Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

DIPLOMA THESIS

Convergence of the Sum-Product Algorithm for Short Low-Density Parity-Check Codes

Institut of Communications and Radio-Frequency Engineering
Vienna University of Technology (TU Wien)

Telecommunications Research Center Vienna (ftw.)

Supervisors:
Univ.Prof. Dipl.-Ing. Dr.techn. Johann Weinrichter (TU Wien)
Dr. Jossy Sayir (ftw.)

Gottfried Lechner
Blumengasse 44/22, 1170 Vienna
Register No. 9525633

Vienna, April 3, 2003

DIPLOMARBEIT

Konvergenz iterativer Decodierung von Blockcodes mit schwach besetzten Prüfmatrizen (LDPC Codes)

Institut für Nachrichtentechnik und Hochfrequenztechnik
Technische Universität Wien (TU Wien)

Forschungszentrum Telekommunikation Wien (ftw.)

Betreuer:
Univ.Prof. Dipl.-Ing. Dr.techn. Johann Weinrichter (TU Wien)
Dr. Jossy Sayir (ftw.)

Gottfried Lechner
Blumengasse 44/22, 1170 Wien
Matrikelnummer 9525633

Wien, 3. April 2003

ABSTRACT

Several construction methods for regular and irregular low-density parity-check (LDPC) codes are investigated. The performance of these randomly and deterministically constructed codes—in terms of the bit error rate—is compared, where the block lengths of interest are between 10^3 and 10^4. These block lengths are suitable for practical applications.

The convergence behavior of the decoding process using the iterative sum-product algorithm is analyzed for these finite length LDPC codes. The decoding process is visualized by animations of the evolution of the messages passed in the decoder. With these visualizations, successful decoding as well as decoding errors are investigated.

Decoding errors of this iterative system are classified into 3 error types and their influence on the performance of the code is analyzed. In contrast to infinite block lengths where the decoder is guaranteed to converge, for finite block lengths it is possible that the decoder will not converge but become unstable.

KURZFASSUNG

Unterschiedliche Konstruktionsmethoden regulärer und irregulärer Blockcodes mit schwach besetzten Prüfmatrizen (low-density parity-check LDPC) werden untersucht. Der erzielbare Codegewinn dieser sowohl zufällig als auch deterministisch konstruierten Codes wird verglichen. Die behandelten Blocklängen sind im Bereich zwischen 10^3 und 10^4, da diese Blocklängen für praktische Anwendungen geeignet sind.

Das Konvergenzverhalten des iterativ arbeitenden Decoders wird für diese endlichen Blocklängen untersucht. Der Decodierungsprozess wird graphisch durch Animationen veranschaulicht. Mit diesen Animationen wird sowohl der Fall einer erfolgreichen als auch der Fall einer fehlerhaften Decodierung untersucht.

Die bei fehlerhafter Decodierung auftretenden Fehler werden in 3 Klassen unterteilt und ihr Einfluss auf die Bitfehlerrate wird analysiert. Im Gegensatz zu Codes mit unendlicher Blocklänge, wo die Konvergenz des Decoders garantiert ist, kann bei endlichen Blocklängen ein instabiles Verhalten des Decoders beobachtet werden.

Contents

1 **Introduction** 1

1.1 Communication System . . . 2

1.2 Channel Capacity . . . 3

2 **Binary Linear Block Codes** 6

2.1 Definitions . . . 6

2.2 Optimal Decoding . . . 8

2.2.1 Complexity . . . 10

2.3 Suboptimal Decoding . . . 10

2.3.1 Factor Graphs . . . 10

2.3.2 Factor Graphs and Block Codes . . . 11

2.3.3 The Sum-Product Algorithm . . . 17

2.3.4 Graphs with Cycles . . . 18

3 **Low-Density Parity-Check Codes** 20

3.1 Definitions . . . 20

3.2 Decoding . . . 21

3.2.1 Cyclefree Graphs and Infinite Block Length . . . 22

3.3 Encoding . . . 22

3.3.1 Encoding with the Generator matrix . . . 22

3.3.2 Encoding with the Parity-Check Matrix . . . 23

3.4 Regular and Irregular Codes . . . 24

3.5 Basic Construction of LDPC Codes . . . 25

3.5.1 Random Codes . . . 26

3.5.2 Deterministic Codes . . . 37

3.6 Extended Construction of LDPC . . . 41

3.6.1 Construction based on Large Girth . . . 41

3.6.2 Triangular Parity-Check Matrix . . . 42

3.7 Comparison of LDPC Codes with Turbo Codes . . . 44

3.8 Conclusion 47

4 Convergence of the Decoding Process 48

4.1 Iterative Decoding of Concatenated Codes 48

4.2 EXIT charts for LDPC codes 52

4.3 Visualization of the Decoding Process 55

4.3.1 Regular Codes 56

4.3.2 Irregular Codes 58

4.3.3 Decoding Errors 59

4.4 Number of Required Iterations 66

4.5 Conclusion 69

5 Conclusion 70

A Calculation of the Gallager Bound 72

ACKNOWLEDGMENT

In February 2002 I started my diploma thesis that was supervised by Prof. Johann Weinrichter from the Technical University of Technology Vienna and Dr. Jossy Sayir from Telecommunications Research Center Vienna (ftw.), where I found a perfect location for scientific work and met a lot of nice people. In May 2002 ftw. offered me a part-time employment which helped me a lot to concentrate on this work.

I want to thank Prof. Weinrichter for giving me freedom in the choice of my thesis and for supporting me during the whole period. I have to thank Jossy Sayir for his excellent explanations and for many valuable discussions during my work. Further thanks go to all my colleagues at ftw. Besides many discussions and suggestions I also had a lot of fun while working on my thesis there.

Last but not least I would like to thank all my friends and of course my parents who supported me all the time and made all this possible.

Gottfried Lechner
Vienna, March 2003

Chapter 1

Introduction

Conventional channel coding techniques with limited decoding complexity are far away from the theoretical limits for transmission rates for noisy channels shown in [16]. Modern coding systems eliminate most of the limitations of decoding complexity by using an iterative decoder which requires low computational complexity. With this technique, it is possible to reduce the difference of required signal to noise ratio between practical systems and theoretical limits to fractions of decibels.

Iterative decoding techniques have a decoding complexity that is linear in the block length. One family of codes that achieve this linear decoding complexity are Turbo codes introduced in [3]. The long block length is achieved by using an interleaver and the decoding complexity is kept small by using simple component codes.

Another family of codes with linear decoding complexity are low-density parity-check (LDPC) codes which were introduced in [10]. The key to the simple decoding process of these codes is the use of sparse parity-check matrices.

The focus of this work is on LDPC codes with finite block lengths between 10^3 and 10^4. This limitation of the block lengths is usually due to delay constraints of the application.

This chapter introduces the model of the communication system used in this work and gives an overview of achievable transmission rates for noisy channels. Chapter 2 introduces basics of binary linear block codes and derives the optimal and a suboptimal decoding algorithm. Chapter 3 compares several construction methods for LDPC codes and the performance of the resulting codes. The convergence behavior of the iterative decoder is treated in chapter 4.

1.1 Communication System

The communication system used in this work is shown in figure 1.1. It consists of the following components:

- *Source Encoder*
  The source encoder removes redundancy from the binary source data and produces a binary vector $\boldsymbol{u}$ of length k (it is assumed that the length of this vector is constant).

- *Channel Encoder*
  The vector $\boldsymbol{u}$ from the source encoder is encoded to a binary codeword $\boldsymbol{x}$ of length n (where $n \geq k$) by adding known redundancy to the information vector $\boldsymbol{u}$.

- *Modulator*
  The binary digits of the codeword are not suited for transmission over the channel. The modulator converts the binary digits to symbols that can be send over the channel.

- *Channel*
  In this work, the channel is assumed to be memoryless with discrete input alphabet $\mathcal{A} = \{-1, 1\}$. The channel adds white Gaussian noise and therefore, the output alphabet of the channel is the set of real numbers. This channel is called a discrete input continuous output memoryless channel.

- *Demodulator*
  The demodulator receives symbols from the channel and provides *soft values* for every digit of the codeword for the following channel decoder.

- *Channel Decoder*
  The task of the channel decoder is to use the knowledge of the added redundancy from the channel encoder to remove transmission errors introduced by the channel. The output of the channel decoder is an estimate of the transmitted information vector.

- *Source Decoder*
  The source decoder simply reverses the source encoding operation and produces the original data if no errors occurred.

Source coding is not considered in this work. Therefore, the source coder and the source decoder will be included in the source and the sink respectively.

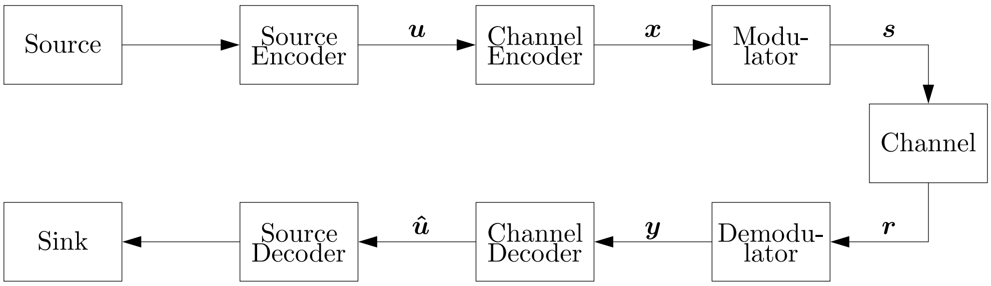

Figure 1.1: Communication system.

1.2 Channel Capacity

Channel capacity for the additive white Gaussian noise (AWGN) channel with continuous input and continuous output can be written as

$$C_{AWGN} = \frac{d}{2} \cdot \log_2 \left(1 + \frac{P_s}{P_z}\right) \text{[bits per channel use]}, \tag{1.1}$$

where d is the number of signal dimensions and P_s and P_z are the signal and the noise power respectively. Defining the two sided power spectral density of the noise as $\sigma^2 = \frac{N_0}{2}$ and the required bandwidth as $2B$, channel capacity is

$$C_{AWGN} = \frac{d}{2} \cdot \log_2 \left(1 + \frac{\frac{E_s}{T}}{\frac{N_0}{2} 2B}\right), \tag{1.2}$$

where T is the symbol duration and E_s denotes the mean energy per symbol. The required bandwidth $2B$ and the symbol duration are related as

$$2B \cdot T = d, \tag{1.3}$$

which leads to

$$C_{AWGN} = \frac{d}{2} \cdot \log_2 \left(1 + \frac{2E_s}{N_0 d}\right). \tag{1.4}$$

For a fair comparison of codes, we are interested in channel capacity as a function of $\frac{E_b}{N_0}$ where E_b is the energy per information bit. Defining R_s as the number of information bits contained in one symbol (R_s contains the rate of the code and the mapping scheme), we can write

$$\frac{2E_s}{N_0 d} = \frac{2R_s E_b}{N_0 d}. \tag{1.5}$$

Capacity is achieved if $R_s = C$ and therefore,

$$C_{AWGN} = \frac{d}{2} \cdot \log_2 \left(1 + \frac{2C_{AWGN}}{d} \cdot \frac{E_b}{N_0}\right), \tag{1.6}$$

which simplifies to

$$C_{AWGN} = \frac{1}{2} \cdot \log_2 \left(1 + 2C_{AWGN} \cdot \frac{E_b}{N_0} \right) \tag{1.7}$$

in the case of one dimension.

If the input of the channel is restricted to two symbols $\{-A, A\}$, the channel is called binary input AWGN (BIAWGN) channel. The capacity of this channel is below the capacity for the continuous input AWGN channel and it is achieved if and only if the two symbols are equally likely ($P(-A) = P(A) = 0.5$).

Channel capacity is defined as the maximum mutual information between the input X and the output Y of the channel. For equally likely input symbols the capacity of the BIAWGN channel is

$$C_{BIAWGN} = h(Y) - h(Y|X), \tag{1.8}$$

where $h()$ denotes differential entropy. The output Y conditioned on the input X is Gaussian distributed and therefore

$$h(Y|X) = \frac{1}{2} \log_2 \left(2\pi e \sigma^2 \right). \tag{1.9}$$

The probability function $f(y)$ is the sum of two Gaussian functions

$$\begin{aligned} f(y) &= P(-A) \cdot \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y+A)^2}{2\sigma^2}} + P(A) \cdot \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-A)^2}{2\sigma^2}} \\ &= \frac{1}{\sqrt{8\pi\sigma^2}} \left[e^{-\frac{(y+A)^2}{2\sigma^2}} + e^{-\frac{(y-A)^2}{2\sigma^2}} \right]. \end{aligned} \tag{1.10}$$

Capacity for the BIAWGN channel can be written as

$$C_{BIAWGN} = -\int_{-\infty}^{\infty} f(y) \log_2 f(y) dy - \frac{1}{2} \log_2 \left(2\pi e \sigma^2 \right). \tag{1.11}$$

The capacities of these two channels are shown in figure 1.2. The figure includes some codes with their spectral efficiency and their required $\frac{E_b}{N_0}$ for achieving a bit error rate of 10^{-5} when used for transmission over a BIAWGN channel (this figure is intended to give an overview only).

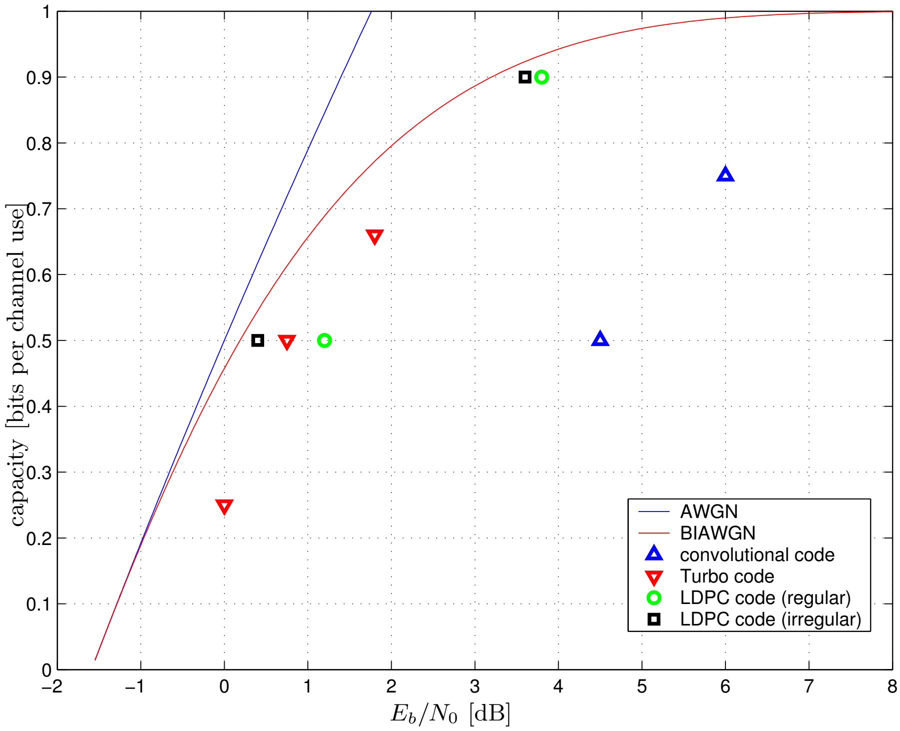

Figure 1.2: Overview of codes.

Chapter 2

Binary Linear Block Codes

Low-density parity-check codes are a subclass of the class of binary linear block codes. This chapter introduces binary linear block codes and the optimal decoder for these codes—the maximum a-posteriori (MAP) decoder. The term 'optimal' refers to minimization of the bit error rate. We will see that the computational complexity of this optimal decoder grows exponentially with the block length and therefore, long block lengths can generally not be used in practical applications. However, we will see that there exists a suboptimal decoding algorithm with linear complexity that allows the use of long block codes.

The first section gives some definitions. In the second section, the optimal decoder is derived and in the last section, we will consider the suboptimal decoding algorithm.

2.1 Definitions

Definition 1 (Binary block code). *A binary block code of length* n *is a set* $\mathcal{C}$ *of* 2^k *distinct vectors in* $GF(2)^n$*, with* $0 \leq k \leq n$*.*

This definition is very general. In this work, only linear codes are considered.

Definition 2 (Linear Code). *A code* $\mathcal{C}$ *is called linear if and only if all linear combinations of codewords are also elements of the code.*

Due to this definition, the all-zero word is always an element of the code.

Definition 3 (Rate of code). *The rate* R *of a code is defined as* $R \overset{\text{def}}{=} \frac{k}{n}$*.*

Definition 4 (Hamming weight). *The Hamming weight* $\operatorname{wt}(\boldsymbol{x})$ *of a vector* $\boldsymbol{x} = x_1, \ldots, x_n$ *is defined as the number of non-zero components of* $\boldsymbol{x}$*.*

Definition 5 (Hamming distance). *The Hamming distance* $\operatorname{dist}(\boldsymbol{x}, \boldsymbol{y})$ *between two vectors* $\boldsymbol{x} = x_1, \ldots, x_n$ *and* $\boldsymbol{y} = y_1, \ldots, y_n$ *is defined as the number of components where* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *differ.*

The weight and the distance are related as

$$\operatorname{wt}(\boldsymbol{x}) = \operatorname{dist}(\boldsymbol{x}, \mathbf{0}) \tag{2.1}$$

$$\operatorname{dist}(\boldsymbol{x}, \boldsymbol{y}) = \operatorname{dist}(\boldsymbol{x} - \boldsymbol{y}, \mathbf{0}) = \operatorname{wt}(\boldsymbol{x} - \boldsymbol{y}), \tag{2.2}$$

where all operations are carried out in $GF(2)^n$.

Definition 6 (Minimum distance). *The minimum distance of a code* $\mathcal{C}$ *is the minimum distance between two distinct codewords:*

$$d \overset{\text{def}}{=} \min_{\substack{\boldsymbol{x}, \boldsymbol{y} \in \mathcal{C} \\ \boldsymbol{x} \neq \boldsymbol{y}}} \operatorname{dist}(\boldsymbol{x}, \boldsymbol{y}). \tag{2.3}$$

Theorem 1. *The minimum distance of a linear code is equal to the minimum weight of all nonzero codewords:*

$$d = \min_{\substack{\boldsymbol{x} \in \mathcal{C} \\ \boldsymbol{x} \neq \mathbf{0}}} \operatorname{wt}(\boldsymbol{x}). \tag{2.4}$$

Proof. Due to the definition of a linear code $\boldsymbol{z} = \boldsymbol{x} - \boldsymbol{y}$ is also an element of the code. Using equation 2.2, we can write

$$d = \min_{\substack{\boldsymbol{x}, \boldsymbol{y} \in \mathcal{C} \\ \boldsymbol{x} \neq \boldsymbol{y}}} \operatorname{dist}(\boldsymbol{x}, \boldsymbol{y}) = \min_{\substack{\boldsymbol{x}, \boldsymbol{y} \in \mathcal{C} \\ \boldsymbol{x} \neq \boldsymbol{y}}} \operatorname{dist}(\boldsymbol{x} - \boldsymbol{y}, \mathbf{0}) = \min_{\substack{\boldsymbol{z} \in \mathcal{C} \\ \boldsymbol{z} \neq \mathbf{0}}} \operatorname{wt}(\boldsymbol{z}). \tag{2.5}$$

□

A binary linear code $\mathcal{C}$ is a subspace of dimension k in the vector space $GF(2)^n$. We can construct a basis for this subspace consisting of k linearly independent vectors of length n. Every codeword can be represented as a linear combination of these basis vectors.

Definition 7 (Generator matrix). *A generator matrix is a matrix* $\boldsymbol{G}$ *whose rows form a basis for a* k*-dimensional subspace* $\mathcal{S}$ *in the vector space* $GF(2)^n$*. Every codeword* $\boldsymbol{x}$ *of a linear code can be represented as a linear combination of these basis vectors, i.e.,*

$$\boldsymbol{x} = \boldsymbol{u} \cdot \boldsymbol{G}. \tag{2.6}$$

Instead of constructing a basis for the k-dimensional subspace of the vector space $GF(2)^n$, we can construct a basis for the orthogonal $(n-k)$-dimensional subspace $\mathcal{S}^{\perp}$.

Definition 8 (Parity-check matrix). *A parity check matrix is a matrix* $\boldsymbol{H}$ *whose rows form a basis for a* $(n-k)$*-dimensional subspace* $\mathcal{S}^{\perp}$ *in the vector space* $GF(2)^n$*. A vector* $\boldsymbol{x}$ *is an element of the code* $\mathcal{C}$ *if and only if it is orthogonal to this subspace*

$$\boldsymbol{x} \in \mathcal{C} \Longleftrightarrow \boldsymbol{x} \cdot \boldsymbol{H}^T = \mathbf{0}. \tag{2.7}$$

Because of the orthogonality of their underlying subspaces, the matrices $\boldsymbol{G}$ and $\boldsymbol{H}$ are related as

$$\boldsymbol{G} \cdot \boldsymbol{H}^T = \mathbf{0}. \tag{2.8}$$

2.2 Optimal Decoding

Let us consider the transmission system shown in figure 2.1. The source generates a binary vector $\boldsymbol{u}$ of length k (the information block). This vector is uniquely encoded to a binary codeword $\boldsymbol{x}$ of length n. The codeword is transmitted over a memoryless channel with discrete input and continuous output with transition probability density function $p(y|x)$. The decoder receives the vector $\boldsymbol{y}$ and makes a decision $\hat{\boldsymbol{u}}$ for the transmitted information block. Because of the unique correspondence between information block and codeword, the decoder can be split into two parts. The first part calculates a decision $\hat{\boldsymbol{x}}$ for the transmitted codeword $\boldsymbol{x}$ and the second part simply reverses the encoding operation. Reversing the encoding operation can be simplified if systematic codes are used, i.e. the first (or the last) k digits of the codeword $\boldsymbol{x}$ are equal to the information vector $\boldsymbol{u}$.

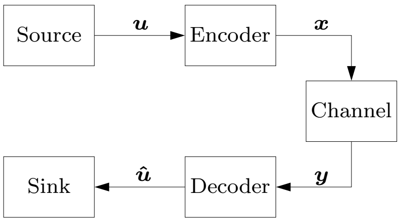

Figure 2.1: Transmission system.

In this work, we understand optimal decoding to mean a decoding procedure that minimizes the average bit error probability. The average probability of a bit error is

$$P_{Biterror} = \frac{1}{n}\sum_{l=1}^{n} P(\hat{x}_l \neq x_l|\boldsymbol{y}) = \frac{1}{n}\sum_{l=1}^{n}\left[1 - P(\hat{x}_l = x_l|\boldsymbol{y})\right], \tag{2.9}$$

where $x_1, \dots, x_n$ are the digits of the codeword.

The quantities $P(\hat{x}_l = x_l|\boldsymbol{y})$ are called a-posteriori probabilities (APP) of x_l given the received vector $\boldsymbol{y}$. Minimizing the bit error probability is equivalent to maximizing the APPs over the symbol alphabet ($\mathcal{A} = \{0, 1\}$ in our case of binary codes) for every digit of the codeword. Therefore, the decoder has to find the values $\hat{x}_l$, $(l = 1, \dots, n)$ that maximize these APPs. The APP for the digit x_l is

$$P(\hat{x}_l = x_l|\boldsymbol{y}) = \frac{P(\hat{x}_l = x_l, \boldsymbol{y})}{P(\boldsymbol{y})} = \frac{\sum_{\sim\{x_l\}} P(\boldsymbol{x}, \boldsymbol{y})\Big|_{x_l=\hat{x}_l}}{P(\boldsymbol{y})}, \tag{2.10}$$

where $\sum_{\sim\{x_l\}}$ denotes the summation over the symbol alphabet over all elements of $\boldsymbol{x}$ except x_l. This so called *summary operator* [12] is defined as

$$\sum_{\sim\{x_l\}} f(\boldsymbol{x}) \overset{\text{def}}{=} \sum_{\substack{x_i \in \mathcal{A} \\ i \neq l}} f(\boldsymbol{x}). \tag{2.11}$$

We can split up the joint probability as

$$P(\boldsymbol{x}, \boldsymbol{y}) = P(\boldsymbol{x})P(\boldsymbol{y}|\boldsymbol{x}) = P(\boldsymbol{x}) \prod_{i=1}^{n} P(y_i|x_i), \tag{2.12}$$

where we used the assumption that the channel is memoryless.

If we assume equally likely codewords, we can write the probability of the transmitted vector $\boldsymbol{x}$ as

$$P(\boldsymbol{x}) = \begin{cases} \frac{1}{|\mathcal{C}|}, & \boldsymbol{x} \in \mathcal{C}, \\ 0, & \boldsymbol{x} \notin \mathcal{C}, \end{cases} \tag{2.13}$$

where $|\mathcal{C}|$ denotes the number of codewords. The vector $\boldsymbol{x}$ is an element of $\mathcal{C}$ if it fulfills all parity-check equations. Let

$$f_j(\boldsymbol{x}) = \begin{cases} 1, & \boldsymbol{h}_j \cdot \boldsymbol{x}^T = 0, \\ 0, & \boldsymbol{h}_j \cdot \boldsymbol{x}^T \neq 0, \end{cases} \tag{2.14}$$

where $\boldsymbol{h}_j$ is the j$^{\text{th}}$ row of the parity-check matrix $\boldsymbol{H}$ of dimension $m \times n$ with $m = n - k$. Therefore,

$$P(\boldsymbol{x}) = \frac{1}{|\mathcal{C}|} \prod_{j=1}^{m} f_j(\boldsymbol{x}). \tag{2.15}$$

Inserting 2.12 in 2.10 and using 2.15 yields

$$P(\hat{x}_l = x_l|\boldsymbol{y}) = \frac{\sum_{\sim\{x_l\}} \left(\frac{1}{|\mathcal{C}|} \prod_{j=1}^{m} f_j(\boldsymbol{x}) \cdot \prod_{i=1}^{n} P(y_i|x_i) \right)\Big|_{x_l=\hat{x}_l}}{P(\boldsymbol{y})}. \tag{2.16}$$

The factors $\frac{1}{P(\boldsymbol{y})}$ and $\frac{1}{|\mathcal{C}|}$ do not influence the maximization with respect to x_l and can be omitted. Finally, the decision rule of the MAP decoder can be written as

$$\hat{x}_l = \operatorname*{argmax}_{x_l \in \mathcal{A}} \sum_{\sim\{x_l\}} \left(\prod_{j=1}^{m} f_j(\boldsymbol{x}) \cdot \prod_{i=1}^{n} P(y_i|x_i) \right). \tag{2.17}$$

For a binary alphabet, it is convenient to use the logarithm of the ratio of the two possible values

$$L(x_l|\boldsymbol{y}) = \log \frac{P(x_l = 0|\boldsymbol{y})}{P(x_l = 1|\boldsymbol{y})}. \tag{2.18}$$

These quantities are called Log-Likelihood ratios (LLR). The maximization of these values over the alphabet values can be written as

$$\hat{x}_l = \begin{cases} 0, & L(x_l|\boldsymbol{y}) > 0, \\ 1, & L(x_l|\boldsymbol{y}) < 0. \end{cases} \tag{2.19}$$

2.2.1 Complexity

The optimal decoding procedure can be described in a closed form. Before we try to implement this procedure, it may be worth taking a look at its computational complexity. For every digit of the codeword, we have to calculate a $(n-1)$-fold sum leading to 2^{n-1} evaluations of the argument. Every argument consists of $m+n = (2-R)n$ factors. The complexity for this optimal decoding procedure rises exponentially with the block length n. Therefore, the block length for practical systems is limited. The next section will describe a suboptimal decoding algorithm with a complexity that is linear in the block length.

2.3 Suboptimal Decoding

To be able to use long block codes, we need a decoding algorithm that can be computed with linear complexity (with respect to the block length). The algorithm described in this section works iteratively by passing messages on an associated factor graph. The next subsection introduces factor graphs and the results will be applied to our decoding problem.

2.3.1 Factor Graphs

Let us start with a simple graphical representation of a function. Consider a function $f(\boldsymbol{x})$, where $\boldsymbol{x}$ is a vector of independent variables $x_1, \ldots, x_n$. If we draw a circle for each variable and a square for the function $f(\boldsymbol{x})$, we can represent the dependence of $f(\boldsymbol{x})$ with a simple graph. An example is shown in figure 2.2. Note that there is an edge in the graph if and only if the value of the function depends on this variable. Since the value of our function depends on all variables, there is an edge from every variable node to the function node. This graph does not tell us very much about the function f, except that it depends on the variables $x_1, \ldots, x_7$.

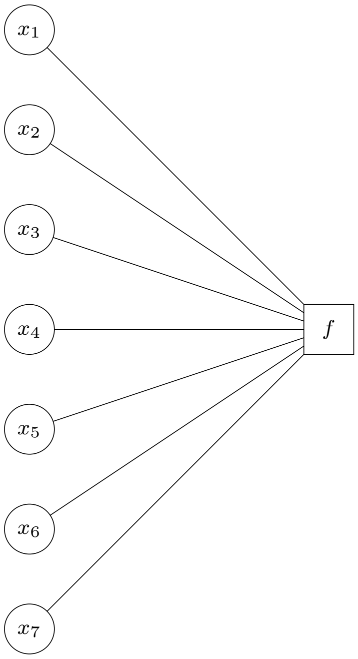

Figure 2.2: Graphical representation of a function.

Many functions can be factorized as a product of simpler functions that do not depend on the whole vector $\boldsymbol{x}$. Let us assume that we can factorize our function $f(\boldsymbol{x})$ in the following way.

$$f(\boldsymbol{x}) = f_1(x_1, x_4, x_5) \cdot f_2(x_2, x_5, x_6) \cdot f_3(x_3, x_6, x_7) \tag{2.20}$$

We can now represent every so called *local function* f_j as a square and draw the corresponding dependencies. This is shown in figure 2.3. The *global function* $f(\boldsymbol{x})$ is the product of all local functions.

2.3.2 Factor Graphs and Block Codes

We will use factor graphs to derive a decoding algorithm for block codes. In the decision rule of the MAP decoder

$$\hat{x}_l = \underset{x_l}{\operatorname{argmax}} \sum_{\sim\{x_l\}} \left(\prod_{j=1}^{m} f_j(\boldsymbol{x}) \cdot \prod_{i=1}^{n} P(y_i|x_i) \right)$$

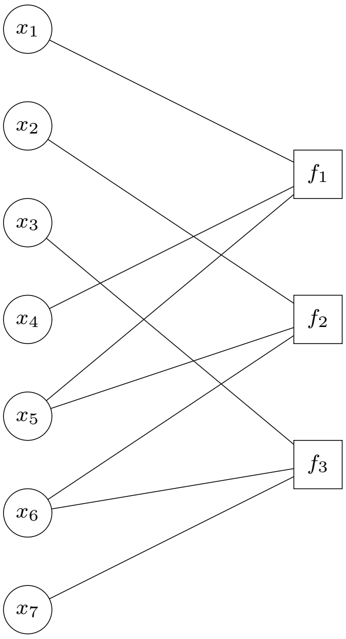

Figure 2.3: Factorization.

we first calculate a function that can be factored. This function is the probability mass function and it depends on the vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. Then, we calculate the marginal probability mass function by applying the summary operator and maximize the result to get the decision for the digits of the codeword. The first step is to represent the argument of the summary operator as a factor graph. The graph consists of two parts—one part represents the channel with its transition probability, and the other part represents the code. The following example demonstrates this representation.

Example 1 (Factor Graph for Decoding). *Let us assume a linear binary block code $\mathcal{C}$ represented by its parity-check matrix*

$$\boldsymbol{H} = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix}, \tag{2.21}$$

$$\mathcal{C} = \{\boldsymbol{x} \in GF(2)^7 | \boldsymbol{x} \cdot \boldsymbol{H}^T = \boldsymbol{0}\}. \tag{2.22}$$

The length of the codewords is $n = 7$ and the number m of parity-checks is equal to 3. The argument of equation 2.17 can be written as

$$f(\boldsymbol{x}, \boldsymbol{y}) = \underbrace{\prod_{j=1}^{3} f_j(\boldsymbol{x})}_{code} \cdot \underbrace{\prod_{i=1}^{7} P(y_i | x_i)}_{channel} \tag{2.23}$$

and factorizes into two parts. The corresponding factor graph is shown in figure 2.4. The left part of the figure represents the channel and the right part represents the code, where every row of the parity-check matrix is drawn as a function node.

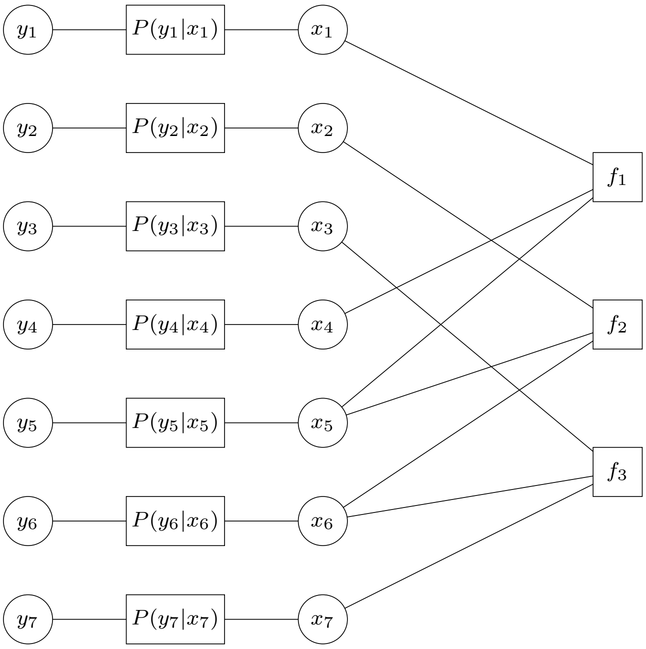

Figure 2.4: Factor graph for code and channel.

With the assumption $P(x_l = 0) = P(x_l = 1) = 0.5$, we can write the received LLR values as

$$\begin{aligned} L(x_l|y_l) &= \log \frac{P(x_l = 0|y_l)}{P(x_l = 1|y_l)} \\ &= \log \frac{P(y_l|x_l = 0)P(x_l = 0)P(y_l)}{P(y_l|x_l = 1)P(x_l = 1)P(y_l)} \\ &= \log \frac{P(y_l|x_l = 0)}{P(y_l|x_l = 1)}. \end{aligned} \tag{2.24}$$

With these quantities we can simplify the factor graph by removing the channel and using only the LLR values. The simplified factor graph is shown in figure 2.5. Every circle is called a *variable node* (it represents a digit of the codeword) and every local function is called a *check node* (it represents a parity-check equation).

After this simplification, we can calculate a LLR value for every digit x_l of the codeword based on a single observation y_l. However, the MAP decoding rule

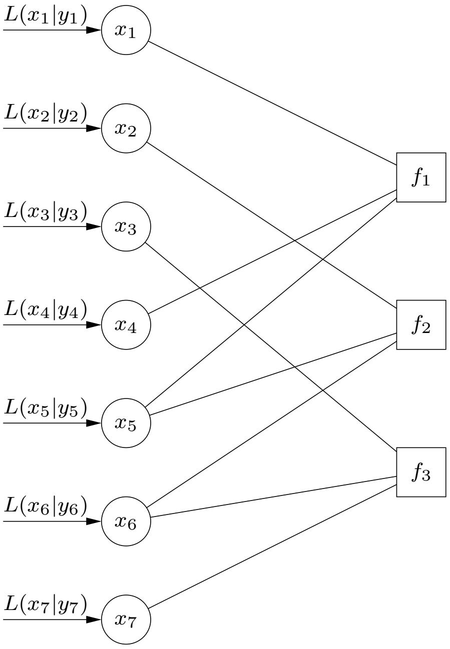

Figure 2.5: Simplified factor graph.

requires the computation of the LLR value based on the complete observed vector $\boldsymbol{y}$.

Suppose we want to calculate the MAP decision for the digit x_5 of the codeword. The first step is to redraw our factor graph, so that the desired variable node is the root of the graph. We assume that the graph can be drawn as a tree, i.e. it does not contain cycles. This assumption will be discussed later. The redrawn factor graph is shown in figure 2.6. All messages in the graph are transmitted from the bottom to the top (to the node x_5) and every node can send its outgoing messages after it has received all incoming messages. Figure 2.7 shows the messages that 'flow' up to the root of the tree (the circled numbers indicate the steps). On their way, they 'collect' all information from the received vector $\boldsymbol{y}$. After the fourth step, the desired variable x_5 has received all messages that are necessary to calculate the desired value $L(x_5|\boldsymbol{y})$. To complete the description of this algorithm, we have to investigate the calculations at the variable and the check nodes.

Computation at a Variable Node

A variable node x receives messages of the form $L(x|\mathcal{Y}_l)$, where $\mathcal{Y}_l$ are sets of received values. Any two sets are disjoint, because we assumed that the graph can

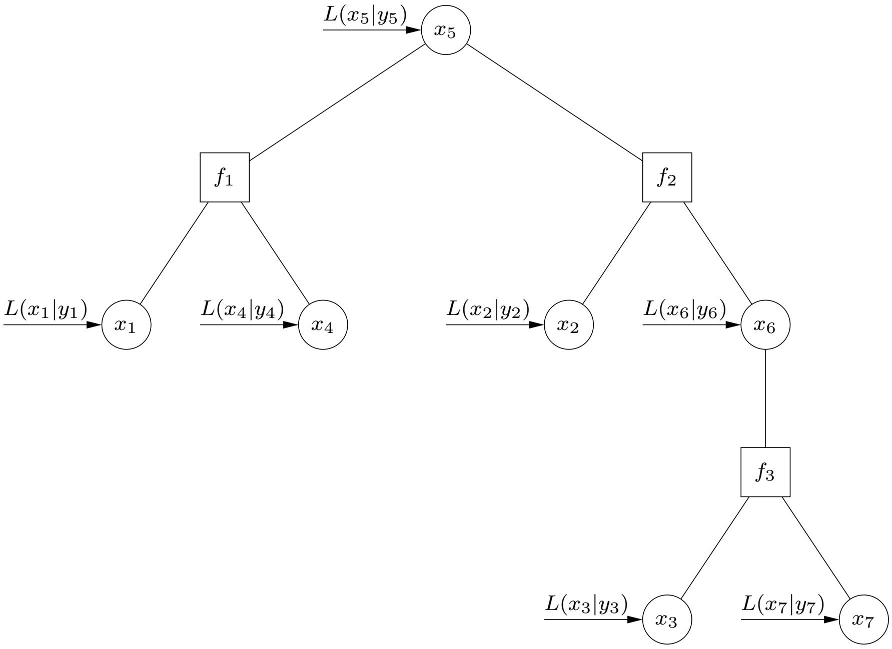

Figure 2.6: Rooted tree.

be drawn as a tree. The output of a variable node can be written as

$$\begin{aligned} L(x|\mathcal{Y}_1 \cup \ldots \cup \mathcal{Y}_k) &= \log \frac{P(x=0|\mathcal{Y}_1 \cup \ldots \cup \mathcal{Y}_k)}{P(x=1|\mathcal{Y}_1 \cup \ldots \cup \mathcal{Y}_k)} = \\ &= \log \frac{P(\mathcal{Y}_1 \cup \ldots \cup \mathcal{Y}_k|x=0)}{P(\mathcal{Y}_1 \cup \ldots \cup \mathcal{Y}_k|x=1)} = \\ &= \log \prod_{l=1}^{k} \frac{P(\mathcal{Y}_l|x=0)}{P(\mathcal{Y}_l|x=1)} = \\ &= \sum_{l=1}^{k} L(x|\mathcal{Y}_l), \end{aligned} \tag{2.25}$$

where in the first step we used the assumption that $P(x=0) = P(x=1)$ and in the second step the fact that the sets $\mathcal{Y}_l$ are disjoint and therefore, statistically independent given x. The variable node sums all incoming messages to calculate a new LLR value for its associated digit of the codeword. The computation at a variable node is illustrated in figure 2.8.

Computation at a Check Node

A check node receives values of the form $L(x_l|\mathcal{Y}_l)$ and has to calculate the LLR value of the variable x that is associated with its parent node, as shown in figure 2.9. As in the case of a variable node, any two sets $\mathcal{Y}_l$ and $\mathcal{Y}_i$ are disjoint.

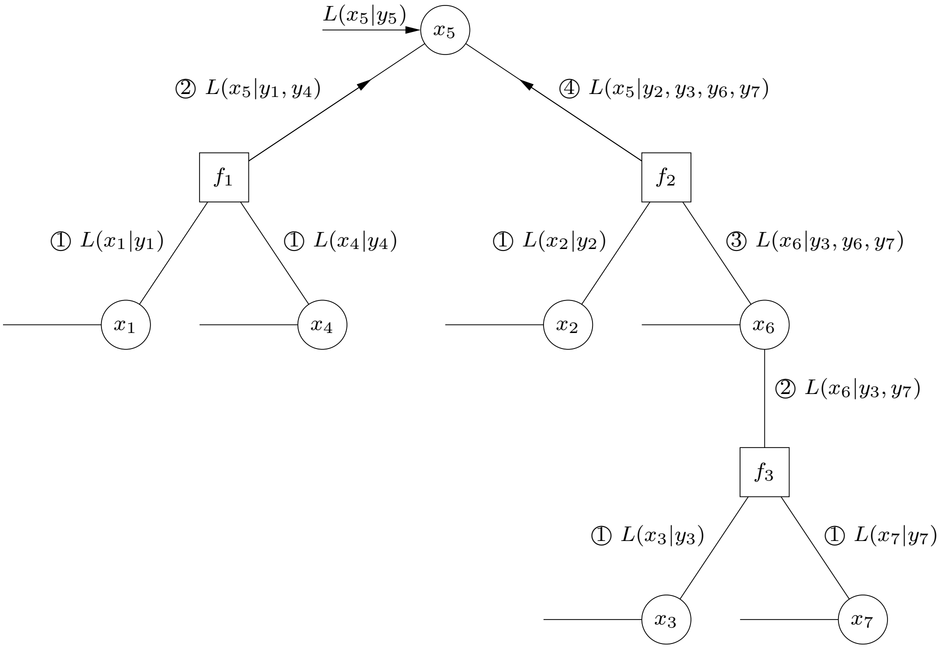

Figure 2.7: Messages after steps 1 to 4.

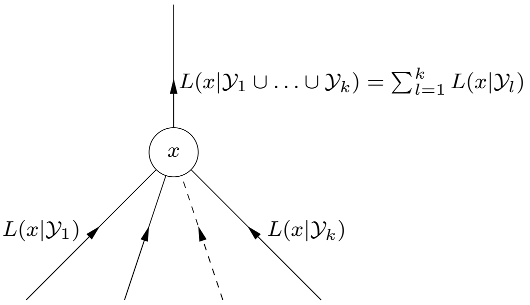
$L(x|\mathcal{Y}_1 \cup \ldots \cup \mathcal{Y}_k) = \sum_{l=1}^{k} L(x|\mathcal{Y}_l)$
x
$L(x|\mathcal{Y}_1)$
$L(x|\mathcal{Y}_k)$

Figure 2.8: Computation at a variable node.

The check node uses the fact that the modulo sum of all connected variable nodes has to be zero. To derive the calculation of a check node, we need the following lemma (taken from [11]).

Lemma 1. *Consider a vector* $\boldsymbol{x} = x_1, \ldots, x_k$ *of independent binary digits, where* $\boldsymbol{P} = P_1, \ldots, P_k$ *denotes the probability of the corresponding digits being a one. The probability that an even number of digits are equal to one can be written as*

$$\frac{1 + \prod_{l=1}^{k}(1 - 2P_l)}{2}. \tag{2.26}$$

Let S denote the event that the parity-check equation is fulfilled and let $\boldsymbol{x} = \{x_1, \ldots, x_k, x\}$ be the digits that are involved in this parity-check equation. Given

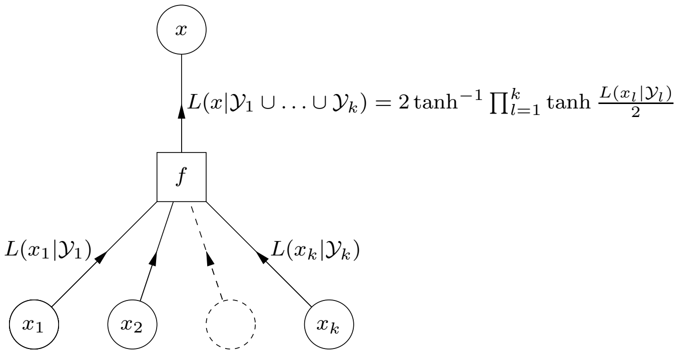
x
$L(x|\mathcal{Y}_1 \cup \ldots \cup \mathcal{Y}_k) = 2\tanh^{-1}\prod_{l=1}^{k}\tanh\frac{L(x_l|\mathcal{Y}_l)}{2}$
f
$L(x_1|\mathcal{Y}_1)$
$L(x_k|\mathcal{Y}_k)$
x_1
x_2
x_k

Figure 2.9: Computation at a check node.

that the digit x is zero, the number of ones in the remaining vector has to be even to fulfill the parity-check equation. If x is one, the number of ones in the remaining vector has to be uneven. Using lemma 1, we can write the ratio of these two cases as

$$\frac{P(x=0|S,\boldsymbol{P})}{P(x=1|S,\boldsymbol{P})} = \frac{P(S|x=0,\boldsymbol{P})}{P(S|x=1,\boldsymbol{P})} = \frac{1+\prod_{l=1}^{k}(1-2P_l)}{1-\prod_{l=1}^{k}(1-2P_l)}, \tag{2.27}$$

where $\boldsymbol{P}$ denotes the vector $P_1,\ldots,P_k$. After transforming this relation to the LLR domain, we get the following result for the computation at a check node

$$\tanh\frac{L(x|\mathcal{Y}_1 \cup \ldots \cup \mathcal{Y}_k)}{2} = \prod_{l=1}^{k}\tanh\frac{L(x_l|\mathcal{Y}_l)}{2}. \tag{2.28}$$

2.3.3 The Sum-Product Algorithm

The previous section showed how to calculate the MAP decoding rule for one digit of the codeword by passing messages on the factor graph that is associated with the parity-check matrix of the code. If we want to compute the MAP decoding rule for every digit of the codeword, we can apply this algorithm for every digit. However, this is very inefficient because many local functions are calculated twice or even more. Before introducing the algorithm that avoids this multiple computation, we will modify the message passing algorithm of the previous section.

In the example, every node was able to send out a message over an edge, when it has received messages from all other edges. Therefore, the total number of messages transmitted in the graph is equal to the number of edges. The resulting LLR value is obtained when the root node has received all its messages. We can modify the algorithm in the following way. The messages from check nodes to variable nodes are initialized with LLR values of zero and every node sends a messages at every step. The algorithm terminates if no message changes its value in the next iteration. It is obvious that the number of steps is equal to the original algorithm and that the number of messages transmitted in the graph is the number of edges times the number of iterations.

After this modification we can formulate the sum-product algorithm that allows the parallel computation of all marginal probability mass functions. Instead of redrawing the factor graph as a tree, we simple initialize the messages from the check nodes to the variable nodes with zero. In the first part of the iteration, the variable nodes send their messages over an edge based on the inputs of all other edges to the check nodes. The second part of the iteration is the computation at the check nodes. Every check node sends out a message over an edge by using all messages received from the other edges. Again, the algorithm terminates if no message changes and the desired LLR values are obtained by combining (summing up) all the incoming messages at every variable node. The number of messages transmitted is again equal to the number of edges times the number of iterations, but the algorithm calculates all the marginal functions in parallel.

Every variable node has to send its associated value received from the channel to every other variable node. The *diameter* of a graph is defined as the longest path between two nodes. At every iteration, a message is sent over two edges (from variable nodes to check nodes and back to the variable nodes). Therefore, the required number of iterations is half the diameter of the graph.

2.3.4 Graphs with Cycles

In the derivation of the sum-product algorithm, we made the assumption that the graph can be drawn as a tree, i.e. it is free of cycles. This assumption is used to calculate the output of a variable node (equation 2.25). This calculation is valid if and only if the sets Y_l are disjoined.

In [7], the authors showed that the minimum distance d of a code, that can be represented by a cycle-free graph, is bounded by

$$d \leq \left\lfloor \frac{n}{k+1} \right\rfloor + \left\lfloor \frac{n+1}{k+1} \right\rfloor . \tag{2.29}$$

This bound reduces to $d \leq 2$ for $R = \frac{k}{n} \geq 0.5$.

Given this result, it seems that the sum-product algorithm is not very useful, because we can apply this algorithm only to very poor codes. However, in practice we can apply the sum-product algorithm to codes with cycles, but the results will not be equal to the results of the MAP decoder and therefore be suboptimal. Cycles in the graph correspond to a feedback. The effect of this feedback is large if the length of the cycles is small. The difficult task is to construct codes with a large minimum distance and only large cycles. One class of codes that can fulfill these requirements for long block lengths are low-density parity-check codes that are treated in the following chapter.

Chapter 3

Low-Density Parity-Check Codes

In 1962 Robert G. Gallager introduced low-density parity-check (LDPC) codes [10]. A more detailed description can be found in his dissertation [11]. LDPC codes are binary linear block codes that can be described by a sparse parity-check matrix, i.e. the density of ones in the matrix is very low—that is the reason why they are called *low-density.* The advantage of LDPC codes is that they allow the application of the sum-product algorithm introduced in the previous chapter because the probability of cycles in the factor graph decreases with increasing block length (The sum-product algorithm assumes a cycle free factor graph.). The fact that the computational complexity of this algorithm grows linearly with respect to the block length allows the application of long block codes that are able to achieve transmission rates close to channel capacity.

For traditional codes like convolutional codes, the length of the code (or the constraint length) is limited by the decoding complexity, because the computational complexity of the decoding process increases exponentially with the block length. The block length of LDPC codes is not limited by the decoding complexity but it is limited by the decoding *delay* that is tolerable for the application.

LDPC codes were rediscovered following the invention of turbo codes by Berrou et al. in 1993 [3]. In a certain sense, turbo codes can be described as a special case of LDPC codes with semi-infinite parity-check matrices [6].

3.1 Definitions

Consider binary linear block codes with block length n and rate R. The number of information bits per codeword is denoted by k. LDPC codes are represented by a parity-check matrix $\boldsymbol{H}$ with dimension $m \times n$, where $m = n - k$ denotes the number of parity-check equations. The rate of the code can be written as

$$R = 1 - \frac{\operatorname{rank}(\boldsymbol{H})}{n}. \tag{3.1}$$

The *design rate* of the code is written as

$$R_d = 1 - \frac{m}{n}. \tag{3.2}$$

If the matrix has full rank, i.e. no linearly dependent rows, rate and design rate are equal. If the matrix contains linearly dependent rows, the rank of the matrix is smaller than m and the rate of the code is higher than the design rate.

The rows and columns of a parity-check matrix are denoted by the vectors r_j ($j = 1, \ldots, m$) and c_i ($i = 1, \ldots, n$) respectively. The row weight and column weight is defined as the weight of the associated row and column vector. The weight of the rows and the columns correspond to the degrees of check nodes and variable nodes in the associated factor graph. Therefore, the terms $d_c(j)$ and $d_v(i)$ are used for the degree of the check nodes and variable nodes (weight of the rows and columns) respectively. For LDPC codes, the row and column weights are *small* and *independent* of the block length. These properties lead to a sparse parity-check matrix if the block length is large enough. A typical value for the column weight is 3, i.e. every digit of the codeword is involved in 3 parity-check equations.

The total number e of ones in the parity-check matrix can be written as

$$e = \sum_{i=1}^{n} d_v(i) = \sum_{j=1}^{m} d_c(j). \tag{3.3}$$

If we assume a code where the row and column weight is constant, i.e. $d_c(j) = d_c$ and $d_v(i) = d_v$, this simplifies to

$$e = n \cdot d_v = m \cdot d_c \tag{3.4}$$

and the design rate of this code can be written as

$$R_d = 1 - \frac{m}{n} = 1 - \frac{d_v}{d_c}. \tag{3.5}$$

3.2 Decoding

LDPC codes are decoded by applying the sum-product algorithm to the associated factor graph. As mentioned in the previous section, the complexity of the sum-product algorithm grows linearly with the number of ones in the parity-check matrix (the number of edges in the associated factor graph). As noted above, the number of ones in the parity-check matrix of a LDPC code grows linearly with the block length (equation 3.4). Therefore, the computational complexity of the sum-product algorithm for decoding LDPC codes grows linearly with the block length.

3.2.1 Cyclefree Graphs and Infinite Block Length

For a cycle-free factor graph, the iterative sum-product algorithm is equivalent to the MAP decoder and is therefore optimal. If the graph contains cycles, the algorithm will be suboptimal. The number and the length of the cycles will determine the gap between the optimal MAP decoder and the suboptimal iterative decoder. Assuming a random construction method, the probability of a cycle is related to the density (number of ones in comparison to number of elements in the matrix) of the parity-check matrix. The density of ones in the matrix for the asymptotic case of infinite block length can be written as

$$\lim_{n\to\infty} \frac{n \cdot d_c}{n \cdot m} = \lim_{n\to\infty} \frac{n \cdot d_c}{n \cdot n \cdot (1-R)} = \lim_{n\to\infty} \frac{1}{n} \cdot \frac{d_c}{1-R} = 0. \tag{3.6}$$

When the block length tends to infinity, the density of ones in the matrix tends to zero and the probability of a cycle tends to zero as well. Therefore, infinite block length leads to optimal performance of the iterative decoding algorithm.

3.3 Encoding

The process of encoding is usually easier than the process of decoding, because the encoding process has to deal with binary values only. However, the block lengths used for LDPC codes are large (larger than 10^3) and therefore, we require the encoding algorithm to also have a computational complexity that is linear with respect to the block length. There are basically two possibilities of encoding a block code and they will be presented in the next two sections.

3.3.1 Encoding with the Generator matrix

The encoding process can be formulated as

$$\boldsymbol{x} = \boldsymbol{u} \cdot \boldsymbol{G}, \tag{3.7}$$

where $\boldsymbol{G}$ denotes the generator matrix of the code with dimension $k \times n$. The easiest way of calculating the generator matrix for a given parity-check matrix is to convert the parity-check matrix to a systematic form (using Gauss-Jordan elimination) and to use the relation

$$H = [\boldsymbol{P}\ \boldsymbol{I}_{n-k}] \Longleftrightarrow G = \left[\boldsymbol{I}_k\ \boldsymbol{P}^T\right]. \tag{3.8}$$

The matrix $\boldsymbol{I}$ is the identity matrix of dimension $(n-k)$ and k respectively and the matrix $\boldsymbol{P}$ is of dimension $k \times (n-k)$.

The encoding process requires $k \cdot (n-k) = R \cdot (1-R) \cdot n^2$ operations and has a computational complexity that is *quadratic* in the block length. The sparseness of the parity-check matrix can not be exploited because after the conversion to systematic form, the sparseness of the matrix is lost in general and therefore the generator matrix is not sparse either. Therefore, this method is not suited for encoding LDPC codes.

3.3.2 Encoding with the Parity-Check Matrix

The relation

$$\boldsymbol{x} \cdot \boldsymbol{H}^T = \boldsymbol{0}$$

provides another possibility for encoding. This relation is a system of linear equations. If we assume a systematic encoder, the number of unknown variables (parity digits) is $m = n - k$ and the number of equations is equal to m [1]. Therefore, the codeword can be calculated by solving this system of equations. The easiest way is to convert the matrix to triangular form (using Gauss-Jordan elimination) and to calculate the unknown variables recursively. This is shown in figure 3.1 where p_i and u_i denotes parity and information digits respectively. The left lower part of the matrix contains only zeros.

As in the previous case, the sparseness of the triangular matrix is lost in general. The parity digit p_i is a linear combination of $u_1, \ldots, u_k$ and $p_1, \ldots, p_{i-1}$. If we assume that every element of the parity-check matrix can be 1 with probability 0.5, these linear combinations contain $\frac{k}{2} + \frac{m}{4}$ elements on average. Therefore, the computation of m parity digits is of the order $\mathcal{O}(n^2)$.

If we restrict the parity-check matrix to triangular form at construction time, we can exploit the sparseness of this matrix. Every row of the matrix contains d_c ones. Therefore, the computation of m parity digits is of the order $\mathcal{O}(m \cdot d_c) = \mathcal{O}(n)$. This method is the preferred method for the encoding process.

---

[1] Assuming that the matrix $\boldsymbol{H}$ has full rank.

$$
\boldsymbol{H} = \begin{bmatrix}
1 & 0 & 0 & 1 & \cdots & 0 & 1 & 0 & \cdots & 0 & 0 & 1 \\
 & 1 & 1 & 0 & \cdots & 0 & 0 & 1 & \cdots & 0 & 1 & 0 \\
 & & 1 & 0 & \cdots & 1 & 1 & 0 & \cdots & 0 & 0 & 0 \\
 & & & 1 & \cdots & 0 & 0 & 0 & \cdots & 1 & 0 & 0 \\
 & & & & \ddots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\
 & & & & & 1 & 0 & 1 & \cdots & 1 & 1 & 0
\end{bmatrix}
$$

$$
\boldsymbol{x} = \begin{bmatrix} p_m & \cdots & \cdots & \cdots & \cdots & p_1 & u_k & \cdots & \cdots & \cdots & \cdots & u_1 \end{bmatrix}
$$

Figure 3.1: Encoding with parity-check matrix.

3.4 Regular and Irregular Codes

If the degrees of the variable and the check nodes are constant, i.e. $d_c(j) = d_c$ and $d_v(i) = d_v$, the resulting LDPC code is called *regular*. If we allow nodes with higher or lower degree, the resulting code is called *irregular*. Irregular codes have a better asymptotic performance and can practically reach channel capacity as shown in [14].

The construction of irregular codes is motivated by the following ideas:

- From the point of view of a variable node, it is best if its degree is high, because every edge delivers useful extrinsic information to the variable node. From the point of view of a check node, the probability that the check function is fulfilled decreases with increasing degree. Therefore, for a check node, it is best if the degree is low. These two types of nodes have to agree to a number of edges that is determined by equation 3.3. By allowing a mixture of higher and lower degrees, these requirements can be fulfilled in a more flexible manner.

- The variable nodes with higher degree receive many messages carrying extrinsic information after a few iterations and converge faster to their convergence point. These nodes, sometimes called *elite nodes*, can provide reliable information in further iterations and help the lower degree nodes to converge.

- With tools like *Extrinsic Information Transfer (EXIT) charts*, codes can be designed to approach channel capacity. This design is only possible if the column weight distribution is irregular. Design of LDPC codes is not considered in this work. Further details can be found in [1] and in [14] where *density evolution* is used for optimization of the code.

Regular codes are specified by their variable and check node degrees. For irregular codes we have to specify *degree distributions*. A degree distribution can be written as a polynomial, e.g.

$$\gamma(x) \overset{\text{def}}{=} \sum_{i\geq 1} \gamma_i x^{i-1} \tag{3.9}$$

satisfying the condition $\gamma(1) = 1$ and $\gamma_i \geq 0$. The coefficients γ_i represent the fraction of edges emanating from a node of degree i.

If we denote the maximum degree of a variable node as $d_{v,max}$ and the maximum degree of a check node as $d_{c,max}$ we can write the degree distribution polynomial $\lambda(x)$ for the variable nodes and $\rho(x)$ for the check nodes as

$$\lambda(x) \overset{\text{def}}{=} \sum_{i=1}^{d_{v,max}} \lambda_i x^{i-1} \tag{3.10}$$

$$\rho(x) \overset{\text{def}}{=} \sum_{i=2}^{d_{c,max}} \rho_i x^{i-1}, \tag{3.11}$$

where the summation for the check node distribution starts at $i = 2$ because check nodes with degree one would correspond to a parity-check equation with only one digit and are therefore not used.

Note that the coefficients λ_i and ρ_i of the degree distributions do not represent the fraction of nodes of degree i, but the fraction of edges emanating from a node of degree i. The fraction of variable and check nodes of degree i ($V(i)$ and $C(i)$ respectively) can be calculated as

$$V(i) = \frac{\frac{\lambda_i}{i}}{\sum_{j\geq 1} \frac{\lambda_j}{j}} \tag{3.12}$$

$$C(i) = \frac{\frac{\rho_i}{i}}{\sum_{j\geq 2} \frac{\rho_j}{j}} \tag{3.13}$$

If we assume constant check node degree, the code can be interpreted as a combination of codes with different rates. Variable nodes with lower degree correspond to codes with higher rate and variable nodes with higher degree correspond to codes with lower rate (see equation 3.5). Lower rate implies higher error protection. Therefore, the systematic part of the codeword should be mapped to the better protected nodes, i.e. the nodes with a higher degree.

3.5 Basic Construction of LDPC Codes

This section is intended to provide an overview of the construction methods for LDPC codes. After introducing the methods, simulation results for the bit error rate when using binary phase shift keying (BPSK) over a channel with additive white Gaussian noise (AWGN) are presented. The simulations will use one code with rate 0.5 and one high rate code with rate about 0.9. The decoder uses the sum-product algorithm with a maximum number of 100 iterations.

3.5.1 Random Codes

One way of constructing LDPC codes is to use a random construction method. The ones in the parity-check matrix are placed randomly with respect to the constraints, e.g. constant number of ones per row and per column for a regular code. By specifying the row and column weights, we do not specify a specific code but a whole *ensemble* of codes. It is possible to provide analytical expressions (bounds for the bit error rate, minimum distance, ...) for these ensembles, but in practical systems and in simulations, a specific instance of the ensemble is implemented. The *concentration theorem* [15] states that quantities such as the bit error probability concentrate around the ensemble average if the block length is large enough. In other words, the performance of instances of the ensemble is nearly equal to the average performance over the ensemble if the block length is sufficiently large. However, for the short block lengths ($n < 1000$) of interest in this work significant differences between instances of the ensemble can be observed.

Random Edge Interleaver

The easiest way of constructing random LDPC codes is to use an *edge interleaver.* The basic idea is illustrated in figure 3.2. The total number of edges e is given by equation 3.3. By using a random interleaver of size e, the bipartite graph and therefore the parity-check matrix of the associated LDPC code can be constructed.

The following example illustrates this construction method:

Example 2 (Construction with Random Edge Interleaver). *We wish to construct a regular LDPC code with block length* 8*, rate* 0.5 *and variable node degree* 2 *using a random edge interleaver. The number of edges in the factor graph is* $8 \cdot 2 = 16$*. Therefore, a random edge interleaver is needed that rearranges the numbers from* 1 *to* 16*. The variable nodes are connected to one side of the interleaver and the check nodes to the other side, as shown in figure 3.3. The parity-check matrix is constructed by placing a one in the matrix for every two nodes that are connected through the interleaver. For this example, the parity-*

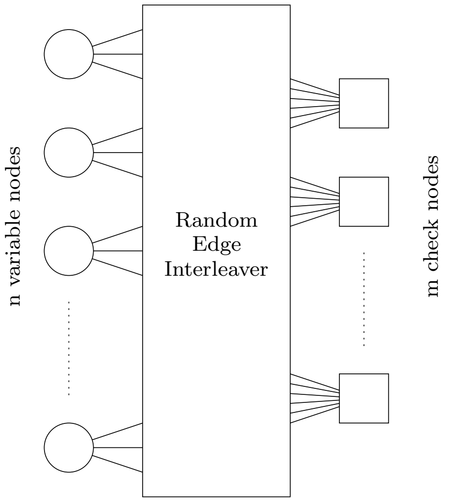

Figure 3.2: Construction with edge interleaver.

check matrix can be written as

$$\boldsymbol{H} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & \textcircled{\mathit{1}} & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \tag{3.14}$$

where the highlighted 1 *corresponds to the connection displayed in figure 3.3.*

This construction method does not ensure that there are no parallel connections between two nodes. Therefore, the number of ones in the parity-check matrix could be smaller than desired. However, for increasing block lengths, the probability of parallel connections decreases.

The disadvantage of this construction method is that additional constraints, like avoiding short cycles, can not be taken in account.

This method can be used for the construction of regular and irregular codes.

Constructing the Matrix by Adding Columns

This method starts with an empty parity-check matrix. A column is constructed by randomly inserting ones into it until the desired column weight is attained. The random number generator used is uniformly distributed over the rows $(1, \ldots, m)$. This procedure is repeated for every column until the complete parity-check matrix is constructed. This method leads to the desired column weights, but the row weights can not be controlled. For increasing block lengths, the row weights tend to become more and more equal.

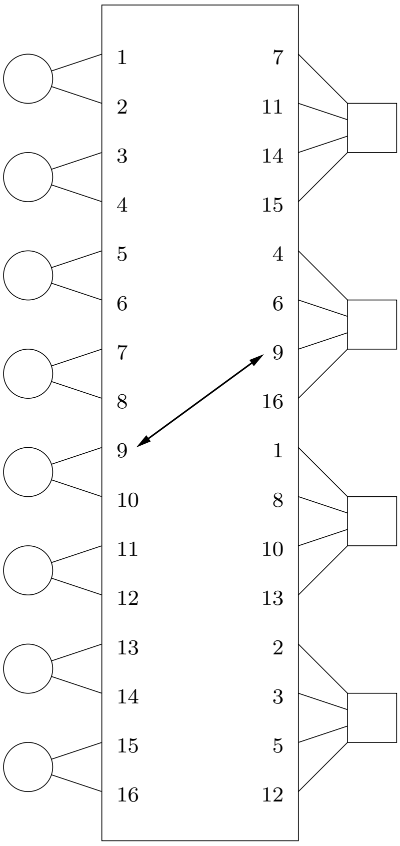

Figure 3.3: Example for construction with edge interleaver.

If we want to force the matrix to have a specific row weight for small block lengths or if we want to construct codes with irregular row weights, the construction method can be improved as follows: given the block length and the column weights, the number of ones e can be calculated as shown in equation 3.3. We construct a vector $\boldsymbol{u}$ of size e, that acts as the *supply.* The elements of $\boldsymbol{u}$ contain the row number where the corresponding one can be placed. For example, if we want to put 5 ones in the first row, $\boldsymbol{u}$ contains a one (corresponding to the first row) 5 times. Instead of selecting a row with the random generator directly, we select an element u of $\boldsymbol{u}$ randomly and put a one in the row number u (if there is already a one, we select another element from the supply). After placing the one in the parity-check matrix, we delete the selected element u from the vector $\boldsymbol{u}$ resulting in a new supply vector $\boldsymbol{u}$ of length $e-1$. The basic principle of this construction method is shown in figure 3.4. This method guarantees the desired row and column weights. For very short block lengths, it may not be possible to construct the last columns with the desired weights, but this effect can be neglected.

This method is illustrated by the following example:

Example 3 (Construction by adding columns). *We wish to construct a regular LDPC code with block length* 8*, rate* 0.5 *and variable node degree* 2 *with the proposed method. The number of edges in the factor graph is* $8 \cdot 2 = 16$*. These edges have to be distributed uniformly over the rows, i.e. every row contains* 4 *ones. The vector* $\boldsymbol{u}$ *contains the row numbers where the ones have to be placed*

$$\boldsymbol{u} = \begin{bmatrix} 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 4 & 4 & 4 & 4 \end{bmatrix}. \tag{3.15}$$

We start with the first column and pick an element from $\boldsymbol{u}$ *at random, for example the* 10^{th} *element. The value of* $\boldsymbol{u}_{10}$ *is* 3 *and therefore, we place a one in our current column (column number* 1*) and in row number* 3*. We now delete the element* $\boldsymbol{u}_{10}$*.* $\boldsymbol{u}$ *now contains elements for the remaining* 15 *ones to be placed*

$$\boldsymbol{u} = \begin{bmatrix} 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 4 & 4 & 4 & 4 \end{bmatrix}. \tag{3.16}$$

The column weight of the current column is equal to 1 *and therefore, we have to place another one in this column to reach the desired column weight of* 2*. We randomly pick element number* 13 *from* $\boldsymbol{u}$ *and place a one in column number* 1 *and row number* 4 *(*$\boldsymbol{u}_{13} = 4$*). After deleting the element from the vector* $\boldsymbol{u}$*, we reach the desired column weight of the first column and continue with the second column. This procedure is repeated until all columns are constructed and the vector* $\boldsymbol{u}$ *is empty.*

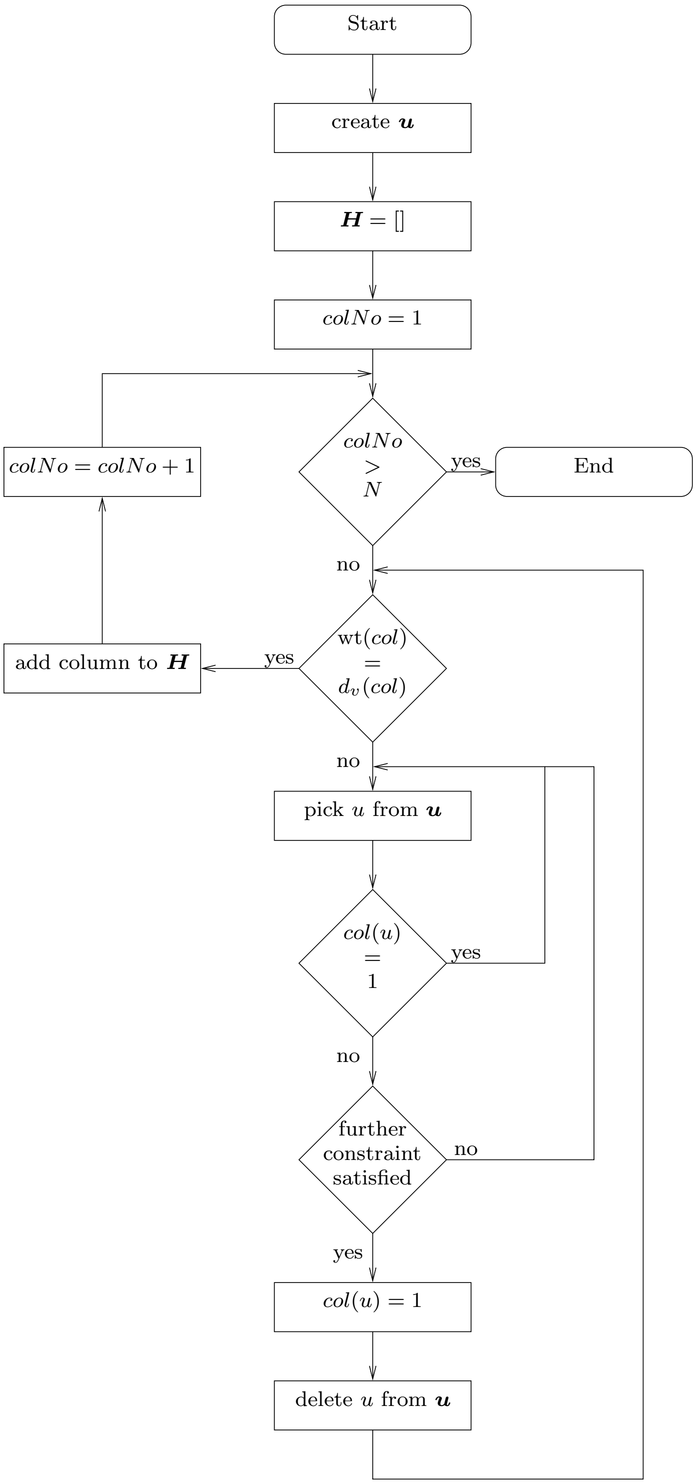
Start
create $\boldsymbol{u}$
$\boldsymbol{H} = []$
$colNo = 1$
$colNo = colNo + 1$
$colNo > N$
yes
End
no
add column to $\boldsymbol{H}$
yes
$\mathrm{wt}(col) = d_v(col)$
no
pick u from $\boldsymbol{u}$
$col(u) = 1$
yes
no
further constraint satisfied
no
yes
$col(u) = 1$
delete u from $\boldsymbol{u}$

Figure 3.4: Random construction of parity-check matrix.

The advantage of this method in comparison to the edge interleaver is that it can easily be improved by adding more constraints. For example, before placing a one in the matrix, we could check whether it creates a cycle[2] of length 4. In that case, we can simply select another element from the supply. This method was used in the rest of this work when randomly constructed codes were required.

This method can also be used to construct irregular codes. In principle, when constructing irregular codes, the construction could start either with the columns with the highest degree, or with the columns with the lowest degree. At the end of the construction, the algorithm may not be able to avoid short cycles, i.e. cycles of length 4. The influence of a short cycle on a node of lower degree is much higher than on a node with higher degree. For example, the worst case is a cycle of length 4 involving two variable nodes of degree 2. This case is shown in figure 3.5 and it is obvious that the minimum distance of this code is 2 (independent of the block length). Therefore, the construction must start with the lower degree nodes to avoid such cycles.

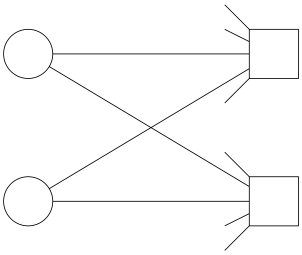

Figure 3.5: Cycle of length 4 involving variable nodes of degree 2.

Simulation Results

Figure 3.6 shows the performance of randomly constructed regular codes with rate 0.5. These codes have a column weight of 3 and a row weight of 6. The simulations show the performance in terms of bit error rate for block lengths ranging from 10^2 to 10^5. With increasing block length, the performance approaches the capacity of the binary input AWGN (BIAWGN) channel. Note that the required computation per digit does not increase with the block length, but the decoding delay increases with the block length. The results show that LPDC codes become attractive for block lengths larger than 10^3.

Figure 3.7 shows the same results for a code of rate 0.9. This code also has a column weight of 3, but a row weight of 30.

---

[2] A cycle of length 4 occurs if any two columns (or rows) have more than one overlapping ones and is therefore easy to detect.

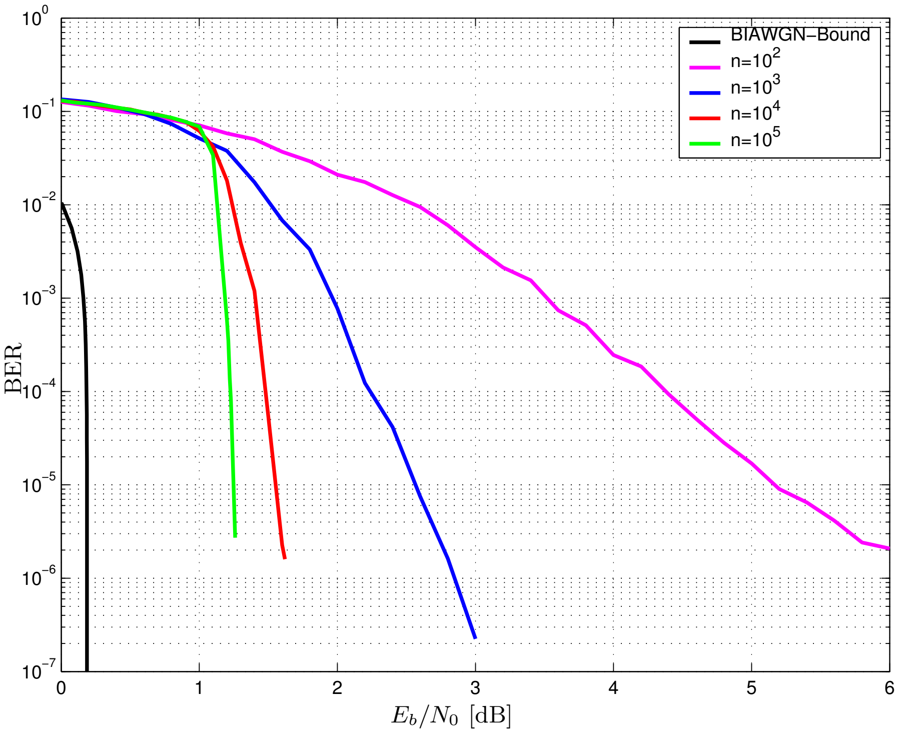

Figure 3.6: Regular random codes with rate 0.5.

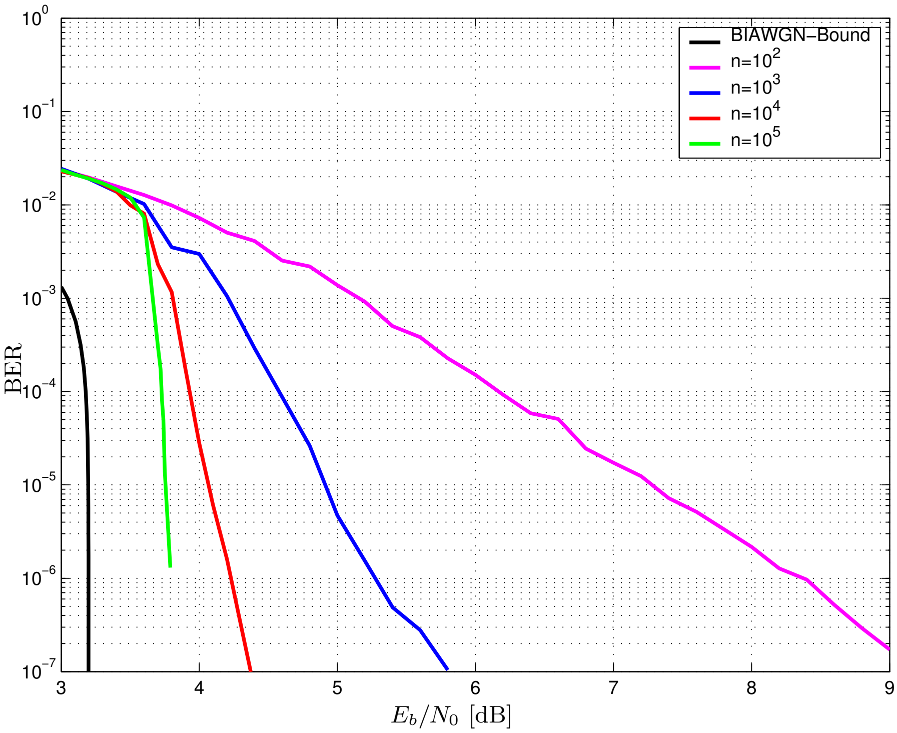

Figure 3.7: Regular random codes with rate 0.9.

For irregular codes, we need to specify degree distributions. These distributions are taken from [14]. The process of calculating these distributions used in [14] can be summarized as follows: first, a check node degree is fixed and the degrees of the variable nodes are optimized. Then the optimized variable node degrees are used and the check nodes are optimized. Therefore, the degrees of the check nodes are approximately constant in comparison to the degrees of the variable nodes. It should be noted that the block lengths must be larger than 10^6 to implement these optimized distributions. For shorter block lengths, the resulting distributions will differ from the optimized distributions due to rounding effects.

For the code with rate 0.5, maximum variable node degrees of 11, 20 and 50 are used for the block lengths 10^3, 10^4 and 10^5 respectively. The degree distributions for these codes are given in tables 3.1 to 3.3.

After converting these distributions from the edge perspective to the node perspective using equations 3.12 and 3.13, the *degree profiles* for the variable nodes as shown in figures 3.8 to 3.10 can be drawn. For the codes of rate 0.9, a maximum variable node degree of 11 with the degree distribution given in table 3.4 is used and the corresponding profile is shown in figure 3.11.

λ_2	0.27684
λ_3	0.28342
λ_9	0.43974
ρ_6	0.01568
ρ_7	0.85244
ρ_8	0.13188

Table 3.1: Degree distribution 9.

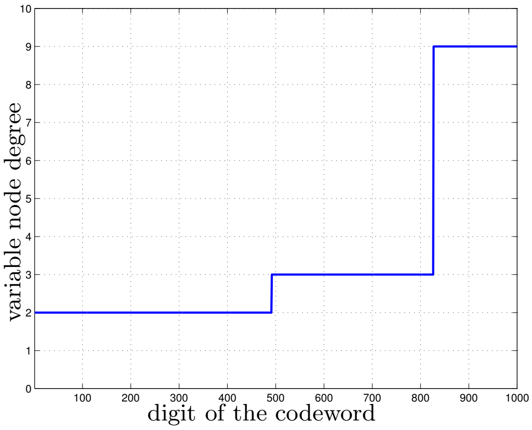

Figure 3.8: Degree profile with maximum degree of 9.

λ_2	0.21991
λ_3	0.23328
λ_4	0.02058
λ_6	0.08543
λ_7	0.06540
λ_8	0.04767
λ_9	0.01912
λ_{19}	0.08064
λ_{20}	0.22798
ρ_8	0.64854
ρ_9	0.34747
ρ_{10}	0.00399

Table 3.2: Degree distribution 20.

λ_2	0.17120
λ_3	0.21053
λ_4	0.00273
λ_7	0.00009
λ_8	0.15269
λ_9	0.09227
λ_{10}	0.02802
λ_{15}	0.01206
λ_{30}	0.07212
λ_{50}	0.25830
ρ_9	0.33620
ρ_{10}	0.08883
ρ_{11}	0.57497

Table 3.3: Degree distribution 50.

λ_2	0.23882
λ_3	0.29515
λ_4	0.03216
λ_{11}	0.43342
ρ_7	0.43011
ρ_8	0.56989

Table 3.4: Degree distribution 11.

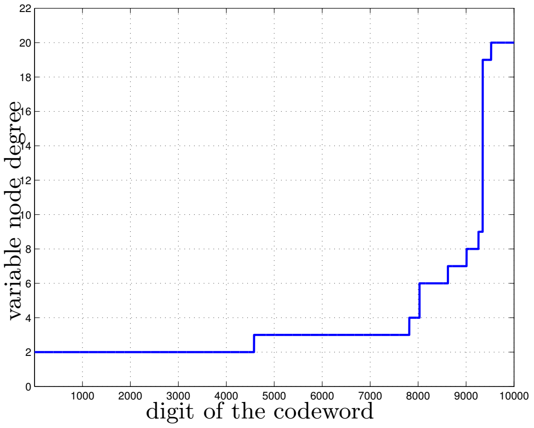

Figure 3.9: Degree profile with maximum degree of 20.

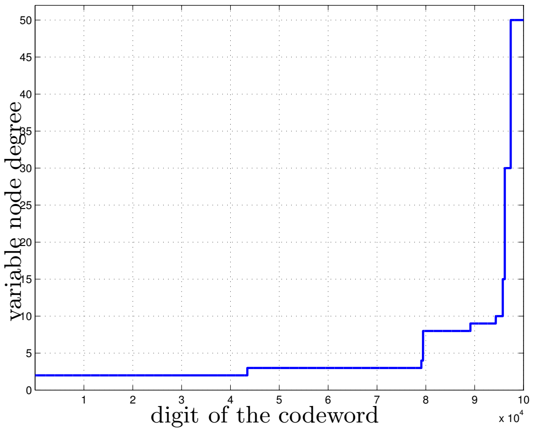

Figure 3.10: Degree profile with maximum degree of 50.

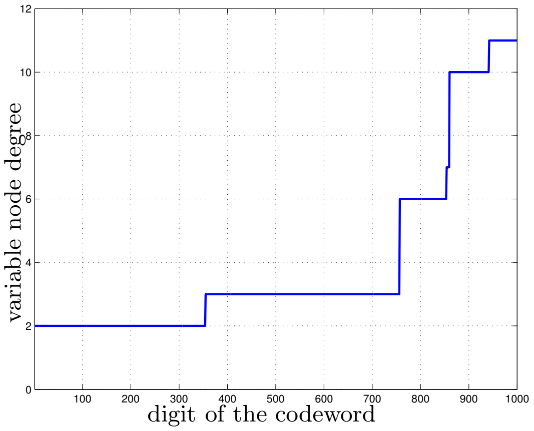

Figure 3.11: Degree profile with maximum degree of 11.

The simulation results in figure 3.12 and 3.13 show that the performance of irregular codes is better than the performance for regular codes. For comparison the figures also contain the results of the regular codes. The difference in required signal to noise ratio (SNR) can be up to 0.6dB in the case of a code with rate 0.5 and block length 10^5. However, although the computational complexity is linear in the block length, it is proportional to the number of messages passed in the graph and therefore increases with growing node degrees. Therefore, the computation time for irregular codes is higher than for regular codes.

In the simulations of regular codes, it was not possible to observe error floors at bit error rates (BER) larger than 10^{-7}. However, irregular codes have error floors at BER of approximately 10^{-5}. This indicates a small minimum distance of irregular codes in comparison to regular codes. Therefore, irregular codes should be combined with an outer code (for example a Reed-Solomon code) to remove the error floor.

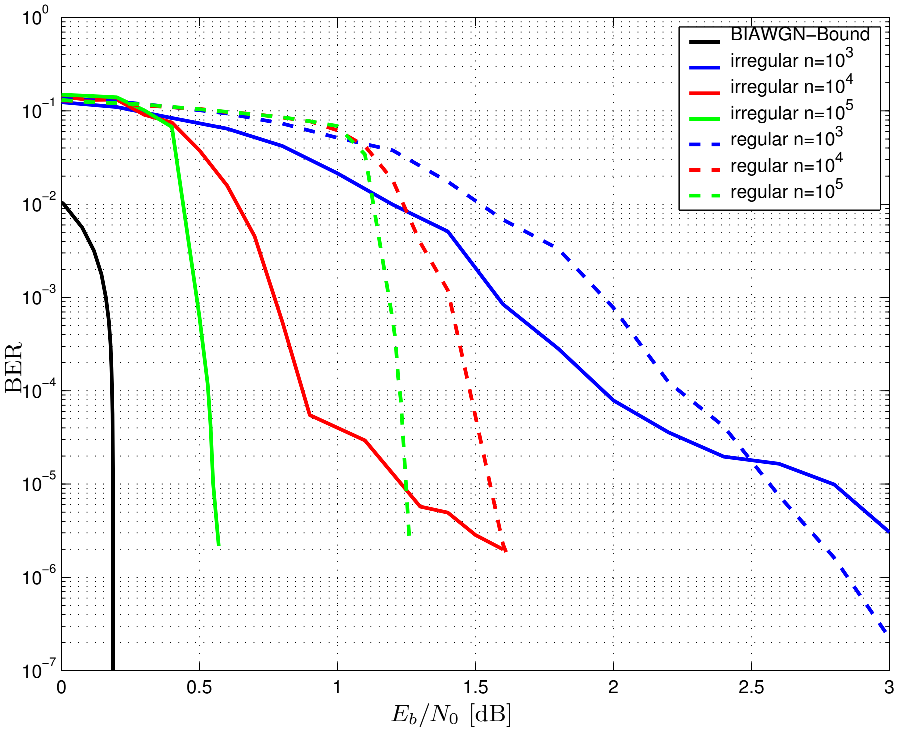

Figure 3.12: Irregular random codes with rate 0.5.

As mentioned in section 3.4, the higher degree nodes are more protected than the lower degree nodes. The following simulation uses the same irregular code with rate 0.5 and block length 10^3 used above. The bit error rate for every variable node degree (2, 3 and 9) is shown in figure 3.14. This simulation confirms the

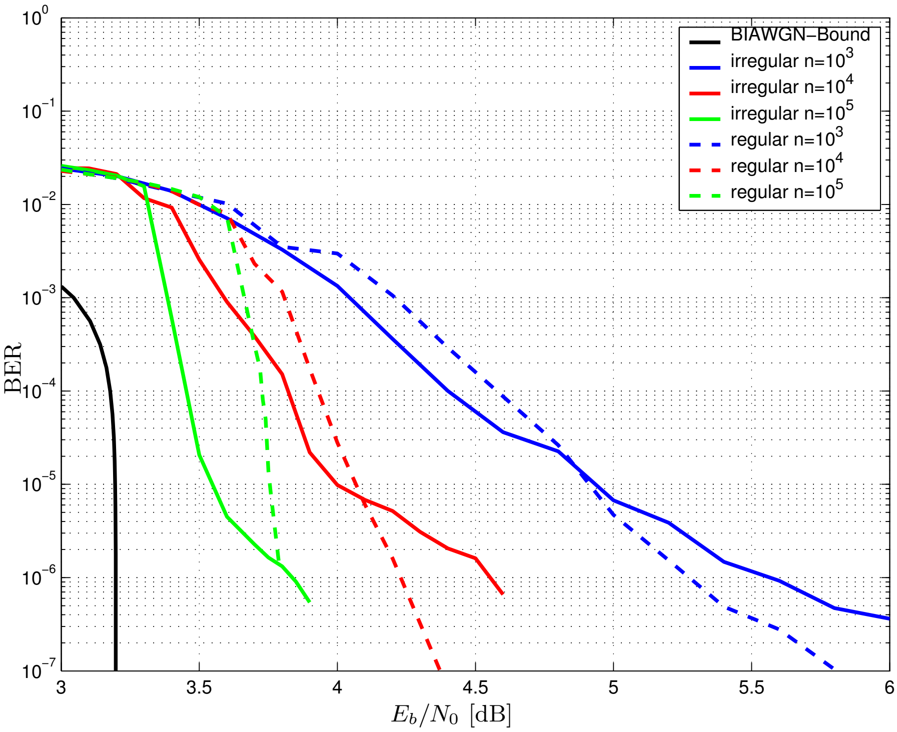

Figure 3.13: Irregular random codes with rate 0.9.

intuitive idea that irregular code provide unequal error protection. The difference in terms of required SNR is approximately 1dB between the variable nodes of degree 2 and degree 9 at a BER of 10^{-5}.

3.5.2 Deterministic Codes

A disadvantage of randomly constructed codes is that they can not be described in a simple manner. The only way of describing them is to provide the complete parity-check matrix, which could be a disadvantage for inclusion in a standard. Also, the encoder and the decoder have to store the complete matrix in memory which enlarges the memory requirements of the system. In contrast, codes constructed in a deterministic way can be described by a few parameters and the corresponding construction method. They are well suited for standardization and the encoder and the decoder can exploit the deterministic structure to avoid storing the complete matrix in memory.

Codes Based on Array Codes

This method has been proposed in [8] where a description of array codes can be found. The resulting construction method is based on building the parity-check

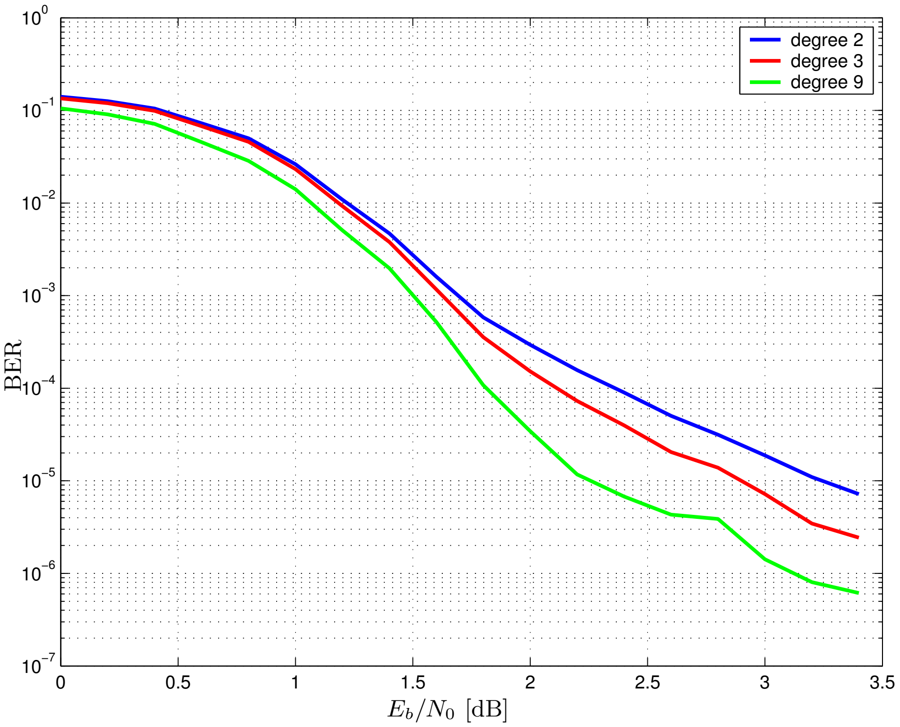

Figure 3.14: Bit error rates of code bits with different variable node degrees.

matrix by combining powers of a single cyclic shift matrix $\boldsymbol{\alpha}$ of dimension $p \times p$, p being a prime, in the following way:

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{I} & \boldsymbol{I} & \cdots & \boldsymbol{I} \\ \boldsymbol{I} & \boldsymbol{\alpha} & \boldsymbol{\alpha}^2 & \cdots & \boldsymbol{\alpha}^{p-1} \\ \boldsymbol{I} & \boldsymbol{\alpha}^2 & \boldsymbol{\alpha}^4 & \cdots & \boldsymbol{\alpha}^{2(p-1)} \\ \boldsymbol{I} & \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{I} & \boldsymbol{\alpha}^{j-1} & \boldsymbol{\alpha}^{2(j-1)} & \cdots & \boldsymbol{\alpha}^{(j-1)(p-1)} \end{bmatrix}, \qquad (3.17)$$

where the number j is smaller than p. The corresponding factor graph is free of cycles of length 4 (see [8]). The single cyclic shift matrix $\boldsymbol{\alpha}$ with for example $p = 5$ is

$$\boldsymbol{\alpha} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \text{ or } \boldsymbol{\alpha} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}. \qquad (3.18)$$

The parameters of the code are related in the following way:

$$n = p \cdot p \qquad (3.19)$$

$$m = p \cdot j \tag{3.20}$$

$$R = 1 - \frac{m}{n} = 1 - \frac{j}{p} = 1 - \frac{j}{\sqrt{n}} \tag{3.21}$$

From equation 3.21, it is clear that it is not possible to choose the rate, the block length and the column weight independently. If the rate is fixed, the column weight increases with the square root of the block length. This is in contradiction to the basic property of LDPC codes (the column weight is *independent* of the block length). In order to fulfill this property, the column weight has to be fixed, and the rate is an increasing function of the block length.

This construction method can be modified by shortening the code, i.e., reducing the number of horizontal repetitions, and using the following scheme:

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{I} & \boldsymbol{I} & \cdots & \boldsymbol{I} \\ \boldsymbol{I} & \boldsymbol{\alpha} & \boldsymbol{\alpha}^2 & \cdots & \boldsymbol{\alpha}^{k-1} \\ \boldsymbol{I} & \boldsymbol{\alpha}^2 & \boldsymbol{\alpha}^4 & \cdots & \boldsymbol{\alpha}^{2(k-1)} \\ \boldsymbol{I} & \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{I} & \boldsymbol{\alpha}^{j-1} & \boldsymbol{\alpha}^{2(j-1)} & \cdots & \boldsymbol{\alpha}^{(j-1)(k-1)} \end{bmatrix}, \tag{3.22}$$

where k and j are smaller than p. The parameters of this shortened code are:

$$n = p \cdot k \tag{3.23}$$

$$m = p \cdot j \tag{3.24}$$

$$R = 1 - \frac{m}{n} = 1 - \frac{j}{k} \tag{3.25}$$

This additional degree of freedom allows an independent choice of the rate, the block length and the column weight. However, if the original code is shortened by a lot, i.e., if k is much smaller than p, the performance of the resulting code deteriorates. This is the reason why this construction method is suited only for high rate codes at the block lengths of practical interest. Of course, this shortened matrix is also free of cycles of length 4.

Simulation Results

Figure 3.15 shows the performance of codes constructed with the method described without shortening. For comparison, regular randomly constructed codes of the same block length and rate are also included in the figures. The simulations show that the difference of the performance between deterministic and randomly constructed codes is small (the required SNR for a given BER differs only in fractions of dB). The parameters for the deterministic codes are given in table 3.5.

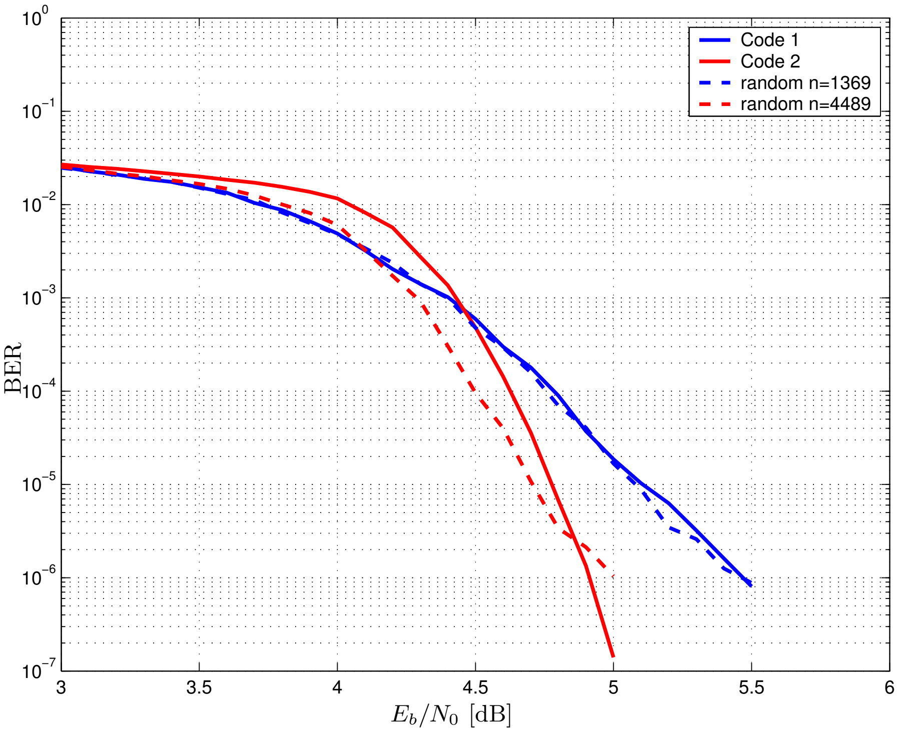

Figure 3.15: Regular deterministic codes with rate 0.9.

	Code 1	Code 2
p	37	67
j	3	5
k	37	67
n	1369	4489
R_d	0.9189	0.9254
R	0.9204	0.9263

Table 3.5: Parameters of the deterministic codes.

3.6 Extended Construction of LDPC

In the previous section, basic construction methods for LDPC codes were introduced. All these methods generate parity-check matrices with the desired column and row weights. In addition, the randomized construction method tried to avoid cycles of length 4 and the deterministic construction method avoids cycles of length 4 by definition. In this section, two additional construction methods will be introduced. The first method is based on maximizing the cycle lengths to get better performance of the iterative decoder. The second method creates parity-check matrices that are suited for *encoding* with linear complexity.

3.6.1 Construction based on Large Girth

In the derivation of the sum-product algorithm, we saw that this decoding algorithm is equivalent to the optimal MAP decoder if and only if the underlying graph is free of cycles. The gap between the iterative decoder and the MAP decoder depends on the number and on the lengths of the cycles in the graph. Therefore, the number of cycles should be made as small as possible and the length of the cycles should be as large as possible.

Large Girth at Construction Time

One way of constructing a parity-check matrix of a LDPC code is to build the matrix column by column and only place ones in the matrix where the influence on the girth is small. This algorithm—known as *bit-filling-algorithm*—was proposed by Campello. In [4], the authors showed simulation results for the constructed codes. They perform slightly better than the codes constructed in the previous section and there is no error floor for irregular codes.

Heuristic Search for Good Codes

The concentration of specific instances around the ensemble average described in [15] is weak for short block lengths. Therefore, it is possible to find codes that are significantly better than the ensemble average. We can make a heuristic search to find those codes by using a simple performance criterion [13]. To define this criterion we need the following definitions:

Definition 9 (Local Girth). *The local girth with respect to a node is the shortest cycle in which the node is involved. If the node is not involved in a cycle, the local girth is defined as* 0*.*

Definition 10 (Local Girth Average). *The local girth average is the local girth averaged over all variable nodes with local girth larger than* 0*.*

In [13], the authors showed simulation results for codes with rate $\frac{1}{3}$ and block lengths of 3480 and 1268. The selected codes with the largest local girth average performed significantly better than a randomly chosen code, especially for high signal to noise ratio.

We constructed regular codes with rate 0.5, block length 1000 and column weight 3 and calculated the local girth average. The histogram of the local girth average is shown in figure 3.16. The performance of the code with the largest and lowest local girth average is shown in figure 3.17. In contrast to [13], our measurements show no significant difference in the performance of these two codes.

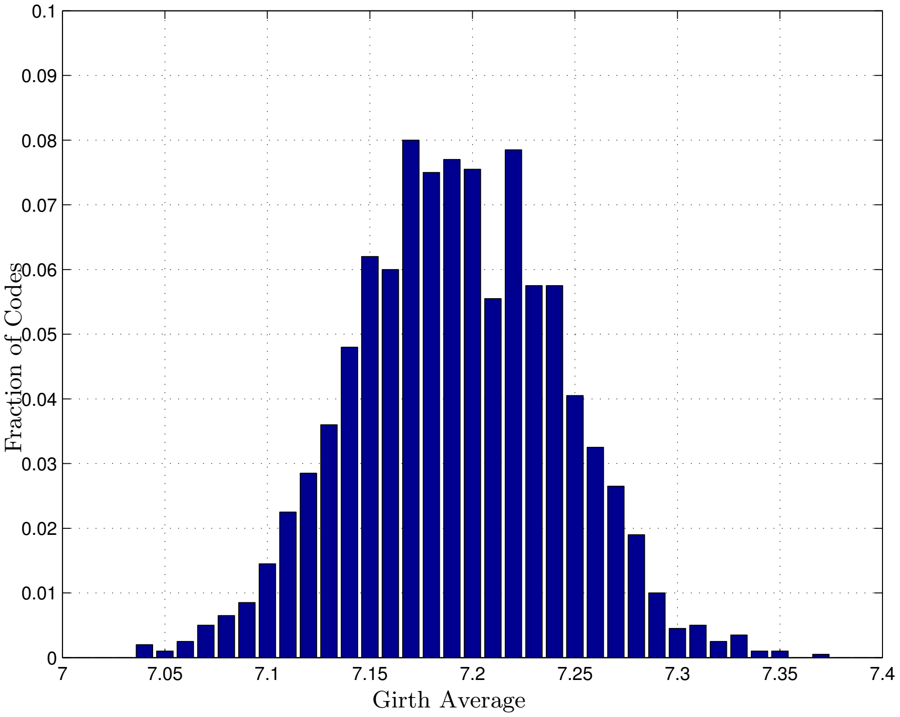

Figure 3.16: Girth average for block length 1000.

If the block length is increased, the variance of the local girth average decreases and the performance of different instances of the ensemble concentrates. Therefore, this heuristic search is only useful for short block lengths ($< 10^4$).

3.6.2 Triangular Parity-Check Matrix

As mentioned in section 3.3.2, a parity-check matrix that is forced to have triangular form by design allows encoding with linear complexity.

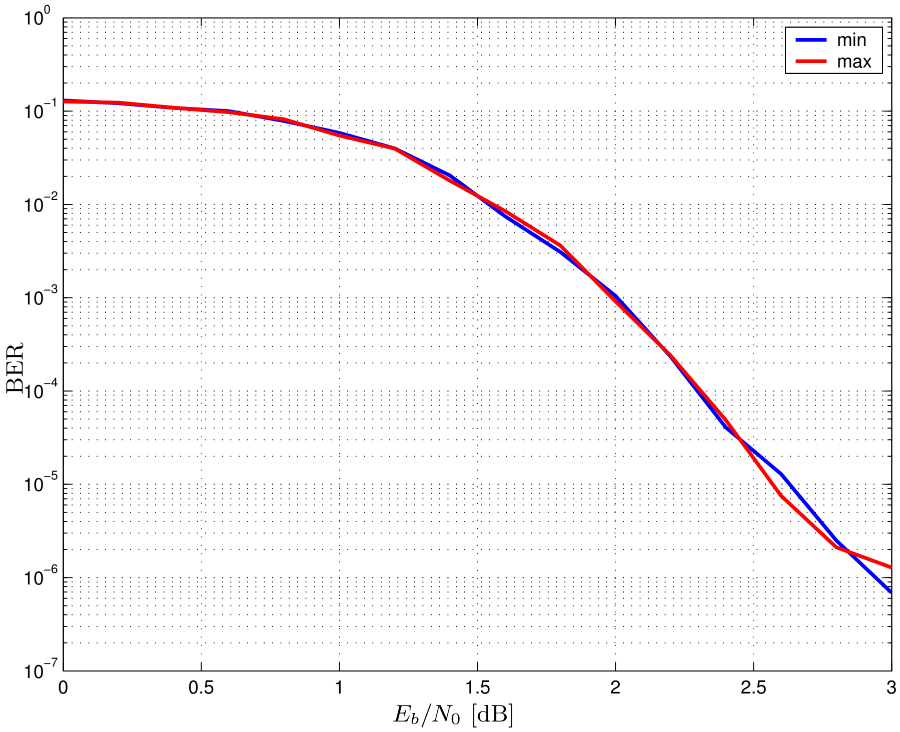

Figure 3.17: Comparison of codes with different girth average.

It is easy to construct random codes with a triangular low-density parity-check matrix. As mentioned in section 3.5.1, the randomized construction algorithm can be constrained to fulfill specific restrictions on the positions of the ones in the matrix. The restriction in this case is simply to avoid placing a one in the lower triangular part of the parity-check matrix. This restriction can be written as

$$\boldsymbol{H}_{ij} = 0 \text{ for } j < i. \tag{3.26}$$

One way of constructing deterministic codes with a triangular parity-check matrix is proposed in [5]. The parity-check matrix $\boldsymbol{H}$ is a shifted version of the matrix shown in equation 3.22

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{I} & \boldsymbol{I} & \cdots & \boldsymbol{I} & \boldsymbol{I} & \cdots & \boldsymbol{I} \\ \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{\alpha} & \cdots & \boldsymbol{\alpha}^{(j-2)} & \boldsymbol{\alpha}^{(j-1)} & \cdots & \boldsymbol{\alpha}^{(k-2)} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} & \cdots & \boldsymbol{\alpha}^{2(j-3)} & \boldsymbol{\alpha}^{2(j-2)} & \cdots & \boldsymbol{\alpha}^{2(k-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \cdots & \vdots \\ \boldsymbol{0} & \boldsymbol{0} & \cdots & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{\alpha}^{(j-1)} & \cdots & \boldsymbol{\alpha}^{(j-1)(k-j)} \end{bmatrix}, \tag{3.27}$$

where the parameters have the same meaning as in section 3.5.2.

Our simulation results show that the performance of these modified randomly and deterministic constructed codes is nearly equal to the performance of the non-triangular codes. The influence of the lower triangular part of the matrix is negligible, especially if the rate of the code is high.

3.7 Comparison of LDPC Codes with Turbo Codes

Besides LDPC codes, there is another popular family of codes that can be decoded iteratively—*turbo codes.* Turbo codes are parallel concatenated, recursive systematic convolutional (RSC) codes that were introduced in [3]. The encoder of a turbo code is shown in figure 3.18. The parity digits $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ are calculated using the original information vector $\boldsymbol{u}$ and an interleaved version $\tilde{\boldsymbol{u}}$ respectively. In this configuration the turbo code has a rate of $\frac{1}{3}$. The rate of the code can be increased by *puncturing* but this is not considered in this work. Therefore, all codes used in this comparison are of rate $\frac{1}{3}$. The decoder shown in figure 3.19 consists of two MAP decoders (implemented using the BCJR algorithm [2]) and the required deinterleaving and interleaving. The extrinsic output of one component decoder is interleaved and fed into the other component decoder as a-priori information. After a fixed number of iterations, the decoder calculates a hard decision of the estimated information vector. One iteration of the decoder requires two uses of the BCJR algorithm in the component decoders.

It is hard to make a fair comparison between the computational complexity of a turbo decoder and a LDPC decoder. Several simplified algorithms for the BCJR decoder and for the LDPC decoder have been proposed. These algorithms have very different computational complexity. For a fair comparison, system parameters like BER at given SNR and maximum delay have to be considered. The computation of one bit in the BCJR algorithm requires the calculation of the transition probabilities and the values of the forward and backward recursion. In comparison, the calculation at a variable node of the LDPC decoder requires only the summation of the incoming messages and the computation at the check nodes can be implemented efficiently too.

The difference in the performance between LDPC codes and turbo codes is very small. For short block lengths, turbo codes perform slightly better but with increasing block lengths, LDPC codes have a better performance. The simulation results shown in figure 3.20 show a turbo code with two RSC encoders as shown in figure 3.21 with a random interleaver and irregular LDPC codes. The turbo decoder uses a maximum of 10 iterations.

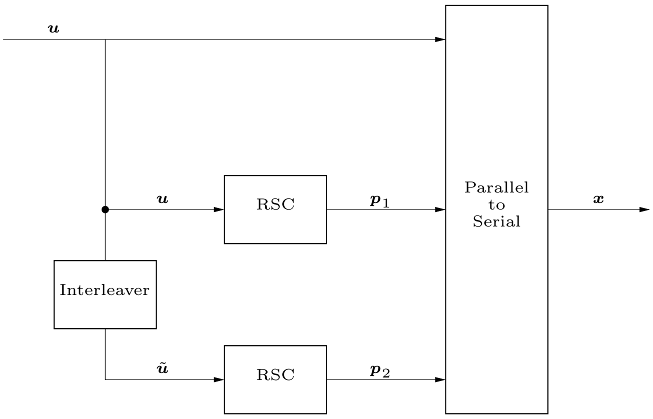

Figure 3.18: Turbo encoder.

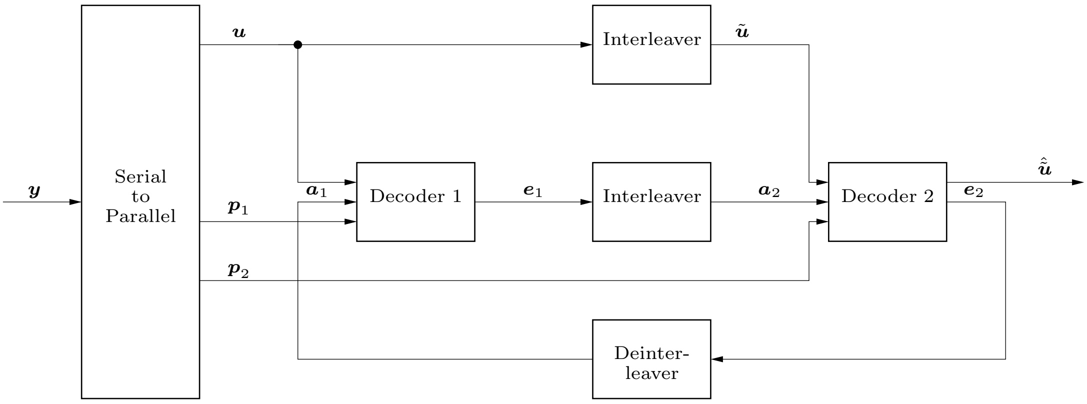

Figure 3.19: Turbo decoder.

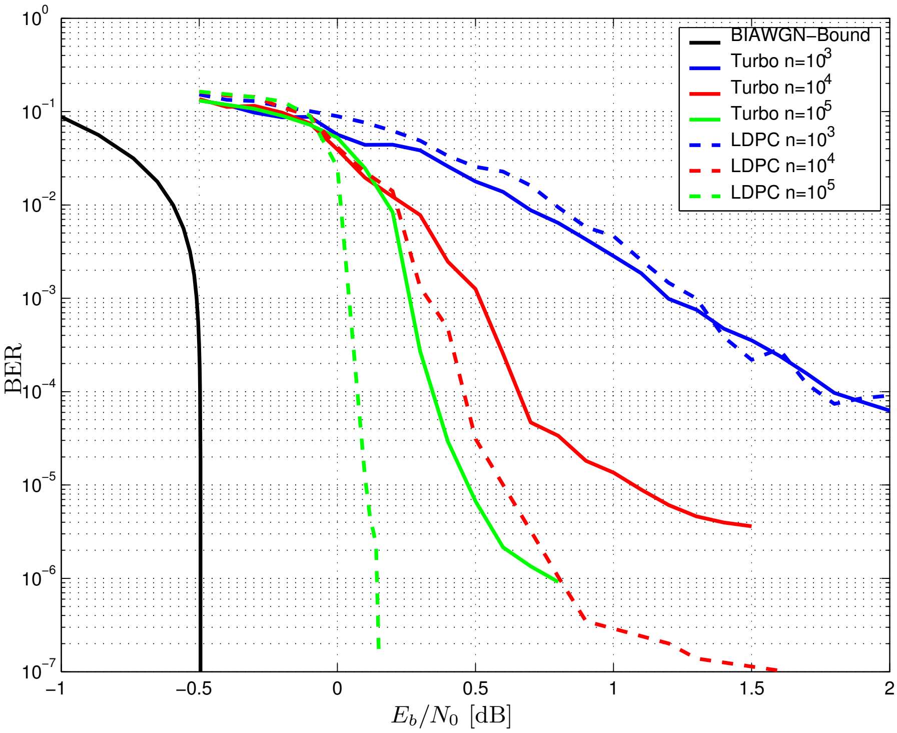

Figure 3.20: Comparison of Turbo codes and LDPC codes.

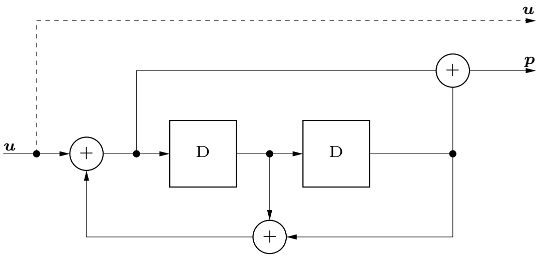

Figure 3.21: RSC encoder.

3.8 Conclusion

As a criterion for comparing codes, the gap of the bit error rate compared to channel capacity was used at a bit error rate of 10^{-5}. The results for the codes of rate 0.5 are shown in table 3.6. These for the codes of rate about 0.9 are shown in table 3.7.

n	random regular	random irregular
10^2	5	
10^3	2.4	2.7
10^4	1.3	1.1
10^5	1	0.4

Table 3.6: Gap to capacity in dB for LDPC codes with rate 0.5.

n	random regular	random irregular	deterministic regular
10^2	4.1		
10^3	1.7	1.7	1.6 ($n = 1369$)
10^4	0.9	0.8	
10^5	0.6	0.4	

Table 3.7: Gap to capacity in dB for LDPC codes with rate 0.9.

These results show that LDPC codes are useful if the block length is larger than or equal to 10^3. Although irregular codes have an asymptotically better performance than regular codes, they exhibit error floors at block lengths smaller than 10^4. Therefore, they have to be combined with outer codes to remove the error floors. The higher computational complexity of irregular codes should also be taken into account when comparing these codes.

Deterministically constructed codes are of practical interest if the rate of the code is high. Shortening the code results in a worse performance. However, for applications that require high rates, codes constructed with a deterministic algorithm have remarkable advantages (standardization, lower memory requirements) in comparison with randomly constructed codes.

Chapter 4

Convergence of the Decoding Process

This chapter deals with the convergence of the sum-product algorithm. The first section introduces a tool developed in [17] for analyzing the asymptotic case of infinite block lengths. This tool allows the prediction of the convergence in a graphical way by observing extrinsic information. It is called *extrinsic information transfer (EXIT) chart*.

After the case of infinite block lengths, the convergence behavior of the sum-product algorithm for finite block lengths is investigated. For this case, an analysis method based on animations was developed within this project. The method allows to identify errors of three types, only one of which is specific to codes of finite block length.

The last part of this chapter deals with the required number of iterations.

4.1 Iterative Decoding of Concatenated Codes

For simplification, the focus will be laid on concatenated systems with two component decoders. Of course, the principle can be extended to more than two component decoders. Figure 4.1 shows the block diagram of this system. The vector $\boldsymbol{y}$ is the received signal given by

$$\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{z}, \tag{4.1}$$

where $\boldsymbol{x}$ is the transmitted signal (the codeword in our case) and $\boldsymbol{z}$ is a noise vector. The received signal is fed to both decoders[1], where the path to the second decoder is usually interleaved. The output of the first decoder $\boldsymbol{v}$ is also interleaved

---

[1] The first decoder could be replaced by a detector to achieve iterative detection and decoding.

and connected to the second decoder[2]. To continue the iterative process, the deinterleaved output of the second decoder $\boldsymbol{w}$ is fed back to the first decoder. Every decoder has two inputs: one for the signals coming from the channel and the other one for the messages coming from previous used decoders carrying some new *a-priori* information. Every decoder combines these two sources of information to calculate output messages. These messages are fed into the other decoder.

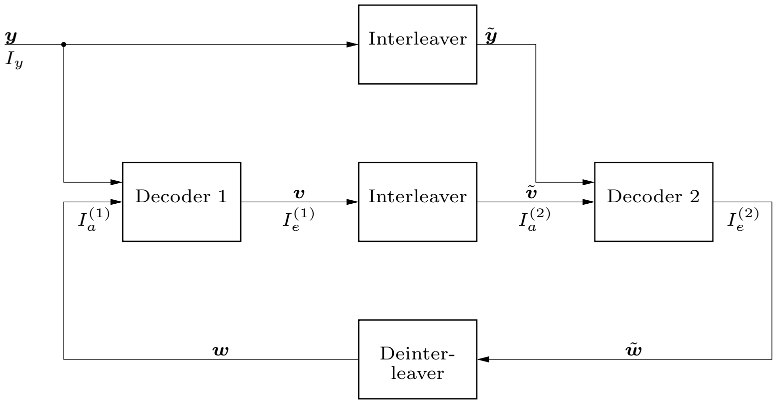

Figure 4.1: General structure of an iterative decoder.

The evolution of the vector valued quantities $\boldsymbol{v}$, $\boldsymbol{w}$ and $\boldsymbol{y}$ would be too complicated to compute. Therefore, we simplify this problem by observing scalar quantities that are derived from these vectors. These scalar quantities are the averaged mutual information between the elements of the vectors $\boldsymbol{v}$, $\boldsymbol{w}$ and $\boldsymbol{y}$ and the transmitted vector $\boldsymbol{x}$ calculated as:

$$I_a^{(1)} = I_e^{(2)} = \frac{I(\boldsymbol{x};\boldsymbol{w})}{n}, \tag{4.2}$$

$$I_e^{(1)} = I_a^{(2)} = \frac{I(\boldsymbol{x};\boldsymbol{v})}{n}, \tag{4.3}$$

$$I_y = \frac{I(\boldsymbol{x};\boldsymbol{y})}{n}, \tag{4.4}$$

where n is the length of the vectors. The subscripts indicate extrinsic information, a-priori information and channel values and the superscripts indicate the decoder. The mutual information between the output of a decoder and the transmitted codeword is equal to the mutual information between the input of the other decoder and the transmitted codeword because rearranging the components of the vectors in the interleavers does not change the mutual information.

With this simplification to the scalar case, each decoder can be described by the function

$$I_e = f(I_a, I_y). \tag{4.5}$$

---

[2] The interleaved quantities are denoted by $\tilde{\boldsymbol{v}}$, $\tilde{\boldsymbol{w}}$ and $\tilde{\boldsymbol{y}}$.

An example of this function is shown in figure 4.2. A binary alphabet is assumed and therefore, the mutual information is at most one. If we do not provide any a-priori information ($I_a = 0$), the decoder calculates extrinsic information based on the received vector from the channel. Therefore, the left point of the curve changes according to I_y. With increasing a-priori information the extrinsic information increases. If the decoder has perfect a-priori information $I_a = 1$, the output of the decoder will also contain no uncertainty about the transmitted codeword.

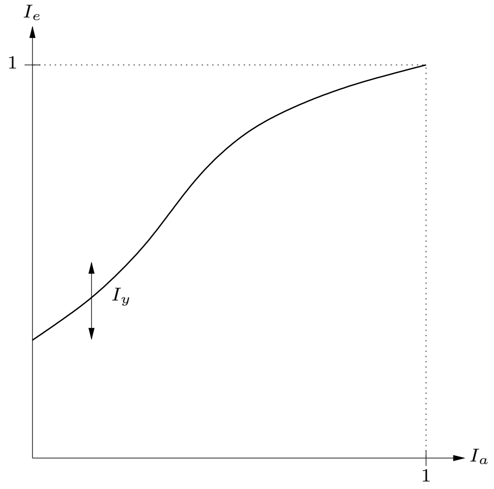

Figure 4.2: Performance of one decoder.

Such a function can be calculated and be drawn for every decoder. In our iterative system, the output of the first decoder is fed to the second decoder and vice versa. Therefore, we can combine these two performance functions and plot them in one figure, where the input/output axes are swapped for the second decoder (the output of one decoder is the input of the other decoder). This representation (EXIT-chart) is depicted in figure 4.3, where two identical decoders are assumed. This assumption leads to curves that are symmetrical with respect to the diagonal line.

Figure 4.3 can be used to illustrate the iterative decoding process. In the first iteration, the first decoder has no a-priori information from the second decoder ($I_a^{(1)} = 0$) and its output messages $\boldsymbol{v}$ carry the information received from the channel ($I_e^{(1)} = I_y$). The messages $\boldsymbol{v}$ are interleaved and passed to the input of the second decoder. The second decoder uses the information provided by $\tilde{\boldsymbol{v}}$ ($I_a^{(2)}$) as a-priori information and calculates output messages $\tilde{\boldsymbol{w}}$ carrying information $I_e^{(2)}$. These messages are deinterleaved and fed back to the input of the first decoder, where the iterative process starts again. The trajectories of this iterative process are shown in figure 4.4, where the blue steps are performed by the first decoder and

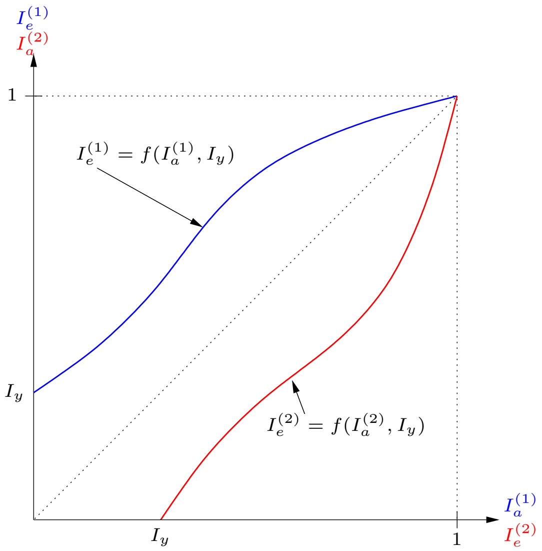

Figure 4.3: EXIT chart.

the red steps by the second decoder. From this analysis, we can observe that the iterative decoding process converges to the point in the top right corner and this point of convergence is the desired point where we have no remaining uncertainty about the transmitted codeword.

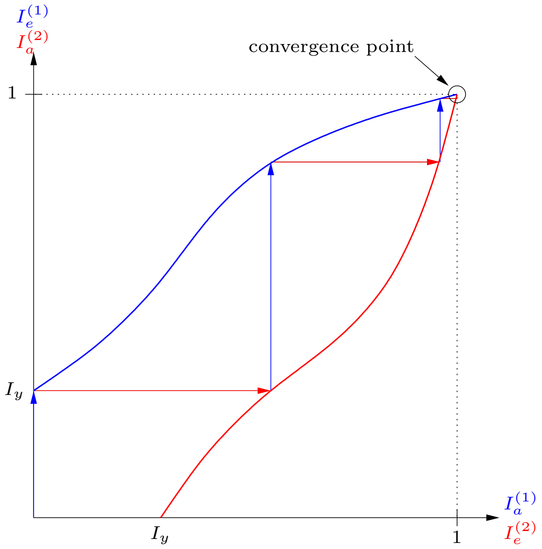

Figure 4.4: Trajectories of an iterative system.

If we use the same decoders as in the previous case but we decrease the information that is received by the channel (by increasing the noise level of the channel), the two curves will eventually intersect. In this case, the decoder also converges, but this convergence point is not the top right corner. This case is shown in figure 4.5.

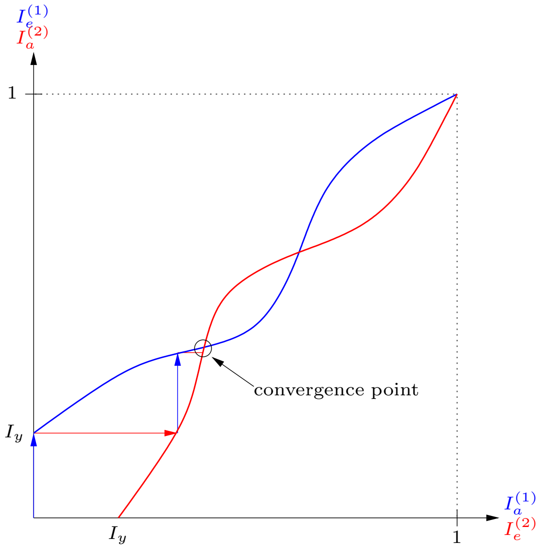

Figure 4.5: Convergence to an undesired point.

If the extrinsic information of each decoder is a monotone increasing function of its a-priori information, it is evident that the iterative system always converges. If the two curves do not intersect, the point of convergence is the top right corner and the decoder is able to successively recover the transmitted codeword. In the case of intersecting curves, the decoder is not able to decode the transmitted codeword and there will be uncertainty left after the decoding process has converged.

4.2 EXIT charts for LDPC codes

This section uses EXIT charts to describe the convergence of LDPC codes. The iterative LDPC decoder (sum-product algorithm) can be represented as the system shown in figure 4.6. The first decoder represents the operations performed at the variable nodes and the second decoder represents the operations performed at the check nodes[3]. Note that the check node decoder is not connected to the channel. Therefore, the extrinsic information of the second decoder depends only on its a-priori information and not on the received vector from the channel.

In the case of the iterative decoder for LDPC codes, the two component decoders (variable and check node decoder) are not identical. Therefore, the two curves are not symmetrical with respect to the diagonal line. An example of an

---

[3] Compare this with the construction method using the edge interleaver in section 3.5.1

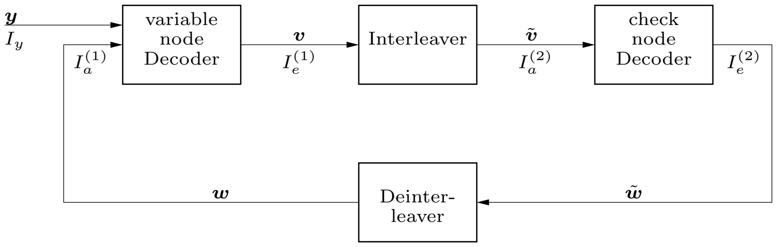

Figure 4.6: Iterative LDPC decoder.

EXIT-chart for a LDPC decoder is shown in figure 4.7. For the binary erasure channel, these two functions can be calculated in closed form. For other channels like the AWGN channel, the functions have to be calculated numerically. A description of calculating these functions can be found for example in [18]. In this work, EXIT charts are only used to illustrate the convergence behavior and therefore, the calculation of EXIT charts will not be treated.

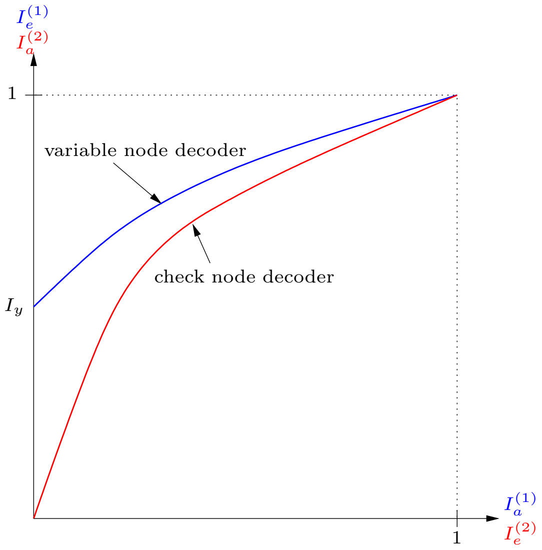

Figure 4.7: EXIT-chart for a LDPC decoder.

The curve for the check node decoder (red) starts at the origin. The check node decoder is not able to provide extrinsic information if no a-priori information is provided. This is because it is not connected to the channel output and therefore, the a-priori messages are the only sources of information for the check node decoder. The curve of the check node decoder is below the diagonal (note that the axes are swapped for this decoder), meaning that the mutual information between the outgoing messages and the transmitted codeword is smaller than the mutual information between the incoming messages and the transmitted codeword. This is in fact a bad decoder, but its performance is sufficient for the iterative process.

The performance curves depend on the type of channel, the mutual information between the received vector and the transmitted codeword and on the decoding algorithm used. For a given type of channel and decoding algorithm, the curve of the variable node decoder varies with the mutual information I_y (the variable node decoder is connected to the channel) but the curve for the check node decoder remains unchanged because this decoder is independent of the channel output. If I_y is decreased (for example by increasing the noise on an AWGN channel), the left point of the variable node decoder curve shifts down and the space between the two curves becomes smaller. If the two curves intersect, the iterative decoding process stops at the crossover point as shown in figure 4.8.

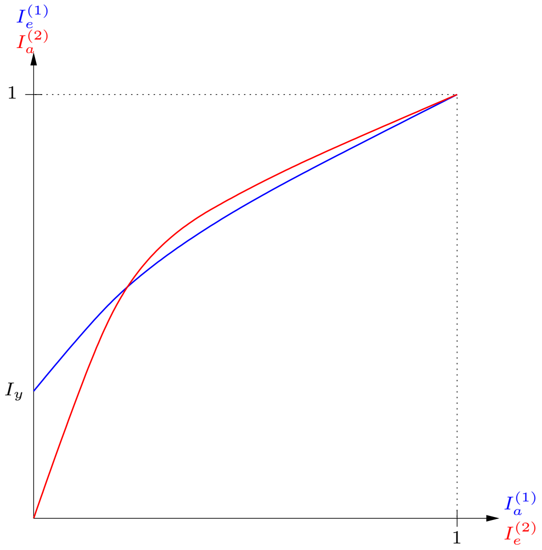

Figure 4.8: EXIT-chart for a LDPC decoder with intersecting curves.

It should be noted that EXIT charts can be used to design the degree distributions for irregular LDPC codes. The trick is to shape the variable node curve to closest match the check node curve for a given channel. This topic is not treated in this work. Details on the design can be found in [1].

The results of this section can be summarized as follows:

- The iterative decoding process is guaranteed to converge to the intersection point of the two transfer curves in the EXIT chart (which could be the at the top right corner of the EXIT chart).

- If the curves of the decoder do not intersect, no uncertainty about the transmitted codeword is left at the convergence point.

- The number of required iterations to reach the convergence point is high if the curves are very close.
- If the curves intersect, the decoder is not able to remove all uncertainty.

These results are only valid for the case of cycle free graphs. The reason for this is that the variable node decoder only calculates true extrinsic information if the the factor graph can be drawn as a tree. The case of a cycle free graph is equivalent with infinite block lengths as the probability of a cycle decreases with increasing block length. The following section treats the case of codes with finite block length, i.e. a corresponding factor graph that contains cycles.

4.3 Visualization of the Decoding Process

In contrast to the previous section, where the behavior of LDPC codes for infinite block length is analyzed, in this section the decoding process of instances of finite length LDPC codes is investigated. The block lengths used in this work are in the order of 10^3 to 10^4, which are suitable for practical applications like digital subscriber lines.

The analysis should not only provide an insight into the behavior of the iterative decoder, but it should also help to investigate errors that occur in iterative decoding. With these results, it should be possible to define additional guidelines for code construction and to further improve LDPC codes and the decoding algorithm.

The iterative decoder works by passing messages between variable nodes and check nodes of the factor graph that is associated with the LDPC code. After every iteration, a current estimate of the LLR values for every variable node can be calculated. These estimates are plotted and merged into a *computer animation*. The components of this visualization are shown in figure 4.9. The main lower plot shows the LLR values of every digit of the codeword (every variable node). After every iteration, new values are drawn in this diagram. This allows the observation of every digit of the codeword. The upper left plot shows the magnitude of the LLR values averaged over the digits of the codeword which corresponds to the average reliability of the decisions. In addition, the upper right plot shows the number of bit errors over the number of iterations.

The all-zero codeword is always transmitted. This is justifiable if the following properties are fulfilled:

- The code is linear and therefore, every codeword has the same properties.

- The channel is output symmetric.
- The decoder is symmetric.

The first two requirements are fulfilled by LDPC codes and an AWGN channel with binary input. The last requirement is fulfilled by the sum-product algorithm. Since we always transmit the all-zero codeword, a positive LLR value corresponds to a correct decision and a negative LLR value indicates a bit error.

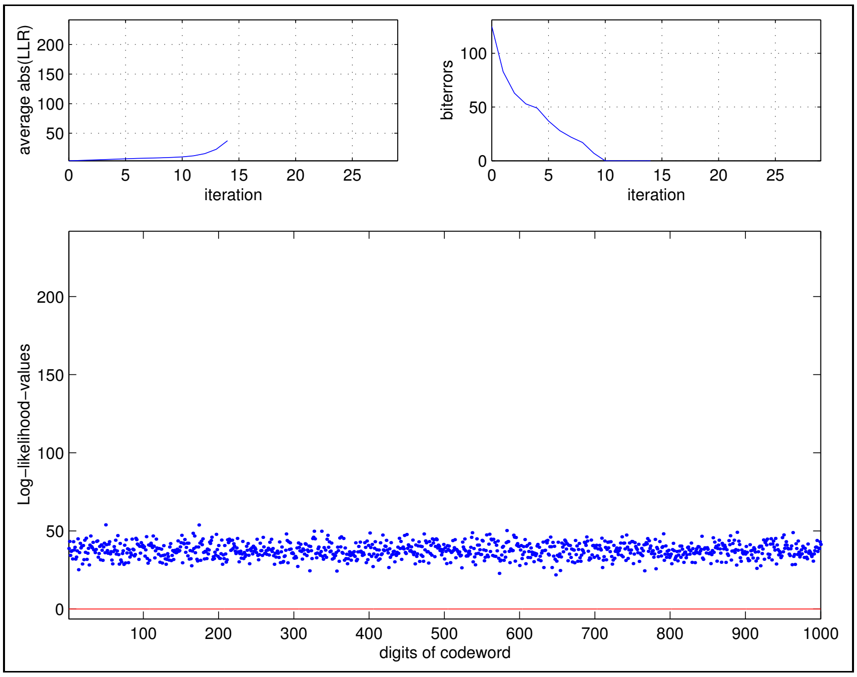

Figure 4.9: Visualization of the decoding process.

In this report screenshots of the animations are provided. The animations are available on the web (`http://ldpc.gottfriedlechner.com`).

4.3.1 Regular Codes

The first codes that we analyzed are regular codes. Because of the constant variable node degree, it is expected that the behavior is equal for every digit of the codeword. If we ignore the influence of cycles, we would expect that the LLR values converge to the true a-posteriori LLRs. The parameters of the code used for the simulation are shown in table 4.1 and the screenshots of the animation are shown in figure 4.10.

In the last figure, we see that the LLR values of the digits tend to infinity (the ceiling in the figure is caused by numerical limitations of the simulation environment). It is evident that these values can not be the true a-posteriori

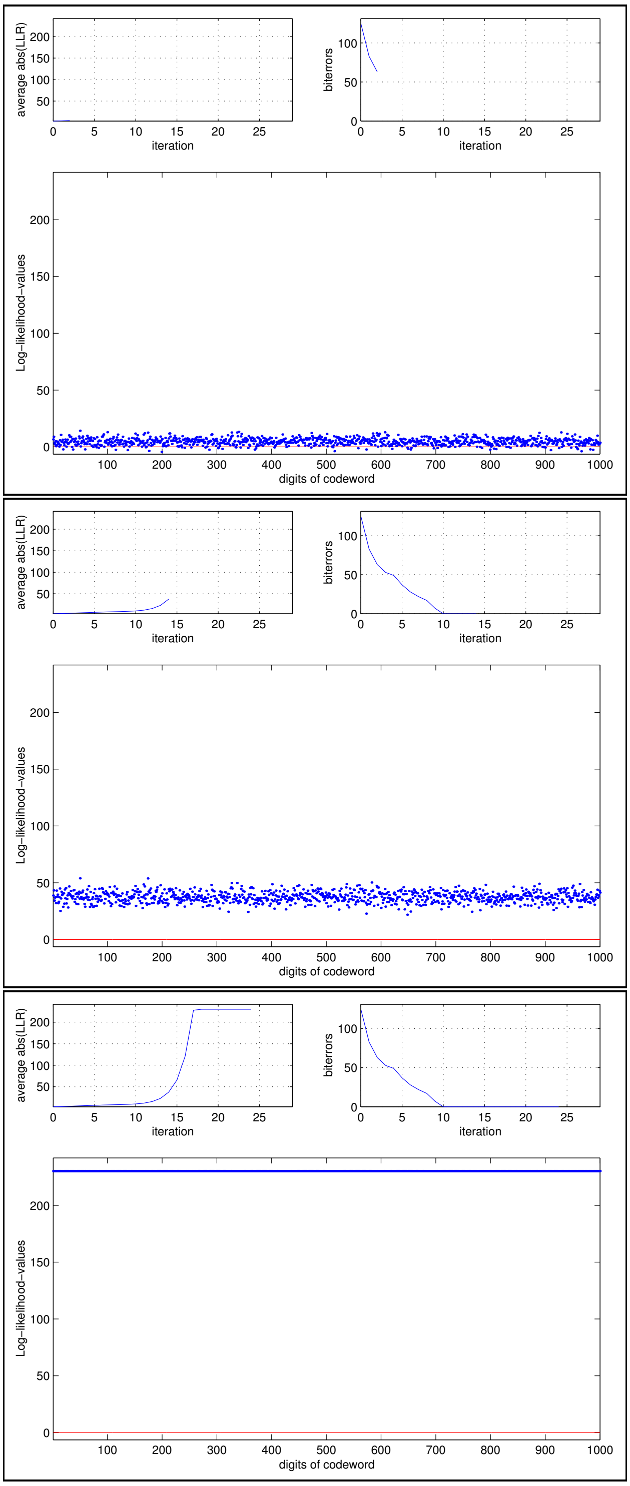

Figure 4.10: Visualization of successful decoding of a regular code.

parameter	value
N	1,000
M	500
R	0.5
d_v	3
d_c	6

Table 4.1: Parameters for regular code.

LLRs (in the presence of noise, the MAP decoder will always provide finite LLR values). The reason for this behavior is the presence of cycles in the graph. They cause incoming messages to the variable nodes to be correlated. This violates the assumption of the sum-product algorithm and leads to infinite LLR values. The fact that every variable node tends to infinity leads to the conclusion that every node is involved in a cycle.

In practice, we are only interested in the sign of the LLR values and therefore, this effect does not influence the performance of the decoder.

4.3.2 Irregular Codes

One idea of allowing irregular column weight distributions is that higher degree nodes should converge faster (because they receive many messages from the check nodes carrying extrinsic information). These nodes can provide reliable information for the lower degree nodes. This intuitive idea is verified by observing the decoding process of an irregular LDPC code. An irregular code with parameters shown in table 4.2 and a degree profile shown in figure 4.11 is simulated. Screenshots of the animation are provided in figure 4.12.

parameter	value
N	1,000
M	500
R	0.5
d_v	$2 \ldots 9$
d_c	6

Table 4.2: Parameters for irregular code.

The animation confirms the intuitive considerations. The higher degree nodes (on the right side of the codeword) converge much faster than the lower degree

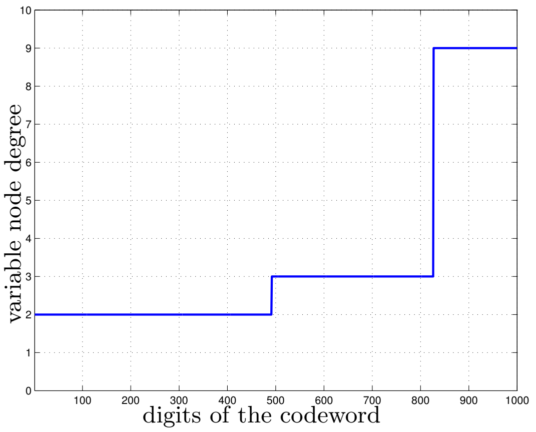

Figure 4.11: Degree profile of the irregular code.

nodes. Note that the degree profile can be recognized in the last screenshot. As in the case of regular LDPC codes, the LLR values tend to infinity if the number of iterations is increased.

4.3.3 Decoding Errors

In the previous sections, successful decoding of regular and irregular codes was simulated. However, the major application of the visualization is the analysis of *decoding errors*. The observed errors can be separated into three different types:

- Error Type I—*Convergence to a vector that is not a codeword*
  This decoding error is shown in figure 4.13. After approximately 50 iterations, the LLR values do not change and therefore, the decoder converges. If a hard decision of the LLR values is made, the resulting vector is not a codeword. The number of bit errors is high. This corresponds to intersecting decoder curves in EXIT charts. This type of error can easily be detected by calculating the syndrome.

- Error Type II—*Oscillation of the decoder*
  The oscillation of the decoder is shown in figure 4.14. The decoder is able to correct most of the bit errors, but is not able to converge to a codeword. Instead, the decoder starts to oscillate and the number of bit errors increases again. This oscillation does not stop. By observing the averaged magnitudes of the LLR values, this type of error can be detected. The observation of the averaged magnitudes can further be used to define a stopping criterion. This enables the decoder to minimize the number of bit errors for this type of error. *This behavior can not be explained with EXIT charts.*

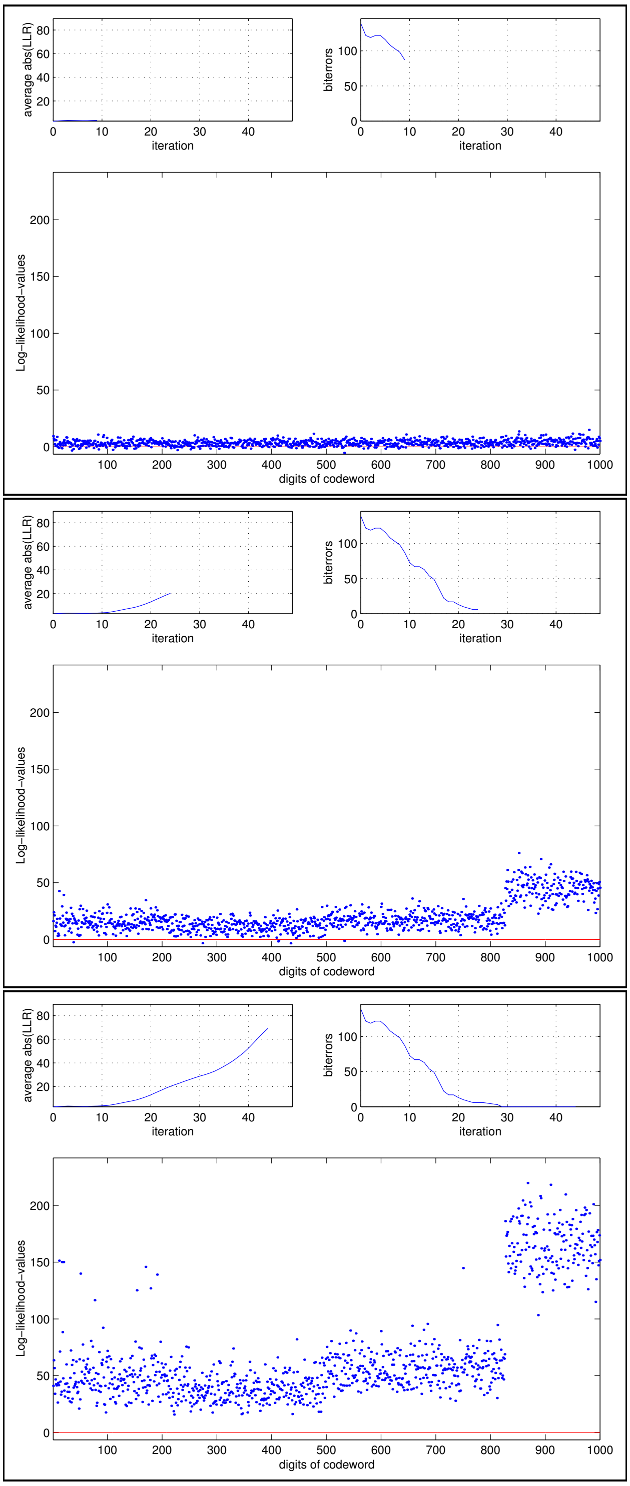

Figure 4.12: Visualization of successful decoding of an irregular code.

- Error Type III—*Convergence to a wrong codeword*
  This error is shown in figure 4.15. The probability for this type of error is high if the minimum distance of the code is small. From the decoder's point of view, the decoding process was successful, but the decoded codeword is not equal to the transmitted one. In most cases the decoded codeword was in fact the most likely codeword. This is a weakness of the code—improving the decoding procedure can not avoid this type of error. In the EXIT chart, this would correspond to non-intersecting curves. This case is similar to the case of successful decoding. The decoder is not able to detect this type of error.

Simulations showed that one type of errors dominates in different SNR regions. At low SNR values, the convergence to a vector that is no codeword is the dominating type. At high SNR values, the convergence to a wrong codeword is the error that was observed in general (most of the times, the decoder worked successfully of course). Between these two regions—in the so called 'waterfall' region—the oscillation of the decoder is the dominating type of error.

The optimal MAP decoder has only two types of error. Either the MAP solution corresponds to a codeword (although it could be different from the transmitted one) or it corresponds to a vector that is not a codeword. Therefore, the additional type of error observed for the iterative decoder (oscillations) determines the gap between the performance of iterative decoding and MAP decoding.

Figure 4.16 shows the block error rate for a regular LDPC code of rate 0.5 and block length 100 (the block length is very short to demonstrate the effects.). This block error rate is decomposed into the three types of error[4]. Also shown in the figure is the Gallager bound on the block error rate. It is very close to the convergence to a vector that is no codeword. This confirms the assumption that the gap between optimal MAP decoding and iterative decoding is determined by the oscillations of the decoder. Increasing the block length leads to a larger minimum distance and therefore, the probability of convergence to a wrong codeword decreases. Furthermore, the performance of the code (in terms of gap to capacity) increases, i.e. the curves get steeper. This is shown in figure 4.17 where the block length is increased to 500. We were not able to observe errors of type III for this block length. The performance of the code in the waterfall region is dominated by the oscillations of the decoder.

---

[4] It should be noted that the error types I and II are usually mixed and the classification depends on a decision threshold. Changing this threshold results only in a shift of the curves.

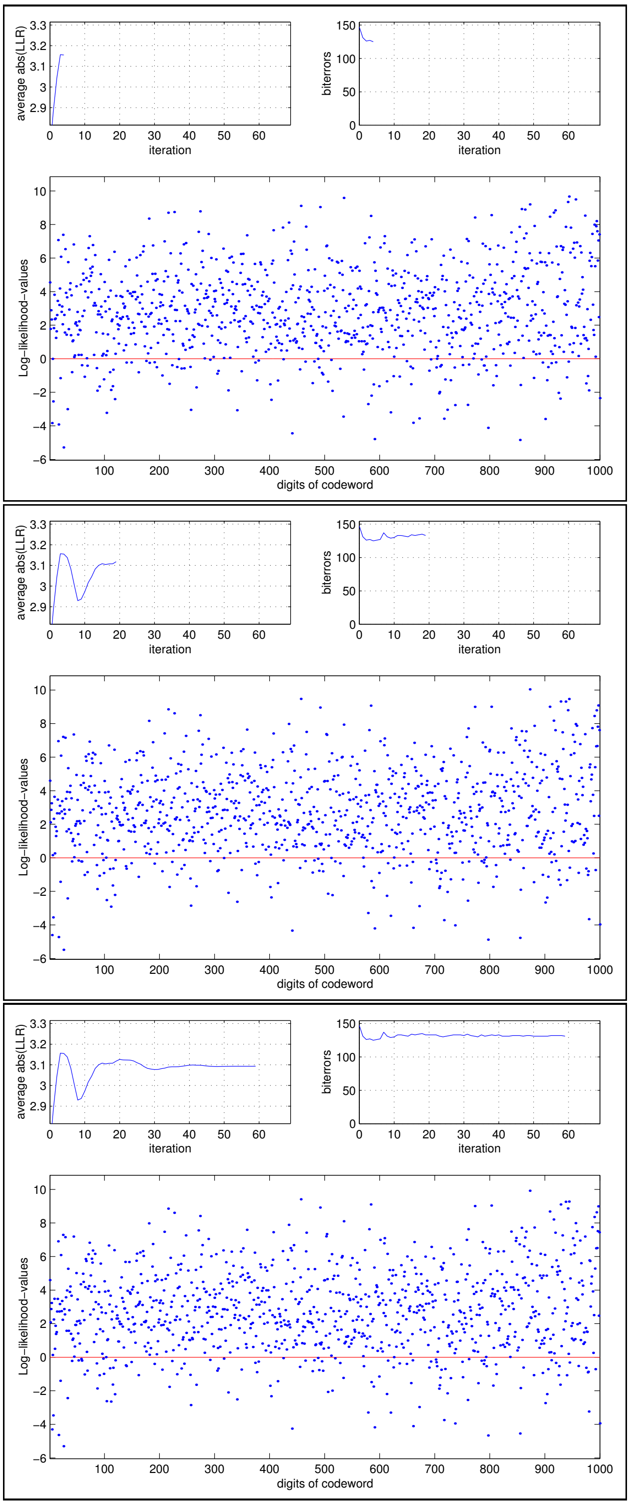

Figure 4.13: Convergence to a vector that is not a codeword.

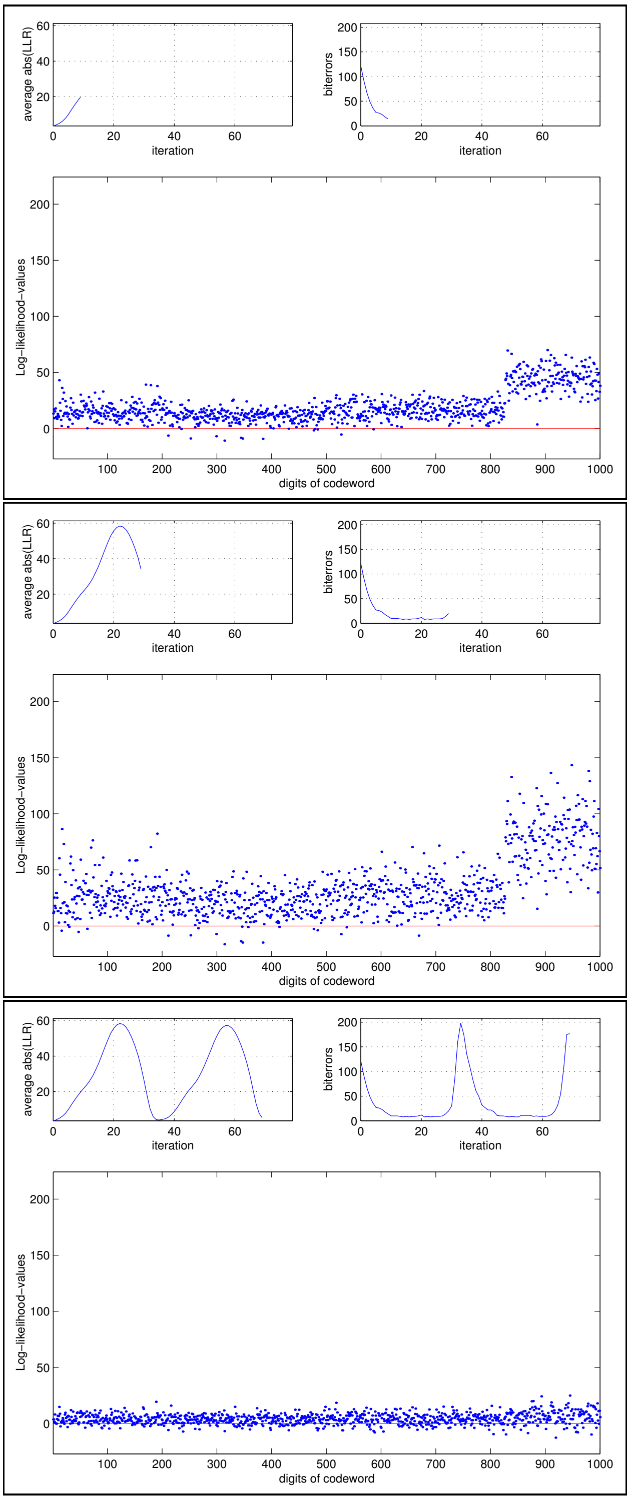

Figure 4.14: Oscillation of the decoder.

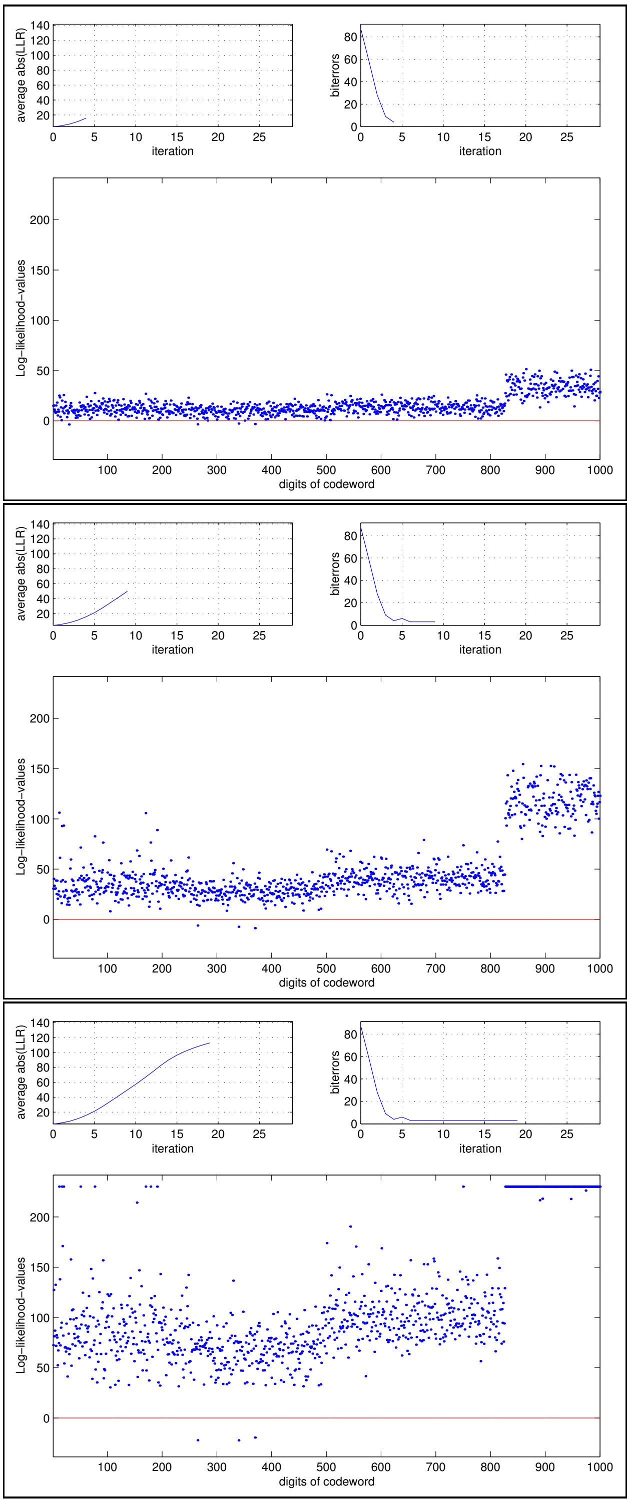

Figure 4.15: Convergence to a wrong codeword.

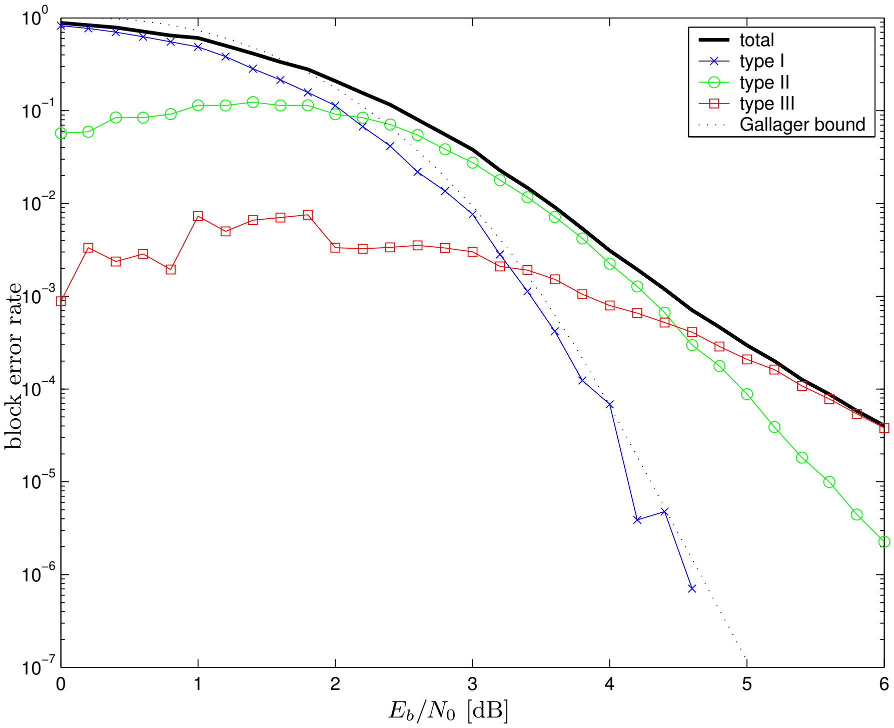

Figure 4.16: Error types at different SNR values for n=100.

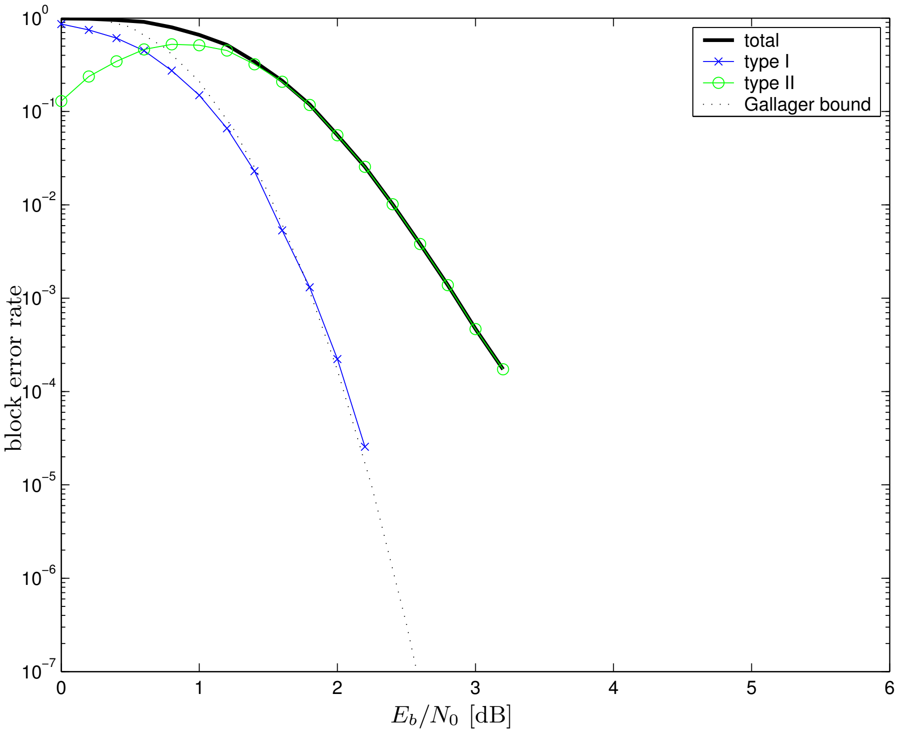

Figure 4.17: Error types at different SNR values for n=500.

4.4 Number of Required Iterations

In practical applications where the maximum decoding delay is an important parameter, the maximum number of iterations of the iterative decoder is limited. Figure 4.18 shows histograms of the required number of iterations when the decoder was able to decode the transmitted codeword. In this simulation, the decoder stops if it detected a valid codeword, i.e. the decoder has to calculate the syndrome after every iteration. The code used for this simulation was a regular LDPC code with block length $n = 1000$, rate $R = 0.5$ and a variable node degree of 3.

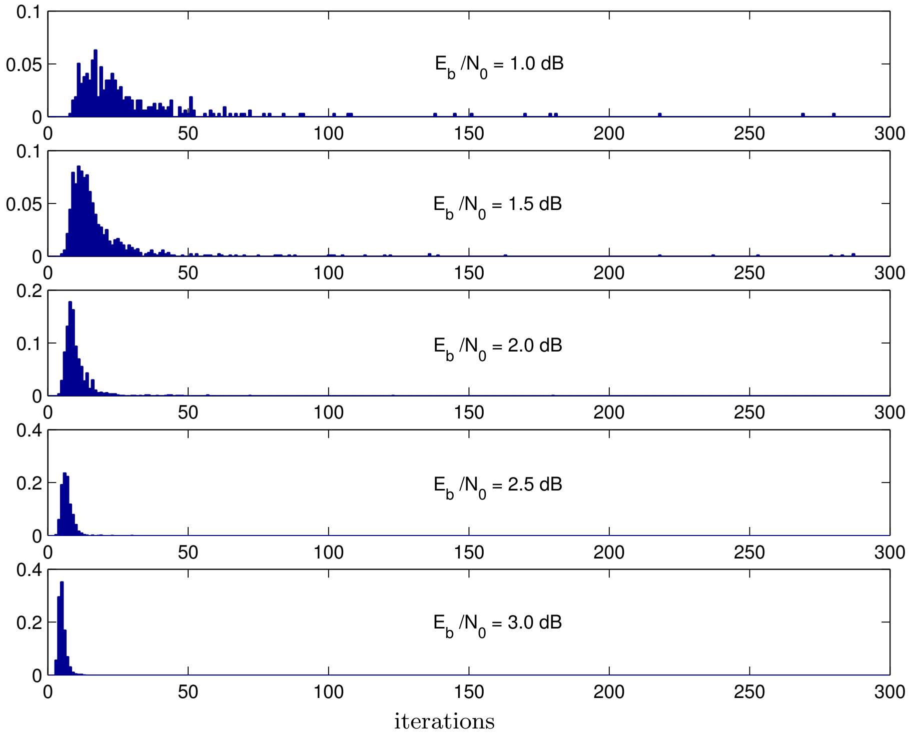

Figure 4.18: Number of iterations for successful decoding.

The mean of the required number of iterations decreases with increasing SNR and the variance of the distribution decreases too. Figure 4.19 shows the average number of required iterations (fur successful decoding) for varying SNR and varying maximum number of iterations.

This simulation and the histograms of the required number of iterations could lead one to conclude that reducing the number of maximum iterations should not influence the performance at high SNR because the histogram is concentrated at a low number of iterations. To test this assumption, the BER and the block error rate of this code for different numbers of maximum iterations were simulated.

The BER of this simulation is shown in figure 4.20 and the block error rate in figure 4.21.

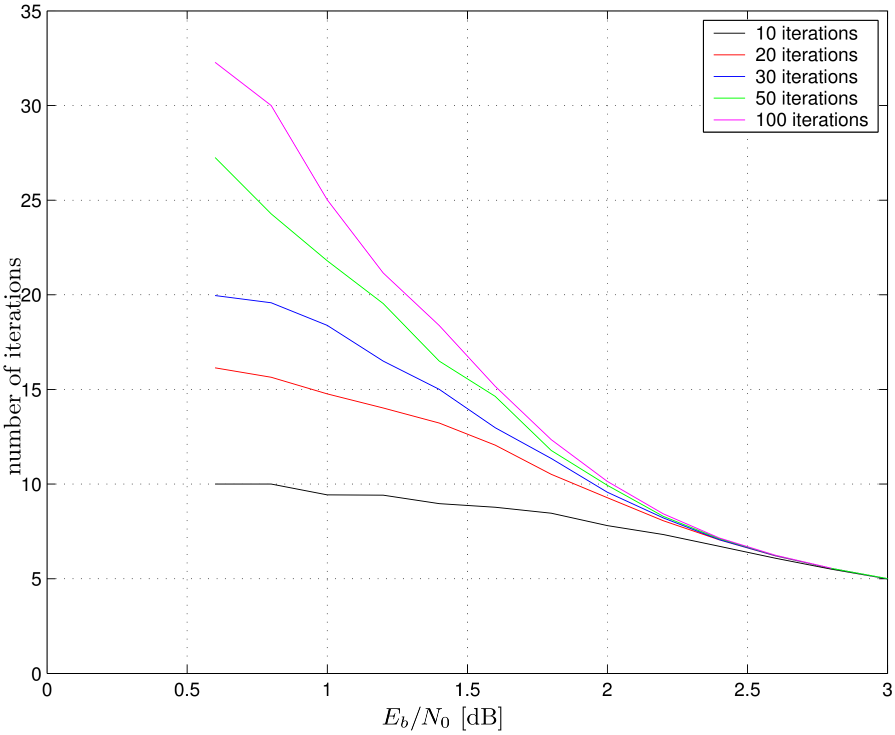

Figure 4.19: Mean value of required iterations.

This simulation showed that contradictionary to the assumption made from observing the histograms, the bit and block error rates increase at high SNR if the number of maximum iterations is reduced. There is a significant difference between the simulation results—the required SNR for a BER of 10^{-5} increases by 0.5dB if the maximum number of iterations is reduced from 100 to 10. Although the majority of blocks is successfully decoded after a few iterations, there are a lot of blocks that require a high number of iterations for successful decoding.

The required *average computation time* at high SNR does not increase if the maximum number of iterations is increased (at 3dB the average number of iterations is approximately 5 and independent of the maximum number of iterations as shown in figure 4.19). However, while the average computation time is not increased, some blocks utilize the high maximum number of iterations and need much more computation time. This results in a large *delay jitter*. Whether this jitter can be tolerated depends on the application.

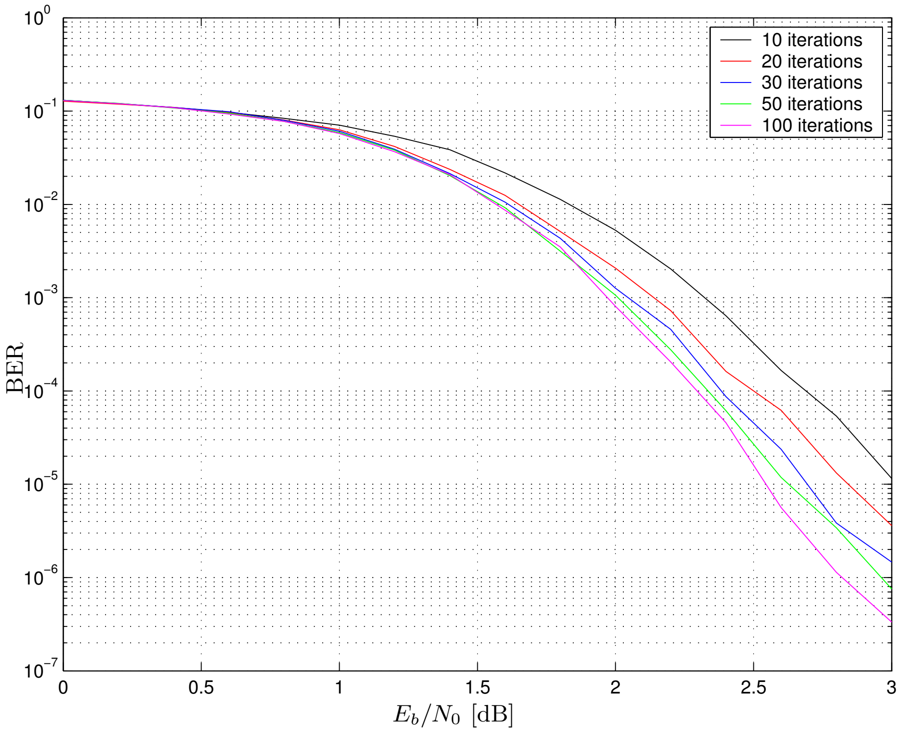

Figure 4.20: BER for different number of maximum iterations.

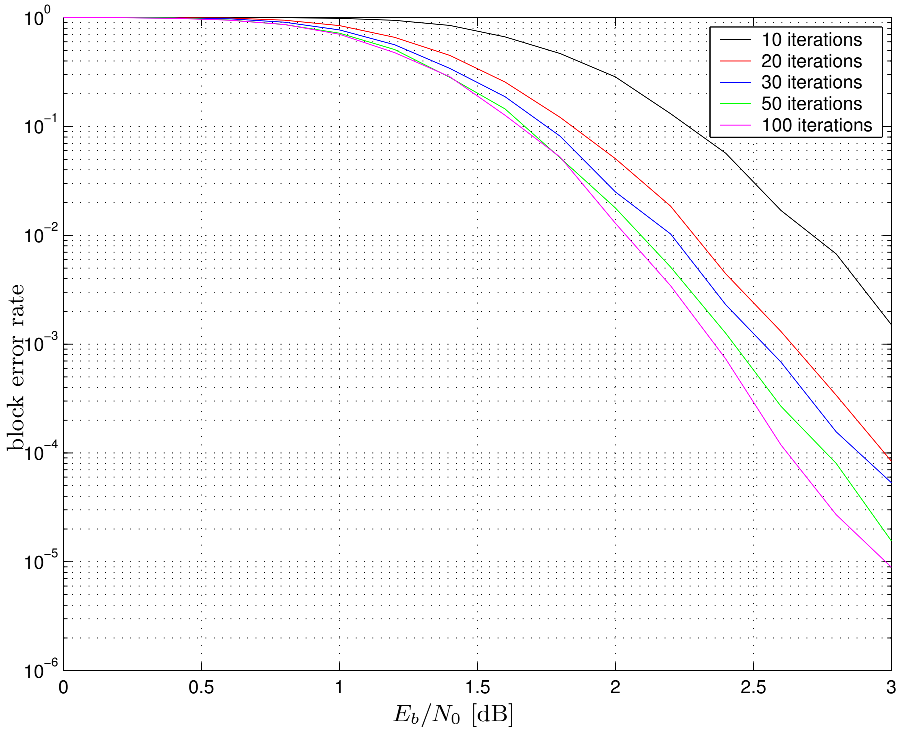

Figure 4.21: Block error rate for different number of maximum iterations.

4.5 Conclusion

In this section, the convergence behavior of the sum-product algorithm for LDPC codes with infinite and finite block lengths was analyzed. The results can be summarized as follows:

- For infinite block lengths, EXIT charts are a useful tool to predict the behavior of the iterative decoder and to determine the number of required iterations.
- For finite block lengths, EXIT charts can be used to determine the *averaged* convergence behavior, but a specific decoding process can differ from the average behavior.
- An additional type or error—oscillations of the decoder—was discovered for short block lengths that determines the performance of the system in the waterfall area. This type of error can not be explained by EXIT charts.
- This additional type of error can be detected by the decoder by observing the averaged magnitudes of the LLR values—a quantity that can be calculated at a negligible computational cost. Further work should be invested to exploit this knowledge at the decoder to improve the performance of the system.
- If the application allows a delay jitter, the maximum allowed number of iterations can be made large, resulting in an improved performance. If the decoder checks for a codeword after every iteration (by calculating the syndrome) the increased computational complexity is small because the averaged number of iterations is nearly independent of the maximum number of iterations if the SNR is sufficiently large.

Chapter 5

Conclusion

Several construction methods for LDPC codes were investigated. The focus was on codes with block lengths in the order of 10^3 to 10^4 which are suitable for many applications. Simulations of randomly constructed codes showed that the performance of LDPC codes—in terms of required SNR for a desired bit error rate—is attractive if the block length of the code is larger than 10^3.

For block lengths larger than 10^4, the required SNR can further be reduced by using irregular codes. However, for irregular codes an error floor was observed that has to be eliminated by concatenating these codes for example with an outer Reed Solomon code.

Codes constructed in a deterministic way were compared with randomly constructed codes. They perform as well as randomly constructed codes if the rate of the code is high. The advantage of deterministically constructed codes is that they can be described by few parameters and are therefore well suited for standardization. In addition, the structure of the deterministically constructed codes can be exploited by the encoder and the decoder to reduce the required memory when implementing these codes.

It was shown that encoding can also be performed with computational complexity that is linear in the block length if a further constraint—a parity-check matrix in triangular form—is introduced. The influence of this constraint on the performance of the code is small.

A comparison of LDPC codes with Turbo codes with the same block lengths showed that the coding gains of these two families are comparable. The advantage of LDPC codes is that they can be designed more flexibly, i.e. they can be constructed for any block length and any rate and they can be optimized for systems that work iteratively over the detection and decoding process.

We presented EXIT charts, which are well suited to predict the convergence of the iterative LDPC decoder if the block length is sufficiently large. We further analyzed the convergence behavior of the LDPC decoder when decoding LDPC codes with finite block lengths. The decoding process for finite block lengths was visualized by computer animations and we identified three different types of errors. Two types of errors were already known and appear also in the case of infinite block lengths. For finite block lengths, we observed an additional type of error that determines the performance of the code in the waterfall region. Further work is planned to analyze this type of error and to improve code design and the decoding algorithm for finite length LDPC codes.

Finally, the required number of iterations was investigated. It was shown that in most cases, the decoder converges to the transmitted codeword after a few iterations if the SNR is sufficiently large. However, the required SNR for achieving a given bit error rate can be reduced significantly if the number of iterations is increased. The average computation time can be kept small by introducing a stopping criterion (calculation of the syndrome) but the application has to tolerate a large delay jitter.

Appendix A

Calculation of the Gallager Bound

The *Gallager bound* used in chapter 4 claims the existence of a block code with block length n and rate $R = \frac{k}{n}$ with a block error probability

$$P_{blockerror} < 2^{-n \cdot E_r(R)}, \tag{A.1}$$

where $E_r(R)$ is the *error exponent* which is determined by the rate and by the channel only. For a channel with discrete input and continuous output, the error exponent can be written as [9]

$$E_r(R) = \max_{0 \leq s \leq 1} \max_{P_x} \left[-sR - \log_2 \int_{\mathcal{A}_{out}} \left(\sum_{x \in \mathcal{A}_{in}} P(x) \cdot P(y|x)^{\frac{1}{1+s}} \right)^{1+s} dy \right]. \tag{A.2}$$

For symmetric channels with discrete input, it is known that the input distribution P_x that maximizes the error exponent is the uniform distribution. The distribution for the AWGN channel with binary input that maximizes equation A.2 is therefore

$$P(0) = P(1) = \frac{1}{2}. \tag{A.3}$$

The output alphabet for this channel is $\mathcal{A}_{out} = \mathbb{R}$. Inserting the input distribution, the output alphabet and the Gaussian probability density function in equation A.2 leads to

$$E_r(R) = \max_{0 \leq s \leq 1} \left[-sR - \log_2 \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{y^2+1}{2\sigma^2}} \cdot \left(\cosh \frac{y}{\sigma^2(1+s)} \right)^{1+s} dy \right], \tag{A.4}$$

where σ^2 is the noise variance of the channel and an input alphabet $\mathcal{A}_{in} = \{+1, -1\}$ is assumed.

This equation can be evaluated by numerical integration. An example for $\sigma^2 = 0.5$ is shown in figure A.1. The capacity of this channel can be calculated as $C = 0.72$. For rates larger than the capacity, the error exponent is zero as shown in the figure.

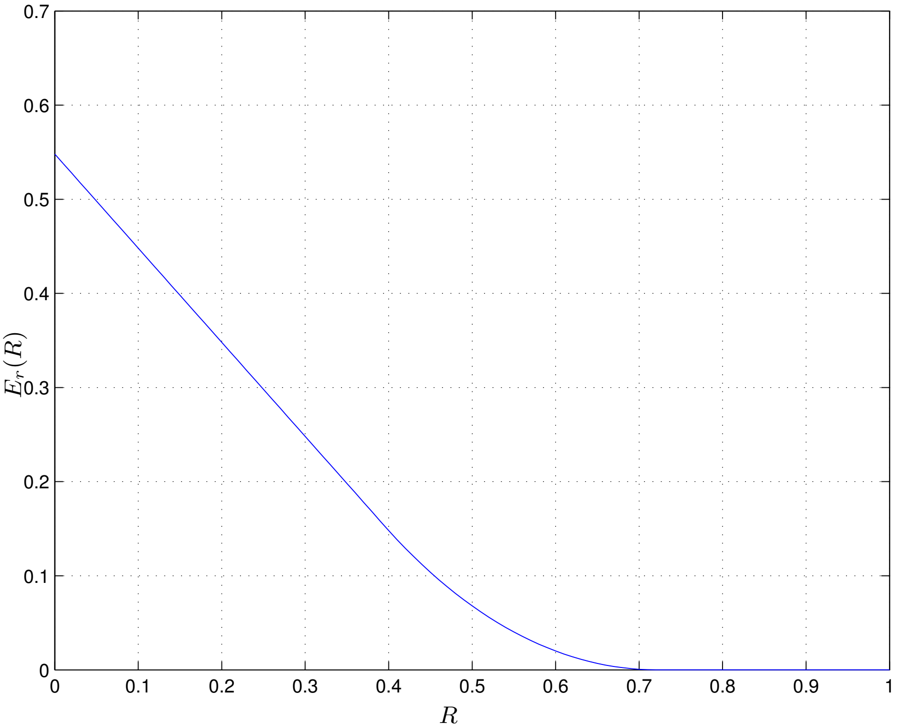

Figure A.1: Error exponent for a binary input AWGN channel with $\sigma^2 = 0.5$.

For a given rate R and block length n, the Gallager bound can be calculated using equation A.1 and equation A.4.

Bibliography

[1] A. Ashikhmin, G. Kramer, and S. ten Brink. Extrinsic information transfer functions: a model and two properties. In *Proc. 2002 Conference Information Sciences and Systems, Princeton, USA*, 2002.

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

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

[4] J. Campello and D.S. Modha. Extended bit-filling and LDPC code design. In *Proc. 2001 IEEE Globecom Conference*, 2001.

[5] E. Eleftheriou and S. Olcer. Low-density parity-check codes for digital subscriber lines. In *Proc. 2002 IEEE International Conference on Communications*, pages 1752–1757, 2002.

[6] K. Engdahl. *Analysis of some convolutional coding constructions*. PhD thesis, Dept. of Information Technology, Lund University, June 2002.

[7] T. Etzion, A. Trachtenberg, and A. Vardy. Which codes have cycle-free tanner graphs? *IEEE Transactions on Information Theory*, 45(6):2173–2180, September 1999.

[8] J.L. Fan. Array codes as low-density parity-check codes. In *2nd International Symposium on Turbo Codes and Related Topics, Brest, France*, 2000.

[9] Bernd Friedrichs. *Kanalcodierung*. Springer, 1995.

[10] R.G. Gallager. Low density parity check codes. *IRE Transactions on Information Theory*, IT-8:21–28, Jan 1962.

[11] R.G. Gallager. *Low Density Parity Check Codes.* Number 21 in Research monograph series. MIT Press, Cambridge, Mass., 1963.

[12] F. R. Kschischang, B. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, 2001.

[13] Y. Mao and A.H. Banihashemi. A heuristic search for good low-density parity-check codes as short block lengths. In *Proc. 2001 IEEE International Conference on Communications*, pages 41–44, 2001.

[14] T. Richardson, A. Shokrollahi, and R. Urbanke. Design of capacity-approaching irregular low-density parity-check codes. *IEEE Transactions on Information Theory*, 47:619–637, Feb 2001.

[15] T. Richardson and R. Urbanke. The capacity of low-density parity-check codes under message-passing decoding. *IEEE Transactions on Information Theory*, 47, 2001.

[16] C.E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948.

[17] S. ten Brink. Convergence of iterative decoding. *Electronic Letters*, 35(10):806–808, May 1999.

[18] S. ten Brink, G. Kramer, and A. Ashikhmin. Design of low-density parity-check codes for multi-input multi-output channels. In *Proc. 2002 Conference on Communication, Control and Computation, Allerton, USA*, 2002.