On the M*-BCJR and LISS Algorithms for Soft-Output Decoding of Convolutional Codes

Jacob Larsson and Fredrik Wesslén

Department of Information Technology
Lund University

Advisors: Rolf Johannesson and Maja Lončar

October 20, 2006

Printed in Sweden
E-huset, Lund, 2006

Abstract

To communicate reliably over noisy wireline or wireless channels, algorithms that add redundancy are used to help reconstruct the transmitted sequence at the receivers. Such error control coding algorithms are today used in all digital communication systems. Optimum decoding algorithms such as the Viterbi and BCJR have very good performances but often too high complexity. Therefore, less complex suboptimum algorithms are of interest.

In this thesis, two suboptimum algorithms for soft output decoding, M*-BCJR and LISS, are evaluated. Their behavior in terms of bit error rate, quality of L-values, and complexity are analyzed. An Additive White Gaussian Noise (AWGN) channel is considered. M*-BCJR and LISS are based on two different decoding strategies, trellis based decoding and sequential decoding, respectively. The simulation results show that M*-BCJR performs better than LISS in terms of bit error rate, quality of L-values, and complexity. When decoding a stand alone code both algorithms perform well, but in an iterative decoding scheme for serially concatenated convolutional codes, the results are less satisfying.

The soft outputs of LISS show a strong linear tendency (with depth) which distorts the quality of the L-values. To tackle this problem a variation of the LISS algorithm is proposed, LISSOM, in which certain states are merged. The LISSOM decreases the complexity but keeps the bit error rate at the same level for stand alone decoding. However, in an iterative decoding scheme the results are improved compared to LISS but still not satisfying.

A suggestion for future work is to further develop the idea behind LISSOM. This may include an investigation of the metric used in LISS aiming to improve the quality of L-values.

Table of Contents

Chapter 1

Introduction

1.1 Scope and purpose

Optimum decoding algorithms such as the Viterbi and the BCJR algorithms have trellis state complexities increasing exponentially with memory of the encoder. Using such algorithms becomes computationally infeasible for longer encoder memories; therefore, suboptimum reduced-complexity algorithms have been developed. However, it is desirable that the decreased complexity does not affect the bit error rate (BER) performance significantly.

In this report the main goal is to investigate relation between the complexity of two algorithms and the quality of their soft outputs as well as their BER performances. The two algorithms are

- The M*-BCJR algorithm, developed by Costello et al. [1], which is a reduced-complexity trellis-based algorithm. It follows the structure of the BCJR algorithm, but dynamically constructs a simplified trellis during the forward recursion. In each trellis section, only the M states with the largest forward metrics are preserved, similarly to the M-BCJR algorithm. Unlike the M-BCJR, however, the remaining states are not deleted, but rather merged into the surviving states.
- The List Sequential (LISS) algorithm, proposed by Hagenauer et al. [2], which is a tree-oriented algorithm based on sequential decoding. It employs a modified stack algorithm for finding a list of the most probable paths, which is used for obtaining approximate symbol reliabilities. The complexity of finding such a list is roughly independent of the trellis complexity of the used code.

The two algorithms are implemented for stand alone decoding and as constituent decoders for iterative decoding of short concatenated convolutional codes. Their behaviors in terms of complexity, bit error rate, and quality of the soft output are analyzed and compared. The complexity is determined by the number of visited nodes in the trellis or tree. The results are also compared with the BCJR algorithm, the optimum maximum *a posteriori* decoding algorithm, serving as a benchmark.

1.2 Reader's guideline

The outline of this thesis is the following. In the Chapter 2 basics of convolutional coding, the channel model used in the simulations, and the BCJR algorithm are presented. This chapter serves as a foundation for the following chapters, where the different decoding algorithms are explained. In Chapters 3 and 4, the principles of the M*-BCJR and LISS algorithms are described, respectively. The simulation results illustrating the performance of the algorithms are presented and analyzed. In end of Chapter 4 an improvement of the LISS algorithm, called LISSOM, is proposed. In Chapter 5, M*-BCJR, LISS, and LISSOM are compared based on simulations results. Finally, Chapter 6 contains the conclusions.

Chapter 2
Convolutional Coding

2.1 Model of a communication system

A generic telecommunication system model is illustrated in Figure 2.1, first described by Shannon in 1948 [3]. A fundamental idea is that communication is essentially digital [4]. The information is expressed as binary digits which are either zero or one.

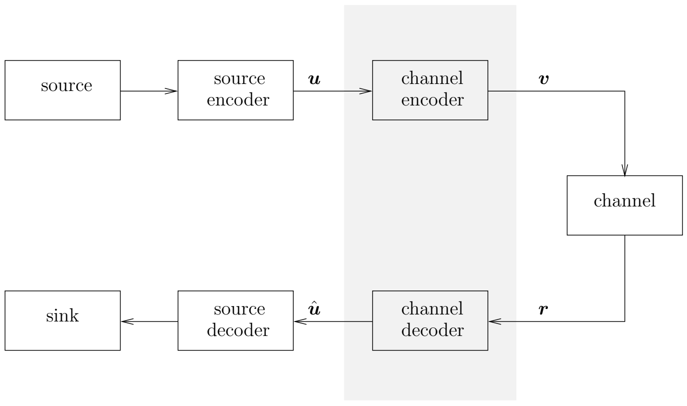


Figure 2.1: A model of a communication system.

The communication system consists of seven elements. We assume that the source data, which could be of any form (e.g., picture or music) is already converted into digital form [5]. The binary information sequence is fed into the source encoder, whose task is to remove the redundant part of source output and produce a compressed version of the source output called the information sequence $\boldsymbol{u}$. Disturbances in the channel can very easily destroy the information sequence. Therefore, some redundancy is added by the channel encoder to obtain the code sequence $\boldsymbol{v}$. The codes considered here are binary convolutional codes (explained in Section 2.3). There is a one-to-one correspondence between $\boldsymbol{u}$ and $\boldsymbol{v}$.

Before sending the code sequence $\boldsymbol{v}$ over the channel, it is mapped into a bipolar sequence $\boldsymbol{x}$ according to $0 \to \sqrt{E_s}$ and $1 \to -\sqrt{E_s}$. The sequence $\boldsymbol{x}$ is transmitted over a noisy channel. The channel distorts the transmitted sequence and the received sequence $\boldsymbol{r}$ is not necessarily the same as $\boldsymbol{x}$. The channel model is presented in Section 2.2. At the receiver, the sequence $\boldsymbol{r}$ is processed by a channel decoder, whose goal is to eliminate errors caused by the channel and produce a decision for the codeword $\hat{\boldsymbol{v}}$ and, hence, for the information sequence $\hat{\boldsymbol{u}}$. Because of the one-to-one correspondence it is clear that $\hat{\boldsymbol{u}} = \boldsymbol{u}$ if and only if $\hat{\boldsymbol{v}} = \boldsymbol{v}$ [6]. A decoding error occurs when $\hat{\boldsymbol{u}} \neq \boldsymbol{u}$, that is, when the decoded sequence differs from the transmitted sequence. From $\hat{\boldsymbol{u}}$ the source decoder constructs a decision for the source output [5] and feeds it to the sink. This thesis focuses on the channel coding and decoding, marked as the gray-shaded area in Figure 2.1.

2.2 Channel model

The channel considered here is an Additive White Gaussian Noise (AWGN) channel with binary input. The noise has a constant power spectral density (PSD), i.e., on the average it disturbs all frequencies equally, and its amplitude follows the Gaussian distribution. If the channel introduces AWGN with zero mean and one-sided PSD N_0, then the output is a Gaussian random variable with variance $\sigma^2 = \frac{N_0}{2}$ [6]. Assume bipolar mapping $0 \to \sqrt{E_s}, 1 \to -\sqrt{E_s}$, where E_s is the signal energy which is often set equal (normalized) to 1. The channel is completely characterized by the following conditional probability functions

$$p(r|x = -\sqrt{E_s}) = \frac{1}{\sqrt{\pi N_0}} e^{-\frac{(r+\sqrt{E_s})^2}{N_0}}$$
$$p(r|x = \sqrt{E_s}) = \frac{1}{\sqrt{\pi N_0}} e^{-\frac{(r-\sqrt{E_s})^2}{N_0}}.$$

To generate the continuous AWGN channel outputs the r250 random number generator (RNG) was used [7]. This RNG has good properties and is well tested [8]. It creates a random number uniformly distributed in the interval $[0, 1]$. Two such random numbers, u_1 and u_2, generated by RNG, are used in the Box-Müller transform to create a random number x from a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$ according to [9]

$$x = \sqrt{-2\ln(u_1)} \cos(2\pi u_2) \cdot \sigma + \mu.$$

Assuming that the all-zero codeword was transmitted the mean value is $\mu = \sqrt{E_s}$, since a 0 is mapped to $\sqrt{E_s}$. The noise variance is calculated according to

$$\sigma = \sqrt{\frac{N_0}{2}} = \sqrt{\frac{E_s}{2R_{\text{system}}} \cdot 10^{-\frac{E_b/N_0\ [\text{dB}]}{10}}}.$$

The ratio $\frac{E_s}{N_0}$ is called the *signal-to-noise ratio per transmitted symbol* [5]. The *signal-to-noise ratio per information bit* is given by

$$\frac{E_b}{N_0} = \frac{E_s}{RN_0}$$

where R is the code rate (R will be explained in Section 2.3). The constant L_c, defined as

$$L_c = \frac{4E_s}{N_0},$$

is often called channel reliability factor. When the signal energy E_s is normalized we have $L_c = 4/N_0$.

2.3 Convolutional codes

The task of a channel encoder is to add properly designed redundancy to the information sequence [6]. An encoder of a convolutional code has a tuple k bits as input, denoted $\boldsymbol{u}_t$, where t is time index. It produces an encoded sequence (code sequence) of n-tuples denoted $\boldsymbol{v}_t$. The encoder has memory denoted m. Each memory element is a delay element, which delays the bits one time instance. Thus, the encoded block depends not only on the corresponding k-bit input tuple at a certain time, but also on the m previous input tuples. The code $\mathcal{C}$ is a set of all possible encoded output sequences that the encoder can produce. An important parameter is the code rate defined as $R = k/n$, that is, it is the number of input bits divided by the number of output bits.

The linear encoding rule is specified by a $k \times n$ generator matrix $\mathbf{G}(D)$. An example of a convolutional encoder is shown in Figure 2.2. This is the encoder whose generator matrix is $\mathbf{G}(D) = (1 + D + D^2 \;\; 1 + D^2)$, or (7 5) in octal notation. If we rewrite the octal number in binary form, the 1's indicate taps. 7_8 and 5_8 are 111_2 and 101_2 in binary form, respectively; hence, there are three connections at the first output and two at the second output. The output sequence is produced by adding the incoming information bit with the bits in the memory elements, where connections are located. A binary alphabet is considered and addition is performed modulo 2. For example, consider the information sequence $\boldsymbol{u} = 1\,1\,0\,1\,0\,0\,0\,0\ldots$, where the leftmost bit enters the encoder first, and assume that the encoder is initially in the

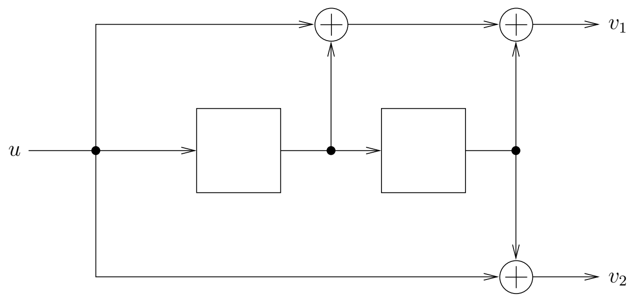


Figure 2.2: The (7 5) convolutional encoder.

all-zero state. Using modulo 2 addition, the encoded sequence $\boldsymbol{v}$ = 11 01 01 00 10 11 00 00 . . . is obtained. A rate $R = k/n$ encoder is called systematic if the k information bits appear unchanged among the n code symbols. The remaining bits are then called parity bits.

In this thesis, the generator matrices are given in octal notation obtained by combining binary 3-tuples into octal numbers, starting from the left. If necessary, additional padding with zeros is performed from the right to achieve full binary 3-tuples. For example, the systematic generator matrix $\mathbf{G}(D) = (1 \;\; 1{+}D{+}D^3{+}D^4)$ corresponds to (1 11011) in binary form, or, after padding with zeros (100 110110) which is (4 66) in octal form.

2.4 Encoder descriptions

At a given time instance t, the contents of the delay elements are called the state of an encoder, denoted as a binary vector $\boldsymbol{\sigma}_t$. The encoding as well as the decoding process can be associated with a series of state transitions. These can be graphically represented in several ways [5]. The encoder state-transition diagram provides a good insight into the behavior of an encoder, see Figure 2.3. Every vertex is an encoder state and a branch between two vertices is labelled with the input/output tuple $\boldsymbol{u}_t/\boldsymbol{v}_t$ corresponding to that state transition.

Another diagram which shows the encoder dynamics over time, called a trellis, is depicted in Figure 2.4. This diagram also uses encoder states $\boldsymbol{\sigma}_t$ as vertices, and the transitions between the states are illustrated by branches. For memory m, rate $R = 1/n$ encoders there are 2^m vertices at each trellis depth, corresponding to possible states at each time instance. Usually, the all-zero state is on the top and the time axis is directed from left to right. The trellis representation is used for decoding as well.

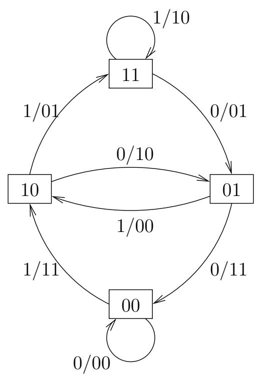


Figure 2.3: The state-transition diagram for the $(7\ 5)$ convolutional encoder.

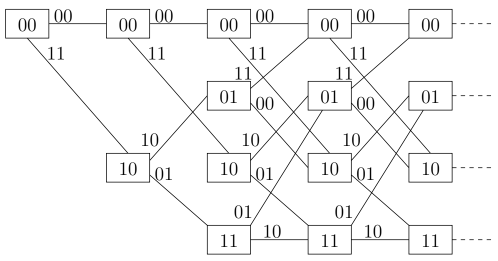


Figure 2.4: The code trellis for the $(7\ 5)$ convolutional encoder.

Another diagram, depicted in Figure 2.5, which also shows the dynamics over time is the code tree [10]. Every node in the code tree corresponds to an encoder state $\boldsymbol{\sigma}_t$. The starting state, usually the all-zero state, corresponds to the leftmost node. This is called the *root* of the tree. For a rate $R = k/n$ encoder there are 2^k branches leaving every node. Every outgoing branch corresponds to an input; when $k = 1$ there are two outgoing branches per node; the upward branch corresponds to input 0 and the downward branch corresponds to input 1. The branches are labeled with the encoder output. From the root to a particular state a certain number of steps is required. This number is called the depth in the tree, denoted d. During simulations it is assumed that the encoder starts and ends in the all-zero state:

$$\boldsymbol{\sigma}_{\text{start}} = \boldsymbol{\sigma}_{\text{end}} = \mathbf{0}.$$

To force the encoder back to the all-zero state, (at most) m dummy bits are

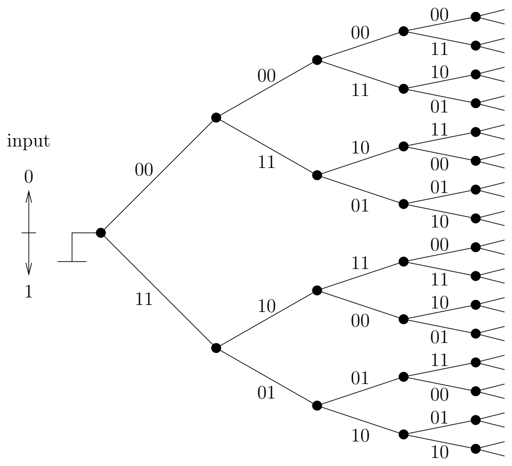


Figure 2.5: Code tree for the (7 5) convolutional encoder.

needed for a rate $R = 1/n$ encoder. This leads to a decrease in the rate of the system

$$R_{\text{system}} = R_{\text{encoder}} \frac{i}{i+m}$$

where i is the number of information bits that will be encoded, m the memory of the encoder, and $R_{\text{encoder}} = 1/n$.

2.5 Optimum decoding

As mentioned before, the information sequence $\boldsymbol{u}$ is encoded into a code sequence $\boldsymbol{v}$ which is sent over the channel. If the channel is perfect, the receiver will receive the code sequence $\boldsymbol{v}$ and therefore also the correct information sequence $\boldsymbol{u}$ [5]. When the channel is noisy the code sequence $\boldsymbol{v}$ will get corrupted. From the received sequence $\boldsymbol{r}$, the receiver must make a decision about the corresponding information sequence $\hat{\boldsymbol{u}}$. There exist two optimal decoding strategies: the maximum *a posteriori* symbol decoding and the maximum *a posteriori* sequence decoding. The first one minimizes **symbol** error probability; the latter one minimizes **sequence** error probability.[1]

[1]The terms *a priori* and *a posteriori* refer to how or on what basis a proposition might be known [11]. A proposition is knowable *a priori* if it is knowable independently of experience. On the other hand, if it is knowable on the basis of experience the proposition is known *a posteriori*.

The maximum *a posteriori* sequence decoder outputs a code sequence $\hat{\boldsymbol{v}}$ that maximizes the *a posteriori* probability $\Pr(\boldsymbol{v}|\boldsymbol{r})$, that is,

$$\hat{\boldsymbol{v}} = \arg\max_{\boldsymbol{v}\in\mathcal{C}}\{\Pr(\boldsymbol{v}|\boldsymbol{r})\}.$$

Using Bayes' rule and assuming that all codewords are equally probable it follows that the maximum *a posteriori* rule is the same as the *maximum-likelihood* (ML) decoding rule

$$\hat{\boldsymbol{v}} = \arg\max_{\boldsymbol{v}\in\mathcal{C}}\{\Pr(\boldsymbol{r}|\boldsymbol{v})\}.$$

An algorithm that realizes ML decoding is the Viterbi algorithm. It fully exploits the error-correcting capability of a convolutional code $\mathcal{C}$.

The maximum *a posteriori* (MAP) symbol decoding algorithm computes the *a posteriori* probabilities

$$\Pr(u_t^{(i)} = u|\boldsymbol{r}), \quad i = 1, 2, \ldots, k, \; u \in \{0, 1\}$$

where $u_t^{(i)}$ is the ith information symbol at time instant t, and outputs the decision $\hat{u}$ that maximizes this probability. A MAP symbol decoder can be realized using the BCJR algorithm, described in the next section.

2.6 The BCJR algorithm

The Bahl-Cocke-Jelinek-Raviv (BCJR) [12] algorithm introduced in 1974 is a maximum *a posteriori* (MAP) symbol decoding method operating on a code trellis [6]. A MAP symbol decoder maximizes the *a posteriori* symbol probabilities, or, equivalently, it minimizes the probability of bit error, that is, the decoder output can be expressed as

$$\hat{u} = \arg\min_{x}\{\Pr(\hat{u} \neq u|\boldsymbol{r})\}$$

where u denotes an information bit. The BCJR algorithm computes the so-called *a posteriori* probability (APP) L-values [6]

$$L(u_l) = \ln\frac{\Pr(u_l = 0|\boldsymbol{r})}{\Pr(u_l = 1|\boldsymbol{r})} \tag{2.1}$$

and the hard-decision from the decoder is given by

$$\hat{u}_l = \begin{cases} 0, & L(u_l) > 0 \\ 1, & L(u_l) < 0. \end{cases}$$

The magnitude of an L-value $L(u_l)$ is the reliability of the hard decision $\hat{u}_l$.

To obtain the APP L-values of the information bits, the BCJR algorithm computes the following three metrics in the trellis

$$\begin{aligned} &\alpha_l(s'), && \text{forward metric} \\ &\gamma_l(s', s), && \text{branch metric} \\ &\beta_{l+1}(s), && \text{backward metric} \end{aligned}$$

where s' denotes a state at time l and s a state at time $l+1$. The forward metric $\alpha_l(s')$ is the probability of being in state s' at time l, given the received sequence up to time l, going forward in the trellis, see Figure 2.6a. Likewise, $\beta_l(s')$ corresponds to the probability of ending up in state s' at time l, going backwards in the trellis, see Figure 2.6b. The $\gamma_l(s', s)$ metric is the channel transition probability from state s' to the state s, for a given received tuple at time l.

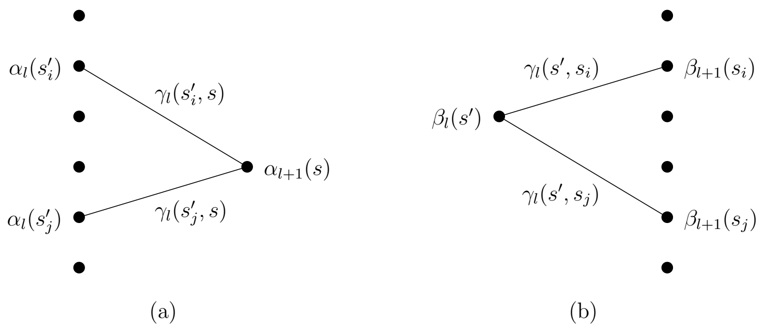


Figure 2.6: The BCJR metrics in the trellis.

When all metrics have been calculated in the trellis, the APP L-value for each bit is calculated as

$$\begin{aligned} L(u_l) \;=\; & \max^*_{(s',s)\in\mathcal{S}_l^0} \left\{\alpha_l^*(s') + \gamma_l^*(s', s) + \beta_{l+1}^*(s)\right\} \\ & - \max^*_{(s',s)\in\mathcal{S}_l^1} \left\{\alpha_l^*(s') + \gamma_l^*(s', s) + \beta_{l+1}^*(s)\right\} \end{aligned}$$

where $\mathcal{S}_l^0$ and $\mathcal{S}_l^1$ denote the sets of all state pairs (s', s) that correspond to the input bit $u_l = 0$ and $u_l = 1$, respectively.

The metrics with a star $*$ are logarithmic, for instance $\alpha_l^*(s') = \ln(\alpha_l(s'))$. The $\max^*$ operation is defined as the logarithm of the sum of exponentials, that is,

$$\max_{1\leq i\leq n}{}^*\{a_i\} \overset{\text{def}}{=} \ln\left(\sum_{i=1}^{n} e^{a_i}\right)$$

and can be recursively computed as

$$\begin{aligned}\max{}^*(a_1, a_2) &= \max(a_1, a_2) + \ln(1 + e^{-|a_1-a_2|})\\ \max{}^*(a_1, a_2, a_3) &= \max{}^*(\max{}^*(a_1, a_2), a_3).\end{aligned}$$

Since the BCJR decoder produces soft symbol reliabilities, it is often used as a component decoder in concatenated coding schemes, where the decoders of the constituent codes exchange soft information in an iterative manner.

Chapter 3

The M*-BCJR Algorithm

The BCJR algorithm has a complexity which grows exponentially with the memory of the code, or, in equalization, with the length of the channel impulse response. To overcome this drawback, the reduced complexity M*-BCJR algorithm was proposed by Marcin Sikora and Daniel J. Costello Jr [1]. It follows the structure of the BCJR algorithm, but, similarly to the M-BCJR algorithm [13], it dynamically constructs a simplified trellis during the forward recursion. Only the M states with the largest forward metric are preserved at each depth. Unlike in the M-BCJR, however, the remaining states are not deleted, but merged with the M best states, and the branches arriving to these states are simply redirected to the M surviving states [1]. Thus, the complexity of the M*-BCJR algorithm is specified by the number M and it does not depend on the memory of the underlying code.

3.1 M*-BCJR Algorithm

When computing the forward recursion, according to the same principle as in the BCJR algorithm, the M*-BCJR algorithm retains a maximum number of M states s_i^j at any depth index i. The branches in the trellis are not deleted, but merely redirected into more likely states.

When explaining the algorithm, $\mathcal{S}_i$ denotes the set of states at time instant i, j denotes the index of a state in the current set $\mathcal{S}_i$, L denotes the information block length in k-tuples, $\boldsymbol{u}_i$ denotes the input k-tuple at time instant i and the rest follows the notation of the BCJR algorithm. In Figure 3.1 a flowchart of the algorithm for the forward recursion is described in more detail. The M*-BCJR algorithm operates in the probability domain as follows

1. Initialize $i := 0$ and set $\alpha(s_0^1) := 1$. Initialize the set of surviving states with $\mathcal{S}_0 = \{s_0^1\}$.

2. Initialize the set of surviving states at depth $i+1$ to the empty set, $\mathcal{S}_{i+1} = \emptyset$.
3. For every state s_i^j in $\mathcal{S}_i$ and every branch leaving this state compute the metric $\gamma(s_i, s_{i+1})$ and add its children nodes s_{i+1} to $\mathcal{S}_{i+1}$.
4. Compute the forward metric of each state s_{i+1}^j from the $\mathcal{S}_{i+1}$ as the sum of $\alpha(s_i)\gamma(s_i^j, s_{i+1}^{j'})$ over all branches visited in step (3) that end in s_{i+1}^j.
5. If $|\mathcal{S}_{i+1}| > M$, proceed to (6) else goto (9).
6. Sort the states in $\mathcal{S}_{i+1}$ according to their metric.
7. Determine the M states with the highest metric and remove the remaining ones into a temporary set $\mathcal{S}'_{i+1}$.
8. For each state in $\mathcal{S}'_{i+1}$ perform the following tasks:
   - Find a state s_{i+1} in $\mathcal{S}_{i+1}$ that differs from s'_{i+1} by the least number of final k-tuples $\boldsymbol{u}_j$.
   - Redirect all branches ending in s'_{i+1} to s_{i+1}.
   - Add $\alpha(s'_{i+1})$ to the metric of $\alpha(s_{i+1})$.
   - Delete s'_{i+1} from $\mathcal{S}'{}_{i+1}$.
9. Increment $i = i+1$.
10. If $i < L + m + 1$ goto (2) else stop.

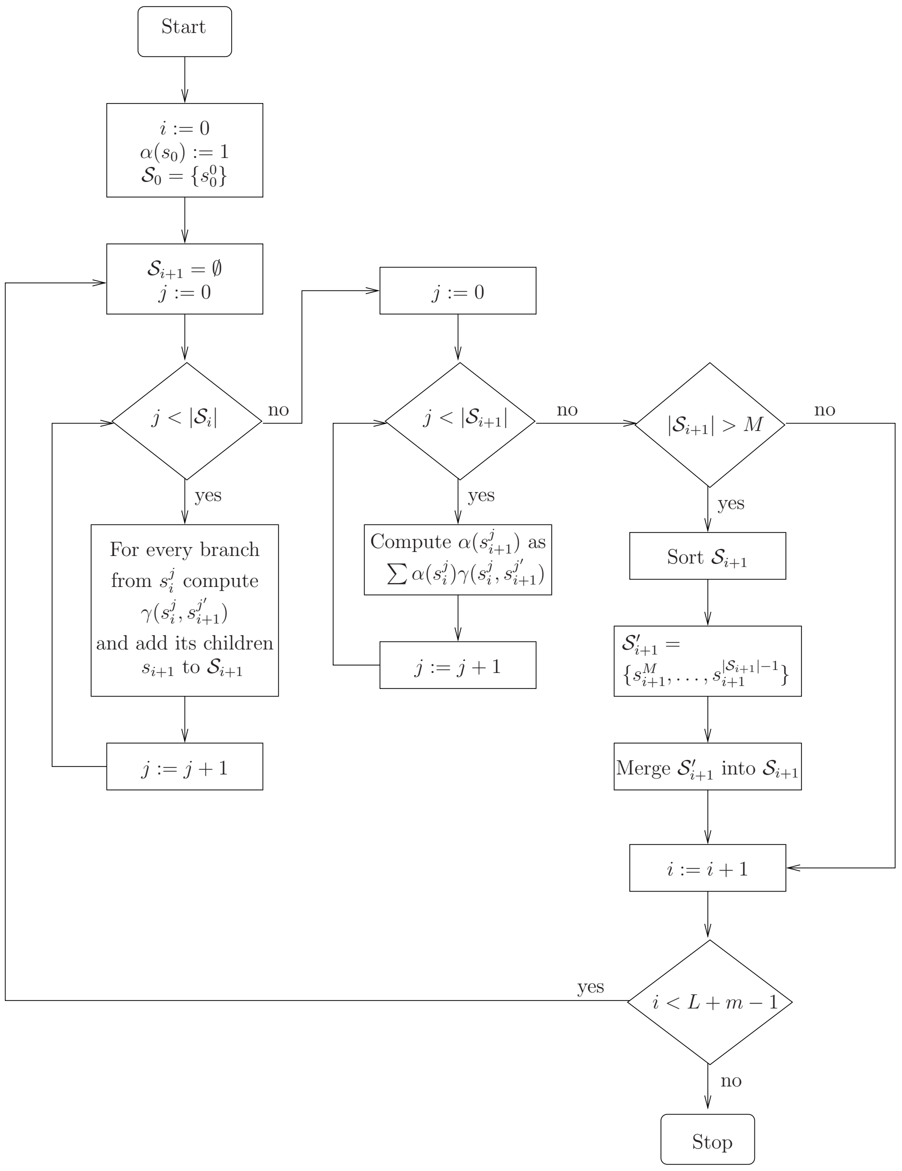

Start
$i := 0$
$\alpha(s_0) := 1$
$\mathcal{S}_0 = \{s_0^0\}$
$\mathcal{S}_{i+1} = \emptyset$
$j := 0$
$j < |\mathcal{S}_i|$
no
yes
For every branch from s_i^j compute $\gamma(s_i^j, s_{i+1}^{j'})$ and add its children s_{i+1} to $\mathcal{S}_{i+1}$
$j := j+1$
$j := 0$
$j < |\mathcal{S}_{i+1}|$
no
yes
Compute $\alpha(s_{i+1}^j)$ as $\sum \alpha(s_i^j)\gamma(s_i^j, s_{i+1}^{j'})$
$j := j+1$
$|\mathcal{S}_{i+1}| > M$
no
yes
Sort $\mathcal{S}_{i+1}$
$\mathcal{S}'_{i+1} = \{s_{i+1}^M, \dots, s_{i+1}^{|\mathcal{S}_{i+1}|-1}\}$
Merge $\mathcal{S}'_{i+1}$ into $\mathcal{S}_{i+1}$
$i := i+1$
$i < L+m-1$
yes
no
Stop

Figure 3.1: Forward recursion of the M*-BCJR algorithm.

The merging of state s'_i into s_i in step (8) is illustrated in Figure 3.2. In the merging process, besides adding the forward metric, the corresponding branches are redirected to s_i and preserved.

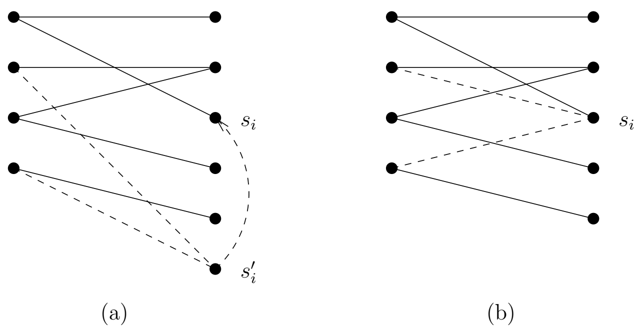


Figure 3.2: Example of trellis section before merging (a) and afterwards (b).

When the forward recursion is finished, a simplified trellis with M states (instead of 2^m) at each depth is obtained. This simplified trellis is subsequently used for the backward recursion and completion phase, when L-values are calculated.

3.2 Simulation results

In the simulations of the M*-BCJR algorithm, an information block length of $K = 100$ bits and the memory $m = 4$ (4 66) systematic encoder was used. The number of surviving states M at each depth was chosen to be 4 or 8, compared to $2^4 = 16$ in the full trellis. In Figure 3.3 the bit error rate (BER) performance is illustrated. As expected, a reduction of M implies a degradation of the BER performance.

The output L-values from the BCJR algorithm follow the Gaussian distribution, since the noise is AWGN [6]. The outputs of the M*-BCJR algorithm exhibit Gaussian-like behavior, as shown in Figures 3.4 and 3.5.

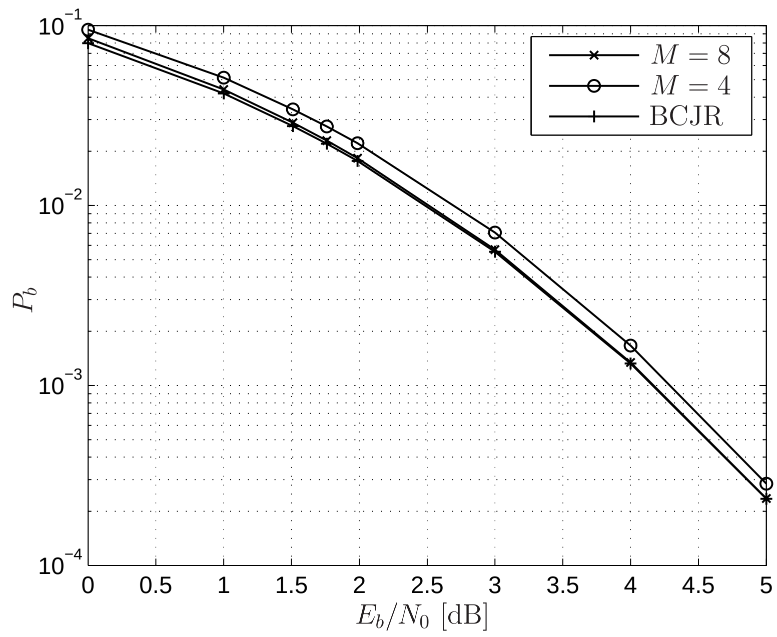


Figure 3.3: Bit error rates of M*-BCJR with the (4 66) encoder.

Furthermore, we compared the L-values of the M*-BCJR algorithm with the corresponding L-values of the BCJR algorithm, which is used as a benchmark. Figure 3.6 shows a snapshot of 30 L-values in one frame. The correspondence between the exact L-values obtained by BCJR and the approximate M*-BCJR values is rather good.

In order to asses the quality of the soft outputs of the M*-BCJR algorithm, as a quality measure we use the average absolute relative discrepancy d with respect to the BCJR output, that is,

$$d = \left| \frac{\Lambda_{\text{BCJR}} - \Lambda_e}{\Lambda_{\text{BCJR}}} \right| \tag{3.1}$$

where Λ_e is the soft output from the evaluated algorithm, in this case the M*-BCJR algorithm. In Figure 3.7, the mean value of the discrepancy $\overline{d}$ is shown for different E_b/N_0. For $M = 8$ (at half the complexity of the BCJR algorithm) the average discrepancy between the M*-BCJR and BCJR outputs is below 10% for E_b/N_0 levels ≥ 2 dB.

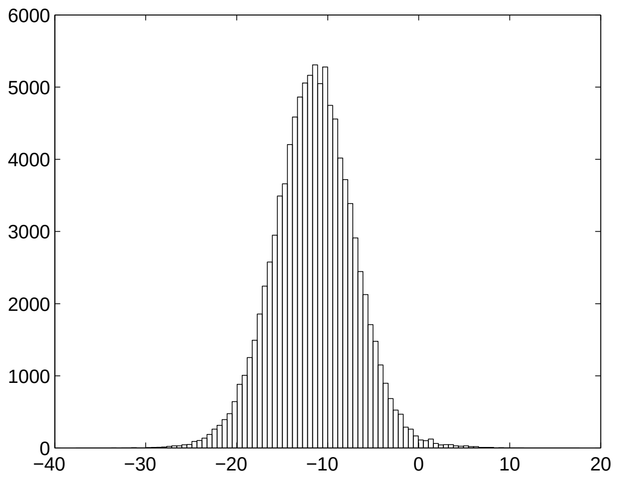


Figure 3.4: Histogram of the L-values for the M*-BCJR algorithm for the memory $m = 4$ $(4\ 66)$ encoder, $E_b/N_0 = 3$ dB, and M=4.

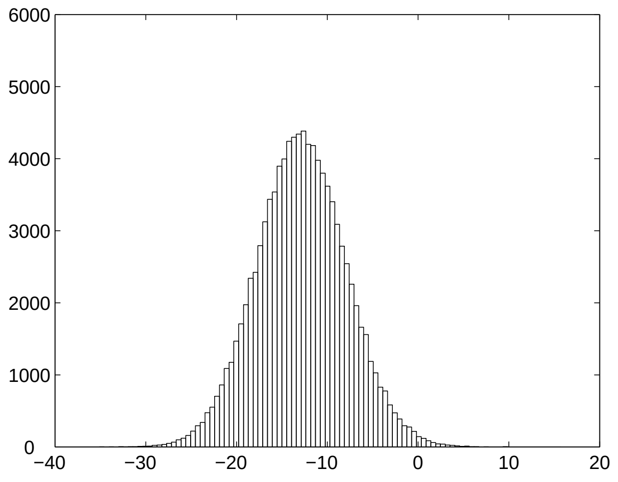


Figure 3.5: Histogram of the L-values of the M*-BCJR algorithm for the memory $m = 4$ $(4\ 66)$ encoder, $E_b/N_0 = 3$ dB, and M=8.

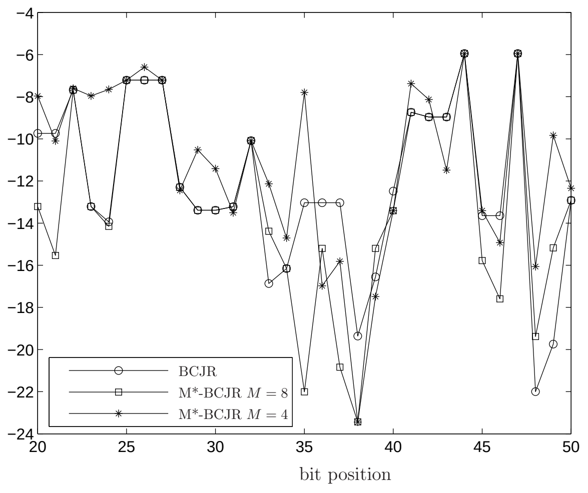


Figure 3.6: L-values of the M*-BCJR algorithm ($M = 4$ and 8) and the BCJR algorithm for the memory $m = 4$ $(4\ 66)$ encoder, and $E_b/N_0 = 3$ dB.

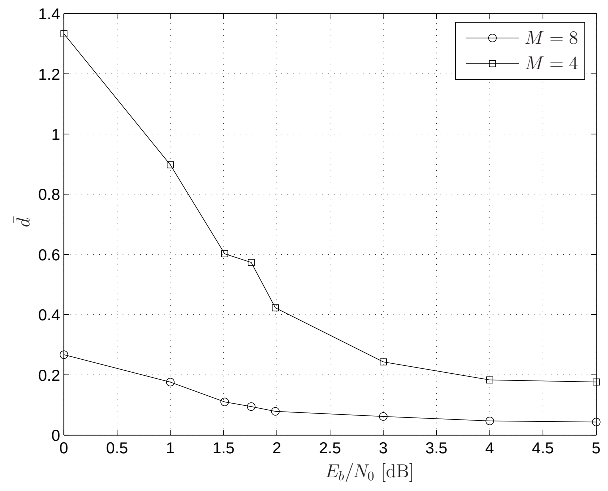


Figure 3.7: The average relative discrepancy between the L-values of the M*-BCJR algorithm ($M = 4$ and 8) and the BCJR algorithm for the memory $m = 4$ $(4\ 66)$ encoder.

Chapter 4

The LISS Algorithm

The List-Sequential (LISS) Algorithm was proposed by Christian Kuhn and Joachim Hagenauer [2]. It has its origin in the Zigangirov-Jelinek (ZJ) stack sequential decoding algorithm. The complexity of sequential decoding is dependent of the noise and the algorithm is sequential in the sense that only one state is evaluated at each depth in the tree [14][6]. Because the number of computations per decoded information block is a random variable, the use of a buffer to store incoming data is needed. Most encoded sequences are decoded very quickly but some may undergo long searches, which can lead to buffer overflow and information loss or can cause erasures. This limits the application of sequential decoding. Traditional sequential decoding provides only a hard decisions of the information bits. The LISS algorithm, however, is a soft-in soft-out sequential decoding algorithm that provides soft information which is required in iterative decoding schemes [2].

4.1 Fano and Fano-like metrics

The Fano metric is derivied for so-called random code trees [15] [14]. The Fano metric for a rate $R = k/n$ random code can be expressed as

$$\mu_F = \sum_{i=0}^{L-1} \sum_{j=1}^{n} \left\{ \log \left(\frac{\Pr(r_i^{(j)} | x_i^{(j)})}{p(r_i^{(j)})} \right) - R \right\} \tag{4.1}$$

where according to the random-tree assumption, all code symbols have probability 1/2. Thus,

$$p(r_i^{(j)}) = p(r_i^{(j)} | v_i^{(j)} = 0)\frac{1}{2} + p(r_i^{(j)} | v_i^{(j)} = 1)\frac{1}{2}. \tag{4.2}$$

Fano metric (4.1) is derived assuming equally probable information bits $u_i^{(j)}$. If non-uniform *a priori* (AP) information about the informations bits is available, it can be incorporated into the sequential decoding process by using the modified Fano metric

$$\mu_{FAP} = \sum_{i=0}^{L-1} \left\{ \sum_{j=1}^{n} \left(\log \frac{p(r_i^{(j)}|x_i^{(j)})}{p(r_i^{(j)})} \right) + \sum_{j=1}^{k} \log \Pr(u_i^{(j)}) \right\}. \tag{4.3}$$

for a random code. Note that (4.3) reduces to (4.1) for $\Pr(u_i^{(j)}) = 1/2$. Let

$$L(x) = \ln \frac{\Pr(x=+1)}{\Pr(x=-1)}$$

denote the *a priori* L-value of the bit $u_i^{(j)}$, and let

$$L_c r_i = \ln \frac{\Pr(r_i|x=+1)}{\Pr(r_i|x=-1)}$$

be the channel log likelihood ratio for the symbol v_i. Then the metric (4.3) can be rewritten in terms of these L-values using the relations

$$\Pr(u_i^{(j)} = u) = \frac{e^{uL_a(u_i^{(j)})/2}}{e^{L_a(u_i^{(j)})/2} + e^{-L_a(u_i^{(j)})}}, \quad u \in \{\pm 1\} \tag{4.4}$$

and

$$\frac{\Pr(r_i^{(j)}|v_i^{(j)})}{p(r_i^{(j)})} = 2\frac{e^{xL_c r_i^{(j)}/2}}{e^{L_c r_i^{(j)}/2} + e^{-L_c r_i^{(j)}/2}}, \quad x \in \{\pm 1\}$$

which follows from (4.2). By substituting (4.4) and (4.1) into the modified Fano metric (4.3) we get

$$\begin{aligned} \mu_{FAP} \quad = \quad & \sum_{i=0}^{L-1} \Bigg\{ n \ln 2 + \sum_{j=1}^{n} \frac{x_i^{(j)} L_c r_i^{(j)}}{2} + \sum_{j=1}^{k} \frac{u_i^{(j)} L_a}{2} \\ - \quad & \sum_{j-1}^{n} \ln \left(e^{L_c r_i^{(j)}/2} + e^{-L_c r_i^{(j)}/2} \right) \\ - \quad & \sum_{j-1}^{k} \ln \left(e^{L_a(u_i^{(j)})/2} + e^{-L_a(u_i^{(j)})/2} \right) \Bigg\}. \end{aligned} \tag{4.5}$$

During the simulations systematic encoders will be used. These encoders lack the randomness properties necessary for a derivation of Fano-like metric. However, a modification of the Fano metric μ_{FAP} (4.5) for systematic encoders was proposed in [16] and also used in [2] for LISS decoding. The modification is obtained by assuming (which is erroneous if we use the required random-tree assumption) that the k systematic bits in each n-tuple have probabilities given by the *a priori* information, while only the $n-k$ parity bits on each branch of the code tree are considered random. This yields the Fano-like metric

$$\begin{aligned}\mu_{APSYS} &= \sum_{i=0}^{L-1}\Bigg\{(n-k)\ln 2 + \sum_{j=1}^{k}\frac{x_i^{(j)}L_a(x_i^{(j)})}{2} + \sum_{j=1}^{n}\frac{x_i^{(j)}L_c r_i^{(j)}}{2}\\ &\quad -\sum_{j=1}^{k}\ln\left(e^{(L_a(x_i^{(j)})+L_c r_i^{(j)})/2} + e^{-(L_a(x_i^{(j)})+L_c r_i^{(j)})/2}\right)\\ &\quad -\sum_{j=k+1}^{n}\ln\left(e^{(L_c r_i^{(j)})/2} + e^{-(L_c r_i^{(j)})/2}\right)\Bigg\}.\end{aligned}$$

Using the approximation

$$\ln(e^x + e^{-x}) \approx |x|$$

the above metric μ_{APSYS} can be approximated (simplified) as

$$\begin{aligned}\mu_{APSYS} &\approx \sum_{i=0}^{L-1}\Bigg\{(n-k)\ln 2 + \sum_{j=1}^{k}\frac{x_i^{(j)}L_a(x_i^{(j)})}{2} + \sum_{j=1}^{n}\frac{x_i^{(j)}L_c r_i^{(j)}}{2}\\ &\quad -\sum_{j=1}^{k}\frac{|L_a(x_i^{(j)}) + L_c r_i^{(j)}|}{2} - \sum_{j=k+1}^{n}\frac{|L_c(r_i^{(j)})|}{2}\Bigg\} =\\ &= \sum_{i=0}^{N-1}\Bigg\{(n-k)\ln 2 + \sum_{j=1}^{k}\frac{x_i^{(j)}\cdot\hat{x}_{i,\mathrm{H}}^{(j)} - 1}{2}|L_a(x_i^{(j)}) + L_c r_i^{(j)}|\\ &\quad + \sum_{j=k+1}^{n}\frac{x_i^{(j)}\cdot\hat{x}_{i,\mathrm{H}}^{(j)} - 1}{2}|L_c r_i^{(j)}|\Bigg\} \qquad (4.6)\end{aligned}$$

where $\hat{x}_\mathrm{H}$ is a hard decision equal to

$$\hat{x}_\mathrm{H} = \mathrm{sign}(L_a(x) + L_c r)$$

for the systematic bits, or

$$\hat{x}_\mathrm{H} = \mathrm{sign}(L_c r)$$

for the parity bits.

4.2 The stack algorithm

The stack algorithm operates on a code tree, where it searches for the path with the largest Fano metric [2]. A path corresponds to an estimated input sequence of bits. The stack algorithm used for LISS decoding uses the Fano-like metric μ_{APSYS} (4.6) as path metric.

The stack algorithm begins with extending the root of the tree. At each depth in the tree only the top path, denoted T, is extended. This path is removed from the stack. By this extension 2^k new paths are created and they are placed in the stack $\mathcal{S}$, according to their calculated metric. The stack is in fact a list containing the leaves of the partially explored tree. Each leaf represents a path. In general, paths in the list have different lengths, and are sorted in decreasing order, according to their metrics. The maximum stack size is defined in advance and denoted L_S. Hereinafter, for simplicity, the Fano-like metric (4.6) of the lth path in the stack is denoted by $\mu(l)$, $l = 1, \ldots, |\mathcal{S}|$. In Figure 4.1 the flowchart of the stack algorithm is given.

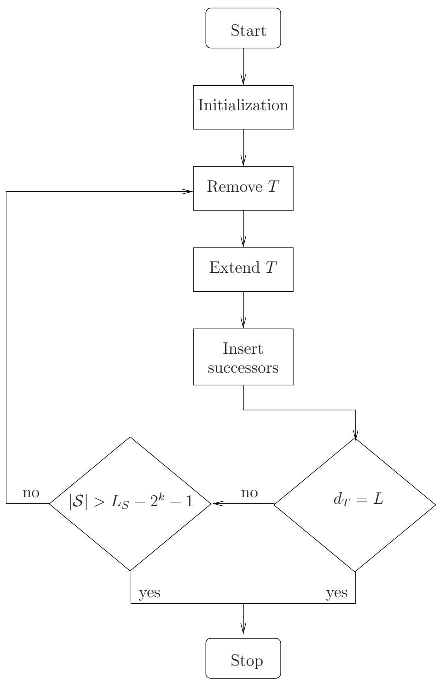


Figure 4.1: Flowchart of the stack algorithm.

The stack algorithm consists of the following steps

1. Initialization: Insert the root node on top of stack, and set its metric to zero and length to zero.
2. Remove T: Remove the path T with the highest metric.
3. Extend T: Determine the 2^k children nodes of T, and for each of them compute the metric by adding the respective branch metric to the parent path metric.
4. Stop criterion: If the number of stack entries is $|\mathcal{S}| > L_S - 2^k - 1$ or if the depth of T has full length L, then stop; else goto (2).

After the completion of the stack algorithm the top path T is of full length, or a buffer overflow has occurred, that is, the stack is filled to maximum size before reaching the full length L. The paths in the stack are further processed by the LISS decoder, as described in the next section.

4.3 LISS decoding

The first part of the LISS algorithm is the Stack algorithm, described in the previous section. The next step is the augmentation step, where all paths in the stack are extended (augmented) to the full length L [2].

In the simulations we have used only the so-called soft augmentation, where the unknown missing systematic bits in the codewords $\boldsymbol{x}$ on each path are replaced by their soft estimates $\bar{x}_i^{(j)}$ obtained from the *a priori* L-values $L_a(x_i^{(j)})$ as

$$\begin{aligned}\bar{x}_i^{(j)} = \mathrm{E}[x_i^{(j)}] &= (+1)\Pr(x_i^{(j)} = +1) + (-1)\Pr(x_i^{(j)} = -1) = \\ &= 1 - 2\Pr(x_i^{(j)} = -1) = \tanh\left(L_a(x_i^{(j)})/2\right). \end{aligned} \tag{4.7}$$

The missing parity bits in the codeword $\boldsymbol{x}$ are calculated using the estimated systematic bits as input to the encoder, where every modulo two adder is replaced with a real number multiplier. The flowchart of the augmentation step is shown in Figure 4.2. After the augmentation, all paths in the stack have full length L. In the part of the tree explored by the stack algorithm the paths have binary code symbols on each branches; in the augmented part of the tree, paths carry soft symbol estimates given by (4.7) based on *a priori* information only.

If the stack algorithm finished successfully (without buffer overflow), the top path has full length and it is the only path that needs no augmentation. In the case of buffer overflow, the top path also needs to be augmented, just as all other stack entries.

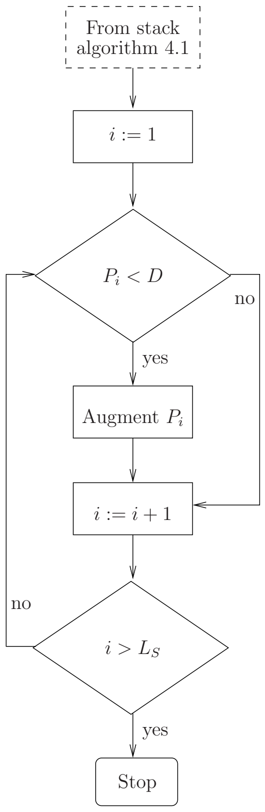


Figure 4.2: Flowchart of the augmentation step in the LISS algorithm.

The third step in LISS decoding is producing soft outputs for each individual bit $x_i^{(j)}$. The *a posteriori* L-value is given by

$$\begin{aligned} L(x_i^{(j)}) &= \ln \frac{\Pr(x_i^{(j)} = +1|\boldsymbol{r})}{\Pr(x_i^{(j)} = -1|\boldsymbol{r})} \\ &= \ln \frac{\sum_{\forall \boldsymbol{x}: x_i^{(j)}=+1} \Pr(\boldsymbol{x}|\boldsymbol{r})}{\sum_{\forall \boldsymbol{x}: x_i^{(j)}=-1} \Pr(\boldsymbol{x}|\boldsymbol{r})} = \ln \frac{\sum_{\forall \boldsymbol{x}: x_i^{(j)}=+1} e^{\ln \Pr(\boldsymbol{x}|\boldsymbol{r})}}{\sum_{\forall \boldsymbol{x}: x_i^{(j)}=-1} e^{\ln \Pr(\boldsymbol{x}|\boldsymbol{r})}} \end{aligned} \quad (4.8)$$

where the summations are performed over all codewords (paths) that have a 0 (+1) and a 1 (−1) on the ith position, respectively.

In the LISS algorithm an approximation of the L-value (4.8) is obtained by only considering the paths explored by the stack algorithm and by simply replacing (this step is not justified!) the $\ln \Pr(\boldsymbol{x}|\boldsymbol{r})$ with the path metric $\mu(l)$ corresponding to the codeword $\boldsymbol{x}$. Additionally, since on an arbitrary position i each path can have either hard bit $x_i^{(j)}$ or soft estimate $\bar{x}_i^{(j)}$, both summations in (4.8) are then performed over the whole stack, and each term is weighted according to the hard bit or soft estimate. Thus, we obtain

$$\begin{aligned} L(x_i^{(j)}) &\approx \ln \frac{\sum\limits_{1\le l\le|\mathcal{S}|} e^{\mu(l)} \Pr(x_i^{(j)}(l)=+1)}{\sum\limits_{1\le l\le|\mathcal{S}|} e^{\mu(l)} \Pr(x_i^{(j)}(l)=-1)} \\ &= \ln \frac{\sum\limits_{1\le l\le|\mathcal{S}|} e^{\mu(l)} (\frac{1+x_i^{(j)}(l)}{2})}{\sum\limits_{1\le l\le|\mathcal{S}|} e^{\mu(l)} (\frac{1-x_i^{(j)}(l)}{2})}. \qquad (4.9) \end{aligned}$$

where $x_i^{(j)}(l)$ is replaced by the soft estimate $\bar{x}_i^{(j)}(l)$ in the augmented parts in the path l. By using the max-log approximation, expression (4.9) further simplifies to

$$\begin{aligned} L(x_i^{(j)}) &\approx \max_{1\le l\le|\mathcal{S}|} \left\{ \mu(l) + \ln \frac{1+x_i^{(j)}(l)}{2} \right\} \\ &- \max_{1\le l\le|\mathcal{S}|} \left\{ \mu(l) + \ln \frac{1-x_i^{(j)}(l)}{2} \right\} \qquad (4.10) \end{aligned}$$

If the ith bit of the lth path is hard, that is, $x_i^{(j)}(l) = \pm 1$, the metric $\mu(l)$ contributes to only one of the two max terms in (4.10). If $x_i^{(j)}(l)$ is a soft value, however, it will contribute to both terms. If the *a priori* information is $L_a(x_i^{(j)}) = 0$, then $x_i^{(j)}(l) = 0$. If $x_i^{(j)}(l) = 0$ for all paths in the stack ($l = 2, \ldots, |\mathcal{S}|$), then the L-value (4.10) equals 0 and it is treated as an erasure.

4.4 Simulation results

In this section simulation results of the LISS algorithm are presented and discussed. Results include the bit error rate, the output L-values, and their

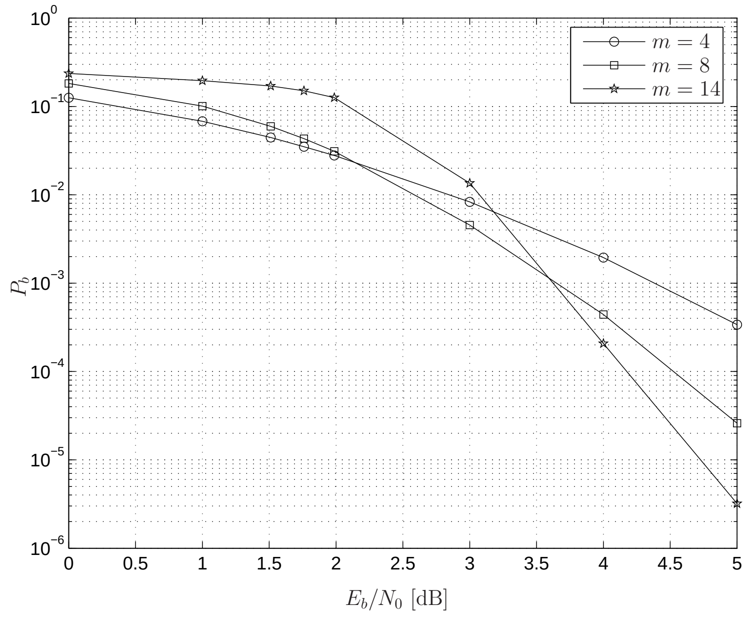


Figure 4.3: Bit error rate of the LISS algorithm for different encoders.

distribution. In Figure 4.3 bit error rates for three different encoders are shown. The encoders are all systematic, have memory $m = 4$, 8 and 14 and their generator matrices are (4 66), (4 671) and (4 71447), respectively. The block length is $K = 500$ and the stack size is limited to $L_S = 2^{10}$. At low E_b/N_0, the high memory encoders have worse performance since if a wrong decision is made it takes more steps in the tree until the decoder is back on the correct path again. Moreover if stack overflow occurs, then the top path T will not have full length and augmentation of the top path is occurs, which also determinates the BER. For high E_b/N_0, large memory encoders rarely make any errors.

In Figure 4.4, we use the (4 66) encoder with different block lengths, $K \in \{100,\ 500,\ 1000\}$. The corresponding maximum stack sizes are $L_S \in \{2^8,\ 2^{10},\ 2^{11}\}$. The BER performance is virtually the same for all three cases, since stack overflow rarely occurs.

In Figure 4.5 the influence of the stack size on the BER performance is illustrated for the (4 66) encoder. The block length $K = 500$ bits. In the case of stack size 2^9, stack overflow occurs often, and soft augmentation to the top

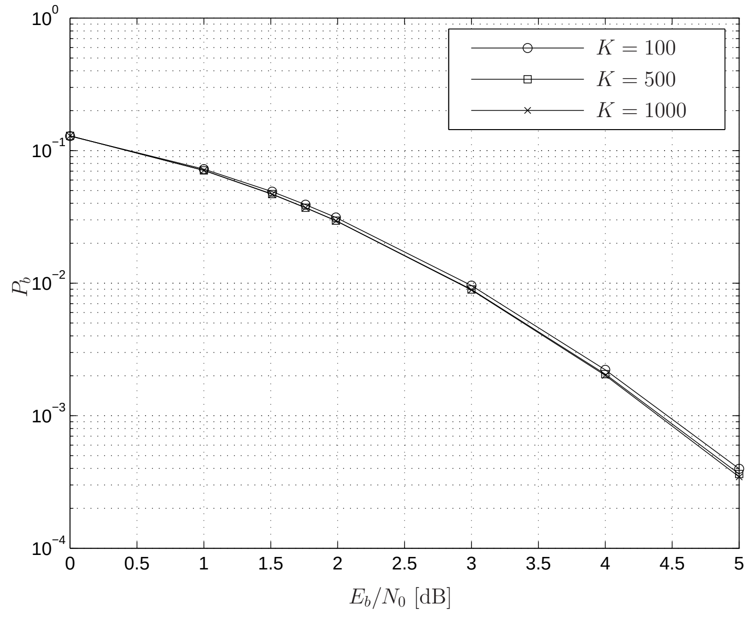


Figure 4.4: Bit error rate of the LISS algorithm for the memory $m = 4$ (4 66) systematic encoder with different block lengths.

path is necessary. Since the *a priori* information is zero for all information bits, the expected soft value of the augmented bits is $\tanh(0) = 0$ which implies coin-flipping for these bits.

The probability density function (PDF) of the true APP L-values (obtained by the BCJR algorithm) is Gaussian. Figure 4.6 shows the histogram of the outputs from the LISS algorithm. Clearly, they have a high non-Gaussian behavior. The reliabilities are highly overestimated by the LISS algorithm.

Average L-values produced by the LISS algorithm are plotted in Figure 4.7 for each position in a frame of $K = 500$ bits. Clearly the L-value have a strong linear dependency on the bit position. The first bits in a frame are very reliable, but as bit position in the frame increases their reliability decreases. This is due to the augmentation part of the algorithm, especially when no *a priori* information is available. Paths that are augmented in the beginning of the tree, will get higher L-values than paths that are augmented at the end. Hence, the soft outputs are more reliable in the beginning of a frame than at the end. Large numerical differences of the L-values will cause problem when

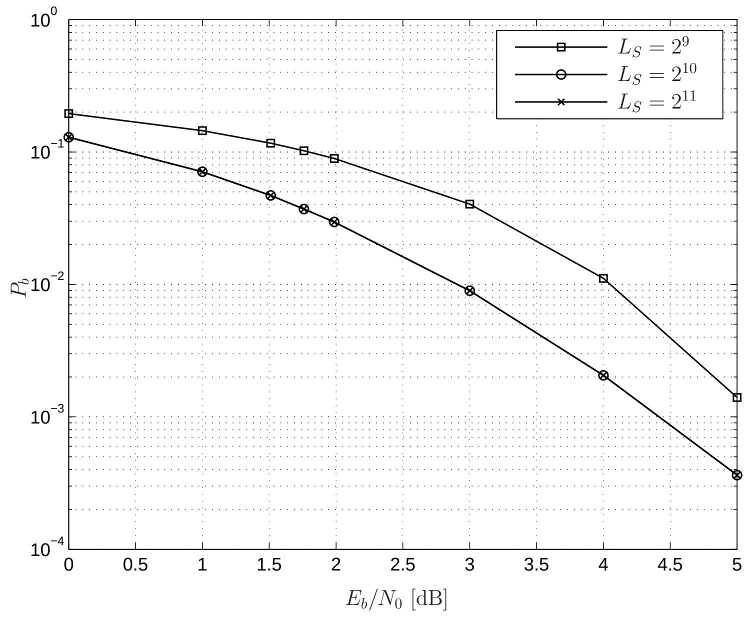


Figure 4.5: Bit error rate of the LISS algorithm with different stack sizes for the memory $m = 4$ (4 66) encoder.

they are used as input to another constituent decoder in an iterative scheme.

A snapshot of $K = 500$ L-values from the LISS algorithm is illustrated in Figure 4.8. The linear tendency is immediate. If the linear trend of the L-values is removed the variations along the removed, line are revealed. In Figure 4.9 the section of 30 error-free L-values marked in Figure 4.8, by a dashed rectangle is examined in more detail. The linear trend was removed by removing the best straight-line fit, which was done in MATLAB using the detrend command. As a reference, the corresponding L-values from the BCJR algorithm, also manipulated by detrend, are plotted in Figure 4.9. The L-values produced by the LISS algorithm are similar to the corresponding values from the BCJR algorithm. Removing the linear trend in this way increases the quality and usability of the soft output of LISS, but increases the algorithm complexity.

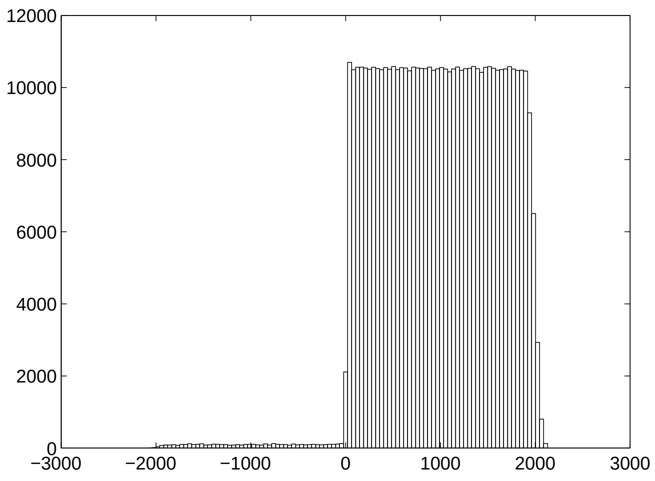


Figure 4.6: Histogram of the L-values of the LISS algorithm for the memory $m = 4$ $(4\ 66)$ encoder at $E_b/N_0 = 3$ dB.

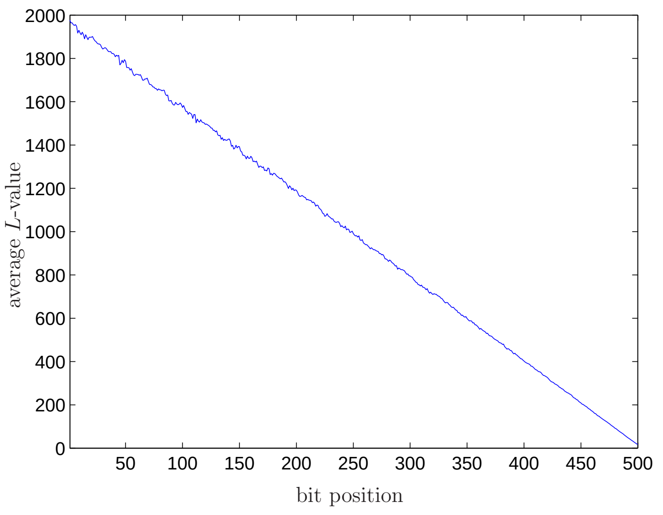


Figure 4.7: L-values versus bit position for the LISS algorithm, block length $K = 500$ bits with stack size $L_S = 2^{10}$, memory $m = 4$ $(4\ 66)$ encoder and $E_b/N_0 = 3$ dB.

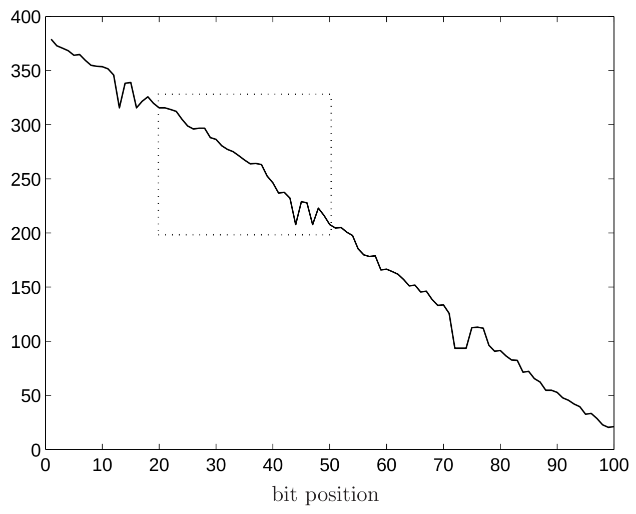


Figure 4.8: A snapshot of 100 L-values for the LISS algorithm, for the memory $m = 4$ (4 66) encoder and $E_b/N_0 = 3$ dB.

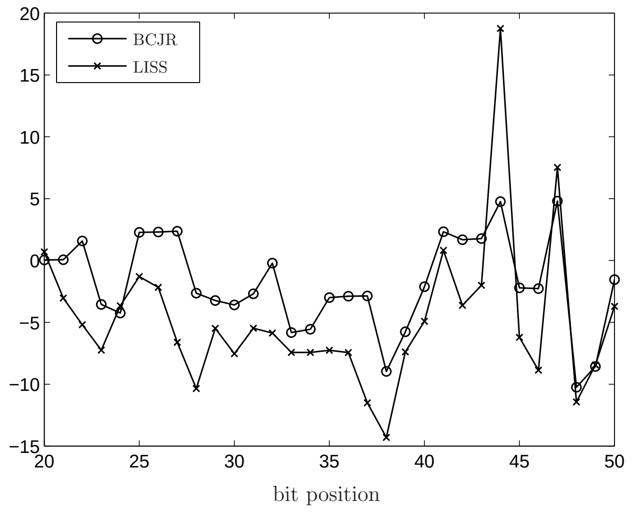


Figure 4.9: A snapshot of 30 L-values from the LISS algorithm compared with the corresponding BCJR values, after applying the detrend operation in MATLAB.

4.5 The LISSOM algorithm

As pointed out in the previous section, soft outputs from the LISS algorithm have a strong linear trend, that is, the magnitude of the L-value depends linearly on the depth in the code tree. Information bits in the beginning of the frame have stronger reliability, i.e., higher L-values, than bits in the end of the frame. This makes the soft output hard to use in an iterative scheme. One way to solve this problem is to remove the tendency numerically from the soft outputs. An example of this was shown in the previous section.

4.5.1 LISSOM—the principle

Instead of using a numerical tool to remove the linear tendency after producing the soft outputs, this can be accomplished by modifying the algorithm itself. To this end, we note that when a path P_l, $l = 2, 3, \ldots, L_s$, is augmented, it will with high probability reach a state which is the same as the state at the same depth on the best path. Thus, we can say that the augmented path *matches* with the best path $P_1 = T$ from the stack algorithm. We write match rather than merge since the two paths are not merged into one path; rather we only check for the possibility of merging (we check if the states are the same and at the same depth) and let the remaining part of the top path T yield the metrics for the path P_l. If the state $\boldsymbol{\sigma}_d^{P_l}$ is equal to the correct path state $\boldsymbol{\sigma}_d^{T}$ at the same depth d, that is, if $\boldsymbol{\sigma}_d^{P_l} = \boldsymbol{\sigma}_d^{T}$, the path P_l has matched with $P_1 = T$ and the remaining path metric can be inherited from T. We call this twist of the LISS algorithm LISSOM, List-Sequential Obtruncated Match.

In the LISSOM algorithm the same path metrics and the soft-output calculation are used as in the LISS algorithm. LISSOM uses hard decisions, described in [17], when estimating the information bits u_i in the augmentation process. In contrast to LISS however, LISSOM uses both *a priori* and channel information when calculating the expectation of u_i, that is,

$$\bar{u}_i = \tanh((L(u_i) + L_c r_i)/2), \qquad i = 0, 1, \ldots, K-1.$$

The hard decision on u_i is then obtained by flipping a coin whose probability of a zero is

$$\Pr(u_i = 0) = \frac{\bar{u} + 1}{2}.$$

Figure 4.10 illustrates the main principle of the LISSOM algorithm in a code tree where the top path has full length L and is terminated with m dummy zeros driving the encoder to the all zero state. The figure shows the two steps of the augmentation process. In the first part, or the matching part, stepwise augmentation is performed until the augmented path P_l matches with the top path T at depth 4. Second, the augmentation of P_l is completed by copying the metric increments for T at each depth. Notice that both paths are extended from depth 4 to depth $L + m$. A different scenario is

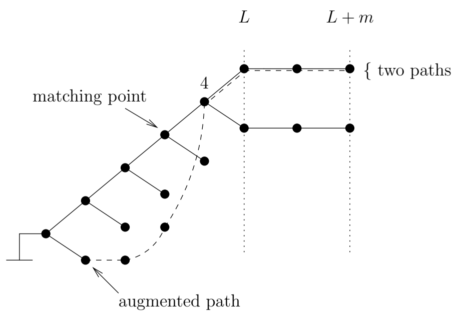


Figure 4.10: Example of obtruncated matching.

illustrated in Figure 4.11. In this case the path P_l does not match with T. The augmentation of P_l will then simply be performed up to the full length L, as in the LISS algorithm.

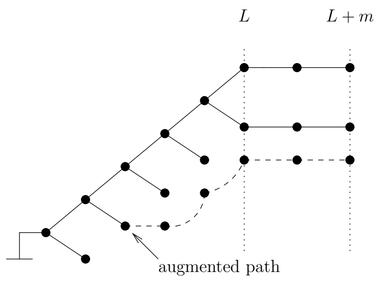


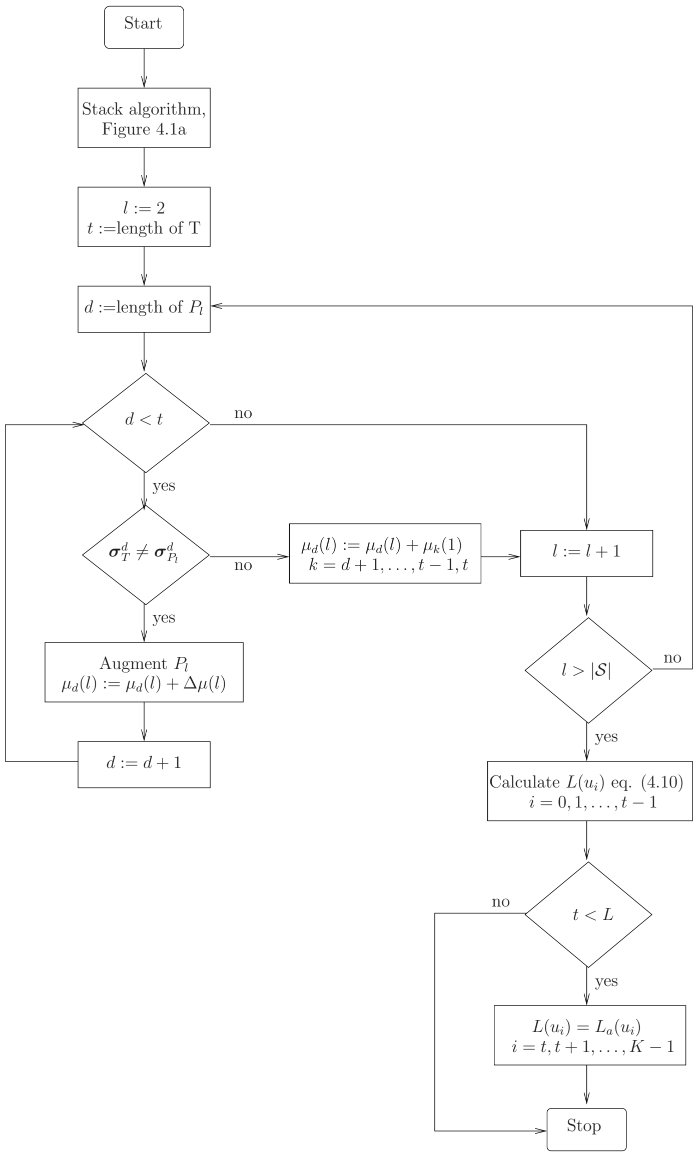
Figure 4.11: Example without matching.

If stack overflow occurs in the stack algorithm, the top path T will not have full length L. Unlike in the LISS algorithm, no augmentation to full length of the top path is carried out in LISSOM. Instead, the paths P_l are only augmented to depth t of the top path. Consequently, the L-values are only calculated for $i = 0, 1, \ldots, t-1$. For the remaining information bits u_i, $i = t, t+1, \ldots, K-1$, $L(u_i)$ are set to the corresponding *a priori* values $L_a(u_i)$.

Since the last part of the augmentation (after a successful merging) will increment the augmented path with the same metrics as the top path, the difference between the two winning candidates in the Max-Log approximation (4.10) of the soft output will be decreased. Additionally, the number of augmentation steps needed for the path P_l to match with T is independent of the depth in the code tree. This significantly reduces the linear tendency of the final L-values. Furthermore, since the calculations are performed only when the augmented path differs from the best path, the complexity is decreased.

The flowchart of the LISSOM algorithm is shown in Figure 4.12, where the input is the partially explored code tree from the stack algorithm just as in the LISS case. The algorithm consists of the following steps:

1. Set $l = 2$ and remove the top path $T = P_1$ from the stack. Let t be the length of T.
2. Remove the next path P_l from the stack and check if its length d is less than t. If yes, goto 3, else goto 6.
3. If the current state P_l differs from the corresponding state of T, goto 4 else goto 7.
4. Increment the path metric $\mu_d(l)$ of P_l at depth d with one augmentation step, i.e., with $\Delta\mu(l)$.
5. If P_l has matched with T goto 6, else goto 4.
6. Augment P_l with the corresponding metric values of T until $d = t$.
7. After the augmentation of P_l, l is incremented and the next path in the stack is augmented.
8. When all paths in the stack are augmented, the soft outputs $L(u_i)$ are calculated in the same manner as in the LISS algorithm if $t = L$. If stack overflow has occurred, then $t < L$ and $L(u_i)$ are set to the corresponding *a priori* information $L_a(u_i)$ for $i = t, t+1, \ldots, K-1$.



Start
Stack algorithm, Figure 4.1a
$l := 2$
$t :=$length of T
$d :=$length of P_l
$d < t$
no
yes
$\boldsymbol{\sigma}_T^d \neq \boldsymbol{\sigma}_{P_l}^d$
no
$\mu_d(l) := \mu_d(l) + \mu_k(1)$
$k = d+1, \ldots, t-1, t$
$l := l + 1$
yes
Augment P_l
$\mu_d(l) := \mu_d(l) + \Delta\mu(l)$
$d := d + 1$
$l > |\mathcal{S}|$
no
yes
Calculate $L(u_i)$ eq. (4.10)
$i = 0, 1, \ldots, t-1$
$t < L$
no
yes
$L(u_i) = L_a(u_i)$
$i = t, t+1, \ldots, K-1$
Stop

Figure 4.12: Flowchart of the LISSOM algorithm.

4.5.2 Simulation results

Simulations of the LISSOM algorithm for stand alone decoding are presented and compared to the LISS algorithm. Performance of LISSOM is evaluated for three systematic encoders with memory $m = 4$, 8, and 14. In all simulations the information block length was $K = 500$ bits and the stack size was $L_S = 2^{10}$. The bit error rate performance of LISSOM is only slightly better than that of the LISS algorithm. No essential difference between the bit error rates of the two algorithms was expected since both have the same code tree as input to the augmentation part and both use the same metric. In Figure 4.13 the bit error rates of the LISS and LISSOM algorithms for a memory 4 code are shown.

In Figure 4.14 the complexities of the two algorithms are compared. The complexity of LISSOM is about 25-32 times lower than that of LISS.

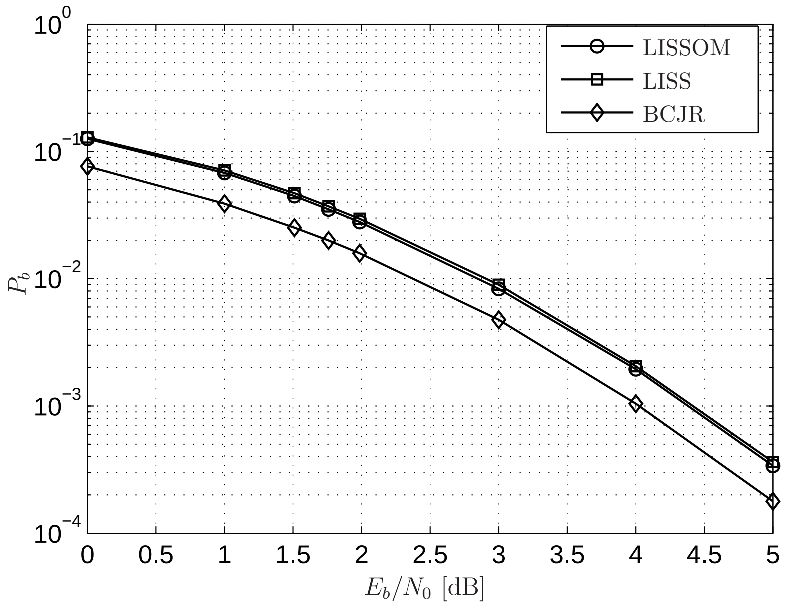


Figure 4.13: Bit error rate of the LISS, LISSOM, and BCJR algorithms for the memory $m = 4$ (4 66) systematic encoder.

The LISSOM algorithm was introduced with the aim to improve the quality of the output L-values by avoiding the linear trend of L-values that LISS suffers from. To investigate if LISSOM fulfills this task the mean value of the

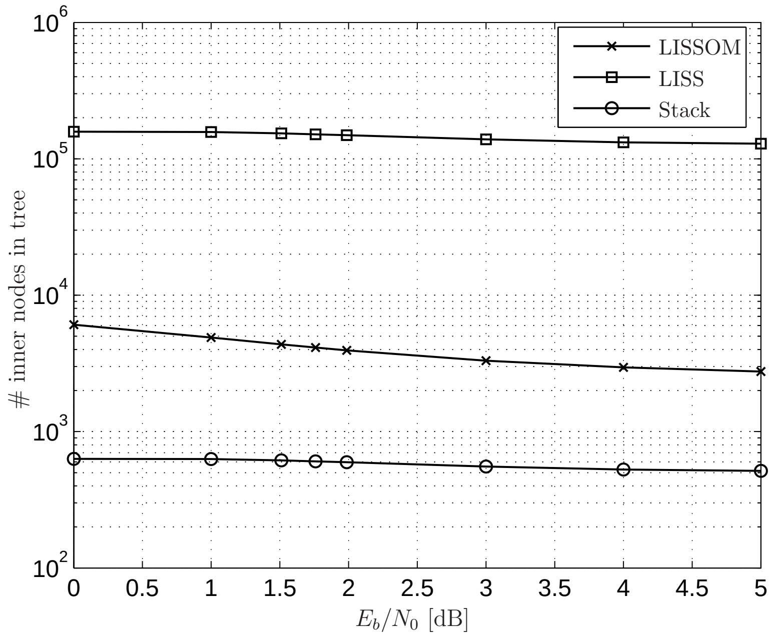


Figure 4.14: Complexity for stack, LISS, and LISSOM algorithms for the memory $m = 4$ (4 66) systematic encoder with stack size $L_S = 2^{10}$.

L-values for each bit position in a frame was obtained by simulations, just as in the case of LISS, and plotted in Figure 4.15. Clearly, the linear tendency with path length has vanished. The numerical values are also rather small compared with the corresponding LISS result, which is shown in Figure 4.7.

The behavior of the soft outputs from LISSOM depends on the code and the E_b/N_0. Figure 4.16 shows the average L-value per bit position for the same conditions as in Figure 4.15 except that a memory 14 code is used. The plot shows that a linear tendency appears in the soft output of LISSOM for the higher memory code. The same kind of tendency also appears at lower E_b/N_0 levels. Both these situations, longer code memory or lower E_b/N_0, yield longer error bursts and, hence, a linear decrease in the L-values with increasing path length is expected.

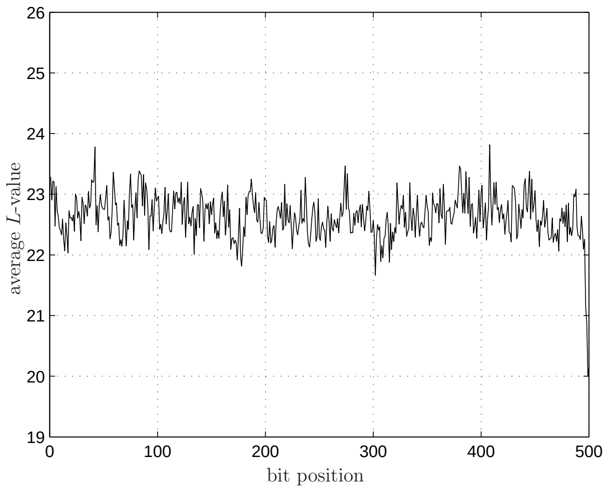


Figure 4.15: Average L-value per bit position for the LISSOM algorithm for the memory $m = 4$ $(4\ 66)$ systematic encoder at $E_b/N_0 = 3$ dB.

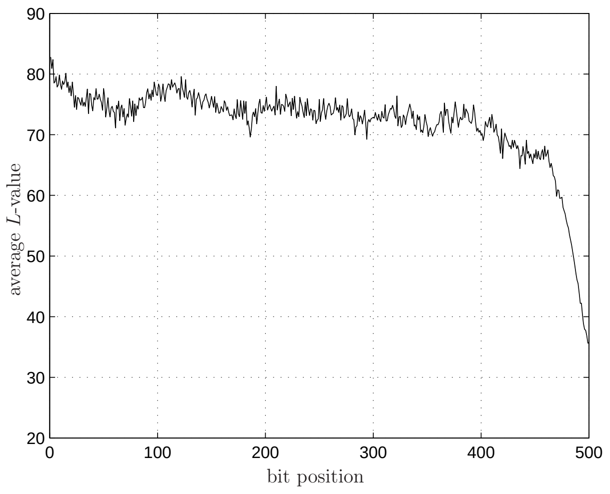


Figure 4.16: Average L-value per bit position for the LISSOM algorithm for the memory $m = 14$ $(4\ 71447)$ systematic encoder at $E_b/N_0 = 3$ dB.

The LISSOM algorithm manages to reduce the linear tendency in the soft output. To further investigate the quality of the soft output, the distribution of the L-values can be studied and a comparison with the BCJR soft outputs can be made. The true L-values obtained by the BCJR algorithm follow the Gaussian distribution. Figure 4.17 shows the histogram of the L-values produced by the LISSOM algorithm, and it illustrates a clear Gaussian-like behavior of the L-values. By increasing the memory of the code the linear tendency will affect the distribution of the L-values as illustrated in Figure 4.18.

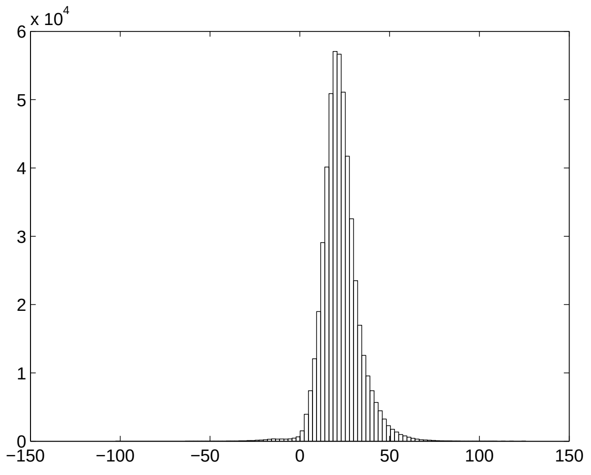


Figure 4.17: Histogram of the L-values of the LISSOM algorithm for the memory $m = 4$ $(4\ 66)$ systematic encoder at $E_b/N_0 = 3$ dB.

In Figures 4.19 and 4.20 a snapshot of 30 soft outputs from the LISSOM algorithm is compared with the corresponding outputs from the BCJR. In Figure 4.19 the L-values are manipulated using the detrend command in MATLAB in the same manner as for LISS in Figure 4.9. In Figure 4.20 the same L-values are plotted without using detrend.

The quality of the L-values expressed in terms of the relative discrepancy defined by (3.1) is illustrated in Figure 4.21. As shown in Figures 4.19, 4.20, and 4.21 the outputs of LISSOM suffers from rather large variations compared to the BCJR outputs, which decreases the usability of the soft outputs in an iterative scheme since it makes the attenuation harder to manage. The variations appear as strong peaks in the L-values. If those variations could be smoothed out, the quality of the soft outputs would improve.

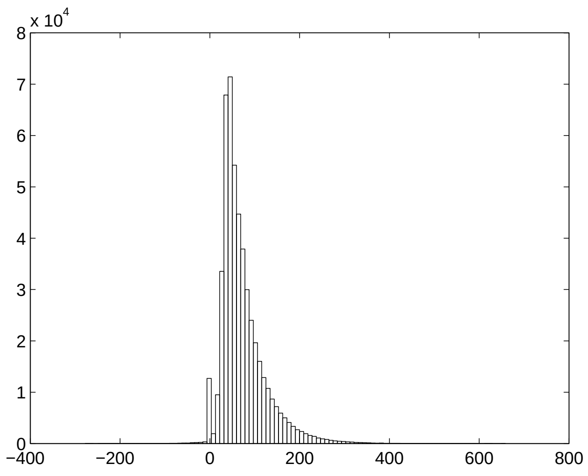


Figure 4.18: Histogram of the L-values for the LISSOM algorithm for the memory $m = 14$ $(4\ 71447)$ encoder at $E_b/N_0 = 3$ dB.

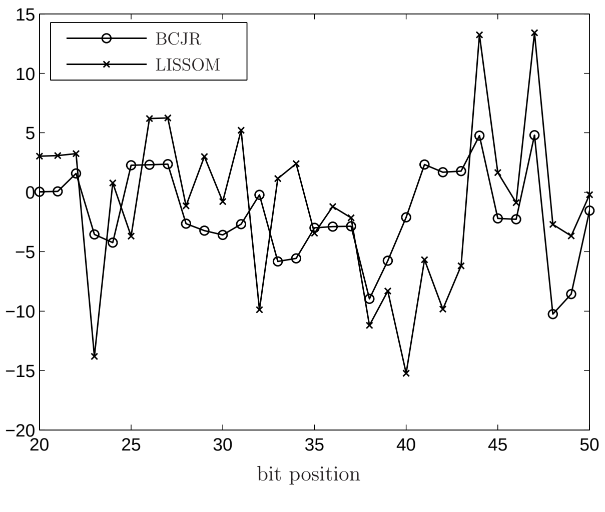


Figure 4.19: A snapshot of output L-values from the LISSOM and BCJR algorithms after applying detrend command in MATLAB for the memory $m = 4$ $(4\ 66)$ encoder and block length $K = 100$.

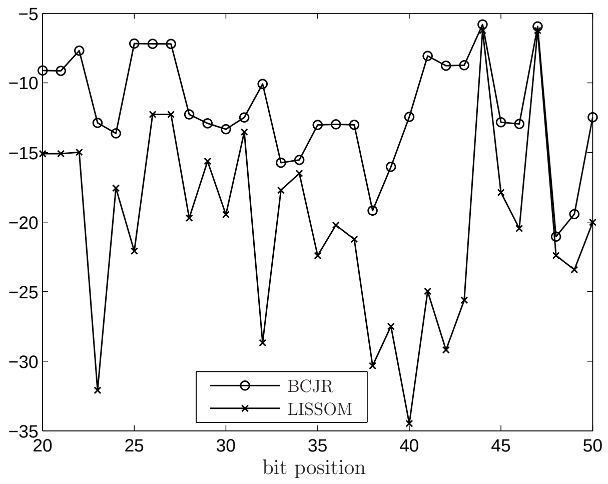


Figure 4.20: A snapshot of L-values (the same as in Figure 4.19) from the LISSOM and BCJR algorithms without manipulation with detrend in MATLAB.

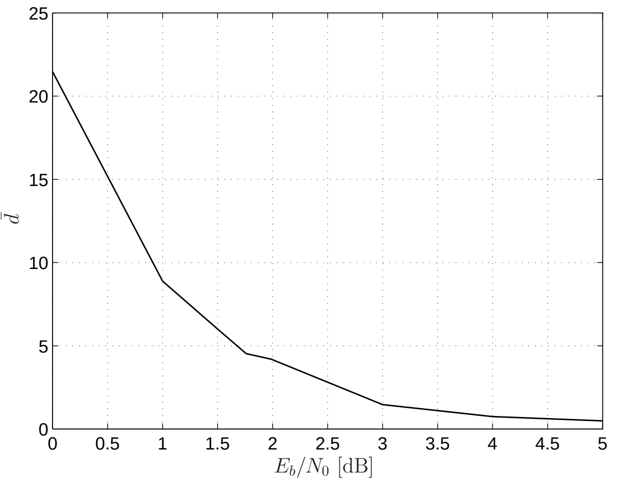


Figure 4.21: Average relative discrepancy between the soft outputs of the LISSOM and BCJR algorithms for the memory $m = 4$ $(4\ 66)$ encoder.

Chapter 5

Comparisons

5.1 Bit error rate

A comparison between the LISS, LISSOM, and M*-BCJR algorithms in terms of the bit error rate is illustrated in Figure 5.1. The memory $m = 4$ (4 66) systematic encoder and block length of $K = 500$ information bits were used for simulations. The stack size for LISS and LISSOM was set to $L_S = 2^{10}$ and the number of surviving states for M*-BCJR were $M = 4$ and 8. The BER of BCJR serves as an optimum reference. The M*-BCJR has better performance than LISS and LISSOM which was already suggested by the simulation results in previous chapters. For $M = 8$ the performance of the M*-BCJR is virtually the same as for BCJR.

5.2 Complexity

The complexity C is expressed in terms of the number of visited (extended) nodes in the code tree or trellis. The computational complexity for each node differs slightly among the algorithms. In LISS and LISSOM algorithms the computational complexity can be compared to the Viterbi algorithm (an add-compare-select operation on each node), while for BCJR and M*-BCJR the complexity for each node is dominated by the max* operation. Figures 5.2 and 5.3 illustrate the average number of visited nodes per information bit. The complexity of BCJR is constant for a particular encoder; it only depends on the memory m, and it is about three times the complexity of the Viterbi algorithm. Thus, for rate $R = 1/n$ codes, the complexity is 3×2^m. The complexity of M*-BCJR is also constant, determined by the number of surviving states M. Compared to the BCJR, the complexity is reduced to $3 \times M$. Note, however, that M*-BCJR and BCJR have the same branch complexity since branches in the reduced-state trellis of the M*-BCJR are not deleted,

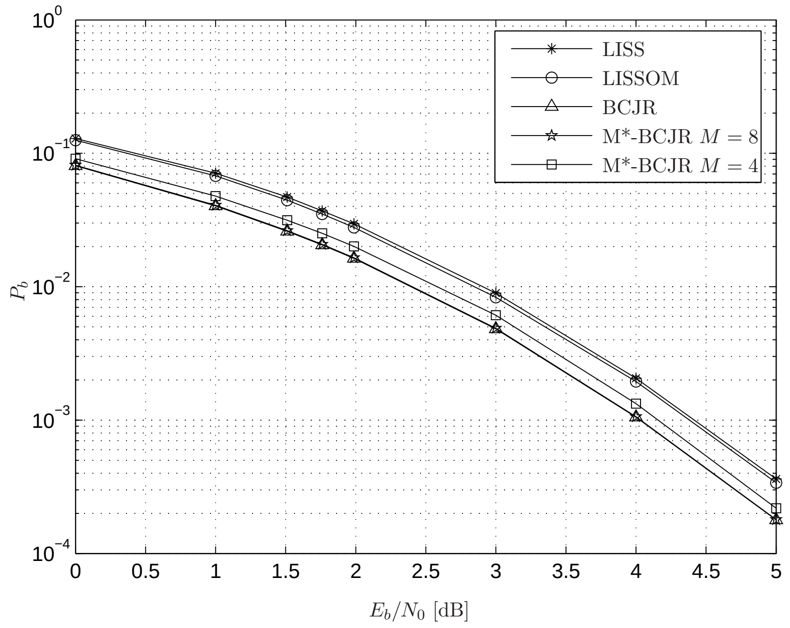


Figure 5.1: Bit error rate of the BCJR, M*-BCJR, LISSOM and LISS algorithms for the memory $m = 4$ $(4\ 66)$ systematic encoder.

but redirected during the forward recursion into the surviving M states. This implies that in the backward recursion of M*-BCJR the computational complexity per each node is slightly increased due to the increased number of branches arriving to each node.

The advantage of sequential decoding is that the decoding complexity is independent of the memory of the code. The complexity of the LISS algorithm is higher than that of the stack algorithm due to the additional augmentation step where all the paths in the stack are extended to full length. Due to this, the complexity of LISS is rather high for low memory codes and even exceeds the complexity of the BCJR, as illustrated in Figure 5.2. The complexity of LISSOM is much smaller than that of all the other algorithms. This is due to the matching operation which significantly shortens the augmentation.

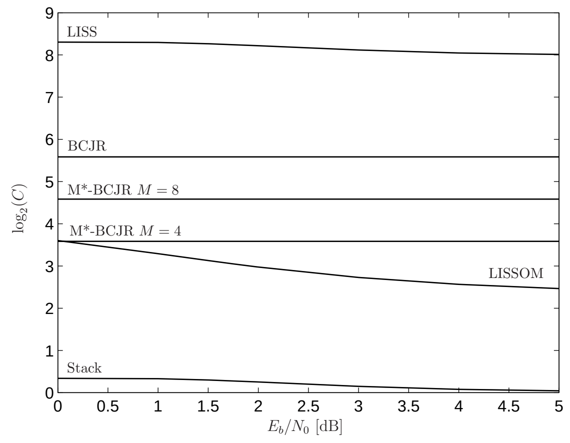


Figure 5.2: Complexities of LISS, BCJR, M*-BCJR, LISSOM, and Stack algorithms for the memory $m = 4$ $(4\ 66)$ encoder; block length $K = 500$ bits and stack size $L_S = 2^{10}$.

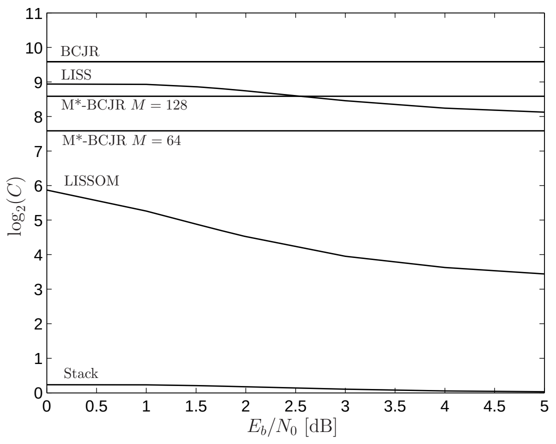


Figure 5.3: Complexities of BCJR, LISS, M*-BCJR, LISSOM, and Stack algorithms for the memory $m = 8$ $(4\ 671)$ encoder; block length $K = 500$ bits and stack size $L_S = 2^{10}$.

5.3 Quality of the soft output

In order to compare the soft outputs from the algorithms, the relative discrepancy d defined by (3.1) is used as a quality measure. In the comparisons we used a memory $m = 4$ (4 66) encoder with block length $K = 100$ information bits. The value of d was averaged over 1000 blocks at $E_b/N_0 = 3$ dB. Figures 5.4 and 5.5 show the quality of M*-BCJR soft outputs. The relative

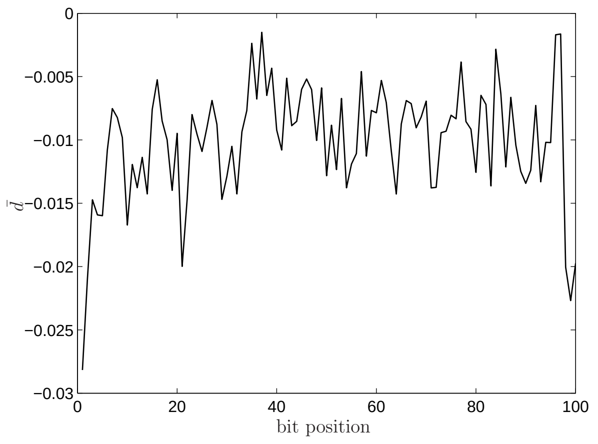


Figure 5.4: Average relative discrepancy $\bar{d}$ between the M*-BCJR and BCJR outputs for M=8, for the memory $m = 4$ (4 66) encoder.

discrepancy d between the LISS and BCJR outputs are shown in Figure 5.6. The value d is rather large, with a strong linear dependency on the bit position. Finally, the quality of the LISSOM outputs is illustrated in Figure 5.7.

In Figure 5.8, the quality of LISSOM and BCJR outputs in terms of the average relative discrepancy $\bar{d}$ is plotted versus the E_b/N_0 level. The outputs of the M*-BCJR are much closer to the true L-values obtained by the BCJR algorithm with the relative discrepancy 10% for $M = 8$ and $E_b/N_0 \geq 1.5$ dB.

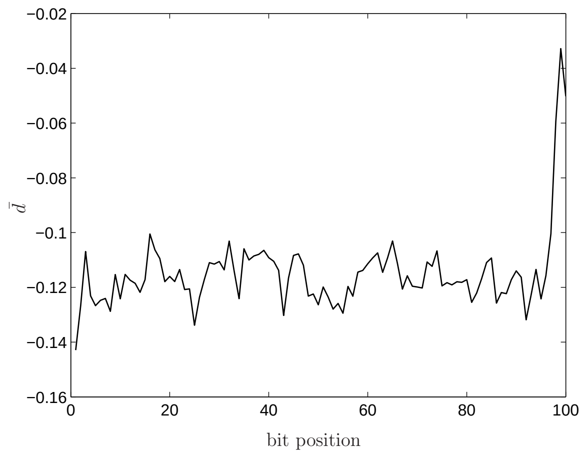


Figure 5.5: Average relative discrepancy $\bar{d}$ between the M*-BCJR and BCJR outputs for M=4 for the memory $m = 4$ $(4\ 66)$ encoder.

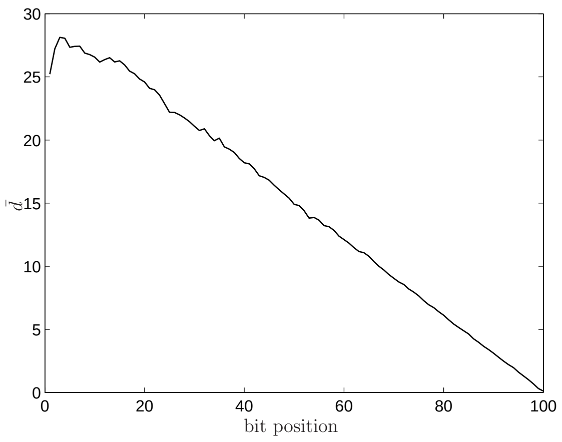


Figure 5.6: Average relative discrepancy $\bar{d}$ between the LISS and BCJR outputs for the memory $m = 4$ $(4\ 66)$ encoder and stack size $L_S = 2^{10}$.

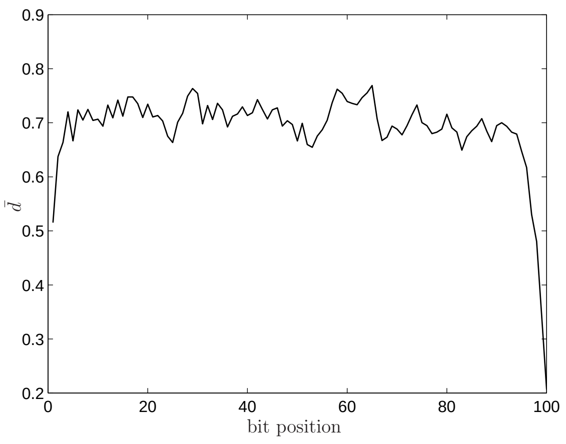


Figure 5.7: Average relative discrepancy $\bar{d}$ between the LISSOM and BCJR outputs for the memory $m = 4$ (4 66) encoder and stack size $L_S = 2^{10}$.

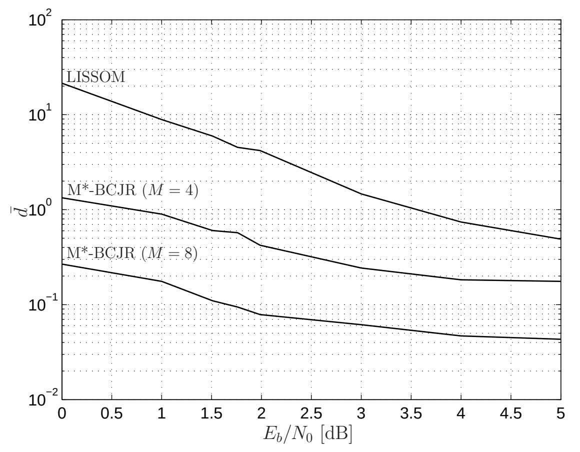


Figure 5.8: Quality of LISSOM and M*-BCJR ($M = 4$ and 8) soft outputs average relative discrepancy $\bar{d}$ for the memory $m = 4$ (4 66) encoder with stack size $L_S = 2^{10}$ and block length $K = 500$ bits.

5.4 Turbo decoding

Turbo decoding was first described in [18]. Here, we consider serially concatenated convolutional codes as illustrated in Figure 5.9. The transmitter consists of two encoders which are serially connected to each other. First, the sequence of binary data is fed into the first encoder, called the *outer encoder*. It produces a code sequence which is fed to an interleaver denoted by π. The purpose of the interleaver is to spread bursts of errors that occur because of the channel and to ensure that the errors appear randomly [19]. It permutes the data sequence according to a certain known pattern. After this the binary sequence is fed into the second encoder, called the *inner encoder*. The overall rate of the concatenated code equals the product of the rates of the outer and the inner encoders, that is, $R = R^o \times R^i$.

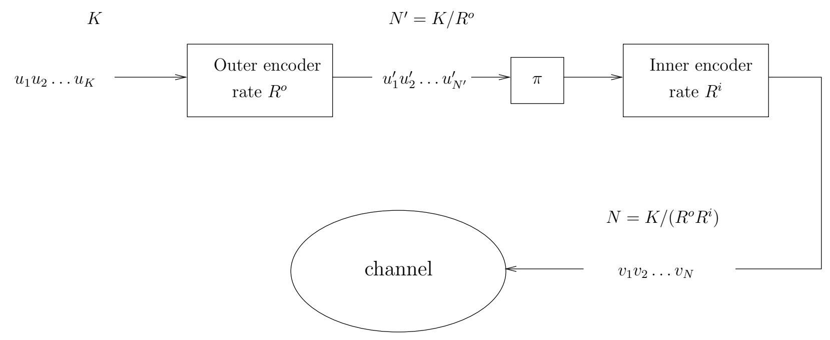


Figure 5.9: Structure of the transmitter in a serially concatenated coding (SCCC) scheme.

The output of the inner encoder is transmitted through the channel which corrupts the sequence. At the receiver the decoding process can be decomposed into two constituent decoders that are arranged in a concatenated scheme [6], as illustrated in Figure 5.10. Between the decoders are both an interleaver and a deinterleaver, denoted by π and π^{-1}, respectively. Both decoders produce soft symbol information, in the form of APP L-values according to (2.1). The soft values of the information bits, produced by the inner decoder, are transferred to the outer decoder and used as *a priori* information about the outer encoded bits in the next iteration. In this iterative scheme each iteration creates a better channel by refining the *a priori* information about the coded bits. A decoder in a turbo decoding scheme can be illustrated as in Figure 5.11 with the *intrinsic* or channel values L_{ch} and *a priori* values L_a as inputs and the *a posteriori* values as output. The L_{APP}-values at the

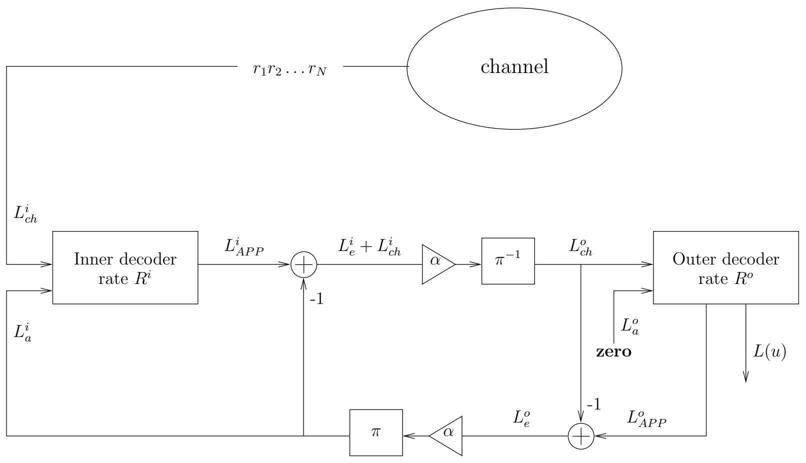


Figure 5.10: Structure of the receiver for a SCCC scheme.

Figure 5.11: General structure of a component decoder in a SCCC scheme.

output consist of three parts [14]

$$\begin{aligned} L_{APP} &= L_a + L_{ch} + L_e = \\ &= \underbrace{\ln \frac{\Pr(v_i = 0)}{\Pr(v_i = 1)}}_{a\ priori \text{ part}} + \underbrace{\ln \frac{\Pr(r_i | v_i = 0)}{\Pr(r_i | v_i = 1)}}_{intrinsic \text{ part}} + \underbrace{\ln \frac{\Pr(\boldsymbol{r}_{\not i} | v_i = 0)}{\Pr(\boldsymbol{r}_{\not i} | v_i = 1)}}_{extrinsic \text{ part}} \end{aligned}$$

where the *extrinsic* part is the information about the current bit found in all the other received bits. It is the additional information about the current bit obtained during decoding.

In the parallel concatenated schemes, only extrinsic information L_e is passed between the decoders [18], and both decoders in the scheme have knowledge about the channel. In the serially concatenated scheme, only the inner decoder has channel knowledge. Therefore, both *extrinsic* and channel information is passed from the inner decoder to the outer as illustrated in Figure 5.10. After deinterleaving this becomes the *a priori* input for the outer decoder. It is often necessary to attenuate the information passed between the decoders with a value $0 < \alpha < 1$ to enable convergence of the decoder [19].

Iterative schemes rely on the assumption that the *a priori* information about each bit is independent of the previous outputs of the algorithm [19]. However, the soft value that one of the decoders calculates about a bit is based on the *a priori* information from the other decoder and thus the inter-dependencies grow during the iterative process. Large interleavers reduce the correlation between the bits in the stream; however, they also imply large decoding delay for turbo codes.

5.5 Turbo scheme simulations

Simulations of the turbo scheme described in Section 5.4 were performed, where LISS, LISSOM, and M*-BCJR algorithms served as inner decoders, while the outer decoder was a BCJR decoder. Results for a setup with the memory $m = 4$ (4 66) outer decoder and the memory $m = 2$ (7 5) inner decoder are shown in Figure 5.12, and compared with the decoder using BCJR as both inner and outer decoder. The number of iterations was limited to 5.

The extrinsic information in the scheme was attenuated in order to improve the performance except when BCJR was used as the inner decoder. The results illustrated in Figure 5.12 were obtained with the attenuation coefficients {0.2 0.2 0.3 0.3 0.4} for both LISS and LISSOM and {0.2 0.2 0.2 0.2 0.2} for M*-BCJR. The choice of attenuation values is obtained by trial and error and can influence the performance significantly.

To improve the results for the LISS/BCJR scheme, two different approaches to attenuate the extrinsic values were tested. At first, a constant attenuation value was chosen for each iteration, where each L-value was attenuated with the same factor independently of the bit position. Since a strong linear tendency is present in the soft outputs, this approach did not yield good results. Another approach was to attenuate the soft outputs in a linear manner, by using the function

$$a(i) = \frac{1}{p(x)i}\,\rho$$

where i is the bit position or depth into the frame, ρ is an appropriately chosen constant, and $p(x)$ is the linear function

$$p(x) = \left(\frac{L(u_0) - L(u_{K-1})}{K}\right) x.$$

Unfortunately, this variant of attenuation did not significantly improve the performance.

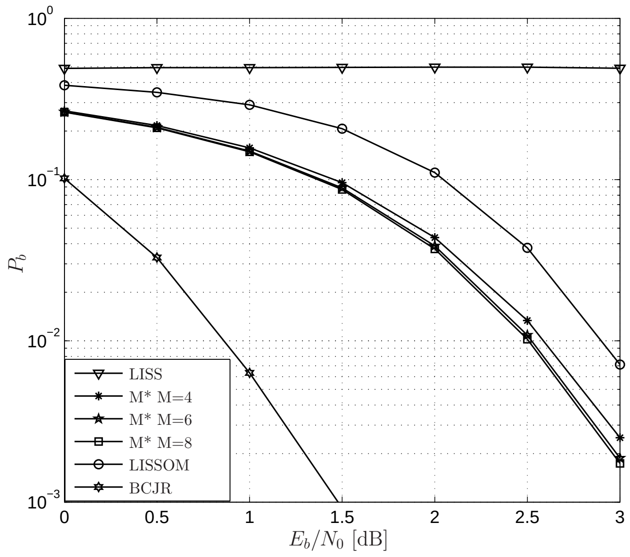


Figure 5.12: BER of a turbo decoder with LISS, LISSOM, M*-BCJR and BCJR algorithms as inner decoders. The outer decoder is BCJR. Block size is $K = 500$ information bits, stack size $L_S = 2^{10}$, and # iterations = 5.

A slight improvement in the bit error rate was achieved when LISS was replaced by LISSOM as the inner decoder. However, the bit error performance is still far from satisfactory. Both LISS and LISSOM outputs suffer from large fluctuations in the magnitude of the L-values (cf. Section 4.5.2) which results in poor convergence of the iterative scheme.

Chapter 6

Conclusions and Outlook

In this thesis, two suboptimum soft-input soft-output decoding algorithms were investigated: LISS and M*-BCJR. Both algorithms perform rather well for stand alone decoding. M*-BCJR has lower bit error rate due to better quality of L-values. For a given memory size, the complexity of M*-BCJR is constant and independent of E_b/N_0. The complexity of LISS is independent of the memory and decreases as E_b/N_0 increases. Hence, in a setup with a high memory encoder the LISS algorithm is preferred, but in other cases M*-BCJR should be considered.

Soft outputs of the LISS algorithm exhibit strong linearly decreasing with increasing bit position. If the tendency is removed, the L-values are similar to the corresponding outputs of the BCJR. By using a numerical tool to remove this tendency the quality of the soft output is improved. An alternative solution is to modify the LISS algorithm in such way that the linear tendency is removed.

A modification of the LISS algorithm, called LISSOM, was proposed in order to improve the quality of the soft output. Additionally, LISSOM has considerably lower complexity than LISS, without affecting the bit error rate. However, the increased quality of the soft output does not have sufficient impact on the iterative decoding scheme.

Both LISS and M*-BCJR were used as inner decoder in an iterative decoding scheme for serially concatenated convolutional codes. The outer decoder was a BCJR decoder. Both of the considered inner decoders have rather poor performance compared to the iterative decoder using two BCJR component decoders. Poor performance of the LISS inner decoder is mainly due to the strong linear tendency in the soft outputs. Different attenuation factors have little effect on the overall performance. M*-BCJR performs better than LISS in an iterative scheme; however, the performance gap with respect to the BCJR-BCJR decoder is significant.

An area of interest for future work is to further investigate the LISSOM algorithm and possible modifications of the metric that could improve its performance. In [2] and [1] it was reported that, unlike in iterative decoding, LISS and M*-BCJR perform very well when used as equalizers of an intersymbol interference channel, in iterative equalization-decoding schemes. Future work should focus first on verifying these findings and then investigating the principle differences of decoding and equalization mechanisms that result in different behavior of the LISS and M*-BCJR algorithms.

References

[1] M. Sikora and D. J. Costello Jr., "A New SISO Algorithm with Application to Turbo Equalization", in *Proc. IEEE Intern. Symposium Inform. Theory, ISIT 2005*, Adelaide, Australia, September 2005.

[2] J. Hagenauer and C. Kuhn "The Revival of Sequential Decoding", in *Proc. 5th Intern. ITG Conf. on Source and Channel Coding*, Erlangen, Germany, January 2004.

[3] Claude E. Shannon, "A Mathematical Theory of Communication", *The Bell System Technical Journal*, vol. 27, pp. 379423 (Part I), July, pp. 623656 (Part II), October 1948.

[4] R. Johannesson, *Informationsteori-grundvalen för (tele-)kommunikation*, Studentlitteratur, Sweden, 1988.

[5] V. Pavlushkov, *Capabilities of Convolutional Codes: Unequal Protection and More*, PhD Thesis, Lund University, Sweden, June 2005.

[6] S. Lin and D. J. Costello Jr., *Error Control Coding*, Second edition, Prentice-Hall, Upper Saddle River, USA, 2004.

[7] S. Kirkpatrick and E. P. Stoll, "A Very Fast Shift-Register Sequence Random Number Generator", *Journal Computational Physics*, vol. 40, no. 2, pp. 517-526, April 1981.

[8] V. Lakshmikantham, S. K. Sen and T. Samanta, "Comparing Random Number Generators Using Monte Carlo Integration", *ICIC International*, vol. 1, no. 2, pp. 143-165, June 2005.

[9] G. E. Box and M. E. Muller, "A Note on the Generation of Random Normal Deviates", *Ann. Math. Stat.*, no. 29, pp. 610-11, 1958.

[10] M. Handlery, *Tales of Tailbiting Codes*, PhD Thesis, Lund University, Sweden, September 2002.

[11] J. S. Baehr, *The Internet Encyclopedia of Philosophy - A Priori and A Posteriori*, http://www.iep.utm.edu/a/apriori.htm, 2006.

[12] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate", *IEEE Trans. Inform. Theory*, vol. 20, pp. 284-287, March 1974.

[13] V. Franz and J. B. Anderson, "Concatenated Decoding with a Reduced-Search BCJR Algorithm", *IEEE J. Select. Areas in Commun.*, vol. 16, no. 2, pp. 186-195, February 1998.

[14] R. Johannesson and K. Sh. Zigangirov, *Fundamentals of Convolutional Coding*, IEEE Press, Piscataway, USA, 1999.

[15] J. L. Massey, "Variable-length Codes and the Fano metric", *IEEE Trans. Inform. Theory*, vol. IT-18, pp. 196-198, 1972.

[16] C. Weiß, S. Riedel and J. Hagenauer, "Sequential Decoding Using *a priori* Information", *Electronic Letters*, vol. 32, no. 13, June 1996.

[17] J. Hagenauer and C. Kuhn, "The List-Sequential (LISS) Algorithm and its Application", submitted to *IEEE Trans. Commun.*, March 2006.

[18] C. Berrou, A. Glavieux and P. Thitimajshima, "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-codes", *IEEE Intern. Conference on Communications, ICC 93*, Geneva, Switzerland, May 1993.

[19] R. Koetter, A.C. Singer and M. Tuchler, "Turbo Equalization", *IEEE Signal Processing Magazine*, pp. 67-80, January 2004.