Examensarbete utfört i Elektroniksystem vid Tekniska högskolan i Linköping av Johannes Lindblom
LITH-ISY-EX--08/4120--SE
Linköping 2008
Examensarbete utfört i Elektroniksystem
vid Tekniska högskolan i Linköping
av

Johannes Lindblom

LITH-ISY-EX--08/4120--SE

Handledare:  Anton Blad
            ISY, Linköpings universitet

Examinator:  Oscar Gustafsson
             ISY, Linköpings universitet

Linköping, 30 April, 2008
Turbo codes was first presented in 1993 by C. Berrou, A. Glavieux and P. Thitima-jshima. Since then this class of error correcting codes has become one of the most popular, because of its good properties. The turbo codes are able to come very close to theoretical limit, the Shannon limit. Turbo codes are for example used in the third generation of mobile phone (3G) and in the standard IEEE 802.16 (WiMAX).

There are some drawbacks with the algorithm for decoding turbo codes. The decoder uses a Maximum A Posteriori (MAP) algorithm, which is a complex algorithm. Because of the use of many variables in the decoder the decoding circuit will consume a lot of power due to memory accesses and internal communication. One way in which this can be reduced is to make early decisions.

In this work I have focused on making early decision of the encoder states. One major part of the work was also to be sure that the expressions were written in a way that as few variables as possible are needed. A termination condition is also introduced. Simulations based on estimations of the number of memory accesses, shows that the number of memory accesses will significantly decrease.
Abstract

Turbo codes was first presented in 1993 by C. Berrou, A. Glavieux and P. Thitimajshima. Since then this class of error correcting codes has become one of the most popular, because of its good properties. The turbo codes are able to come very close to theoretical limit, the Shannon limit. Turbo codes are for example used in the third generation of mobile phone (3G) and in the standard IEEE 802.16 (WiMAX).

There are some drawbacks with the algorithm for decoding turbo codes. The decoder uses a Maximum A Posteriori (MAP) algorithm, which is a complex algorithm. Because of the use of many variables in the decoder the decoding circuit will consume a lot of power due to memory accesses and internal communication. One way in which this can be reduced is to make early decisions.

In this work I have focused on making early decision of the encoder states. One major part of the work was also to be sure that the expressions were written in a way that as few variables as possible are needed. A termination condition is also introduced. Simulations based on estimations of the number of memory accesses, shows that the number of memory accesses will significantly decrease.
Acknowledgements

First I want to thank my examiner Oscar Gustafsson and my supervisor Anton Blad for their support. I also want to thank my opponent Erik Lindahl and of course my relatives who always have supported my studies.

Finally I also want to thank my office mate Roy Wang. Our time together has been very good when it comes to practicing my poor English.
Nomenclature

Most of the reoccurring abbreviations and symbols are listed and described here.

Symbols

- \( k \): Time index
- \( L \): Constraint length
- \( M = L - 1 \)
- \( K = 2^M \)
- \( s_k \): The encoder state at time \( k \)
- \( m \): Value of encoder state at time \( k \)
- \( m' \): Value of encoder state at time \( k - 1 \)
- \( d_k \): Information bit at time \( k \)
- \( x_k^s = d_k \)
- \( x^s = (x_1^s, x_2^s, \ldots, x_N^s) \)
- \( x_k^p = x_k^p (m', i) \)
- \( x_k = (x_k^s, x_k^p) \)
- \( a_k \)
- \( x^p = (x_1^p, x_2^p, \ldots, x_N^p) \)
- \( x = (x_1, x_2, \ldots, x_N) \)
- \( y_k = (y_k^s, y_k^p) \)
- \( y^p = (y_a, y_{a+1}, \ldots, y_b) \)
- \( y = (y_1, y_2, \ldots, y_N) \)
- \( \sigma^2 \)
- \( \Lambda_c = 2/\sigma^2 \)
- \( R \)
- \( N \)
- \( \Lambda_e (d_k) \)
- \( \alpha_k (m) \)
- \( A_k (m) = \log \alpha_k (m) \)
- \( \beta_k (m) \)
- \( B_k (m) = \log \beta_k (m) \)
- \( \gamma_k (m', i) \)
- \( G_k (m', i) = \log \gamma (m', i) \)
- \( \Lambda_e \): Extrinsic information
- \( \alpha \): Forward state metrics
- \( \beta \): Backward state metrics
- \( \gamma \): Branch metrics
- \( \Lambda \): Code rate
- \( \sigma \): Block length
- \( \alpha \): Noisy version of \( x_k \)
- \( \sigma \): The transmitted codeword
- \( \Lambda \): Noisy received codeword
- \( \sigma \): Noise variance
Abbreviations

AWGN  Additive White Gaussian Noise
APP   A Posteriori Probability
BER   Bit error rate
DEC1  Decoder 1
DEC2  Decoder 2
ENC1  Convolutional encoder 1
ENC2  Convolutional encoder 2
FSM   Finite State Machine
I     Interleaver
I⁻¹   Deinterleaver
LAPP  Logarithm A Posteriori Probability
LLR   Logarithm of Likelihood Ratio
LUT   Look Up Table
MAP   Maximum A Posteriori
NSC   Non Systematic Convolutional
RSC   Recursive Systematic Convolutional
SNR   Signal to Noise Ratio
5 Results

5.1 Used Thresholds .................................................. 33
   5.1.1 Termination Condition .................................... 33
   5.1.2 LLR for Decided Bits .................................... 34
   5.1.3 Likelihood for Deciding State .......................... 34
   5.1.4 Start of Making Early Decisions ....................... 34

5.2 Comments on the Results .................................... 34
   5.2.1 Error Correcting Performance .......................... 34
   5.2.2 Estimation of the Number of Memory Accesses ........ 34
   5.2.3 Number of iterations ................................... 38

5.3 A Mistake ......................................................... 38

6 Discussion and Future Work ................................. 41

   6.1 Discussion .................................................... 41
   6.2 Future Work .................................................. 41

Bibliography .......................................................... 43
Chapter 1

Introduction

This chapter gives a motivation for this thesis work. It will describe the task for the work, the methods used during the work and the outline of this report.

1.1 Background

The theoretical limit of the transmission capacity has been known since Claude E. Shannon in 1948 introduced his famous results, see [8]. Researchers have since then tried to reach this limit.

The turbo codes were invented by C.Berrou, A. Clavieux and P. Thitimajshia in 1993. Turbo codes have become one of the most powerful error correcting codes and one of the error correcting codes which has come closest to the theoretical limit. Because of this, it immediately became one of the important research topics in the area.

Some examples of where Turbo codes nowadays are used are in the third generation of mobile phone and the wireless metropolitan network standard IEEE 802.16 (IEEE stands for Institute of Electrical and Electronics Engineers) also known as WiMAX.

Despite of these good things, there are drawbacks with turbo codes. The decoding algorithm has low throughput and high latency. Furthermore, the energy consumption is high due to the high number of memory accesses and internal chip communication.

1.1.1 A Communication System

In Fig. 1.1 a communication system is shown. The turbo encoder and decoder belongs to the blocks channel coding and channel decoding. The modulator block maps the information symbols onto signals that can be efficient transmitted over the channel.

The received signal is a noisy version of the transmitted signal. The demodulator will produce some hard or soft decision of the transmitted bits. The channel
decoder will hopefully correct the errors that have occurred during the transmission. This is the block that this thesis will treat.

If the reader want to know more about digital communications systems, we refer to [1].

1.2 The Task

The task was to investigate if it is possible to use early decision in the algorithm for decoding turbo codes. The division of Electronics system has earlier studied this area, but for low density parity check (LDPC) codes, see e.g. [5]. This research area is a sensitive area. The theorists mean that early decisions decrease the error correcting capacity. That is true, but with smart decisions the capacity will be only slightly worse and the energy consumption will decrease significantly.

Early decision means that the decoder makes hard decision during the decoding. Normally the hard decision are based on the soft decision after the whole decoding. For soft decision the output will be a likelihood or a likelihood ratio. When the decoder make hard bit decision the output will be 0 or 1.

This work focuses on reduction of the number of memory accesses. But also the number of computations will decrease. Or more properly, the number of computations decrease which leads to a decrease of the number of memory accesses.

1.3 Method of Solving

The starting point for this work was to get an understanding of the decoding algorithm. The way this was done was to implement the original algorithm. After evaluation of the code, early decisions were introduced in different steps.

One important thing was to be sure that no unnecessary memory accesses were done, e.g. if one in advance knows that a value will be zero it is meaningless to
store and calculate with this value.

No complete survey has been made in this work. This means that every code parameter, threshold, limit and word length are not optimal. The ideas was instead to see if it is possible to use this kind of methods.

This work does not contain any implementation in hardware. But there will be discussions about this in meaning of processing elements and use of memory.

The software used for simulations has been developed in C. An estimation of the number of memory accesses has been made. MATLAB has been used to plot the results. Due to the estimations the presented plots are maybe not totally reliable.

\section*{1.4 Report Outline}

A theoretical background of the turbo encoder is given in Chapter 2. In Chapter 3 the original turbo decoder will be described. In Chapter 4 early decisions will be introduced. In both Chapter 3 and 4 some ideas about a hardware implementation will be presented. Chapter 5 contains results from simulations and comments about the result. Some ideas of improvements of the algorithm and future work will be discussed in Chapter 6.

If the reader of this thesis wants to know more about turbo codes there are a paper written by A. Burr [6]. This paper is more accessible than many other papers.
Chapter 2

Turbo Encoders

In this chapter the turbo encoder will be described. It begins with a description of the convolutional codes and the interleaver which are two of the building blocks. In Section 2.4 the parameters used during this work will be presented.

The turbo encoder is built using parallel concatenation of two identical convolutional encoders and one interleaver. It is also possible to use a puncturer to change the code rate. See Fig. 2.1. The output of the turbo encoder consists of the original data and two parity bits per each data bit. The puncturer will remove some of the parity bits.

![Diagram of Turbo Encoder](image)

**Figure 2.1.** Principle of a turbo encoder. The block I is an interleaver and ENC1 and ENC2 are two identical convolutional encoders. PM is the puncturing mechanism.

### 2.1 Convolutional Codes

There are many books that treat convolutional codes. One of these is [11]. The intention of this section is to give a brief introduction.

The approach of the convolutional codes is substantially different from that of block codes, such as BCH and Reed-Solomon codes. Block encoders segment the data stream into blocks of some fixed length \( k \). A convolutional encoder, on the
other hand, encodes the entire data stream, regardless of its length. The common idea is to introduce extra bits, parity bits, to obtain redundancy.

The convolutional code can either be a Non Systematic Convolutional (NSC) code or Recursive Systematic Convolutional (RSC) code. There are also non recursive systematic codes but these will have a higher bit error rate (BER) than NSC. A systematic code is a code which outputs are both the original data stream and a coded data streams.

The differences between these two kind of codes can be seen in Figs. 2.2(a) and 2.2(b) respectively. In this work only RSC codes have been used.

\[
d_k = a_k
\]

(a)

\[
d_k
\]

(b)

**Figure 2.2.** a) Classical non systematic code. b) Recursive systematic code. The blocks with a T are delay elements and the plus signs in a circle are XOR gates

### 2.1.1 Linear Convolutional Encoders

The linear convolutional encoder is based on a shift register, consisting of a series of delay elements, and XOR-gates (eXclusive OR, or addition modulo 2).

Consider a binary rate \( R = 1/2 \) convolutional encoder with constraint length \( L \) and memory \( M = L - 1 \). \( M \) is the number of delay elements in the encoder, see Fig. 2.2. The NSC code can be described by the two encoder generators \( G_1 : \{g_{1i}\} \) and \( G_2 : \{g_{2i}\} \). The generators are generally expressed as a number in octal form.

The input to the encoder at time \( k \) is a bit \( d_k \) and the corresponding codeword \( C_k \) for a NSC is the binary couple \( (x_{k}^{s}, x_{k}^{p}) \) with
2.1 Convolutional Codes

\[
x_{1,k}^p = \sum_{i=0}^{L-1} g_{1i} d_{k-i} \mod 2 \quad g_{1i} \in \{0, 1\} \quad (2.1)
\]

\[
x_{2,k}^p = \sum_{i=0}^{L-1} g_{2i} d_{k-i} \mod 2 \quad g_{2i} \in \{0, 1\} \quad (2.2)
\]

To obtain a binary rate \( R = 1/2 \) RSC code the NSC code is fed back and one of the two outputs from the NSC encoder is set equal to the input bit \( d_k \). Here one have \( x_k^s = d_k \). First an expression for \( a_k \) is derived as

\[
a_k = g_{10} d_k + \sum_{i=1}^{L-1} g_{1i} a_{k-i} \mod 2. \quad (2.3)
\]

Then the second output, \( x_k^p \), is derived as

\[
x_k^p = g_{20} a_k + \sum_{i=1}^{L-1} g_{2i} a_{k-i} \mod 2. \quad (2.4)
\]

Figure 2.3. The generally structure for a RSC encoder with \( M = 4 \).

Equations (2.3) and (2.4) can be derived from Fig. 2.3. \( g_{10} \) always takes the value one. Otherwise the output should be constant zero. [3] shows that \( g_1 \) and \( g_2 \) have odd values \( (g_{jL-1} = 1, \ j = 1, 2) \) for all good codes. For the future work it will be assumed that this holds.

2.1.2 State diagram and trellis

The convolutional encoder is a finite state machine (FSM). Associated with this is a state diagram and a code trellis. A state diagram shows the state transitions and
outputs at given state and input. The trellis is an extension of the state diagram that shows the passage of time.

The state diagram and the trellis are described by Example 2.1.

---

**Example 2.1: State diagram and code trellis**

In this example a RSC code with \( g_1 = 7_8 \) and \( g_2 = 5_8 \) has been used. It has memory length \( M = 2 \) and the number of states is \( K = 2^M = 4 \). Fig. 2.4(a) shows the state diagram for the encoder. The circles correspond to the states. The branches correspond to the state transitions. The symbols on the branches are the input and the outputs (\( d_k/x_k^r,x_k^p \)).

When the trellis is constructed the encoder is considered to start and terminate in state \( m = 0 \). This means that for \( k = 0 \) and \( k = 1 \) only two and four state transitions are allowed respectively. The last \( M \) bits (here 2 bits) are chosen to terminate the encoder. The termination bits \( (d_{N-M+1}, \ldots, d_N) \) are chosen so that \( a_k + d_k = 0 \mod 2 \).

![State Diagram](image)

![Trellis](image)

**Figure 2.4.** a) State diagram for the encoder in Example 2.1. b) Trellis for the encoder in Example 2.1.
2.2 Interleaver

Usually the interleaver is used to protect the transmission against burst error. In turbo codes the interleaver is used to satisfy that the parity data streams are well separated from each other.

In this work a block interleaver has been used. One can think of the block interleaver as an \( m \times n \) matrix. The symbols are stored in the matrix row by row. When the matrix is full, the data is read column-wise. In this work an \( n \times n \) matrix has been used. Example 2.2 gives an explanation of the functionality.

This kind of interleaver is not the best one neither when comes to latency or ability to separate the symbols. The reason why this interleaver has been used is that it is the easiest one to understand and implement. Furthermore the deinterleaver is exactly the same as the interleaver.

---

**Example 2.2: \( n \times n \) block interleaver**

Fig. 2.5 is expected to describe the block interleaver. This example uses a \( 3 \times 3 \) matrix.

\[
\left( a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9 \right)
\]

(a)

(b)

\[
\left( a_1, a_4, a_7, a_2, a_5, a_8, a_3, a_6, a_9 \right)
\]

(c)

**Figure 2.5.** a) Symbol sequence to be interleaved. b) Full interleaver matrix. c) Interleaved data.

The sequence of symbols in Fig. 2.5(a) is fed row by row into the block in-
terleaver in Fig. 2.5(b). The output sequence in Fig. 2.5(c) is obtain by reading column-wise.

In this report will the interleaving be denoted $I(\cdot)$ and the deinterleaving will be denoted $I^{-1}(\cdot)$.

**2.3 Parallel Concatenation of RSC Codes**

It is common to use serial concatenation schemes for error correction. One way to do this is to first code a data block with a block encoder. The output from the block encoder is then interleaved and coded by a convolutional encoder.

With the introduction of RSC codes it is possible to use a new concatenation scheme called parallel concatenation. Both ENC1 and ENC2 inputs uses the same bit $d_k$. Due to the presence of the interleaver the input sequences are different [4]. One article that describe the concatenation in a good way is [6].

It is possible to use a puncturing mechanism to obtain higher code rates. This means that not all information bits (i.e. $d_k$) are sent. Instead some parity bits are removed to obtain the required code rate.

**2.4 Code Specifications**

During this work some assumptions have been made regarding the encoder and the channel. These are presented in this section. The chosen structures are maybe not the most optimal. But the intention of this work was, as described in chapter 1 not to find an optimal encoder. Two works that deal with finding good convolutional codes and interleavers for different SNR are [3] and [13].

**2.4.1 Encoder Specifications**

In this section are the encoder structure, block length, termination, puncturing mechanism and interleaver specified.

**Encoder Structure**

The convolutional encoder has the generators $G_1 = 37_8 = 11111_2$ and $G_2 = 21_8 = 10001_2$, respectively. In Fig. 2.2(b) the chosen RSC encoder is shown.

Both ENC1 and ENC2 start a new block in state $s_0 = 0$ and ENC1 end the block in state $s_N = 0$.

**Block Lengths**

During the simulations data sequences of length $N = 2^{2j}$, where $j$ is an integer, have been used. The last $M$ bits are used to terminate ENC1 into the zero-state.
2.4 Code Specifications

Termination

To know in which state the encoder ends in termination bits are used. This was introduced in Example 2.1.

When termination is used the input to the encoder shift register shall be $a_k = 0$. By rewriting (2.3) the termination bits are derived as

\[ 0 = d_k + \sum_{i=1}^{L-1} g_{1i}a_{k-i} \mod 2 \iff \]
\[ d_k = \sum_{i=1}^{L-1} g_{1i}a_{k-i} \mod 2. \quad (2.5) \]

Only ENC1 is terminated. The reason for this is that the termination bits are interleaved and cannot be used for termination of ENC2. It is possible to put the bits produced in the termination at the end of the code sequence. This means that this termination bits are not interleaved.

Puncturing Mechanism

The puncturing mechanism removes every other parity bit in a special pattern for reducing the code rate to $R = 1/2$. It is not a good idea to remove all bits which corresponds to odd $k$. Instead the parity bits belong to an even $k$ from ENC1 and an odd $k$ from ENC2 are removed.

Also the parity bits produced during the termination of the encoders are removed. The bit sequence $x_{N-M+1}^N = (x_{N-M+1}^s, \ldots, x_N^s)$ are not information bits. This leads to an exact code rate of $R = (N-M)/2N$.

To be able to obtain the same bit error rate (BER) the transmitter have to send with more power than for an unpunctured code. On the other hand fewer bits are sent.

Interleaver specification

The interleaver was chosen as an $n \times n$ block interleaver. As told in Section 2.2 this interleaver is not optimal, but it is the most simple one to explain and implement in software.

2.4.2 Channel Specifications

The channel is considered to be an additive white gaussian noise (AWGN) discrete memoryless channel. This assumption will result in mathematical simplicity after taking the logarithm. Because of the central limit theorem (noise is sum of many sources) this is a relevant assumption. Antipodal signaling has been used. This means that instead of sending zeros and ones the sent symbols are $-1$ and $+1$. 
The received symbols can be described as the random variables $y_{k}^s$, $y_{1,k}^p$ and $y_{2,k}^s$ where

\begin{align*}
    y_{k}^s &= 2x_{k}^s - 1 + n_{k}^s \quad (2.6) \\
    y_{1,k}^p &= 2x_{1,k}^p - 1 + n_{1,k}^p \quad (2.7) \\
    y_{2,k}^p &= 2x_{2,k}^p - 1 + n_{2,k}^p \quad (2.8)
\end{align*}

and $n_{k}^s$, $n_{1,k}^p$ and $n_{2,k}^p$ are three independent noises with the same variance $\sigma^2$. 
Chapter 3

The Turbo Decoder

It is common to use the Viterbi algorithm [9] together with convolutional encoders. This algorithm is an optimal decoding method which minimizes the probability of sequence error for convolutional codes. One wants the turbo decoder to output a posteriori probabilities, or soft decision. Unfortunately the original Viterbi algorithm outputs hard decisions.

The Bahl et al. (Sometimes called the BCJR algorithm.) algorithm [2] is a relevant algorithm, which minimizes the bit error probability in decoding convolutional and linear block codes. This algorithm yields the APP for each decoded bit. Because of the use of RSC codes, the Bahl algorithm must be modified in order to take into account their recursive character.

The Bahl et al. algorithm is a Maximum A Posteriori (MAP) algorithm. The MAP algorithm minimizes the probability of bit errors by using the entire received sequence to identify the most probable bit at each stage of the trellis. The MAP algorithm does not constrain the set of bit estimates to necessarily correspond to a valid path through the trellis. The Viterbi decoder always identifies the most probable valid path through the trellis [2].

The first the notation will be explained in Section 3.1. Then the so called modified Bahl et al. algorithm will be explained in Section 3.2. After that the algorithm is adapted for iterative decoding, see Section 3.3. In Section 3.4 the log-MAP algorithm will be introduced to simplify the computations and reduce the complexity. The computational path will be described by Algorithm 3.1 in Section 3.5. At the end of this chapter, in Section 3.6, the hardware implementation will be discussed. This will focus on the use of memory.

A schematic view of the decoder can be seen in Fig. 3.1.

The algorithm described in this chapter is a mix between the algorithms in [4] and [2]. The ideas are from the tutorial paper [7], but the algorithm has been modified to better suit a implementation.
3.1 Notation

The notation for time indexes can be confusing. The intention of this section is to unravel it.

The encoder begins in time \( k = 0 \), but the first information bit has time index \( k = 1 \). Encoding a block of \( N \) bits leads to an end state for the encoder at \( k = N \). Therefore the encoder passes \( N + 1 \) states when the block length is \( N \).

For more readable expression the parity bits are denoted \( x^p_k \). Actually they are a function of the encoder state \( s_{k-1} = m' \) and the information bit \( d_k = i \).

The two functions \( f(i, m) \) and \( b(i, m) \) are introduced. \( f(i, m) \) is the next state given an input \( i \) and an encoder state \( m \) and \( b(i, m) \) is the encoder state going backwards in time from state \( m \) on the previous branch corresponding to input \( i \). This two functions are described by the Algorithms 4.2 and 4.3 respectively.

3.2 Modified Bahl et al. algorithm for RSC codes

The output from the decoder should be the Logarithm of the Likelihood Ratio, LLR, \( \Lambda (d_k) \), that is associated with the decoded bit \( d_k \). \( \Lambda (d_k) \) is the logarithm APP (LAPP). This value holds information about the certainty of the bit \( d_k \). A large positive value indicates that it is a strong one. In the same way a large negative value indicates a strong zero. The LLR is calculated as

\[
\Lambda (d_k) = \log \frac{Pr\{d_k = 1|\text{observation}\}}{Pr\{d_k = 0|\text{observation}\}}.
\]  

(3.1)

The decoder can make a hard decision by comparing \( \Lambda (d_k) \) to a threshold equal to zero. See (3.2).

\[
\hat{d}_k = 1 \quad \text{if} \quad \Lambda (d_k) > 0 \\
\hat{d}_k = 0 \quad \text{if} \quad \Lambda (d_k) < 0
\]  

(3.2)

Consider a RSC code with constraint length \( L \). At time \( k \) the encoder state \( s_k \) is a \( M \)-tuple
\[ s_k = (a_{k-1}, a_{k-2}, \ldots, a_{k-M}) \] (3.3)

where \( a_l, \ l = k, \ldots, k - M + 1 \) are calculated in (2.3) and \( M \) is the length of the encoder memory.

Suppose that the information bit sequence \( \{d_k\} \) is made up of \( N \) independent bits \( d_k \) where \( Pr\{d_k = 0\} = Pr\{d_k = 1\} = 1/2. \)

The output from the encoder, i.e. the codeword sequence

\[ x = \{x_1, \ldots, x_k, \ldots, x_N\} \] (3.4)

is the input to a discrete gaussian memoryless channel whose output is the sequence

\[ y = \{y_1, \ldots, y_k, \ldots, y_N\}. \] (3.5)

\( x_k \) and \( y_k \) can be written as

\[ x_k = (x^s_k, x^p_k) = (d_k, x^p_k) \] (3.6)

and

\[ y_k = (y^s_k, y^p_k) \] (3.7)

respectively.

Incorporating the code’s trellis, (3.1) can be written as [7]

\[ \Lambda (d_k) = \log \frac{\sum_{m'} p(s_{k-1} = m', s_k = f(1, m'), y)/p(y)}{\sum_{m'} p(s_{k-1} = m', s_k = f(0, m'), y)/p(y)}. \] (3.8)

Observe that it is possible to cancel \( p(y) \) in (3.8). This means that it only requires an algorithm for computing \( p(m', i, y) = p(s_{k-1} = m', s_k = f(i, m'), y) \). The Bahl et al. algorithm [2] for doing this is

\[ p(m', i, y) = \alpha_{k-1}(m') \gamma_k(m', i) \beta_k(f(i, m')) \] (3.9)

where

\[ \gamma_k(m', i) \triangleq p(y_k, d_k = i|s_{k-1} = m'). \] (3.10)

\[ \alpha_k(m) \triangleq p(s_k = m, y^p_1), \] often called the forward state metric, is computed recursively for \( k = 1, 2, \ldots, N \) as

\[ \alpha_k(m) = \sum_{i=0}^{1} \alpha_{k-1}(b(i, m)) \gamma_k(b(i, m), i). \] (3.11)

This expression is obtained by summing over all paths leading into the state \( m \). However there are only two possible paths leading to \( m \). One path corresponding to \( i = 0 \) and the other to \( i = 1. \)

The encoder state for \( k = 0 \) is known to be 0. This leads to the boundary conditions
The Turbo Decoder

\[ \alpha_0 (m) = \begin{cases} 1, & m = 0 \\ 0, & m \neq 0 \end{cases} \]

The probabilities \( \beta_k (m) \triangleq p (y_{k+1}^N | s_k = m) \) in (3.9) are computed in a backward recursion for \( k = 1, 2, \ldots, N \) in the same way as for \( \alpha_k (m) \):

\[ \beta_{k-1} (m') = \sum_{i=0}^{1} \beta_k (f (i, m')) \gamma_k (m', i). \] (3.12)

\( \beta_k (m) \) is called the backward state metric. When the encoder terminates in state 0 after \( N \) input bits the boundary conditions of \( \beta_k (m) \) for \( k = N \) are

\[ \beta_N (m) = \begin{cases} 1, & m = 0 \\ 0, & m \neq 0 \end{cases} \]

A graphical representation of the calculations of \( \alpha_k (m) \) and \( \beta_k (m) \) are shown in Fig. 3.2.

**Figure 3.2.** Graphical representation of calculations of \( \alpha_k (m) \) and \( \beta_k (m) \).

The canceling of the divisor \( p (y) \) made in (3.8) leads to a numerically unstable algorithm. This problem can be solved by including the division by \( p (y) / p (y_k) \) in the Bahl et al. algorithm by defining the modified probabilities

\[ \tilde{\alpha}_k (m) = \frac{\alpha_k (m)}{p (y_1^k)} \] (3.13)

and

\[ \tilde{\beta}_k (m) = \frac{\beta_k (m)}{p (y_1^k)}. \] (3.14)

Dividing (3.9) by \( p (y) / p (y_k) = p (y_1^{k-1}) p (y_{k+1}^N | y_1^k) \) one obtain

\[ p (m', i | y) = \tilde{\alpha}_{k-1} (m') \gamma_k (m', i) \tilde{\beta}_k (f (i, m')). \] (3.15)
Since \( p(\mathbf{y}_1^k) = \sum_m \alpha_k (m) \), the values \( \tilde{\alpha}_k (m) \) may be computed as

\[
\tilde{\alpha}_k (m) = \alpha_k (m) / \sum_m \alpha_k (m). \tag{3.16}
\]

It is not desirable to store both \( \{\alpha_k (m)\} \) and \( \{\tilde{\alpha}_k (m)\} \). By using (3.11) in (3.16) it is possible to obtain a recursion involving only \( \{\tilde{\alpha}_k (m)\} \).

\[
\tilde{\alpha}_k (m) = \frac{\sum_{i=0}^{l} \alpha_{k-1} (b(i,m)) \gamma_k (b(i,m), i)}{\sum_{n} \sum_{i=0}^{l} \alpha_{k-1} (b(i,n)) \gamma_k (b(i,n), i)}
\]

\[
= \frac{\sum_{i=0}^{l} \tilde{\alpha}_{k-1} (b(i,m)) \gamma_k (b(i,m), i)}{\sum_{n} \sum_{i=0}^{l} \tilde{\alpha}_{k-1} (b(i,n)) \gamma_k (b(i,n), i)} \tag{3.17}
\]

In (3.17) the second equality follows by dividing the numerator and the denominator by \( p(\mathbf{y}_1^{k-1}) \). In the denominator the state index \( m \) has been changed to \( n \) for avoiding mixing up the summation index and the parameter. By noticing that

\[
p(\mathbf{y}_1^N | \mathbf{y}_1^{k-1}) = p(\mathbf{y}_1^k) p(\mathbf{y}_1^{k+1} | \mathbf{y}_1^k) \frac{p(\mathbf{y}_1^{N} | \mathbf{y}_1^{k})}{p(\mathbf{y}_1^{k-1})}
\]

\[
= \sum_m \sum_{i=0}^{l} \alpha_{k-1} (b(i,m)) \gamma_k (b(i,m), i) \frac{p(\mathbf{y}_1^{N} | \mathbf{y}_1^{k})}{p(\mathbf{y}_1^{k-1})} \tag{3.18}
\]

it is possible to obtain the recursion for \( \tilde{\beta}_k (m) \) by dividing (3.12) by (3.18).

\[
\tilde{\beta}_{k-1} (m') = \frac{\sum_{i=0}^{l} \tilde{\beta}_k (f(i,m')) \gamma_k (m', j)}{\sum_{n} \sum_{i=0}^{l} \tilde{\alpha}_{k-1} (b(i,n)) \gamma_k (b(i,n), i)} \tag{3.19}
\]

Notice that the denominators in (3.17) and (3.19) are the same. Define the variable \( \tilde{\alpha}_k^{\text{Denom}} \). When \( \tilde{\alpha}_k (m) \) is computed \( \tilde{\alpha}_k^{\text{Denom}} \) is used. For \( \tilde{\beta}_{k-1} (m'), \tilde{\alpha}_k^{\text{Denom}} \) is also used. Notice the time indexes. This means that it is enough to calculate them once. Also notice that \( \tilde{\alpha}_k (m) \) and \( \tilde{\beta}_k (m) \) have the same boundary conditions as \( \alpha_k (m) \) and \( \beta_k (m) \), respectively.

The LAPP ratio \( \Lambda(d_k) \) for the modified Bahl et al. algorithm is calculated by combining (3.8) and (3.15).

\[
\Lambda(d_k) = \log \frac{\sum_{m'} \tilde{\alpha}_{k-1} (m') \gamma_k (m', 1) \tilde{\beta}_k (f(1, m'))}{\sum_{m'} \alpha_{k-1} (m') \gamma_k (m', 0) \tilde{\beta}_k (f(0, m'))} \tag{3.20}
\]

Computations of the probabilities \( \gamma_k (m', i) \) will be discussed in Section 3.3. The described algorithm works well in a software implementation, but a hardware implementation will employ the so called log-MAP algorithm. This is described in Section 3.4.
First one need to rewrite $x_k$ and $y_k$ from (3.6) and (3.7) respectively as

$$x_k = \left( x^s_k, x^p_{1,k}, x^p_{2,k} \right) = \left( d_k, x^p_{1,k}, x^p_{2,k} \right)$$  \hspace{1cm} (3.21)$$

and

$$y_k = \left( y^s_k, y^p_{1,k}, y^p_{2,k} \right).$$  \hspace{1cm} (3.22)$$

In the expression below are only $x^p_k$ and $y^p_k$ used. But for DEC1 shall the subscript be 1, $k$ and 2, $k$ for DEC2.

By using Bayes’ rule, the LAPP ratio for an arbitrary MAP decoder can be written as

$$\Lambda (d_k) = \log \frac{Pr\{y|d_k = 1\}}{Pr\{y|d_k = 0\}} + \log \frac{Pr\{d_k = 1\}}{Pr\{d_k = 0\}}$$  \hspace{1cm} (3.23)$$

where the second term is the a priori information. Since $Pr\{d_k = 1\} = Pr\{d_k = 0\} = 1/2$ typically, this term is usually zero. But for iterative decoders, DEC1 receives extrinsic or soft information for each $d_k$ from DEC2. This information serves as a priori information to DEC1. In the same way, DEC2 receives extrinsic information from DEC1. The idea behind extrinsic information is that DEC2 provides soft information to DEC1 for each $d_k$, using information, which not is available to DEC1 (i.e. parity bits from ENC2). DEC1 does likewise for DEC2.

The name Turbo Codes comes from the fact that the iterative scheme of the decoder can be compared to a turbo engine. Often the word turbo is used when something is very good. The turbo codes are good but there is a parallel to the turbo engines where the energy in the exhaust gases are used to give the engine an extra boost.

The second term in (3.23) is defined as

$$\Lambda^e (d_k) \triangleq \log \frac{Pr\{d_k = 1\}}{Pr\{d_k = 0\}}.$$  \hspace{1cm} (3.24)$$

Now it is possible to show how the extrinsic information is extracted from the modified Bahl et al. version of the LAPP ratio (3.23). First it can be observed that $\gamma_k (m’, i)$ can be written as

$$\gamma_k (m’, i) = p(y_k|d_k = i, s_{k-1} = m’) Pr\{d_k\}.$$  \hspace{1cm} (3.25)$$

The definition in (3.24) can be written as

$$Pr\{d_k\} = \frac{\exp \left[-\Lambda^e (d_k) / 2 \right]}{1 + \exp \left[-\Lambda^e (d_k) / 2 \right]} \exp \left[d_k \Lambda^e (d_k) / 2 \right] = A_k \exp \left[d_k \Lambda^e (d_k) / 2 \right].$$  \hspace{1cm} (3.26)$$
To proof the first equality in (3.26) define $P_+ \triangleq P_r \{ d_k = 1 \}$ and $P_- \triangleq P_r \{ d_k = 0 \}$ respectively. Then $\exp [\Lambda^e (d_k)]$ can be expressed as $P_+/P_-$. 

$$\frac{\sqrt{P_-/P_+}}{1 + P_-/P_+} \sqrt{P_+/P_-} = P_+ \text{ when } d_k = 1 \text{ and } \frac{\sqrt{P_-/P_+}}{1 + P_-/P_+} \sqrt{P_+/P_-} = P_- \text{ when } d_k = 0.$$ 

Recall $y_k = (y_k^e, y_k^p)$ and $x_k = (x_k^e, x_k^p) = (d_k, x_k^p)$. Now $p (y_k | d_k = i, s_{k-1} = m')$ can be written as

$$p (y_k | d_k = i, s_{k-1} = m') \propto \exp \left[ -\frac{(y_k^e - (2d_k - 1))^2}{2\sigma^2} - \frac{(y_k^p - (2x_k^p - 1))^2}{2\sigma^2} \right] = \exp \left[ -\frac{y_k^e^2 + y_k^p^2}{2\sigma^2} \right] \exp \left[ \frac{(2d_k - 1) y_k^e + (2x_k^p - 1) y_k^p}{2\sigma^2} \right] = B_k \exp \left[ \frac{(2d_k - 1) y_k^e + (2x_k^p - 1) y_k^p}{2\sigma^2} \right].$$

(3.27) Remember that $x_k^p$ actually is a function of $m'$ and $i$. The expression would be even more messed up if this would be written.

Equations 3.26 and 3.27 together give

$$\gamma_k (m, i) \propto A_k B_k \exp [d_k \Lambda^e (d_k)/2] \exp \left[ \frac{(2d_k - 1) y_k^e + (2x_k^p - 1) y_k^p}{2\sigma^2} \right].$$

(3.28) Since $\gamma_k (m', i)$ appears in numerator and denominator of (3.20) the factor $A_k B_k$ will be canceled. The reason for is that this factor is independent of $d_k$. Let $\Lambda_c \triangleq \frac{2}{\sigma^2}$ where $\sigma^2$ is the noise variance of the channel. Also define

$$\gamma_k^c (m', i) \triangleq \exp \left[ \frac{1}{2} \Lambda_c y_k^p (2x_k^p - 1) \right].$$

(3.29) Now it is possible to write (3.28) as

$$\gamma_k (m, i) \propto \exp \left[ \frac{1}{2} (2d_k - 1) (\Lambda^e (d_k) + \Lambda_c y_k^p) + \frac{1}{2} \Lambda_c (2x_k^p - 1) d_k \right] \gamma_k^c (m', i).$$

(3.30) Inserting (3.30) in (3.20) results in the expression

$$\Lambda (d_k) = \log \sum_{m'} \hat{\alpha}_{k-1} (m') \gamma_k^c (m', 1) \hat{\beta}_k (b(1, m')) C_k (1) \sum_{m'} \hat{\alpha}_{k-1} (m') \gamma_k^c (m', 0) \hat{\beta}_k (b(0, m')) C_k (0) = \Lambda_c y_k^p + \Lambda^e (d_k) + \log \sum_{m'} \hat{\alpha}_{k-1} (m') \gamma_k^c (m', 1) \hat{\beta}_k (b(1, m')) \gamma_k^c (m', 0) \hat{\beta}_k (b(0, m'))\ (3.31)$$
where \( C_k (d_k) \triangleq \exp \left[ \frac{1}{2} (2d_k - 1) (\Lambda^e (d_k) + \Lambda_c y_k^e) \right] \). Sometimes the first term in (3.31) is called channel value, the second term represents the a priori information about \( d_k \) provided by a previous decoder. The last term represents the extrinsic information that can be passed to a subsequent decoder. For any iteration, the decoder computes

\[
\Lambda (d_k) = \Lambda_c y_k^e + \Gamma^{-1} (\Lambda_{21}^e (d_k)) + \Lambda_{12}^e (d_k)
\]  

(3.32)

where \( \Lambda_{21}^e (d_k) \) is the extrinsic information passed from DEC2 to DEC1. Remember that \( \Gamma^{-1} (\cdot) \) is the deinterleaver operator. \( \Lambda_{12}^e (d_k) \) is to be used as extrinsic information from DEC1 to DEC2. When it comes to implementation only \( \Lambda_{21}^e (d_k) \) and \( \Lambda_{12}^e (d_k) \) respectively will be computed. After the last iteration (3.32) will be computed.

### 3.4 log-MAP decoding

The decoding algorithm contains a lot of multiply operations. This cause a high decoding complexity which likely would decrease. This can be achieved by taking the logarithm of the algorithm. Remember that this already is done by the computation of the extrinsic information. Now, the multiplications in the algorithm are converted to additions, which are much easier to implement. Unfortunately the computations essentially resorts to such calculations as

\[
f(x_1, x_2, \ldots, x_n) = \log \sum_{i=1}^{n} \exp [x_i].
\]  

(3.33)

For the two-variable case the Jacobian logarithm is

\[
f(x_1, x_2) = \log (e^{x_1} + e^{x_2})
\]

\[
= \max (x_1, x_2) + \log \left( 1 + e^{-|x_1 - x_2|} \right)
\]

\[
= \max (x_1, x_2) + f_c (|x_1 - x_2|)
\]  

(3.34)

where \( f_c (x) = \log (1 + e^{-x}) \), \( x > 0 \) is a correction function which correct the error induced by the max approximation. This function or approximations of it can easily be implemented in a look up table (LUT) [10].

Together with the DEC1 Jacobian logarithm the operator \( \mathcal{E} \) is introduced:

\[
\mathcal{E}_{i=1}^{n} x_i = x_1 \mathcal{E} x_2 \mathcal{E} \ldots \mathcal{E} x_n = f(x_1, x_2, \ldots, x_n)
\]  

(3.35)

Define

\[
A_k (m) = \log \tilde{\alpha}_k (m)
\]  

(3.36)

\[
B_k (m) = \log \tilde{\beta}_k (m)
\]  

(3.37)

\[
G_k (m', i) = \log \gamma_k (m', i)
\]  

(3.38)

\[
G_k^e (m', i) = \log \gamma_k^e (m', i)
\]  

(3.39)
3.5 Decoding Algorithm

Now the MAP algorithm becomes

\[ \Lambda^e (d_k) = E \sum_{m' = 0}^{2^M - 1} A_k (m') + G_k^e (m', 1) + B_k (f (m', 1)) \]
\[ - E \sum_{m' = 0}^{2^M - 1} A_k (m') + G_k^e (m', 0) + B_k (f (m', 0)) \] (3.40)

\[ A_k (m) = \varepsilon_{i=0}^1 A_{k-1} (b (i, m)) G_k (b (i, m), i) \]
\[ - \varepsilon_{n=0}^{2^M - 1} \varepsilon_{i=0}^1 A_{k-1} (b (i, n)) G_k (b (i, n), i) \] (3.41)

\[ B_{k-1} (m') = \varepsilon_{i=0}^1 B_k (f (i, m')) G_k (m', i) \]
\[ - \varepsilon_{n=0}^{2^M - 1} \varepsilon_{i=0}^1 A_{k-1} (b (i, n)) G_k (b (i, n), i) . \] (3.42)

From (3.29) and (3.30)

\[ G_k^e (m', i) = \frac{1}{2} \Lambda_c y_k^p (2x_k^p - 1) \] (3.43)
\[ G_k (m', i) = \frac{1}{2} d_k (\Lambda^e (d_k) + \Lambda_c y_k^p) + G_k^e . \] (3.44)

are obtained respectively.

3.4.1 Boundary conditions for \( A \) and \( B \)

When the log-MAP algorithm is introduced it is also necessary to change the boundary conditions for \( A_0 (m) \) and \( B_N (m) \). For DEC1 the boundaries are

\[ A_0 (m) = \begin{cases} 0, & m = 0 \\ -\infty, & m \neq 0 \end{cases} \]

and

\[ B_N (m) = \begin{cases} 0, & m = 0 \\ -\infty, & m \neq 0 \end{cases} \]

respectively. ENC2 does not terminate after one block, but it begins in state 0. Therefore DEC2 \( A_0 \) is initialized as for DEC1 and \( B_N (m) = A_N (m) \).

3.5 Decoding Algorithm

As some kind of conclusion this section will present the decoding algorithm based on the earlier equations in this chapter. The algorithm is described in Algorithm 3.1.

--- Algorithm 3.1: log-MAP turbo decoding algorithm ---

This algorithm does not take care about puncturing. To adapt it for the case that puncturing is used one can notice that \( G_k^e (m', i) \) will not be computed for all \( k \).

The algorithm have a termination condition: When all LLRs have had the same sign in three iterations the decoder will terminate. To control this the flag \( frm \) is used.

Initialization
The decoder receives the sequence $y$, see (3.5).

The noise variance is estimated as

$$\sigma^2 = \frac{1}{3N-1} \sum_{k=1}^{N} \left[ |y_k^s|^2 + |y_{1,k}^p|^2 + |y_{2,k}^p|^2 \right].$$

(3.45)

Then $\Lambda_c = 2/\sigma^2$ can be obtained. If the noise variance is known in advance it is not necessary to compute $\sigma^2$ and $\Lambda_c$. $1/2 \cdot \Lambda_c y_{1,k}^p$, $1/2 \cdot \Lambda_c y_{2,k}^p$ and $\Lambda_c y_k^s$ will be stored instead of $y_{1,k}^p$, $y_{2,k}^p$ and $y_k^s$ respectively. If puncturing is used one does not need to store all the parity symbols.

- Set $f_{rm} = 1$

**Initialization of DEC1**

- $A_0 (m)$ and $B_N (m)$ are initialized as

$$A_0 (m) = \begin{cases} 0, & m = 0 \\ -\infty, & m \neq 0 \end{cases}$$

and

$$B_N (m) = \begin{cases} 0, & m = 0 \\ -\infty, & m \neq 0 \end{cases}$$

respectively.

- Since there are no extrinsic information for the DEC1 in the first iteration

$$\Lambda_{21}^e (d_k) = 0, \quad k = 1, 2, \ldots, N.$$

By a smart implementation this initialization is unnecessary.

**Initialization of DEC2**

- $A_0 (m)$ is initialized as

$$A_0 (m) = \begin{cases} 0, & m = 0 \\ -\infty, & m \neq 0 \end{cases}.$$  

- Since ENC2 does not terminate $B_N (m)$ cannot be initialized here.

**The $n$:th iteration: DEC1**

for $k = 1, 2, \ldots, N$:

- Get the values of $y_k^s$, $y_{1,k}^p$ and $\Lambda_{21}^e (d_k)$.

- Compute the second term of (3.41) as

$$A_k^{\text{Denom}} = \mathbf{c}_m^2 \mathcal{L}_m^1 A_{k-1} (b(i,m)) G_k (b(i,m), i)$$
where the superscript Denom stands for Denominator, compare to (3.17). For reduction of the number of stored variables, \( G_k (m', i) \) and \( G^c_k (m', i) \) are not computed in advance. Therefore, \( G_k (m', i) \) is replaced by

\[
G_k (m, i) = \frac{1}{2} d_k (I^{-1} (A_{21}^c (d_k)) + \Lambda_c y_k^p) + \frac{1}{2} \Lambda_c y_{1,k}^p x_k^p
\]

which is obtained by inserting (3.43) in (3.44). The value of \( A^\text{Denom}_k \) is stored in a memory.

- \( A_k (m) \) is then obtained as

\[
A_k (m) = \mathcal{E}^1 \sum_{i=0}^{m-1} A_{k-1} (b (i, m)) G_k (b (i, m), i) - A^\text{Denom}_k \quad m = 0, \ldots, 2^M - 1
\]

and stored in a memory.

for \( k = N, N - 1, \ldots, 1 \):

- \( B_{k-1} (m') \) can be computed as

\[
B_{k-1} (m') = \mathcal{E}^1 \sum_{i=0}^{m-1} B_k (f (i, m')) G_k (m', i) - A^\text{Denom}_k.
\]

Only the last set of \( B_k (m) \) needs to be stored. That means that the memory for all \( B_k (m) \) only needs to hold \( K = 2^M \) values.

- Finally the extrinsic information is computed as

\[
\Lambda_{12}^c (d_k) = \mathcal{E} \sum_{m'=0}^{2^M-1} \mathcal{E}^1 \sum_{i=0}^{m-1} A_{k-1} (b (i, m)) G_k (b (i, m), i) - \mathcal{E} \sum_{m'=0}^{2^M-1} A_k (m') + G^c_k (m', 0) + B_k (f (m', 0))
\]

where \( G^c_k (m', i) \) is replaced by (3.43). The value of \( \Lambda_{12}^c (d_k) \) is stored in a memory which also can be used as an interleaver. By computing this value in the reverse order the memories does not need to hold all values of \( B_k (m) \).

**The \textit{n}:th iteration: DEC2**

for \( k = 1, 2, \ldots, N \):

- Get the values of \( y_k^s, y_{2,k}^p \) and \( \Lambda_{12} (d_k) \).
- Compute the second term of (3.41) as

\[
A^\text{Denom}_k = \mathcal{E} \sum_{m=0}^{2^M-1} \mathcal{E}^1 \sum_{i=0}^{m-1} A_{k-1} (b (i, m)) G_k (b (i, m), i)
\]

where \( G_k (m', i) \) is replaced by

\[
G_k (m, i) = \frac{1}{2} d_k (I (\Lambda_{12}^c (d_k)) + \Lambda_c y_k^p) + \frac{1}{2} \Lambda_c y_{2,k}^p (2x_k^p - 1).
\]

The value of \( A^\text{Denom}_k \) is stored in a memory.
A\_k (m) is then obtained as
\[ A_k (m) = \epsilon_1^{i=0} A_{k-1} (b (i, m)) G_k (b (i, m), i) - A_k^{\text{Denom}}, \quad m = 0, \ldots, 2^M - 1 \]
and stored in a memory.

\(B_k\) is initialized as \(B_N (m) = A_N (m)\) for all \(m\),
for \(k = N, N - 1, \ldots, 1\):

- \(B_{k-1} (m')\) can be computed as
  \[ B_{k-1} (m') = \epsilon_{i=0}^{1} B_k (f (i, m')) G_k (m', i) - A_k^{\text{Denom}}. \]

- The extrinsic information is computed as
  \[
  \Lambda_{e21} (d_k) = \epsilon_{m'=0}^{2M-1} A_k (m') + G_k^e (m', 1) + B_k (f (m', 1)) \\
  - \epsilon_{m'=0}^{2M-1} A_k (m') + G_k^e (m', 0) + B_k (f (m', 0)) \tag{3.46}
  \]
where \(G_k^e (m', i)\) is replaced by (3.43). The value of \(\Lambda_{e21} (d_k)\) is stored in a memory which also can be used as an interleaver.

- Compute
  \[ \Lambda (d_k) = \Lambda_{c y_k} + I^{-1} (\Lambda_{e21} (d_k)) + \Lambda_{e12} (d_k) \]
and store the value. If the new \(\Lambda (d_k)\) has the same sign as the old value, then \(f_{\text{term}} := f_{\text{term}}\) else \(f_{\text{term}} := 0\).

**After the \(n\):th iteration**

- If \(f_{\text{term}}\) is equal to a threshold the decoder is terminated. Otherwise \(f_{\text{term}} := f_{\text{term}} + 1\).
- If the decoder terminated the hard decisions are made for all \(k\) as
  \[ \hat{d}_k = 1 \quad \text{if} \quad \Lambda (d_k) > 0 \]
  \[ \hat{d}_k = 0 \quad \text{if} \quad \Lambda (d_k) < 0 \]

A schematic view of how the values are exchanged between the subdecoders is shown in Fig. 3.1.

### 3.6 Implementation of the log-MAP algorithm

In this section will the implementation of Algorithm 3.1 be discussed. This text will focus on the use of memory. There will also be some comments on the computing elements. In this section only the log-MAP algorithm will be considered.
3.6 Implementation of the log-MAP algorithm

First, all the $G_k^{e} (m', i)$ and $G_k (m', i)$ values will be computed. Together, there are $2 \cdot 2^M = 2^{M+2}$ values. If all these values should be stored they need to be read twice for each time $k$ and $G_k (m', i)$ will be recomputed each iteration. By noticing that $G_k^{e}$ and $G_k$ can be computed from the three values $\frac{1}{2} \cdot \Lambda_c y_k^p$, $L_s y_k^p$ and $\Lambda^e (d_k)$ it is sufficient to only store these values. The cost of this is more computations (additions, shifts, and LUTs). After the first iteration only $\Lambda^e (d_k)$ needs to be stored.

In (3.17) and (3.19) the denominators are the same except for the time index. This means that this value only needs to be calculated once. On the other hand the value needs to be stored in a memory. But it is still better than compute the value again. The functions $f (i, m')$ and $b (i, m)$ can be implemented in LUTs.

In the paper [10] a number of possible implementations of the jacobian logarithm are described. It shows how it can be implemented and how the correction function can be approximated. DEC1 and DEC2 can be implemented in the same block with shared memories. Also there need to be a memory for storing $\Lambda (d_k)$. In the software implementation the decoding is terminated when all the values of $\Lambda (d_k)$ have had the same sign in the last three iterations. This value (three) can be changed. If all bits are the same in this number of iterations it is very likely that they will not change.

If only $\Gamma^{-1} (\Lambda_{21}^e (d_k))$ is used for making hard decisions all available information will not be used, see (3.31). One can also think of how much the hardware implementation can be parallelized. The more parallel it will be, the less will the latency be, but at the same time that will demand more hardware.

### 3.6.1 Use of Memory

In table 3.1 the memory used for one iteration (i.e. DEC1 and DEC2) are listed. This table does not say anything about the word lengths. That is something that must be done in future work.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Number of words</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_k^p$</td>
<td>$N$</td>
</tr>
<tr>
<td>$y_{1,k}^p$</td>
<td>$N$, if puncturing is not used</td>
</tr>
<tr>
<td>$y_{2,k}^p$</td>
<td>$N$, if puncturing is not used</td>
</tr>
<tr>
<td>$A_k (m)$</td>
<td>$N \cdot 2^M$</td>
</tr>
<tr>
<td>$B_k (m)$</td>
<td>$2^M$</td>
</tr>
<tr>
<td>$\Lambda_{1,2}^e (d_k)$ and $\Lambda_{21}^e (d_k)$</td>
<td>$N$</td>
</tr>
<tr>
<td>$\Lambda (d_k) = \Lambda_c y_k^s + \Lambda_{1,2}^e (d_k) + \Lambda_{21}^e (d_k)$</td>
<td>$N$</td>
</tr>
</tbody>
</table>

**Table 3.1.** Use of memory in a hardware implementation.

By computing $\Lambda^e (d_k)$ directly after the computing of $B_k (m)$ it is sufficient to only save the last set of $B_k$. 

Chapter 4

Turbo Decoding Using Early State Decisions

As one can see in Chapter 3 the decoding algorithm uses a lot of variables. There are some different strategies to reduce the number of memory accesses in the decoding algorithm. Some of these will be discussed more comprehensively in Chapter 6. The strategy chosen in this work is to decide the encoder state at each time index.

If the value of $A_k(m)$ is very close to zero in some sense it means that it is very likely that the encoder state at corresponding time index was $m$.

As can be seen in the trellis diagram in Chapter 2, Fig. 2.4(b), the number of branches are reduced if a state is known (i.e. decided). This idea is showed by Example 4.1.

Example 4.1: Trellis diagram when state decision is used

This example will use the trellis from Example 2.1 with $N = 9$. At time $k = 4$ and time $k = 6$ the states are decided to $m = 1$ and $m = 2$ respectively. Fig. 4.1 shows what happens with the trellis when the state at $k = 4$ are decided. The total number of state transitions are here reduced from 52 to 32.

In the same way Fig. 4.1 shows what happens when the states at both $k = 4$ and $k = 6$ are decided. Notice that for $k = 5$ only $m = 0$ is possible. This also leads to the fact that information bits $d_4$ and $d_5$ will be decided, because there is only one branch out from the corresponding state.

The trellis will now have only 20 possible state transitions instead of 52 for the trellis where no decision have been done.

To be able to determine the possible state transitions to and from the states at time $k$ one must know the possible states at times $k-1$, $k$ and $k+1$. Variations of equation (2.3) will be frequently used in this chapter when the possible transitions will be determined.
4.1 How to Find Possible States

To find the possible states at $k$ the first thing to do is to check if the state at this time is decided or not. If it was decided it is very easy, there is only one possible state. If it was not decided it is harder to determine possible states. One needs to check the states in the time interval $[k - M + 1, k + M - 1]$. Begin with the timestamps closest to $k$. If one of these states are decided this will affect the possible states at time $k$.

The developed algorithm is searching both backward and forward from times $k$. From both directions a set with possible states for $k$ are returned. By computing the intersection between this two sets the possible states are achieved.

Recall Example 4.1. When the decoder stands at time $k = 5$ it knows that the state for $k = 4$ and $k = 6$ are decided. The backward search will return the set $\{0, 2\}$ and the forward search returns the set $\{0, 1\}$. The intersection between the two sets is then $\{0\}$.

The variables holding information whether the states are decided or not are
called \( D^*_k \). If the state at time \( k \) is decided \( D^*_k \) takes the value of the state. This value is in the interval \([0, 1, \ldots, K-1]\). If the state is not decided \( D^*_k = -K \). With this values it is enough to look at the most significant bit of \( D^*_k \) to determine if the state is decided or not.

Each subdecoder (e.g. DEC1 and DEC2) has one set of these variables. The subdecoders have to take care about states for \( N+1 \) time stages (remember that the encoder begins at \( k = 0 \) and ends at \( k = N \)). The first state is always known because the subdecoders know that the encoders begin in state 0. To calculate the possible states there need to be a window of size \( 2M - 1 \) over the variables that tell whether a state is decided or not. Consider the decoder to work at time \( k \). The window will then be the multiple

\[
W_k = \left( D^*_{\text{max}\{0,k-M+1\}}, \ldots, D^*_{\text{min}\{N,k+M-1\}} \right).
\] (4.1)

Let the binary vector \( S^{\text{poss}} \) hold information about whether a state is possible or not. If element \( i \) is 1 it means that state \( i \) is possible. Let the vectors \( S^{\text{poss}}_{\text{bw}} \) and \( S^{\text{poss}}_{\text{fw}} \) hold information about the possible state for a backward search and a forward search respectively. Then

\[
S^{\text{poss}} = S^{\text{poss}}_{\text{bw}} \& S^{\text{poss}}_{\text{fw}}
\] (4.2)

where \& is the bitwise and operator.

---

**Algorithm 4.1: Determination of possible states**

Assume the decoder to be at time \( k \) and the states at time \( k - a \) and time \( k + b \) are decided. \( a \) and \( b \) are positive integers less than \( M \) and no states are decided inside the interval \([k - a, k + b]\) but maybe outside.

```plaintext
for j = 0; j < 2^M; j = j + 1;
S^{\text{poss}}_{\text{bw}} [j] = 0
S^{\text{poss}}_{\text{fw}} [j] = 0
for i = 0; i < 2^a; i = i + 1;
S^{\text{poss}}_{\text{bw}} [D^*_{k-a} \gg a + i \ll a] = 1
for i = 0; i < 2^b; i = i + 1;
S^{\text{poss}}_{\text{fw}} [D^*_{k-a} \ll b + i] = 1
```

In Algorithm 4.1 two types of shift operators were used. \( a \ll b \) stands for \( b \) steps logic shift to the left of the variable \( a \). In the same way are the operator \( \gg \) is the logic shift to the right.

### 4.2 Possible State Transitions

A problem that needs to be solved is to check if a state transition is allowable. The decoder stands in state \( m' \) at time \( k \) and it needs to derive the state \( m \) for time \( k+1 \) given the bit \( d_{k+1} = i \). The state \( m \) can be derived from the function \( f(i, m') \) which is described by Algorithm 4.2.
Algorithm 4.2: Determination of the next state - \( m = f(i, m') \)

The \( M - 1 \) most significant bits in \( m' \) are equal to the \( M - 1 \) least significant bits in \( m \). The most significant bit in \( m \) can be obtained as

\[
a_k = i + \sum_{j=1}^{L-1} g_{1j} a_{k-j} \mod 2 \tag{4.3}
\]

where \( m' = (a_{k-1}, \ldots, a_{k-L+1}) \). \( m \) can now be written as

\[
m = m' \gg 1 + a_k 2^M. \tag{4.4}
\]

The other problem is to derive state \( m' \) at time \( k - 1 \) when the state is \( m \) for time \( k \) and \( d_k = i \). The state \( m' \) can be derived from the function \( b(i, m) \) which is described by Algorithm 4.3.

Algorithm 4.3: Determination of the last state - \( m' = b(i, m) \)

The \( M - 1 \) least significant bits in \( m \) are equal to the \( M - 1 \) most significant bits in \( m' \). By rewriting (4.3) one obtains

\[
a_{k-L+1} = i + \sum_{j=1}^{L-2} g_{1j} a_{k-j} \mod 2. \tag{4.5}
\]

Note that the assumption that both \( g_1 \) and \( g_2 \) are odd was made in section 2.4.1. That mean that \( g_1 L - 1 = 1 \). Now \( m' \) can be written as

\[
m' = m \ll 1 + a_{k-L+1}. \tag{4.6}
\]

To check whether the derived state is possible or not, one need to check if elements \( m' \) and \( m \) of \( S^{\text{poss}} \) are equal to one. If they are, the state is possible.

4.3 Decided Bits

No early bit decisions are made by taking the LLR into consideration. But with the knowledge of which states that are decided it is possible to decide the bits. This knowledge can be utilized by both DEC1 and DEC2. Unlike the information about which states that are decided the two subdecoders can share the information of the decided bits.

A bit, \( d_k \), is decided when the states at time \( k - 1 \) and time \( k \) are decided. The value of the bits LLR will now be set to a threshold. The number of possible state transitions will be halved for the stage corresponding to \( d_k \). If DEC1 has decided the states for time \( k - 1 \) and time \( k \), the corresponding bit will be decided. The
memory that holds the information about the state of the bits is interleaved. Say that the bit after interleaving has time index \( l \). Bit \( d_l \) are now decided, but that does not mean that the states at time \( l - 1 \) and time \( l \) are decided. But if one of them are decided it is possible to decide the other one.

Algorithm 4.4 describes how the bit can be decided.

---

**Algorithm 4.4: Decision of \( d_k \) when \( m \) and \( m' \) are decided**

Let \( m' \) be the state at time \( k \) and \( m \) be the state at time \( k + 1 \). The \( M - 1 \) most significant bits in \( m' \) are the same as the \( M - 1 \) least significant bits in \( m \). The value of \( d_k \) can be derived as

\[
d_k = a_k + \sum_{j=1}^{L-1} g_{1j} a_k - j \mod 2
\]

(4.7)

where \( a_k \) is the most significant bit in \( m \) and \( m' = (a_{k-1}, \ldots, a_{k-L+1}) \).

---

### 4.4 Implementation

To be able to implement this modified algorithm some additional variables are needed. When a state has been decided this has to be stored in a memory. First the decoder needs to know if a state is decided or not. After that it needs to know the value of the state. To store this information for both DEC1 and DEC2 there must be two memories with word length \( M + 1 \) (\( M \) bits for the state and 1 bit to decided if the state is decided or not).

The reason why there must be two memories to store the variables above is that the encoder states for ENC1 and ENC2 can not be related due to the interleaving. The use of these variables leads to more memory access, but as one can see in Chapter 5 it is worth this small performance loss.

When it comes to hardware implementation with fix point arithmetics there will be problem when one will decide whether a state is likely enough to be decided. The value of \( A_k \) needs to be very close to 0. But with quantization too large values will be rounded to 0. To solve this problem one can look at the second largest value. If that value is less than a threshold it is possible to decide the state. Example 4.2 shows how this idea works.

---

**Example 4.2: How to decide if a state can be decided.**

Assume that the threshold for deciding a state is \( A_k \) \( > \) \( (m) \) \( > \) \( -10^{-5} \). By quantization this value will be rounded to zero. The threshold corresponds to the probability \( e^{10^{-5}} \approx 1 - 10^{-5} \). This mean that the total probability of the other states need to be less than this threshold.

Assume that there are four possible states and that they are equal likely, \( p = e^a \) where \( a \) is a threshold. Then \( a \) can be calculated as

\[
3p < 10^{-5} \iff a = \log \frac{10^{-5}}{3} < -13.
\]
This example shows that checking if the second greatest value is less than a given threshold is sufficient for deciding whether a state should be decided or not.

The information about which states that are possible can be hold by two LUTs. One LUT is for $S_{bw}^{\text{poss}}$ and the other one for $S_{fw}^{\text{poss}}$. In each direction it is possible to take $M - 1$ steps. For each time there are $K = 2^M$ possible states. Therefore the LUTs need to hold $(M - 1)2^M$ words, each word with the length $2^M$, in each direction. The contents in the LUTs can be calculated as in Algorithm 4.1.

The window in (4.1) can be implemented in a shift register. At each time stage a new value is read from decided state memory and put in the register.

The results of Algorithms 4.2-4.4 can be implemented in LUTs. Note that for Algorithm 4.4 it seems to be $2^{2M}$ combinations, but only $2^{M+1}$ of them are possible. So with some smart representation it is sufficient to do the implementation so that only $2^{M+1}$ values need to be stored.

### 4.4.1 Use of Memory

In table 4.1 the memory used for one iteration (i.e. DEC1 and DEC2) are listed. Again this table does not say anything about the word lengths, and hence that is something to consider in future work.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Number of words</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y^s_k$</td>
<td>$N$</td>
</tr>
<tr>
<td>$y^p_{1,k}$</td>
<td>$N$, if puncturing is not used</td>
</tr>
<tr>
<td>$y^p_{2,k}$</td>
<td>$N$, if puncturing is not used</td>
</tr>
<tr>
<td>$A_k (m)$</td>
<td>$N \cdot 2^M$</td>
</tr>
<tr>
<td>$B_k (m)$</td>
<td>$2^M$</td>
</tr>
<tr>
<td>$\Lambda_{12} (d_k)$ and $\Lambda_{21} (d_k)$</td>
<td>$N$</td>
</tr>
<tr>
<td>$\Lambda (d_k) = \Lambda_c y^s_k + \Lambda_{12} (d_k) + \Lambda_{21} (d_k)$</td>
<td>$N$</td>
</tr>
<tr>
<td>Decided states DEC1</td>
<td>$N - 1$</td>
</tr>
<tr>
<td>Decided states DEC2</td>
<td>$N$</td>
</tr>
<tr>
<td>Decided bits</td>
<td>$N$</td>
</tr>
</tbody>
</table>

Table 4.1. Use of memory in a hardware implementation of early state decision method.

In total there are $N + 1$ state for ENC1 and ENC2 for each block. Both encoders start in a known state and ENC1 ends in a known state. That is why the length of the decided states memory for DEC1 need to have the length $N - 1$ and DEC2 the length $N$.

The word lengths of the variables in the decided states memories is $M + 1$. The decoder need one bit to determine whether the state is decided or not and $M$ bits for knowing which state it is decided to. In the same way the word length for the variables in the decided bits memory is two bits. One bit is for telling the decoder if the bit is decided or not. The other bit is the value of the decided bit (0 or 1).
Chapter 5

Results

In this chapter the results of the simulations are presented. The encoder parameters has been chosen as in Section 2.4.1. The block lengths was chosen to $N = 256$, $N = 1024$, $N = 4096$ and $N = 16384$ respectively. The number of blocks simulated for each block length is shown in Table 5.1.

<table>
<thead>
<tr>
<th>Block length, $N$</th>
<th>Number of blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>16384</td>
</tr>
<tr>
<td>1024</td>
<td>4096</td>
</tr>
<tr>
<td>4096</td>
<td>1024</td>
</tr>
<tr>
<td>16384</td>
<td>256</td>
</tr>
</tbody>
</table>

Table 5.1. Number of blocks and bits in simulation for each $N$. The total number of bits per block length is $2^{22} = 4194304$ bits.

The threshold used for the simulations is listed in Section 5.1. The results of the simulations are shown and commented in Section 5.2. Some mistakes were made when the initial code was written. This will be mentioned and discussed in Section 5.3.

5.1 Used Thresholds

During the simulations some thresholds have been used. They are listed in this section.

5.1.1 Termination Condition

The decoder will terminate when all bits have had the same sign of their LLRs the last three iterations. If the decoder has not terminated before, it will terminate after 30 iterations. For example it is a large difference between doing 5 or 15 iterations. If the LLRs have had the same sign the last three iteration, it is likely that they will hold them. The decoder can not run towards infinity. Somewhere it needs to be terminated. Maybe 30 iterations would be enough.
This threshold are applied to both the early decision method and the original algorithm.

5.1.2 LLR for Decided Bits
When a bit has been decided, it has been given the value 35 or -35 for one or zero, respectively. This corresponds to \( Pr\{d_k = 1\} \approx 1 - 6.7 \cdot 10^{-16} \) or \( Pr\{d_k = 0\} \approx 1 - 6.7 \cdot 10^{-16} \), respectively.

5.1.3 Likelihood for Deciding State
The threshold for deciding a state was that the second largest value of \( A_k \) should be less than -22. This leads to a probability of \( m \) that is greater than \( 1 - 4 \cdot 10^{-9} \).

5.1.4 Start of Making Early Decisions
The decoder starts to make early decisions after three iterations. After looking at the results in Sections 5.2.1 and 5.2.2 this could maybe have been made earlier if the likelihood threshold was set to a higher value.

5.2 Comments on the Results
In this section will the results of the simulations be shown and commented.

5.2.1 Error Correcting Performance
As one can see in Fig. 5.1 there are only some minor losses for the early decision method compared to the original algorithm. For the shortest block length the error correcting capacity are slightly better for the early decision method. The loss in error correcting capacity is so low that the profits of using the early decision method balances it.

5.2.2 Estimation of the Number of Memory Accesses
From the plots in Fig. 5.2 and Fig. 5.3 one can see that the profit of using the early decision method is pretty large for some values of the noise. For the set of threshold used in this work the profit is about 15 – 30% for large SNR. For very low SNR the early decision algorithm does need more memory accesses than the original one. The reason for this is that the early decision algorithm will check if a state is decided. But for low SNR the likelihood for a decided state is very small. On the other hand, one never wants to send at such low SNR.

The profit is the largest for long blocks. For \( N = 16384 \) \( SNR = 1.1 \) dB the early decision algorithm only needs 40% of the number of memory accesses that the original algorithm needs. The greatest profit are made around \( SNR = 1.2 \) dB for all these block lengths. This point is where there is a knee in the error probability plot and the slope of the curve is smaller than earlier.
Figure 5.1. Simulated bit error rates with and without early state decisions. The dashed lines are the result of simulations with the original algorithm.
Figure 5.2. The mean number of memory access with and without early state decisions for different values of the noise. The dashed lines are the result of simulations with the original algorithm.
Figure 5.3. Relative number of memory access for the early decision method compared to the original algorithm.
5.2.3 Number of iterations

The number of iterations used says something about the latency in the decoder. The differences between the two methods are small, see Fig. 5.4. It seems like the early decision method uses slightly fewer iterations.

5.3 A Mistake

One should remember that the number of memory access is an estimate that will give an indication of the real number of memory access. After the last large simulations I remembered that the same code was used for both with and without early state decision. That means that the variables that stores information about decided bits and states are checked even if early decisions are not used.

I choose to do a new simulation, but with less number of blocks. The large simulations above simulated $2^{24}$ bits for each block length and noise level for $\text{SNR} \gtrsim 0.9\text{dB}$. Now are the number of simulated bits $2^{22}$ for all noise levels. The less number of bits leads to plots, which are not so smooth as they would be for a higher number of bits.

Despite the described mistake I think that the differences between using early decision and the original algorithm are significant. I think that one need to see a hardware implementation to be sure about the real profits of this method.
Figure 5.4. The mean number of iterations with and without early state decisions. The dashed lines are the result of simulations with the original algorithm.
Chapter 6

Discussion and Future Work

The results of this work and the design decisions will be discussed in this chapter.

6.1 Discussion

As one can see in Chapter 5 the method with early state decisions works well. For low SNR this method is slightly worse than the original one. The reason for this is that the algorithm will look for early decisions in the memories, but there will not be any early decisions with that much of noise. During the writing of this report I came up with the idea that it is possible to introduce a flag that tells if any early decision has been made. If there are no early decisions these memories will not be checked. In fact it is very seldom the transmission goes over this bad links.

I have not tried to find the best interleaver and code generators as described in Chapter 2. The result may become even better with a better encoder. It would be interesting to see how much the performance could be inflicted by the choice of better codes and interleavers. Two works that deal with this are [3] and [13].

There are many papers that try to find efficient number representations and implementations of the original log-MAP algorithm or the max-log-MAP-algorithm which does not use the correction function in the jacobian logarithm. One paper that describes a method for complexity reduction is [12]. This method uses reduced block lengths for symbols with unreliable detection after some initial iterations.

We have not found any paper that describes that method that are developed in this thesis. Maybe this method can be used together with the method in [12].

To be able to get smoother BER-curves for high SNR it maybe is better to simulate until a fixed number of bit errors (e.g. 1000 bit errors) have occurred. I have made the simulations will a fixed number of bits.

6.2 Future Work

It will be very interesting to see a working hardware implementation of the new algorithm described here. There are some important design decisions to make.
One of these are how the variables should be read from the memories - in parallel or serial. Another important thing to work more with is the word lengths. In this work I only had time for recognizing some problems that will come up when a fix point implementation is used. Especially it would be interesting to study how different approximation of the function $f_c(x)$ will affect the decoding performance.

In this work I have focused on the reduction of the number of memory access for reduction of power. One can think of how much less power the decoder will consume with this algorithm. There are some overhead hardware, it is maybe necessary to use some other processing elements. But still there are fewer number computations so I imagine that this will be better.

It is still possible to work with the thresholds in the decoder. In which iteration shall the decoder starts making early decisions and how sure must the decoder be to make a decision?

During the work I tried to use early bit decision. That means that if the absolute value of the LLR is greater than a threshold the bit will be decided to a corresponding value. I could not get this method to work. I do not think that it should gives as good results as early state decision in meaning of the number of memory accesses, but it would be easier to implement.

Notice that the method that I have developed also have information about decided bits. But the decisions are only made on the only possible state transition between two decided states. This is information that can be used by the other subdecoder.

I have only worked with a AWGN-channel in this thesis. It would be interesting to see what will happens in fading channels. One thing that needs to be changed is the calculations of $G_k$ and $G_k^c$ in Chapter 3. The corresponding equations need to be modified to fit a rician or rayleigh distribution.

To find the best thresholds one can use a fix SNR and then change the thresholds. It is possible that the optimal threshold will differ between different values of SNR.
Bibliography


Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/

© Johannes Lindblom