver we can see that when we use the random interleaver we get lower bit-error rates than we do when we use the block interleaver. Comparing the $128 \times 128$ block interleaver with the $128 \times 128$ random interleaver we see that the difference in performance is bigger when we use larger interleaver sizes, especially when the SNR increases. The performance for the block interleaver does not increase very much, even though the size of the interleaver is increased with a factor of 16. With larger encoder memory, the performance for the block interleaver increases more with bigger interleaver sizes, but the random interleaver is always better, so there is no reason for using the block interleaver.

Another important parameter when using the iterative decoding algorithm is of course the number of iterations. In Figure 8.2 the influence the number of iterations on the bit-error probability is illustrated. We can see that the effect of increasing the number of iterations is larger with higher SNR ratio, i.e. the bit-error decreases faster with the number of iterations. The figure also shows that to reach sufficiently low bit-error levels we need more iterations at

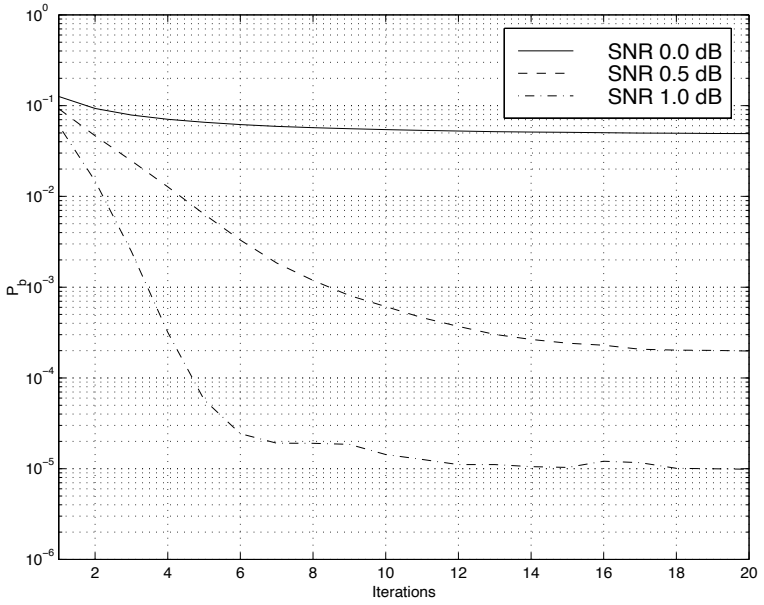



Figure 8.2: Diagram showing bit-error probabilities for different numbers of iterations. (Two-way algorithm, $64 \times 64$ random interleaver, $m = 2$)

low SNR ratios than at high SNR ratios.

In Figure 8.3 there is a comparison between different memory sizes for the two-way algorithm using the same $64 \times 64$ random interleaver and 10 iterations. We can see that the memory 4 improves the bit-error rate faster with increasing signal-to-noise ratio compared to the memory 2, but memory 6 has worse performance for low SNRs. This can be explained by that we get longer burst errors with larger encoder memory. The simulations also show that there is a limit where the curves flattens, the so called error floor.

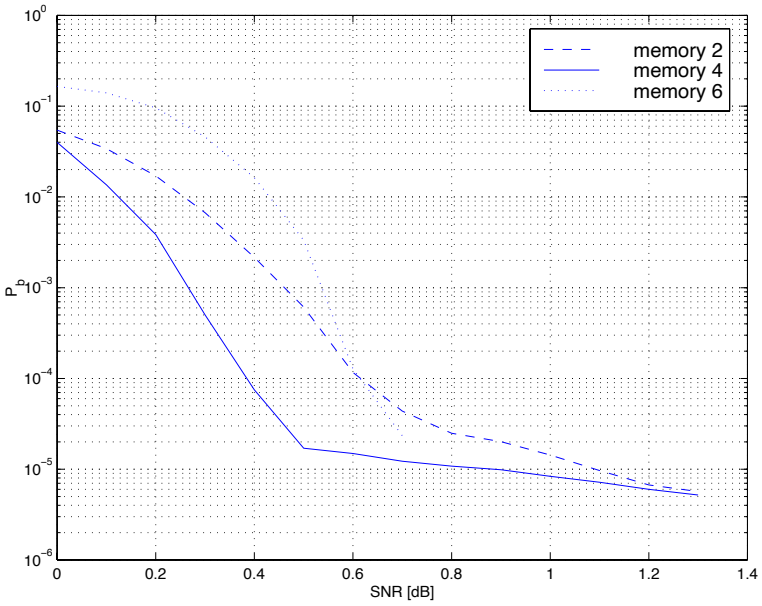



Figure 8.3: Diagram showing bit-error probabilities for different encoder memories. (Two-way algorithm, $64 \times 64$ random interleaver, 10 iterations)

The next diagram, Figure 8.4, shows a comparison between the one-way and the two-way algorithm. The performances of the algorithms are approximately the same. The one-way algorithm has the advantage over the two-way algorithm that the code sequences do not have to be terminated, i.e. we do not have to send the terminating bits. Because of this the one-way algorithm can have better performance than the two-way algorithm when small blocks (interleavers) are used.

Figure 8.5 shows the bit-error rate for the one-way algorithm with window sizes $\tau = 5, 10$ and 20. We can see that the bit-error rate improves with bigger $\tau$. In Figure 8.4 we could see that with $\tau = 20$ the performance of the one-way algorithm is the same as for the two-way algorithm.

The influence of the window size $\tau$ for the one-way algorithm is shown in Figure 8.6. Here, like in Figure 8.5, we can see that larger $\tau$ improves the bit-error rate, but here we also can see that the bit-error rate only improves up to a certain limit.

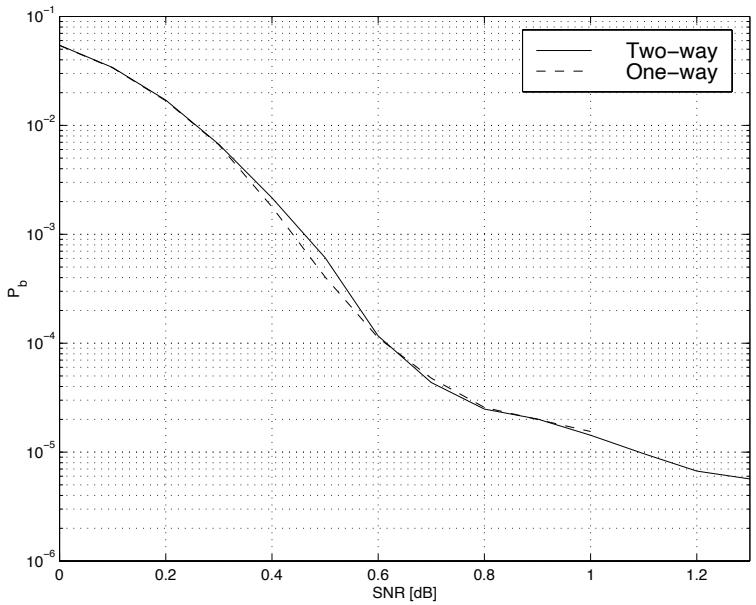



Figure 8.4: Comparison between the one-way and the two-way algorithms ($32 \times 32$ and $64 \times 64$ random interleavers, 10 iterations, $m = 2$. For one-way $\tau = 20$)

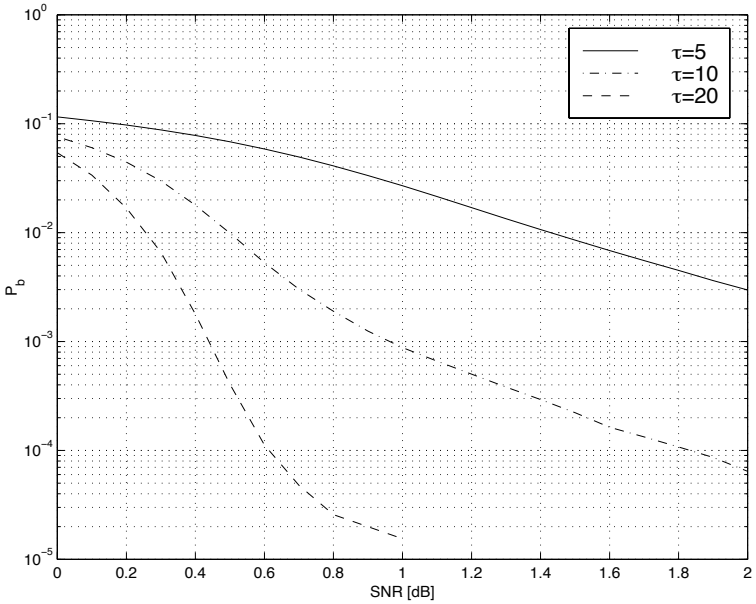



Figure 8.5: Bit-error rates for $\tau = 5, 10$ and 20 for the one-way algorithm ( $64 \times 64$ random interleaver, 10 iterations )

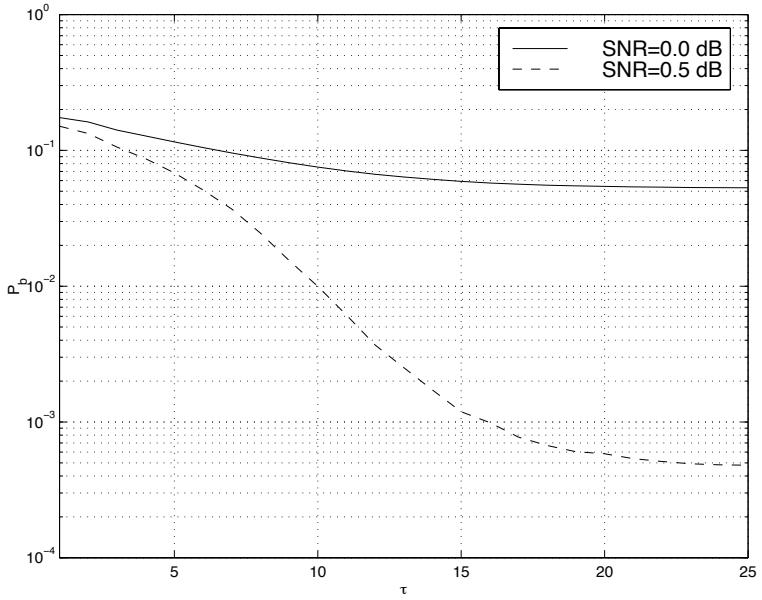



Figure 8.6: Bit-error rates for different values of $\tau$ for the one-way algorithm ( $64 \times 64$ random interleaver, 10 iterations )

# Chapter 9

# Conclusions

In this thesis we have examined performance of different decoding algorithms for the turbo encoder. We have concluded that the one-way and two-way implementations have essentially the same performance. The advantages of the one-way algorithm are that it can be implemented in a pipelined structure and that it can be used for non-terminated codes. The main disadvantages of the one-way implementation are that is uses more memory and has a longer decoding delay than the two-way implementation.

We also found that the choice of interleaver is essential to get good performance. The interleaver size should be as large as possible. The choice of interleaver size is a tradeoff between better performance and longer decoding delay. The type of interleaver is also important. Our results show that the pseudo-random interleaver is a good choice compared to the block interleaver. Much effort has been devoted to finding good interleavers for turbo-codes, but it seems that the performance gain compared to the pseudo-random interleaver is small, or as Wozencraft once said "You should choose the interleaver at random and avoiding being unlucky".

Choosing the size of the encoder memory optimally, also improves the performance but larger memory means higher decoding complexity. Our simulations show that the optimal memory size does not need to be the largest. With very large memory sizes the performance decreases due to longer burst errors.

From our research we can conclude that turbo codes is an efficient method to protect data from errors. By increasing the interleaver size we get better performance without increasing the complexity in the decoder, only the delay and the memory consumption are affected. The number of iterations should be optimized for different SNR ratios to save power. A comparison between the iterative decoder and the Viterbi decoder show that the iterative decoder has better performance but longer delay at similar complexity.

# Appendix A

# Tables

## A.1 Simulation results for the two-way implementation

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
| 5 | 0.072 | 0.0598 | 0.0482 | 0.0373 | 0.028 | 0.0205 | 0.0146 |
| 10 | 0.0649 | 0.0517 | 0.0394 | 0.0285 | 0.0202 | 0.0139 | 0.00941 |
| 20 | 0.0627 | 0.0492 | 0.0371 | 0.0261 | 0.018 | 0.0124 | 0.00811 |

Table A.1: Block interleaver size=32x32, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| 5 | 0.0101 | 0.007 | 0.00477 | 0.00326 | 0.00221 | 0.00154 | 0.00107 |
| 10 | 0.00631 | 0.00426 | 0.00295 | 0.00199 | 0.0014 | 0.00102 | 0.000773 |
| 20 | 0.00544 | 0.00364 | 0.00251 | 0.00173 | 0.00123 | 0.000917 | 0.000675 |

Table A.2: Block interleaver size=32x32, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 1.9 | 2.0 |
| 5 | 0.000751 | 0.000538 | 0.000393 | 0.000279 | 0.000187 | 0.000128 | 9.46e-005 |
| 10 | 0.000523 | 0.000389 | 0.000284 | 0.000227 | 0.000165 | 0.000109 | 7.18e-005 |
| 20 | 0.000471 | 0.000361 | 0.000264 | 0.000211 | 0.000152 | 0.000105 | 6.56e-005 |

Table A.3: Block interleaver size=32x32, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
| 5 | 0.069 | 0.0569 | 0.0453 | 0.0349 | 0.026 | 0.0189 | 0.0133 |
| 10 | 0.0614 | 0.0478 | 0.0355 | 0.0255 | 0.0176 | 0.0119 | 0.008 |
| 20 | 0.0587 | 0.0445 | 0.032 | 0.0222 | 0.0149 | 0.01 | 0.00668 |

Table A.4: Block interleaver size=64x64, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| 5 | 0.00919 | 0.00621 | 0.00419 | 0.00282 | 0.0019 | 0.00129 | 0.000902 |
| 10 | 0.00536 | 0.00361 | 0.0025 | 0.00169 | 0.00121 | 0.000877 | 0.000648 |
| 20 | 0.00445 | 0.00303 | 0.0021 | 0.00151 | 0.00107 | 0.000798 | 0.000606 |

Table A.5: Block interleaver size=64x64, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 1.9 | 2.0 |
| 5 | 0.000651 | 0.000472 | 0.000367 | 0.000264 | 0.000201 | 0.000153 | 0.000114 |
| 10 | 0.000502 | 0.00038 | 0.000297 | 0.000224 | 0.000177 | 0.000125 | 0.000101 |
| 20 | 0.000478 | 0.000365 | 0.000289 | 0.000214 | 0.000171 | 0.000117 | 9.7e-005 |

Table A.6: Block interleaver size=64x64, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
| 5 | 0.0692 | 0.0568 | 0.0449 | 0.0344 | 0.0253 | 0.0181 | 0.0128 |
| 10 | 0.0613 | 0.0472 | 0.0349 | 0.0245 | 0.0168 | 0.0114 | 0.00747 |
| 20 | 0.0585 | 0.0441 | 0.0311 | 0.0213 | 0.0144 | 0.00938 | 0.00604 |

Table A.7: Block interleaver size=128x128, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| 5 | 0.00875 | 0.0059 | 0.00395 | 0.00267 | 0.00184 | 0.00127 | 0.000886 |
| 10 | 0.00499 | 0.00337 | 0.00235 | 0.00166 | 0.00121 | 0.00088 | 0.000654 |
| 20 | 0.00409 | 0.00281 | 0.00201 | 0.00143 | 0.0011 | 0.000775 | 0.000588 |

Table A.8: Block interleaver size=128x128, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 1.9 | 2.0 |
| 5 | 0.000633 | 0.000453 | 0.000332 | 0.000225 | 0.000156 | 0.00011 | 8.82e-005 |
| 10 | 0.0005 | 0.000349 | 0.000272 | 0.000164 | 0.000125 | 9.45e-005 | 7.88e-005 |
| 20 | 0.000442 | 0.000332 | 0.000254 | 0.000157 | 0.000123 | 9.22e-005 | 7.98e-005 |

Table A.9: Block interleaver size=128x128, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
| 5 | 0.0688 | 0.0565 | 0.0449 | 0.0344 | 0.0255 | 0.0184 | 0.0129 |
| 10 | 0.0612 | 0.0473 | 0.035 | 0.0247 | 0.017 | 0.0114 | 0.0076 |
| 20 | 0.0585 | 0.0439 | 0.0313 | 0.0217 | 0.0144 | 0.00962 | 0.00631 |

Table A.10: Block interleaver size=256x256, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| 5 | 0.00892 | 0.00601 | 0.00401 | 0.00268 | 0.00181 | 0.00123 | 0.000905 |
| 10 | 0.00512 | 0.00352 | 0.00241 | 0.00167 | 0.00118 | 0.000842 | 0.000658 |
| 20 | 0.00426 | 0.00293 | 0.00217 | 0.00149 | 0.00108 | 0.00077 | 0.000605 |

Table A.11: Block interleaver size=256x256, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 1.9 | 2.0 |
| 5 | 0.000635 | 0.000441 | 0.000336 | 0.000255 | 0.000195 | 0.000144 | 0.000116 |
| 10 | 0.000507 | 0.000365 | 0.000284 | 0.000207 | 0.000152 | 0.000109 | 7.61e-005 |
| 20 | 0.000467 | 0.000352 | 0.000266 | 0.000197 | 0.000146 | 0.000101 | 7.51e-005 |

Table A.12: Block interleaver size=256x256, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
| 5 | 0.0691 | 0.0555 | 0.0428 | 0.0311 | 0.0212 | 0.0137 | 0.00832 |
| 10 | 0.0609 | 0.0461 | 0.0325 | 0.0212 | 0.0126 | 0.00709 | 0.00373 |
| 20 | 0.0585 | 0.0434 | 0.03 | 0.0188 | 0.0109 | 0.00594 | 0.00289 |

Table A.13: Random interleaver size=32x32, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| 5 | 0.00475 | 0.00257 | 0.00134 | 0.000692 | 0.000361 | 0.000215 | 0.000132 |
| 10 | 0.0017 | 0.000785 | 0.000384 | 0.000225 | 0.00014 | 9.52e-005 | 5.86e-005 |
| 20 | 0.00124 | 0.000566 | 0.000302 | 0.000163 | 0.000106 | 7.06e-005 | 3.8e-005 |

Table A.14: Random interleaver size=32x32, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
| 5 | 0.0655 | 0.0501 | 0.0354 | 0.0224 | 0.0127 | 0.00638 | 0.00283 |
| 10 | 0.0544 | 0.034 | 0.0171 | 0.00669 | 0.00216 | 0.000611 | 0.000117 |
| 20 | 0.0491 | 0.0263 | 0.0107 | 0.00327 | 0.000819 | 0.000198 | 5.95e-005 |

Table A.15: Random interleaver size=64x64, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| 5 | 0.00114 | 0.000425 | 0.000155 | 5.74e-005 | 2.77e-005 | 1.67e-005 | 1.14e-005 |
| 10 | 4.36e-005 | 2.49e-005 | 2.01e-005 | 1.43e-005 | 9.68e-006 | 6.71e-006 | 5.67e-006 |
| 20 | 2.12e-005 | 1.59e-005 | 1.43e-005 | 9.77e-006 | 7.86e-006 | 6.34e-006 | 5.61e-006 |

Table A.16: Random interleaver size=64x64, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
| 5 | 0.0778 | 0.0662 | 0.055 | 0.045 | 0.0367 | 0.031 | 0.0287 |
| 10 | 0.0906 | 0.0839 | 0.0764 | 0.0714 | 0.0643 | 0.0585 | 0.0512 |
| 20 | 0.087 | 0.0799 | 0.0717 | 0.0673 | 0.0639 | 0.0613 | 0.0605 |

Table A.17: MIL interleaver size=333, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| 5 | 0.0305 | 0.0379 | 0.051 | 0.0699 | 0.0935 | 0.12 | 0.145 |
| 10 | 0.0448 | 0.0391 | 0.0375 | 0.0401 | 0.0488 | 0.0626 | 0.0824 |
| 20 | 0.0599 | 0.0579 | 0.0589 | 0.0577 | 0.0577 | 0.0587 | 0.0632 |

Table A.18: MIL interleaver size=333, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 1.9 | 2.0 |
| 5 | 0.167 | 0.183 | 0.187 | 0.183 | 0.17 | 0.153 | 0.131 |
| 10 | 0.101 | 0.000219 | 0.000105 | 6e-005 | 2.68e-005 | 2.23e-005 | 1.27e-005 |
| 20 | 0.0713 | 0.0775 | 0.0828 | 0.0856 | 0.0809 | 0.0711 | 0.0595 |

Table A.19: MIL interleaver size=333, encoder memory=2

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
| 5 | 0.0882 | 0.0659 | 0.0452 | 0.029 | 0.0168 | 0.00847 | 0.00378 |
| 10 | 0.0724 | 0.0497 | 0.0308 | 0.0172 | 0.00844 | 0.00353 | 0.00135 |
| 20 | 0.0677 | 0.045 | 0.0273 | 0.0141 | 0.00641 | 0.00267 | 0.000874 |

Table A.20: Random interleaver size=32x32, encoder memory=4

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| 5 | 0.00156 | 0.000616 | 0.000179 | 7.46e-005 | 2.21e-005 | 3.15e-005 | 1.75e-005 |
| 10 | 0.000454 | 0.000155 | 7.44e-005 | 2.7e-005 | 2.21e-005 | 1.95e-005 | 1.7e-005 |
| 20 | 0.000322 | 8.98e-005 | 6.32e-005 | 2.6e-005 | 2.22e-005 | 1.76e-005 | 1.55e-005 |

Table A.21: Random interleaver size=32x32, encoder memory=4

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
| 5 | 0.0732 | 0.0436 | 0.0197 | 0.00663 | 0.00177 | 0.000337 | 6.68e-005 |
| 10 | 0.04 | 0.0136 | 0.00386 | 0.000501 | 7.49e-005 | 1.71e-005 | 1.49e-005 |
| 20 | 0.0289 | 0.00828 | 0.0022 | 0.000132 | 1.99e-005 | 1.64e-005 | 1.48e-005 |

Table A.22: Random interleaver size=64x64, encoder memory=4

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| 5 | 2.14e-005 | 1.46e-005 | 1.12e-005 | 8.75e-006 | 7.08e-006 | 5.98e-006 | 5.33e-006 |
| 10 | 1.22e-005 | 1.08e-005 | 9.88e-006 | 8.35e-006 | 7.19e-006 | 6.01e-006 | 5.2e-006 |
| 20 | 1.18e-005 | 1.05e-005 | 9.39e-006 | 8.18e-006 | 6.95e-006 | 5.99e-006 | 5.16e-006 |

Table A.23: Random interleaver size=64x64, encoder memory=4

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
| 5 | 0.0751 | 0.0386 | 0.0118 | 0.00201 | 0.000257 | 3.85e-005 | 1.29e-005 |
| 10 | 0.0241 | 0.00145 | 2.97e-005 | 6.59e-006 | 5.67e-006 | 4.77e-006 | 3.79e-006 |
| 20 | 0.00705 | 0.000504 | 7.06e-006 | 6.07e-006 | 4.97e-006 | 4.07e-006 | 3.73e-006 |

Table A.24: Random interleaver size=128x128, encoder memory=4

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| 5 | 3.73e-006 | 3.73e-006 | 3.73e-006 | 3.73e-006 | 3.73e-006 | 3.73e-006 | 3.73e-006 |
| 10 | 3.73e-006 | 3.73e-006 | 3.73e-006 | 3.73e-006 | 3.73e-006 | 3.73e-006 | 3.73e-006 |
| 20 | 3.73e-006 | 3.73e-006 | 3.73e-006 | 3.73e-006 | 3.73e-006 | 3.73e-006 | 3.73e-006 |

Table A.25: Random interleaver size=128x128, encoder memory=4

## A.2 Simulation results for the one-way implementation

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
| 5 | 0.0797 | 0.0673 | 0.055 | 0.0434 | 0.0332 | 0.0243 | 0.017 |
| 10 | 0.0756 | 0.0619 | 0.0485 | 0.0364 | 0.0262 | 0.0177 | 0.0116 |
| 20 | 0.0746 | 0.0606 | 0.047 | 0.035 | 0.0246 | 0.0165 | 0.0107 |

Table A.26: Random interleaver size=32x32, encoder memory=2, $\tau = 10$

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| 5 | 0.0116 | 0.00751 | 0.00478 | 0.00305 | 0.00196 | 0.00129 | 0.000824 |
| 10 | 0.00739 | 0.00459 | 0.00292 | 0.0019 | 0.00126 | 0.000875 | 0.000617 |
| 20 | 0.00671 | 0.00425 | 0.00273 | 0.00176 | 0.00121 | 0.00085 | 0.000598 |

Table A.27: Random interleaver size=32x32, encoder memory=2, $\tau = 10$

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
| 5 | 0.0794 | 0.0662 | 0.0528 | 0.0401 | 0.0287 | 0.0192 | 0.0121 |
| 10 | 0.0758 | 0.0602 | 0.0443 | 0.0299 | 0.0179 | 0.00988 | 0.0053 |
| 20 | 0.0753 | 0.0588 | 0.0421 | 0.0271 | 0.0153 | 0.00816 | 0.00457 |

Table A.28: Random interleaver size=64x64, encoder memory=2, $\tau = 10$

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| 5 | 0.00723 | 0.00422 | 0.00243 | 0.00144 | 0.000912 | 0.000608 | 0.00043 |
| 10 | 0.00302 | 0.0019 | 0.00125 | 0.000879 | 0.000661 | 0.000501 | 0.000382 |
| 20 | 0.00275 | 0.0018 | 0.00122 | 0.000881 | 0.000666 | 0.000502 | 0.000381 |

Table A.29: Random interleaver size=64x64, encoder memory=2, $\tau = 10$

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
| 5 | 0.116 | 0.107 | 0.0982 | 0.089 | 0.0795 | 0.07 | 0.0608 |
| 10 | 0.116 | 0.107 | 0.098 | 0.0886 | 0.0789 | 0.0692 | 0.0596 |
| 20 | 0.115 | 0.106 | 0.0971 | 0.078 | 0.0876 | 0.0682 | 0.0587 |

Table A.30: Random interleaver size=128x128, encoder memory=2, $\tau = 15$

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| 5 | 0.0517 | 0.0433 | 0.0357 | 0.029 | 0.0233 | 0.0185 | 0.0146 |
| 10 | 0.0502 | 0.0417 | 0.0339 | 0.0272 | 0.0216 | 0.0171 | 0.0136 |
| 20 | 0.0495 | 0.0411 | 0.0334 | 0.0268 | 0.0212 | 0.0168 | 0.0134 |

Table A.31: Random interleaver size=128x128, encoder memory=2, $\tau = 15$

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
| 5 | 0.115 | 0.107 | 0.0974 | 0.0882 | 0.079 | 0.0699 | 0.061 |
| 10 | 0.115 | 0.106 | 0.097 | 0.0876 | 0.0781 | 0.0688 | 0.0598 |
| 20 | 0.115 | 0.106 | 0.0969 | 0.0875 | 0.078 | 0.0687 | 0.0596 |

Table A.32: Random interleaver size=32x32, encoder memory=2, $\tau = 15$

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| 5 | 0.0525 | 0.0445 | 0.0372 | 0.0307 | 0.025 | 0.0201 | 0.0162 |
| 10 | 0.0511 | 0.0429 | 0.0355 | 0.029 | 0.0236 | 0.019 | 0.0152 |
| 20 | 0.0509 | 0.0427 | 0.0354 | 0.0289 | 0.0234 | 0.0189 | 0.0151 |

Table A.33: Random interleaver size=32x32, encoder memory=2, $\tau = 15$

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
| 5 | 0.116 | 0.107 | 0.0975 | 0.0882 | 0.0787 | 0.0693 | 0.06 |
| 10 | 0.115 | 0.106 | 0.0972 | 0.0877 | 0.078 | 0.0683 | 0.0587 |
| 20 | 0.115 | 0.106 | 0.0972 | 0.0877 | 0.078 | 0.0682 | 0.0586 |

Table A.34: Random interleaver size=64x64, encoder memory=2, $\tau = 15$

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| 5 | 0.0511 | 0.0428 | 0.0354 | 0.0287 | 0.023 | 0.0182 | 0.0144 |
| 10 | 0.0495 | 0.041 | 0.0334 | 0.027 | 0.0215 | 0.017 | 0.0134 |
| 20 | 0.0494 | 0.0408 | 0.0333 | 0.0269 | 0.0214 | 0.0169 | 0.0134 |

Table A.35: Random interleaver size=64x64, encoder memory=2, $\tau = 15$

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
| 5 | 0.0661 | 0.0525 | 0.04 | 0.0288 | 0.0196 | 0.0125 | 0.00754 |
| 10 | 0.058 | 0.0428 | 0.0296 | 0.019 | 0.0111 | 0.00619 | 0.00319 |
| 20 | 0.0555 | 0.04 | 0.0267 | 0.0164 | 0.00921 | 0.00491 | 0.00251 |

Table A.36: Random interleaver size=32x32, encoder memory=2, $\tau = 20$

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| 5 | 0.00431 | 0.00234 | 0.00125 | 0.000627 | 0.000352 | 0.0002 | 0.000114 |
| 10 | 0.00156 | 0.000802 | 0.000362 | 0.000171 | 0.000121 | 7.8e-005 | 4.78e-005 |
| 20 | 0.00114 | 0.000576 | 0.00027 | 0.00014 | 8.98e-005 | 6.54e-005 | 4.61e-005 |

Table A.37: Random interleaver size=32x32, encoder memory=2, $\tau = 20$

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
| 5 | 0.0651 | 0.0498 | 0.035 | 0.0224 | 0.0126 | 0.00621 | 0.00269 |
| 10 | 0.0542 | 0.0338 | 0.0168 | 0.00663 | 0.00179 | 0.000406 | 0.000112 |
| 20 | 0.049 | 0.0264 | 0.0108 | 0.00293 | 0.00064 | 0.000125 | 7e-005 |

Table A.38: Random interleaver size=64x64, encoder memory=2, $\tau = 20$

| | *SNR* | | | | | | |
|---|---|---|---|---|---|---|---|
| Iterations | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| 5 | 0.00107 | 0.00039 | 0.000149 | 6.56e-005 | 2.81e-005 | 1.78e-005 | 1.2e-005 |
| 10 | 4.78e-005 | 2.58e-005 | 1.55e-005 | 1.92e-005 | 1.29e-005 | 9.18e-006 | 7.54e-006 |
| 20 | 3.04e-005 | 2.42e-005 | 2e-005 | 1.43e-005 | 1.27e-005 | 9.09e-006 | 7.45e-006 |

Table A.39: Random interleaver size=64x64, encoder memory=2, $\tau = 20$

# Bibliography


[ARI98] ARIB. *Japan's Proposal for Candidate Radio Transmission Technology on IMT-2000:W-CDMA*, 1998.

[BCJR74] 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*, pages 284–287, March 1974.

[BGT93] C. Berrou, A. Glavieux, and P. Thitimajshima. Near Shannon limit error-correcting coding and decoding: Turbo-codes. In *Proc. 1993 IEEE International Conference on Communications, Geneva, Switzerland*, pages 1064–1070, 1993.

[For66] David G. Forney. *"Concatenated Codes"*. MIT Press, Cambridge, Mass., 1966.

[JZ97] Alberto Jiménez and Kamil Sh. Zigangirov. "Periodic time-varying convolutional codes with low-density parity-check matrices". *submitted to IEEE Trans. on Inform. Theory.*, 1997.

[JZ98] Rolf Johannesson and Kamil Sh. Zigangirov. *"Fundamentals of Convolutional Codes"*. IEEE Press, 1998.

[Lin97] Göran Lindell. *"Introduction to Digital Communication"*. Unfinished working manuscript, 1997.

[Sha48] C.E. Shannon. "A Mathematical Theory of Communication". *Bell System Tech. J., vol.27, pp.379-423 and pp.623-656*, 1948.

[Tro96] A. Trofimov. Soft Output Decoding Algorithm for Trellis Codes. Submitted to *IEEE Transactions on Information Theory*, November 1994 (rejected 1996).

[VO79] Andrew J. Viterbi and Jim K. Omura. *"Principles of Digital Communications and Coding"*. McGraw-Hill, 1979.

[Zig98] Kamil Sh. Zigangirov. App Decoding of Convulutional Codes. Submitted to *Probl. Peradachi Inform*, Januari 1998.