Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Mikael Simberg

Linear-time encoding and decoding of low-density parity-check codes

Master's Thesis
Espoo, 2015

Supervisor:	Professor Petteri Kaski
Advisors:	Professor Petteri Kaski
	Professor Camilla Hollanti

Aalto University
School of Science

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

AUTHOR:	Mikael Simberg		
TITLE:	Linear-time encoding and decoding of low-density parity-check codes		
DATE:	2015	PAGES:	108 + 12
MAJOR:	Systems and Operations Research	CODE:	F3008
SUPERVISOR:	Professor Petteri Kaski		
ADVISORS:	Professor Petteri Kaski Professor Camilla Hollanti		

ABSTRACT:

Low-density parity-check (LDPC) codes had a renaissance when they were rediscovered in the 1990's. Since then LDPC codes have been an important part of the field of error-correcting codes, and have been shown to be able to approach the Shannon capacity, the limit at which we can reliably transmit information over noisy channels. Following this, many modern communications standards have adopted LDPC codes. Error-correction is equally important in protecting data from corruption on a hard-drive as it is in deep-space communications. It is most commonly used for example for reliable wireless transmission of data to mobile devices. For practical purposes, both encoding and decoding need to be of low complexity to achieve high throughput and low power consumption.

This thesis provides a literature review of the current state-of-the-art in encoding and decoding of LDPC codes. Message-passing decoders are still capable of achieving the best error-correcting performance, while more recently considered bit-flipping decoders are providing a low-complexity alternative, albeit with some loss in error-correcting performance. An implementation of a low-complexity stochastic bit-flipping decoder is also presented. It is implemented for Graphics Processing Units (GPUs) in a parallel fashion, providing a peak throughput of 1.2 Gb/s, which is significantly higher than previous decoder implementations on GPUs. The error-correcting performance of a range of decoders has also been tested, showing that the stochastic bit-flipping decoder provides relatively good error-correcting performance with low complexity. Finally, a brief comparison of encoding complexities for two code ensembles is also presented.

KEYWORDS:	bit-flipping, coding theory, error-correcting codes, graphics processing unit, linear time complexity, low-density parity-check codes
LANGUAGE:	English

Acknowledgments

First and foremost, I would like to thank my supervisor Petteri Kaski for his great help and guidance throughout the preparation of this thesis, and for introducing me to the exciting world of error-correcting codes and the study of decoding using stochastic bit-flipping with Alexander Mozeika and Pekka Orponen.

I also wish to thank Camilla Hollanti for fruitful discussions and helpful comments throughout the thesis work, and for introducing me to and providing insights about information transmission standards.

I gratefully acknowledge the use of computing resources available via project "Science-IT" at Aalto University School of Science and via CSC–the Finnish IT Center for Science. As regards the latter I would especially like to thank Maarit Mantere for her advice that NVIDIA Tesla K40 graphics processing units are available at CSC.

Finally, I would like to thank my wonderful girlfriend and friends for always being there, and my family for showing interest in my work and fully supporting me in what I do.

Mikael Simberg
Espoo, 15.12.2014

Contents

Abstract ii

Acknowledgments v

List of acronyms xi

List of mathematical symbols xiii

1 Introduction 1

2 Error-correcting codes 5

- 2.1 Preliminaries 5
- 2.2 Noisy channels 7
- 2.3 Low-density parity check codes 9
  - 2.3.1 Tanner graph representation 9
- 2.4 LDPC code constructions 10
  - 2.4.1 Regular codes 10
  - 2.4.2 Irregular codes 11
  - 2.4.3 Code ensembles and their properties 13
  - 2.4.4 Quasi-cyclic LDPC codes 17
- 2.5 Codes used in practice 18
  - 2.5.1 Digital Video Broadcasting 18
  - 2.5.2 WiMAX and WiFi 19
  - 2.5.3 Ethernet 19
- 2.6 Summary 19

3 Encoding of LDPC codes 21

- 3.1 Encoding using the systematic generator matrix 21

3.2 Encoding with approximate lower-triangular parity-check matrices 23
3.2.1 Irregular codes with a fixed gap 27
3.2.2 Quasi-cyclic LDPC codes with approximate lower-triangular parity-check matrices 27
3.3 Summary 28

4 Decoding of LDPC codes 29

4.1 Naïve decoding 31
4.2 Bit-flipping decoding 31
4.2.1 Gallager's bit-flipping decoder 31
4.2.2 Weighted bit-flipping decoders 32
4.2.3 Schedules for bit-flipping decoders 34
4.2.4 Stochastic bit-flipping decoders 35
4.2.5 Stochastic bit-flipping decoder with hard channel values 37
4.3 Message-passing decoding 39
4.3.1 Exact inference on factor graphs and the sum-product algorithm 40
4.3.2 The sum-product algorithm for decoding 47
4.3.3 Implementing the sum-product decoder 52
4.3.4 Binary message-passing decoder 55
4.3.5 Message-passing schedules 57
4.4 Turbo codes 58
4.5 Summary 59

5 Decoder implementations 61

5.1 Hardware simulations 62
5.2 GPU implementations 63
5.3 A GPU implementation of a stochastic-bit flipping decoder 64
5.3.1 Architecture and programming of CUDA devices 65
5.3.2 Decoder implementation 67

6 Experimental results 77

6.1 Comparison of decoders 77
6.1.1 Choosing the decoder parameters 79
6.1.2 Results 80
6.2 Complexity of approximate lower triangular encoding 85
6.2.1 Results 85

6.3 The stochastic bit-flipping decoder on the GPU 89
6.3.1 Results 90

7 Conclusion 95

Bibliography 99

A Derivations 109

A.1 Derivation of the gradient-descent bit-flipping decoder 109
A.2 Derivation of the tahn-rule for the sum-product decoder 110

B Parameter searches 115

B.1 Parameter searches for the stochastic bit-flipping decoder 116
B.2 Parameter searches for the gradient-descent bit-flipping decoder 119

List of acronyms

LDPC code	Low-density parity-check code
QC-LDPC code	Quasi-cyclic low-density parity-check code
BEC	Binary erasure channel
BSC	Binary symmetric channel
BAWGNC	Binary additive white Gaussian noise channel
BER	Bit-error rate
FER	Frame-error rate
CPU	Central processing unit
RAM	Random access memory *or* random access machine
MBRAM	Random access machine with addition, subtraction, multiplication, division and bitwise Boolean operations
GPU	Graphics processing unit
CUDA	Compute unified device architecture
PTX	Parallel thread execution
ISA	Instruction set architecture

List of mathematical symbols

H	Parity-check matrix
G	Generator matrix
H_{ai}	The ith element of the ath row of a matrix H
H^T	The transpose of a matrix H
$\mathbf{x}$	A vector
x_i	The ith element of the vector $\mathbf{x}$
$\mathbf{x} \setminus x_i$	The vector $\mathbf{x}$ with the ith element removed
I_n	Identity matrix of dimension $n \times n$
$\mathbf{0}$	An all-zero vector
$\mathbb{F}_2$	The binary field
$N(i)$	The set of neighbors of a node i in a graph
$N(i) \setminus j$	The set of neighbors of a node i excluding the node j
P_b	Bit-error rate
P_B	Block-error rate
$[P]$	Iverson's bracket notation, evaluates to 1 if the predicate P is true and 0 otherwise
$\mathcal{N}(\mu, \sigma^2)$	The Gaussian distribution with mean μ and variance σ^2
$\operatorname{sgn}(x)$	The sign function
$\tanh(x)$	The hyperbolic tangent function
$\tanh^{-1}(x)$	The inverse hyperbolic tangent function

Chapter 1

Introduction

The field of information theory was started by Shannon in 1948 with his "A mathematical theory of communication" (Shannon, 1948). In it he presented the problem of transmitting information reliably over noisy channels, together with an initial solution on how to tackle the problem. Although there is a limit to how much information we can transmit over a channel—the *Shannon capacity*—Shannon also showed that we can come arbitrarily close to this limit using *error-correcting codes*. Shannon's work gave answers to a practical and common problem, and error-correcting codes are today in use in practically every communications scenario, from mobile networks to deep-space communications, in addition to reliable storage.

The classical communications problem is the following: we have a string of bits that we want to send from one place to another. The problem we are facing is that when we send the bits, some amount of noise will be added along the way and the receiving end will not receive what we originally sent. Error-correcting codes attempt to solve this problem by *encoding* the given string of bits into a longer string of bits, adding some form of *redundancy* to the bits that are sent. The longer string of bits is then sent over the channel and, on the receiving end, even with a certain amount of noise added to the bits, one can *decode* the sent bits uniquely to the original bits. Figure 1.1 presents the situation graphically. The problem is usually thought of as involving physical transmission of data from one point to another. However, the problem applies equally well in a situation where bits are not physically transferred at all. A hard-drive is one such example, where errors will accumulate over time.

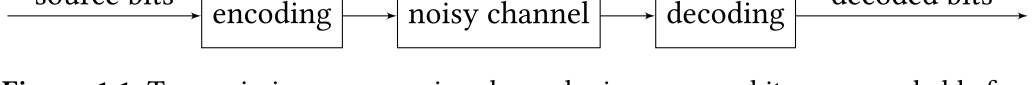


Figure 1.1: Transmission over a noisy channel: given source bits are encoded before transmission over a noisy channel; decoding of the encoded noisy bits attempts to recover the source bits.

How could one start to approach the problem of making sure that the right bits are received? Say that one would like to send the following string of 8 bits:

$$01001101.$$

We want to make sure that the receiver does not mistake the bits for any of the other $2^8 - 1$ possible bit strings of length 8 in the case that some of the bits are flipped along the way. One way of dealing with this is to send the following string of bits instead:

$$000\,111\,000\,000\,111\,111\,000\,111.$$

What we have done is replaced each bit by three occurrences of the same value. After receiving the longer string of bits, but with possibly some bits flipped, we can then decide that we decode the received bits so that for each group of three bits we set the value of that bit to the majority value. That is, if for example two or three of the three bits in a group are ones, we decide that the group of three bits represents a one. With this scheme one of the bits in each group of three bits can be flipped and we will still decode the correct string of bits.

The above scheme is called a repetition code which does not work well in practice. It does, however, demonstrate the essence of error-correcting codes: to allow us to recover from as many errors as possible, and to do so efficiently. The repetition code does the first if we repeat each bit enough times, but loses on efficiency as we have to repeat each bit many times to achieve reliable transmission. On the other hand, keeping the number of repetitions low and fixed maintains efficiency but we cannot recover from many errors. The codes considered in this thesis, low-density parity-check codes, are one type of codes that can do both tasks well.

Low-density parity-check codes were first introduced by Gallager (1962). Despite the current knowledge of their good properties, the codes were largely forgotten after their discovery until the 1990's. The codes were rediscovered independently by MacKay (1995), as well as Sipser and Spielman (1996). It was quickly realized that the new codes were largely equivalent to the codes Gallager first presented. Following the rediscovery there has been a considerable amount of research into low-density parity-check codes. They have been shown to have good error-correcting properties and have practical algorithms for both encoding and decoding. Recently, the codes have also been included in standards for wireless and wired communication, joining and in some cases replacing Turbo codes (Berrou et al., 2005) which have, together with low-density parity-check codes, been shown to be able to approach the Shannon capacity.

As the field of low-density parity-check codes has grown more mature since their rediscovery there has been more focus on making the codes perform better, as well as improving the algorithms used for encoding and decoding of the codes. For example, encoding can as of today not be done in linear time for general low-density parity-check codes. On the decoding side, most decoders are linear-time, but there is room for improvement in terms of the constants involved in implementations. The need for faster practical encoders and, more importantly, decoders is twofold. First, power consumption is an important factor in mobile devices, and reducing the complexity of hardware implementations can have a positive effect on the power consumption. Second, faster encoders and decoders can allow improvements in a combination of throughput, latency and error-correction. Linear-time encoders and decoders allow us to scale the so-called block length of low-density parity-check codes, ultimately leading to better error-correction. Most importantly, encoders and decoders with small constants give practical improvements to error-correction, throughput and latency. For this reason it is important to consider practical issues when implementing encoding and decoding algorithms, and not only the theoretical properties of the algorithms. With this in mind, we will in this thesis review the current state-of-the-art in encoding and decoding of low-density parity-check codes together with experimental results on encoding complexity and the error-correcting performance of decoders. Most importantly, this thesis presents a low-complexity decoder for GPUs.

We will begin with an overview of definitions relating to coding theory, define what low-density parity-check codes are, and look into various code constructions in Chapter 2. Chapter 3 contains a summary of useful encoding methods, one of which allows linear-time encoding for a limited but useful set of codes. A majority of the research is focused the on decoding of low-density parity-check codes. In Chapter 4 we review the advances in decoding algorithm designs, which aim to reduce the complexity of the decoders to reach higher throughputs, and to improve the error-correcting properties of the decoders. In Chapter 5 we present existing work on implementing decoders in hardware, where the focus generally is to achieve high throughput. Partly, the aim of this thesis is to provide a comprehensive overview of the current state-of-the-art in low-density parity-check codes. However, the main contribution of this thesis is the implementation of a simple decoder implemented for GPUs. The need for simpler algorithm designs becomes apparent as decoders are implemented in hardware. The implementation aims at achieving high decoding throughputs by using a simple design which is easily parallelized. In Chapter 6 we will present some experimental results concerning encoding complexity, compare five decoders in terms of error-correcting performance, and present results on the performance of the GPU implementation. We conclude the thesis in Chapter 7.

Chapter 2

Error-correcting codes

In this chapter we set up the terminology and basic concepts concerning error-correcting codes. In Section 2.1 we will present the basic concepts concerning error-correcting codes in general and in Section 2.2 we present three noisy channels. In Section 2.3 we present low-density parity-check codes, some constructions of low-density parity-check codes and the most important results relating to them. By the end of this chapter we will have the prerequisites to consider encoding and decoding of low-density parity-check codes in the following chapters.

2.1 Preliminaries

There are two main types of codes, of which the first adds redundancy to continuous *streams* of data and the second does so to *blocks* of data. We will concern ourselves with the second type of codes, *block codes*, as low-density parity-check codes are of this second type. Block codes work with finite blocks of data, encoding each block of data into a longer block, and each block is *independent* from each other. With this, we can define a block code more formally, and since we will only consider one type of block codes in this thesis we will refer to block codes simply as codes. We will largely follow the notation and terminology of Richardson and Urbanke (2008).

Definition 1 (Code)

Let $\mathbb{F}$ be a finite field. A code *C of* block length *n and cardinality M is a set of $M \geq 2$ elements from $\mathbb{F}^n$.*

We call the elements of a code its *codewords*. In contrast, a *word* refers to a vector which is part of $\mathbb{F}^n$, but is not necessarily a codeword. We will denote vectors, or words, from $\mathbb{F}^n$ by small bold letters, and the ith element of a vector $\mathbf{x}$ by x_i.

Definition 2 (Linear code)

A linear code C is a code which satisfies

$$\alpha\mathbf{x} + \alpha'\mathbf{x}' \in C, \quad \forall \mathbf{x}, \mathbf{x}' \in C \text{ and } \forall \alpha, \alpha' \in \mathbb{F}.$$

Low-density parity-check codes are linear codes. A direct consequence of linearity is that the all-zero word is always a codeword. A second consequence of the linearity of a code is that the error-correcting performance of the code is independent of the sent codeword. Hence we usually assume that the all-zero codeword was sent when examining decoding performance.

The weight of a vector and the distance between two vectors are useful concepts for analyzing error-correcting codes in terms of the *minimum distance of a code.*

Definition 3 (Weight of a vector and distance between two vectors)

The weight *of a vector* $\mathbf{x}$*, denoted by* $w(\mathbf{x})$*, is the number of nonzero entries in* $\mathbf{x}$*. The* distance *between two vectors* $\mathbf{x}$ *and* $\mathbf{x}'$ *is* $d(\mathbf{x}, \mathbf{x}') = w(\mathbf{x} - \mathbf{x}')$.

Definition 4 (Minimum distance of a code)

The minimum distance of a code C is the smallest distance between any two distinct elements of the code. More precisely, the minimum distance of a code is defined as

$$\min_{\substack{\mathbf{x}, \mathbf{x}' \in C \\ \mathbf{x} \neq \mathbf{x}'}} d(\mathbf{x}, \mathbf{x}').$$

The essence of an error-correcting code is that we map a set of shorter strings to a set of longer strings, the codewords. In doing so we can increase the distance between any two elements in the code resulting in the code becoming more robust to errors. The extent to which we have improved the error-correcting properties of a code is partly captured by the minimum distance of the code, as a larger minimum distance means that the code can tolerate more errors while still allowing decoding to the correct codeword.

Although codes can be formulated on larger finite fields, the binary field $\mathbb{F}_2$ consisting of the elements $\{0, 1\}$ is most commonly used. On the binary field the addition operation is the logical XOR operation. More precisely, $0 + 0 = 1 + 1 = 0$ and $1 + 0 = 0 + 1 = 1$. The binary field can also be represented by the elements in the set $\{1, -1\}$. Instead of mod-2 addition, we now use multiplication, meaning $1 \cdot 1 = (-1) \cdot (-1) = 1$ and $(-1) \cdot 1 = 1 \cdot (-1) = -1$. The field will be assumed to be the binary field for the rest of this thesis.

We say that a binary code with M elements has $\log_2 M$ *information bits*, as this is the number of bits of information that we are sending over the channel in each codeword. The rate of a code is then defined as the ratio of information bits in a codeword to the total number of bits in a codeword.

DEFINITION 5 (RATE OF A CODE)

The rate of a code C with block length n and cardinality M is the number of information bits sent over the number of total bits sent. More precisely, the rate $r(C)$ is

$$r(C) = \frac{\log_2 M}{n}.$$

It will later be convenient to consider *ensembles* of codes. They are essentially sets of codes, constructed using some random process, from which codes are chosen at random. Shannon's random ensemble is one example.

EXAMPLE 1 (SHANNON'S RANDOM ENSEMBLE)

Let $\mathrm{Shannon}(n, M)$ *denote Shannon's random ensemble where each code has block length n and M elements. A code is chosen from the ensemble by choosing each of the M codewords uniformly at random from $\mathbb{F}_2^n$.*

2.2 NOISY CHANNELS

Noisy channels are the fundamental reason that error-correcting codes are used. For that reason we will take a small detour to look at channels before continuing to low-density parity-check codes. A *binary channel* is a mapping $\{0, 1\} \to S$ where S can be a finite or infinite set. We will in general denote the sent codeword by $\mathbf{x}$ and the received codeword after transmission over a noisy channel by $\mathbf{y}$.

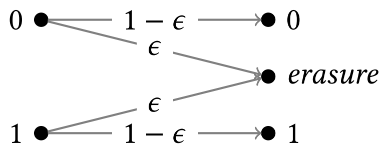


Figure 2.1: The transition diagram for the BEC. The input can have the value 0 or 1, and will after transmission have changed to an erasure with probability ϵ and stayed at the input value with probability $1 - \epsilon$.

The *binary erasure channel* (BEC) simply performs the following operation: with some probability ϵ, a bit becomes an *erasure*. The set of symbols received thus has the added *erasure* symbol. Put differently, on the binary erasure channel we assume that the receiver knows that a bit has been erased but simply does not know what value the erased bit had originally. The transition diagram of the BEC is shown in Figure 2.1.

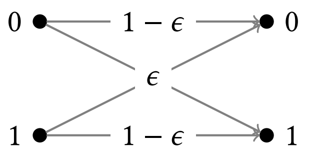


Figure 2.2: The transition diagram for the BSC. The input can have the value 0 or 1, and will after transmission on the BSC have flipped to the other value with probability ϵ.

The *binary symmetric channel* (BSC) has the same input and output symbols and is determined by the parameter ϵ, often called the *crossover probability*. The crossover probability determines the amount of noise in the channel in the sense that a bit passing through the channel is simply flipped from 0 to 1 or from 1 to 0 with probability ϵ. The transition diagram of the BSC is shown in Figure 2.2.

The *binary additive white Gaussian noise channel* (BAWGNC) maps the set of input symbols to the set of real numbers. For the BAWGNC it is convenient to use $\{1, -1\}$ as the set of input symbols. The mapping for the BAWGNC is

$$y_i = x_i + e_i$$

for each bit x_i that we wish to send, where e_i is an independently and normally distributed random variable with zero mean and variance σ^2. That is,

$$e \sim \mathcal{N}(0, \sigma^2).$$

Two quantities are useful when considering the BAWGNC. The *signal-to-noise ratio* (SNR) is the ratio of the energy per transmitted bit E_s to the energy of the noise σ^2. That is, SNR $= \frac{E_s}{\sigma^2}$. Here, $E_s = 1$ because the set of input symbols is $\{1, -1\}$. A related quantity is the ratio of the energy per transmitted information bit $E_b = \frac{E_s}{r}$ to the double-sided power spectral density $N_0 = 2\sigma^2$. Here we have simply written r for the rate of a code C. The quantities are usually shown in dB. That is, they are shown as $10 \log_{10}\left(\frac{E_s}{\sigma^2}\right)$ and $10 \log_{10}\left(\frac{E_s}{2r\sigma^2}\right)$. For codes of rate $\frac{1}{2}$ the two quantities are the same. The BAWGNC is useful as a model for real channels.

2.3 Low-density parity check codes

A *low-density parity-check code*, or LDPC code, is defined by a matrix $H \in \{0,1\}^{m \times n}$. The defining characteristic of LDPC codes is that H is *sparse*. This means that H has $\mathcal{O}(n)$ nonzero elements. Given a matrix H the set of codewords is defined by

$$C = \{\mathbf{x} \in \{0,1\}^n \mid H\mathbf{x}^T = \mathbf{0}^T\},$$

where $\mathbf{0}$ is the zero vector and we assume that all vectors are *row* vectors. The matrix H is referred to as the *parity-check matrix*, since it requires that a codeword $\mathbf{x}$ have even parity in the so-called parity-check equations

$$s_a = \sum_{i=1}^{n} H_{ai} x_i = 0,$$

for all $a = 1, 2, \ldots, m$. Each parity-check equation is often called a *checksum* or *check*, and the vector $\mathbf{s}$ of sums s_a is called the *syndrome*. If a check has even parity we say that the check is *satisfied* and otherwise we say that the check is *unsatisfied*. Assuming that the parity-check matrix of an LDPC code is of full rank, the cardinality of the code is $M = 2^{n-m}$. The rate of an LDPC code with a parity-check matrix of full rank is $\frac{n-m}{n}$. A parity-check matrix uniquely defines a code up to row operations, meaning row permutations and additions of one row to another.

2.3.1 Tanner graph representation

It is often convenient to consider a graph representation of a parity-check matrix. The graph representation of a parity-check matrix is more commonly referred to as its *Tanner graph* (Tanner, 1981). The Tanner graph is a bipartite graph with two sets of nodes. The first set consists of the *check nodes*, each corresponding to the rows, or checks, of H. The second set consists of the *variable nodes*, each corresponding to a column of H, or a symbol of a word. There is an edge between a check node a and a variable node i if and only if $H_{ai} = 1$. Put differently, the Tanner graph corresponding to a parity-check matrix H has an adjacency matrix given by

$$\begin{pmatrix} 0 & H \\ H^T & 0 \end{pmatrix}.$$

We denote the neighbors of a node i by $N(i)$. As a shorthand, the notation $N(i) \setminus j$ means the neighbors of the node i excluding the node j. A parity-check matrix and a Tanner graph are equivalent up to permutation of rows and columns. In general, we will also treat a *code* as equivalent to a parity-check matrix or Tanner graph, even though a code can be defined by more than one parity-check matrix, and likewise by more than one Tanner graph.

2.4 LDPC code constructions

The definition of LDPC codes does not state how the codes should be constructed. Next we will look at some ways of doing so. Most importantly, we will present *regular* and *irregular* code ensembles. We will, in relation to code ensembles, also state two central theorems regarding the performance of codes. In addition, we will present so-called *quasi-cyclic* LDPC codes and briefly look at LDPC codes used in communications standards.

2.4.1 Regular codes

A simple way to construct an LDPC code results in what is called a *regular* code. An (l, r)-regular LDPC code is defined by a parity-check matrix H where each column of H has weight l and each row has weight r. Such a code has, by double counting, $e = mr = nl$ ones in its parity-check matrix or, equivalently, e edges in its Tanner graph representation.

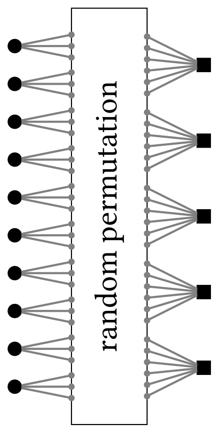


Figure 2.3: Drawing a random bipartite graph with the configuration model. In the above example each node on the left has degree 3 and thus 3 "sockets" indicated by the small gray nodes connected to each node. Likewise, each node on the right has degree 6 and thus 6 sockets. Thus, there are in total $10 \cdot 3 = 5 \cdot 6 = 30$ sockets on each side. We can then, using a random permutation, connect each socket on the left to a unique socket on the right. The result is a bipartite graph which may not be simple. That is, it may have more than one edge connecting a pair of nodes.

One way of constructing a regular code is with the help of a random permutation. For the construction of a regular code it is easiest to think of the code in terms of its graph representation. As the graph is bipartite we know that the two sets of nodes will both have e edges incident to them. We then define each node to have a number of "sockets" equal to the degree of the node. Each socket will be used to attach one end of an edge to it. If we number the sockets from 1 to e on one side (say, the variable nodes) we can then, given a permutation of $1, 2, \ldots, e$, connect the sockets of the check nodes to a socket of a variable node with a number given by the permutation. The model for constructing a graph in such a way is called the *configuration model* (Bollobás, 2001). In this way, we can construct a random bipartite graph but the graph is not necessarily simple. That is, there may be multiple edges between a pair of nodes. However, we can simply draw a new permutation until we get a simple graph. In other words, we perform rejection sampling to get a simple bipartite graph. More importantly, the probability of drawing a simple graph approaches zero exponentially only in the degrees of the nodes and not in the block length n. The probability of drawing a simple bipartite graph approaches a nonzero constant in the block length given fixed degrees (Greenhill et al., 2006).

Example 2 (Parity-check matrix of a (3,6)-regular LDPC code)

The following matrix defines a randomly generated $(3, 6)$*-regular LDPC code of block length 20. That is, parity-check matrix has 3 ones in each column and 6 ones in each row.*

$$H = \begin{pmatrix} 0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&1&0&1&0 \\ 0&1&1&1&0&0&0&0&1&1&0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&1&0&1&0&0&0&1&0&0&0&0&1&0&1&0&0&1 \\ 0&0&0&0&0&1&1&0&0&0&0&0&1&1&0&1&0&1&0&0 \\ 0&1&0&0&1&1&0&0&1&0&0&0&1&0&0&0&0&1&0&0 \\ 0&0&0&0&1&0&1&0&0&0&1&0&0&0&1&0&0&0&1&1 \\ 1&1&1&0&0&0&0&0&1&1&0&0&0&0&0&0&0&1&0&0 \\ 1&0&0&0&0&0&0&1&0&0&1&1&0&0&0&1&1&0&0&0 \\ 0&0&0&1&1&0&1&1&0&0&0&0&0&1&1&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0&0&0&0&1&1&1&0&0&0&0&1&1 \end{pmatrix} \tag{2.1}$$

Example 3 (Tanner graph of a (3,6)-regular code)

The Tanner graph corresponding to the parity-check matrix in (2.1) *is shown in Figure 2.4. The convention for drawing Tanner graphs is to draw the variable nodes as circles on the left-hand side and the check nodes as squares on the right hand side.*

2.4.2 Irregular codes

Irregular codes are a more general class of codes, which include the regular case. The terminology and notation of irregular degree distributions was first presented by Luby et al. (1997). They also presented the idea of *optimizing* the properties of a degree distribution using linear programming. Richardson et al. (2000, 2001) expanded on the work.

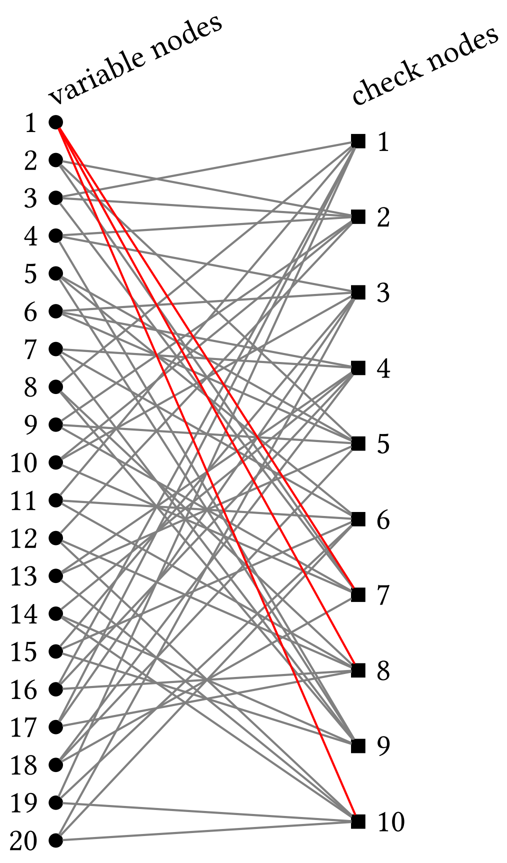


Figure 2.4: The Tanner graph of the parity-check matrix (2.1). The nodes on the left are the variable nodes and the nodes on the right are the check nodes. The edges corresponding to the nonzero entries of the first column of (2.1) have been highlighted in red.

For a code of length n, let Λ_i be the number of variable nodes of degree i. Thus $\sum_i \Lambda_i = n$. Likewise, let P_i be the number of check nodes of degree i, such that $\sum_i P_i = m$. It must hold that the number of edges emanating from both sides are equal, so $\sum_i i\Lambda_i = \sum_i iP_i = e$. For convenience, we represent the degree distributions in terms of the following polynomials:

$$\Lambda(x) = \sum_{i=1}^{d_v} \Lambda_i x^i \quad \text{and} \quad P(x) = \sum_{i=1}^{d_c} P_i x^i,$$

where d_v and d_c are the maximum degrees of the variable and check nodes, respectively. Using this representation we can write

$$\Lambda(1) = n \quad \text{and} \quad P(1) = m.$$

We also define the *normalized* degree distributions

$$L(x) = \frac{\Lambda(x)}{\Lambda(1)} \quad \text{and} \quad R(x) = \frac{P(x)}{P(1)}.$$

The normalized degree distributions give the fraction of nodes with a given degree. That is, a term $L_i x^i$ in $L(x)$ means that there are $\frac{L_i}{n}$ variable nodes of degree i, and likewise for the check nodes.

An (l,r)-regular code is a special case of an irregular code. We can define a (l,r)-regular code with block length n with a degree distribution $(\Lambda(x), P(x)) = (nx^l, \frac{l}{r}nx^r)$. Alternatively, in terms of a normalized degree distribution, the (l,r)-regular code is defined by $(L(x), R(X)) = (x^l, x^r)$.

As in the case of a regular code, we can generate irregular codes by assigning unique names to sockets at each variable node, where the number of sockets at each node is equal to the degree of the node. Using a permutation we then again assign edges to a pair of variable and check nodes. The only difference to the regular case is that the number of sockets at each node need not be constant for the variable and check nodes. The probability of drawing a simple graph again approaches a nonzero constant in the block length n (Greenhill et al., 2006).

2.4.3 Code ensembles and their properties

We have already seen Shannon's ensemble as an example of a code ensemble. We can also, with the help of the construction in the previous section, define an ensemble of irregular LDPC codes. More precisely, an ensemble of *irregular* LDPC codes consists of codes constructed using the configuration model corresponding to the degree distributions $\Lambda(x)$ and $P(x)$, where the permutations are drawn uniformly at random. To further simplify analyzing code ensembles we can make the additional simplification that we do not reject any codes even if they have duplicate edges. Instead we take the number of edges between two nodes modulo 2. This way we may end up with parity-check matrices which are not of full rank, but the probability approaches zero as the block length is increased (Di et al., 2002). We will denote the ensemble of LDPC codes with block length n constructed using the configuration model by $\text{LDPC}(n, \Lambda, P)$.

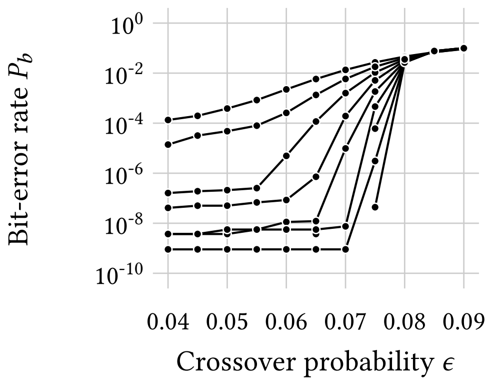
One of the main results is that for an ensemble of irregular codes constructed using the configuration model the codes from the ensemble are concentrated around the average in terms of error-correcting performance. A second, perhaps more important theorem is the *channel coding theorem* by Shannon, which states that there is a limit to the rate at which we can send information over a channel with a given level of noise. We will present the case for the BSC here.

Before we state the theorems, let us introduce some additional terminology commonly used when talking about the performance of codes and decoders. The error-correcting performance is generally stated in terms of the *bit-error rate* of an ensemble of codes paired with a decoder. In the case of codes that have for example been defined in standards, we will of course talk about the bit-error rate of the single code paired with a decoder. The bit-error rate P_b is simply the probability that a transmitted bit has, after decoding, the wrong value. The bit-error rate is often abbreviated BER. One often also talks about the *block-* or *frame-error rate*, which we denote by P_B. This is the probability of a decoded block differing from the transmitted block in at least one bit. This is sometimes abbreviated FER. Figure 2.5 shows typical behavior of an LDPC code in terms of the bit-error rate and the block-error rate, as a function of the noise and the block length. Bits were encoded with codes from the (3,6)-regular ensemble (without duplicate edges) and transmitted over the BSC. The resulting words were decoded using the so-called sum-product algorithm. One can see a clear limit where increasing the block length of the code does not improve error-correction. This is called the *threshold* of the code and decoder, which we will define more precisely with Theorem 2.2. The region where the bit-error rate decreases sharply towards lower levels of noise is called the *waterfall region*. Following that, one can see the so-called *error floor*. The error floor is the flat region at lower levels of noise than the waterfall region. It is typical for LDPC codes to exhibit this error floor, although it is not desirable. The error floor is often caused by some inherent weakness in the code which leaves some bits particularly prone to being erroneous (Richardson, 2003).

With the above terminology, we can now state the theorems more precisely. The theorem regarding concentration around the average code in an ensemble enables us to choose essentially any code from an ensemble and be nearly certain that we have picked a code which is representative of the ensemble in general. We will state the following theorems as presented by Richardson and Urbanke (2008), omitting the proofs.

Theorem 2.1 (Concentration around the ensemble average)

Let a parity-check matrix H be chosen uniformly at random from an ensemble LDPC(n, Λ, P) and let transmission occur over the BEC with erasure probability ϵ. We decode the received word with l iterations of message-passing decoding and let $P_b(H, \epsilon, l)$ denote the final bit-error probability. Then, for a fixed number of iterations l and for any given $\delta > 0$, there exists an $\alpha > 0$, $\alpha = \alpha(\Lambda, P, \epsilon, \delta, l)$, such



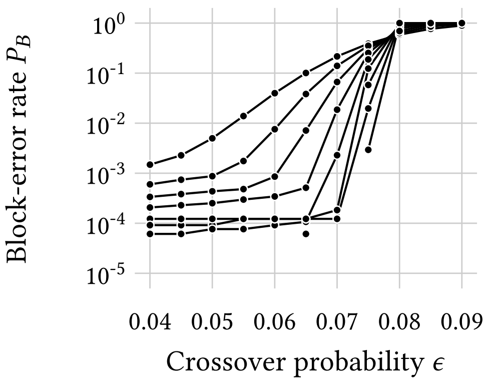


Figure 2.5: Bit-error rate (left) and block-error rate (right) as a function of the crossover probability ϵ on the BSC. The following was repeated for each block length 2^i for $i = 10, 11, \ldots, 20$ until 2^{30} total bits were transmitted: draw a new code from the (3,6)-regular ensemble, assume the all-zero codeword and simulate transmission over the BSC, decode using 20 iterations of the sum-product decoder. The vertical axis is logarithmic. A bit- or block-error rate of 0 is not plotted.

that

$$P\left\{|P_b(H,\epsilon,l) - \mathbb{E}_{H'\in LDPC(n,\Lambda,P)}\left[P_b(H',\epsilon,l)\right]| > \delta\right\} \le e^{-\alpha n}.$$

Shannon's channel coding theorem gives us limits on how much information we can hope to send over a channel. In this setting we consider transmission over the BSC and look at the block error probability after so-called maximum a posteriori decoding. The binary entropy function is defined by

$$h(p) = -p\log_2(p) - (1-p)\log_2(1-p).$$

Theorem 2.2 (Shannon's channel coding theorem for the BSC)

Assume transmission over the BSC with crossover probability ϵ. Let $P_B(C,\epsilon)$ be the block-error rate after transmission on the BSC with crossover probability ϵ using a code C and decoding using maximum a posteriori decoding. If the rate r satisfies $0 < r < 1 - h_2(\epsilon)$ then

$$\min_{C\in Shannon(n,2^{\lfloor rn\rfloor})} P_B(C,\epsilon) \xrightarrow{n\to\infty} 0.$$

The above theorem says that we can only hope to transmit with a bit-error rate that approaches zero if the rate of the code is below $h(\epsilon)$ on the BSC. This is also called the *Shannon capacity* of the channel. Perhaps more importantly, the converse also holds: if we transmit at a rate above the capacity of the channel, the error probability is bounded away from 0 asymptotically in the block length. It is often more convenient to consider the behavior with a fixed rate r and let the crossover probability ϵ vary. In the case that the rate is fixed we want to find the largest ϵ such that the error probability still approaches 0 in the block length. We will refer to this as the *threshold* of the channel. That is, the threshold of the channel gives the limit given any code and decoder with a fixed rate. While the limit given any code and decoder is useful, we will more often consider the threshold of a combination of a code and decoder. With a given code and decoder we then mean by the threshold the largest ϵ such that the error probability approaches 0 in the block length, using the given code and decoder. Equivalent results can be shown for the BEC and BAWGNC (Richardson and Urbanke, 2008).

Luby et al. (1998, 2001a,b, 1997) as well as Richardson et al. (2001); Richardson and Urbanke (2001a) presented a majority of the tools used to analyze random irregular LDPC code ensembles. An important tool is that of *density evolution*, which is a method for determining the threshold under message-passing decoding. An approximation to density evolution was presented by Chung et al. (2001b). Chung et al. (2001a); Richardson et al. (2000) presented optimized degree distributions with thresholds approaching the Shannon capacity and Richardson and Urbanke (2001a) generalized many results to more general channels.

Finally, while Chung et al. (2001a); Richardson et al. (2000) presented ensembles that approach the Shannon capacity, this was done in a setting where the number of iterations and the block length tend to infinity. Clearly, neither of these assumptions are feasible in practice. Degree distributions that approach the Shannon capacity asymptotically can perform badly with a finite number of iterations and finite block lengths. For this reason, designing good codes in the finite setting requires slightly different tools. Richardson et al. (2001) note that density evolution can be applied to some extent to codes of finite length which are decoded using a finite number of iterations. The performance of codes with practical limitations has been studied by Amraoui et al. (2009); Di et al. (2002); Richardson (2003); Richardson et al. (2002), and some ways to construct good finite-length codes are presented by Mao and Banihashemi (2001); Yue et al. (2007). Of particular importance for the error-correcting performance of codes decoded using message-passing algorithms are so-called *stopping sets* and short cycles in the Tanner graph. Since the focus of this thesis is more on encoding and decoding of given codes, rather than code constructions, we will not go into detail about what makes a good code. However, the intuition behind the reason for stopping sets and short cycles being problematic is easy to see. Stopping sets are subsets of the nodes of a Tanner graph in which it is in some sense difficult to resolve what the real values of the variable nodes should be. Variables nodes in a stopping set will therefore often be decoded wrongly, and if the stopping sets are large the error floor of a code will be higher. We will take a closer look at message-passing decoders in Chapter 4 and why short cycles in the Tanner graph are harmful for them.

2.4.4 Quasi-cyclic LDPC codes

Different constructions of *quasi-cyclic* LDPC (QC-LDPC) codes were presented by Tanner et al. (2004) and Myung et al. (2005). The general structure of the parity-check matrix of a QC-LDPC code is the following: H consists of several square submatrices, each of which is either the zero matrix or the identity matrix with the diagonal shifted cyclically to the right by some amount. More precisely, for a code of length n, the parity-check matrix consists of submatrices of size $z \times z$. The code is compactly described by a smaller matrix of size

$$m_M \times n_M = \frac{m}{z} \times \frac{n}{z},$$

where we assume that z divides both m and n. The smaller matrix H_M is called the *model* matrix. Each entry of the model matrix specifies the kind of submatrix at that position in H. A non-negative integer entry specifies an identity matrix shifted by the given entry, and "–" specifies a zero matrix.

Example 4 (WiMAX model matrix)

The rate-$\frac{1}{4}$ code with block length 2304 *and submatrix size* 96 *in the WiMAX standard is specified by the model matrix*

$$H_M = \begin{pmatrix} 6 & 38 & 3 & 93 & - & - & - & 30 & 70 & - & 86 & - & 37 & 38 & 4 & 11 & - & 46 & 48 & 0 & - & - & - & - \\ 62 & 94 & 19 & 84 & - & 92 & 78 & - & 15 & - & - & 92 & - & 45 & 24 & 32 & 30 & - & - & 0 & 0 & - & - & - \\ 71 & - & 55 & - & 12 & 66 & 45 & 79 & - & 78 & - & - & 10 & - & 22 & 55 & 70 & 82 & - & - & 0 & 0 & - & - \\ 38 & 61 & 1 & 66 & 9 & 73 & 47 & 64 & - & 39 & 61 & 43 & - & - & - & - & 95 & 32 & 0 & - & - & 0 & 0 & - \\ - & - & - & - & 32 & 52 & 55 & 80 & 95 & 22 & 6 & 51 & 24 & 90 & 44 & 20 & - & - & - & - & - & - & 0 & 0 \\ - & 61 & 31 & 88 & 20 & - & - & - & 6 & 40 & 56 & 16 & 71 & 53 & - & - & 27 & 26 & 48 & - & - & - & - & 0 \end{pmatrix}. \qquad (2.2)$$

QC-LDPC codes are convenient because they have a compact description. In addition, the same model matrix can be used for a range of block lengths. For example, in the WiMAX standard (IEEE, 2009), the same model matrix defines codes for block lengths from 576 bits to 2304 bits. The description of QC-LDPC codes above still leaves room for choosing the model matrix in different ways. One possibility is presented by Myung et al. (2005). The degrees of the model matrix are retained when it is expanded to the full parity-check matrix and so Myung et al. propose to choose an appropriate degree distribution for the model matrix to obtain a parity-check matrix which has similar properties to an irregular code with the same degree distribution. The positions of the nonzero block matrices and the shifts are chosen to maximize the girth, meaning the length of the shortest cycle, of the resulting parity-check matrix, or more accurately of its Tanner graph.

The parity-check matrix in Example 4 has additional structure on the right-hand side, which ensures that the code can be encoded in linear time. We will present the construction in more detail in the following chapter.

2.5 Codes used in practice

LDPC codes have been included in several standards in the recent years. We review some of them here as they are of interest for decoding purposes. A code, once in a standard, remains fixed for the lifetime of the standard. However, decoders can be changed independently of the codes as inprovements are made to decoders. Thus it is useful to know the structure of codes used in practice, and use them as benchmarks for decoding algorithms.

2.5.1 Digital Video Broadcasting

The Digital Video Broadcasting (DVB) standards for terrestrial, satellite and cable broadcasts (ETSI, 2012, 2013a,b) employ a compact scheme for describing the codes. The DVB codes are defined by o lists of offsets, each of length o_i. The offsets determine the positions of the ones in the parity-check matrix. The parity-check matrix has the following structure:

$$\begin{pmatrix} H_0 & H_1 & \cdots & H_{s-1} & H_p, \end{pmatrix}$$

where H_p is an $m \times m$ matrix with ones only on the full diagonal and in one position below the diagonal. The matrices H_i for $i = 0, 1, \ldots, s-1$ are each of size $m \times 360$, where 360 is a constant defined for all LDPC codes in the DVB standards. The ith list of offsets determines the positions of the ones in H_i. Let b_{ia} denote the ath element of the ith list of offsets. Then, for column j of the full parity-check matrix H, where the column is within a submatrix H_i, the positions of the ones in that column are determined by

$$(b_{ia} + (j \bmod 360)Q) \bmod m, \quad \forall a \in 1, 2, \ldots, o_i.$$

The value Q is another constant defined in the standard which is dependent on the rate of the code. The right-hand side of each parity-check matrix in the DVB standards is lower triangular which, as we will see in the next chapter, means that the code can be encoded in linear time.

2.5.2 WiMAX and WiFi

The WiMAX standard defines a few different code types for error-correction. One of them is LDPC codes (IEEE, 2009). The code type used in the WiMAX standard is a quasi-cyclic code. It defines codes of lengths between 576 and 2304 bits. Example 4 shows the rate-$\frac{1}{4}$ code defined in the standard. The standard has defined codes of rates $\frac{1}{2}$ and $\frac{1}{4}$.

The WiFi standard also uses a QC-LDPC code for error correction. The structure is the same as in the WiMAX standard. The block lengths are 648, 1296 and 1944 bits with rates of $\frac{1}{2}$, $\frac{2}{3}$, $\frac{3}{4}$ and $\frac{5}{6}$.

2.5.3 Ethernet

The Ethernet standard (IEEE, 2012a) also uses LDPC codes for error-correction. Figure 2.6 shows the parity-check matrix of the rate-0.84 LDPC code defined in the Ethernet standard.

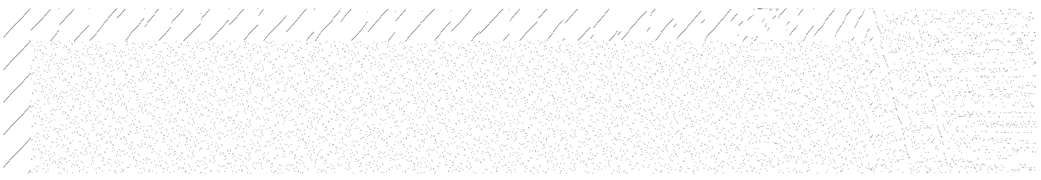


Figure 2.6: The parity-check matrix of the rate-0.84 LDPC code with block length 2048 defined in the Ethernet standard. Each black dot denotes a 1 in the parity-check matrix.

2.6 Summary

We have now presented the basics of codes and in particular LDPC codes, which are a class of binary linear codes with sparse parity-check matrices. Particularly important are the code constructions. The irregular codes based on the configuration model are the basis for codes that can in general be encoded in linear time and perform well under message-passing decoding. The tools developed for irregular codes can also be used for QC-LDPC codes, and in these are used in practice in many standards.

One type of LDPC codes that we have left out are codes based on finite geometries (Kou et al., 2001). Kou et al. have shown that finite geometry codes can have good theoretical properties. However, they often contain high degree variable and check nodes, which increases the complexity significantly (Cho et al., 2010). To the best of the author's knowledge, they have also not been included in any current communications standards.

Chapter 3
Encoding of LDPC codes

Although much of the focus in research of LDPC codes has been on *decoding* of the codes, encoding is equally important in terms of time complexity. For general LDPC codes, encoding can not as of today be done in linear time. The biggest contribution to date is that of Richardson and Urbanke (2001b). They show that LDPC codes with a carefully chosen degree distribution can be encoded in linear time. In addition, they show that these codes are good for message-passing decoders in the sense that they can approach the Shannon limit using message-passing decoders. Simpler code constructions can also yield linear time encoding but may not be as good for error-correction.

Encoding consists of taking k information bits and adding *parity bits* such that the information bits and the parity bits together form a codeword. The parity bits are essentially chosen by first setting the information bits in fixed positions of the codeword. Once the information bits have been fixed, the parity bits can be determined by solving a set of linear equations. The naïve method, which we will present first in Section 3.1, does this by Gaussian elimination. This method is simple but has quadratic time complexity. In Section 3.2 we will present the method by Richardson and Urbanke (2001b) where the rows and columns of a parity-check matrix are only *permuted* such that the system of linear equations we are solving is defined by a nearly triangular matrix. Doing this reduces complexity significantly.

3.1 Encoding using the systematic generator matrix

An LDPC code is defined by its parity-check matrix H of size $m \times n$. By setting H in an appropriate form we can form what is called a *generator matrix* of the code. With the help of the generator matrix G, we can then encode the binary vector $\mathbf{u}$, also called the *information bits*, into a codeword $\mathbf{x}$ by $\mathbf{u}G = \mathbf{x}$. We assume that all vectors are *row* vectors. We let $k = n - m$ be the number of information bits.

Let I_m be the identity matrix of size $m \times m$. To begin with, we transform the parity-check matrix into its *systematic form*

$$H = \begin{pmatrix} -P^T & I_m \end{pmatrix}.$$

A parity-check matrix can always be brought into the systematic form using Gaussian elimination without changing the code. This means that we only use standard row operations: permutations and additions of rows. We can thus for the purposes of encoding assume that H is in systematic form. Now, given H, we denote

$$G = \begin{pmatrix} I_{n-m} & P \end{pmatrix}.$$

Any word generated by the generator matrix G is a codeword, which we can see by checking that the syndrome is the zero vector:

$$\begin{aligned} Hx^T &= H\left(\mathbf{u}G\right)^T \\ &= \begin{pmatrix} -P^T & I_m \end{pmatrix} \left(\mathbf{u}\begin{pmatrix} I_{n-m} & P \end{pmatrix}\right)^T \\ &= \begin{pmatrix} -P^T & I_m \end{pmatrix} \begin{pmatrix} \mathbf{u} & \mathbf{u}P \end{pmatrix}^T \\ &= (-P\mathbf{u} + P\mathbf{u})^T \\ &= \mathbf{0}^T. \end{aligned}$$

We can also see that the resulting codeword is of the form $\mathbf{u}G = \mathbf{x} = (\mathbf{u}\ \mathbf{x}_p) = (\mathbf{x}_s\ \mathbf{x}_p)$. The first part of the codeword, $\mathbf{x}_s$, has length k and is called the systematic part of the codeword. It is also exactly the information bits we wish to send. The second part, $\mathbf{x}_p$, is of length m and contains the *parity bits*. If we assume that the parity-check matrix can be brought into systematic form it implies that the parity-check matrix is of full rank, and if we assume that the parity-check matrix is of full rank there will be a unique codeword for each word we wish to encode. We will for the remainder of this thesis assume that the parity-check matrix is of full rank. As an example, Figure 3.1 shows the generator matrix of the code specified in the Ethernet standard in systematic form.

While this method can be useful for shorter codes, it becomes impractical for larger block lengths. For general LDPC codes, the matrix P will be dense, meaning it will have $O(n^2)$ nonzero elements as a result of the Gaussian elimination. This leads to $O(n^2)$ time complexity for the encoding due to the matrix-vector multiplication. Given that an LDPC code by definition only has a linear number of elements in its parity-check matrix, one could wish that encoding could also be done in linear time. In the next section we will see how this can be done for certain codes.

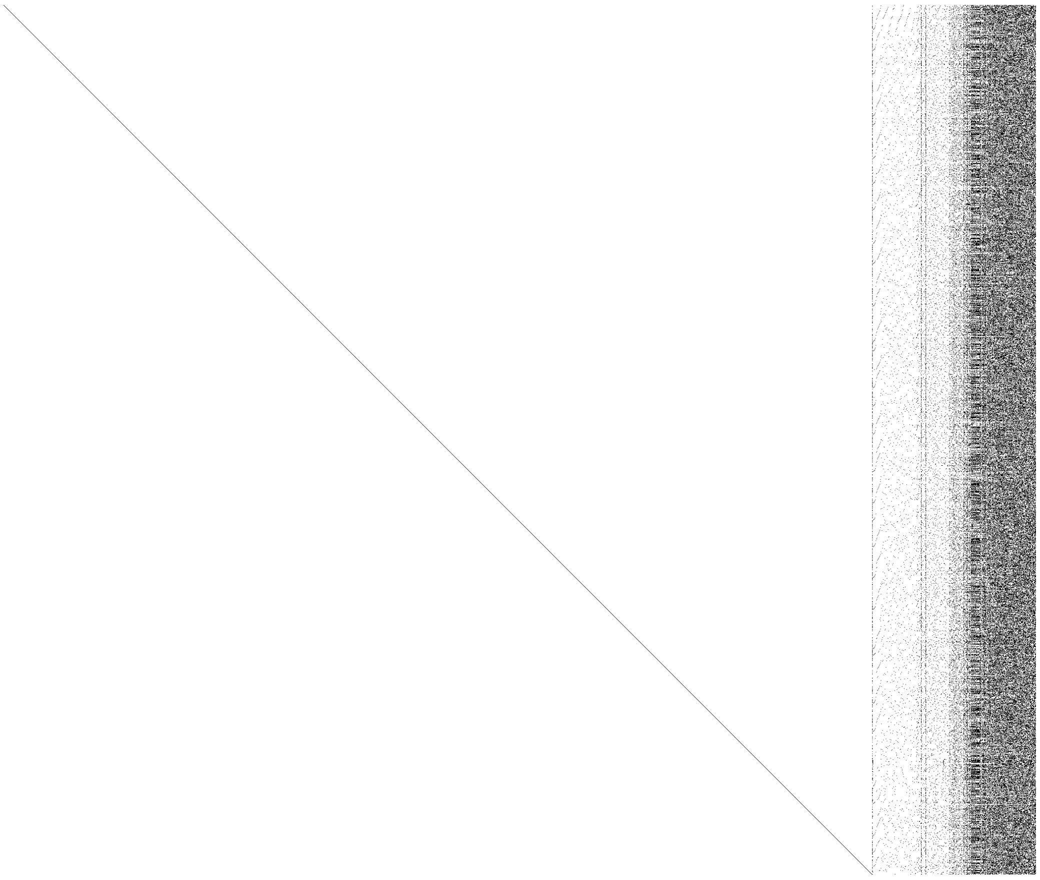


Figure 3.1: The generator matrix of the rate-0.84 LDPC code with block length 2048 defined in the Ethernet standard. Each black dot denotes a 1 in the matrix.

3.2 Encoding with approximate lower-triangular parity-check matrices

MacKay (1999) introduced the idea of LDPC codes that can be encoded in linear time with parity-check matrices that are almost lower triangular. In that case we can perform back-substitution for most of the parity-check bits, but some additional work still has to be done. Richardson and Urbanke (2001b) expanded on this idea and generalized the results on when random irregular LDPC codes can be encoded in linear time. They showed that if the degree distribution of an ensemble of irregular codes is chosen appropriately encoding of a code in that ensemble can in expectation be done in linear time. They also showed that if an ensemble has a degree distribution for which linear time encoding is possible, that ensemble of codes will also behave well under message-passing decoding.

This is a particularly interesting result. While the problem of linear time encoding for general LDPC codes was not solved, the codes that one often wants to use in practice can be encoded in linear time.

However, with the rising popularity of bit-flipping decoders, the above result still calls for an answer to the question whether or not all LDPC codes can be encoded in linear time. Optimal degree distributions for message-passing algorithms do not necessarily lead to optimal performance for bit-flipping algorithms. It is also good to note here that the linear time complexity really only applies to the *encoding* process, and ignores a preprocessing step which needs to be done only once for a code.

The encoding method of Richardson and Urbanke has three steps: (i) a preprocessing step, in which the parity-check matrix is brought into an appropriate form by only column and row permutations, (ii) a second preprocessing step, in which a matrix inverse is calculated; and (iii) the actual encoding step, in which the parity-check matrix from the previous step can be used to encode the information bits $\mathbf{u}$ in linear time. We will begin by looking at the second and third steps, but it is useful to keep in mind that in the first step we only use row and column *permutations*, meaning that the modified parity-check matrix still has a linear number of nonzero elements.

In the encoding step, we assume that the parity-check matrix has been brought into the following form:

$$H = \begin{pmatrix} A & B & T \\ C & D & E \end{pmatrix}.$$

The matrix A is of size $m - g \times k$, B is of size $m - g \times g$, T is of size $m - g \times m - g$, C is of size $g \times k$, D is of size $g \times g$ and E is of size $g \times m - g$. The matrix T is a lower triangular matrix. We say that the parity-check matrix is in an *approximate lower-triangular form.* The parameter g is called the *gap* of the parity-check matrix. If the gap is zero, encoding can be done simply by back-substitution: first set the information bits in the first positions of the codeword after which the parity bits can be solved by back-substitution. If the gap is of size $O(\sqrt{n})$, it turns out that encoding can still be done in linear time. To proceed with the encoding, we first premultiply H by

$$\begin{pmatrix} I_{m-g} & 0 \\ -ET^{-1} & I_g \end{pmatrix}$$

resulting in

$$\begin{pmatrix} A & B & T \\ -ET^{-1}A + C & -ET^{-1}B + D & 0 \end{pmatrix}.$$

As we are premultiplying by an invertible matrix the code remains equivalent to the original code. After this step, encoding corresponds to solving two sets of linear equations:

$$\begin{aligned} A\mathbf{x}_s^T + B\mathbf{x}_{p_1}^T + T\mathbf{x}_{p_2}^T &= \mathbf{0}^T, \\ (-ET^{-1}A + C)\mathbf{x}_s^T + (-ET^{-1}B + D)\mathbf{x}_{p_1}^T &= \mathbf{0}^T. \end{aligned}$$

Similarly to the case of systematic encoding, the codeword is $\mathbf{x} = (\mathbf{x}_s\ \mathbf{x}_{p_1}\ \mathbf{x}_{p_2})$. The parity bits are now split up into two parts, $\mathbf{x}_{p_1}$ and $\mathbf{x}_{p_2}$, where $\mathbf{x}_{p_1}$ is of length g and $\mathbf{x}_{p_1}$ of length $m - g$.

We then define $\phi = -ET^{-1}B + D$ and assume that it is invertible. Then, given ϕ^{-1} we can solve the first set of parity bits by

$$\mathbf{x}_{p_1}^T = -\phi^{-1}(-ET^{-1}A + C)\mathbf{x}_s^t.$$

Once we have solved for $\mathbf{x}_{p_1}$, we can determine $\mathbf{x}_{p_2}$ by back-substitution from

$$T\mathbf{x}_{p_2}^T = -A\mathbf{x}_s^T - B\mathbf{x}_{p_1}^T$$

as T is lower triangular.

In practice, one performs the premultiplication step by Gaussian elimination. The matrix ϕ, however, might not be invertible after performing Gaussian elimination. In that case we permute columns from the left part of the parity-check matrix into ϕ so that it is invertible. If this is in fact not possible, the parity-check matrix is not of full rank, but as we mentioned earlier the probability of this happening decreases with the block length and in practice we need not worry about this.

On the whole, we now have that if the parity-check matrix is in an approximate lower-triangular form, and the matrix inverse ϕ^{-1} has been precalculated we can perform encoding in $O(n + g^2)$ time. Let us see why this is so. First, inverting T need not be done explicitly and can be done by backsubstitution. Additionally, since T is sparse this requires $O(n)$ operations. Second, all matrix-vector multiplications involve sparse matrices with $O(n)$ elements, again requiring $O(n)$ operations. Finally, we can avoid performing matrix-matrix multiplications and instead always perform matrix-vector multiplications. The exception to sparse matrix-vector multiplication is the multiplication by ϕ^{-1} which is a dense matrix-vector multiplication. This contributes the g^2 term in the running time.

What remains is to show that the original parity-check matrix can be brought into the needed approximate lower-triangular form. Richardson and Urbanke do not show a way to find the minimum gap, but propose a greedy algorithm which they show is good enough for achieving linear-time encoding complexity for some codes. The algorithm proceeds in rounds, incrementally building up the right-hand side lower-triangular matrix T. Following the terminology of Richardson and Urbanke (2008) we let two parameters, g and t run during the algorithm. The current gap is given by g and the iteration number is given by t. The *residual* parity-check matrix H is at each step the submatrix of H consisting of rows 1 to $m - g - t - 1$ and columns 1 to $n - t - 1$. The *residual degree* of a column or row in the residual parity-check matrix is equivalent to the *weight* of that column or row in the residual matrix.

The algorithm consists of two main steps, *extend* and *choose*. In the extend step we assume that there exists a column of residual degree one. We choose one column with residual degree one uniformly at random. Let c be the chosen column and r the row containing the only nonzero entry of that residual column. We then swap column c with column $n - t - 1$ and row r with row $m - g - t - 1$. This places the nonzero entry in the lower-right corner of the residual matrix and extends the diagonal on the right-hand side by one. Finally, t is incremented by one.

The choose step is performed when there are no columns with residual degree one. We choose a column uniformly at random from the column or columns with the minimum residual degree d, and call this column c. Then, choose an arbitrary row with a nonzero entry in the residual part of column c and call it r. Swap column c and row r into the lower-right corner of the residual matrix as in the extend step. Move the remaining $d - 1$ rows with nonzero entries in column c to the bottom of the full parity-check matrix. Finally, increment t by one and g by $d - 1$.

The algorithm begins by considering the full parity-check matrix H and setting $t = g = 0$. The algorithm stops when $t + g = m$. If there is at least one column with residual degree one, perform the extend step, and otherwise perform the choose step. At the end of the algorithm, the resulting parity-check matrix is in approximate lower-triangular form with gap g.

Finally, for linear time encoding, we would need to show that the gap g will on average be of size $O(\sqrt{n})$ if we choose the degree distribution appropriately. We will not show how to achieve this here because of the lengthy details, but the general idea is to model the residual degree distributions of the parity-check matrix and the gap using a set of differential equations along the course of the greedy upper triangulation algorithm. This way one can arrive at an asymptotic value for the gap such that elements from the ensemble will have a gap that is close to the asymptotic value with high probability. Although Richardson and Urbanke (2001b) give more detailed conditions on when the average gap will be small enough for linear-time encoding, the intuition is that there needs to exist a large enough number of variable nodes of degree two. When this holds, there will more often exist a column of residual degree one in the extend step meaning that the gap is not increased. If no column of residual degree one exists, the gap will still often be increased by only one in the choose step because of the large number of variable nodes with degree two.

3.2.1 Irregular codes with a fixed gap

Although Richardson and Urbanke (2001b) show that codes with certain irregular degree distributions can be brought to a form where the parity-check matrix has a gap which is proportional to $\sqrt{n}$, and the codes corresponding to those distributions can therefore be encoded in linear time, Freundlich et al. (2007) present an alternative approach to achieve this. Instead of letting all configurations be possible with a given degree distribution pair, they propose to constrain the construction of the code so that it will have a predetermined gap g. They achieve this by essentially fixing the diagonal elements of T in the approximate lower-triangular decomposition to be ones, such that T is of size $m-g$. After that one proceeds to again randomly assign edges between variable and check nodes, but disallowing edges that go above the diagonal in T, enforcing the approximate lower-triangular form. They show through examples that forcing the gap to be proportional to $\sqrt{n}$ does not noticeably impact the performance of the code, while it of course guarantees linear-time encoding. On the other hand, setting the gap to be 0 or close to 0 does decrease the error-correcting performance of the codes.

3.2.2 Quasi-cyclic LDPC codes with approximate lower-triangular parity-check matrices

The encoding method of Richardson and Urbanke was applied to quasi-cyclic codes to achieve linear-time encoding when the cyclic shifts of the submatrices are chosen appropriately (Myung et al., 2005). The construction leads to the matrix ϕ being an identity matrix, meaning that the encoding method becomes a linear-time operation even when one includes preprocessing. For this to hold, the model matrix specifying the parity-check matrix must be of the form

$$H_M = \left(\begin{array}{c|cccccc} & b_1 & 0 & - & \cdots & - & - \\ & - & b_2 & 0 & \cdots & - & - \\ & \vdots & - & b_3 & \cdots & - & - \\ H_I & y & \vdots & \vdots & \cdots & \vdots & \vdots \\ & \vdots & \vdots & \vdots & \cdots & 0 & - \\ & - & - & - & \cdots & b_{m-1} & 0 \\ & x & - & - & \cdots & - & b_m \end{array}\right),$$

where the part of the model matrix corresponding to the information bits H_I can be freely chosen. The shift values x, y and b_i for $i = 1, \ldots, m$ must, however, fulfill one of the following two criteria:

$$x \equiv \sum_{i=1}^{m} b_i \bmod L \quad \text{and} \quad y \equiv -\sum_{i=l+1}^{m} b_i \bmod L \tag{3.1}$$

or

$$\sum_{i=1}^{m} b_i \equiv 0 \bmod L \quad \text{and} \quad x \equiv y + \sum_{i=l+1}^{m} b_i \bmod L\,. \tag{3.2}$$

We will without proof state that if the shift values fulfill one of the two criteria (3.1) or (3.2), then ϕ becomes the identity matrix in which case it is trivially invertible. We also do not need to perform the matrix-vector multiplication which causes the $O(g^2)$ term in the general encoding algorithm.

The authors performed limited tests with these kinds of codes but showed that a QC-LDPC code of this kind performed equally well as a non-quasi-cyclic code with the same degree distribution. The degree distribution of the random code was an optimized distribution obtained by Richardson and Urbanke (2001b). The degree distribution for the quasi-cyclic code was chosen to be the same as for the non-quasi-cyclic code but the coefficients were rounded to fit the block structure of the code. The LDPC codes in the WiMAX standard are quasi-cyclic LDPC codes with the structure presented here allowing them to be encoded in linear time. In the code presented in (2.2), the set of equations in (3.1) is satisfied by setting $b_1 = x = 48$ and $y = b_2 = b_3 = \ldots = b_6 = 0$.

3.3 Summary

While the state of encoding of LDPC codes is in some aspects a solved problem in practice, the lack of a linear-time encoding algorithm for *all* types of LDPC codes still leaves something to be desired. The QC-LDPC construction is a convenient construction that has proven itself to work well enough for inclusion in communications standards. In addition, it allows for a compact implicit description of the parity-check matrix. The irregular ensembles of LDPC codes which can be encoded in linear time have the additional benefit of working well with message-passing decoders. However, being able to use arbitrary LDPC codes that can still be encoded in linear time may allow the use of codes which have better error-correcting capabilities.

Chapter 4 Decoding of LDPC codes

Decoding of LDPC codes has, unlike encoding, essentially been a linear-time operation since Gallager's introduction of LDPC codes. More precisely, there are linear-time algorithms for *approximate* decoding. Optimal decoding, on the other hand, is difficult. Berlekamp et al. (1978) showed that a certain decision problem related to binary linear codes is NP-complete. The approximate schemes are often, however, good enough in practice and can in some cases approach the capacity of the channel asymptotically in the block length n. In addition, any decoding algorithm that is superlinear is bound to be impractical for arbitrarily large block lengths n as in most situations the decoding throughput needs to equal the channel throughput. The reason for wanting to use longer block lengths is the improvement in error-correcting performance as the block length is increased. One of the reasons for this is that a longer code is more robust to noise in the sense that variations in the level of noise are less significant for larger block lengths. For example on the BSC, we can consider the number of bits that are flipped by the channel constant for large enough block lengths. Another aspect is that correlated noise is more likely to corrupt whole blocks if the block length is short. A code with longer block length is robust to longer *bursts* of errors.

One should keep in mind that when we talk about decoding we mostly talk about the process of recovering a codeword given word received from the channel. However, the full process of decoding of course includes recovering the original information bits but, as we saw in Chapter 3, doing this is easy as the information bits will generally explicitly be part of the codeword.

Gallager (1962) introduced two types of decoders: (i) a simple, so called bit-flipping decoder, which was then deemed insufficient in its ability to decode; and (ii) a message-passing decoder which performed much better but was more complex. The bit-flipping decoders and the message-passing decoders are the two major classes of decoders being actively researched at this point. A third class of decoders into which research is being done is that based on linear programming as first presented by Feldman (2003); Feldman et al. (2005). To the best of the author's knowledge, these decoders are to date neither efficient enough nor good enough at correcting errors compared to the two other classes of decoders. However, their benefits lie in easy analysis with established tools from linear programming theory, and work is also being made to reduce the complexity of the decoders (Burshtein, 2009; Burshtein and Goldenberg, 2011; Goldin and Burshtein, 2013; Vontobel and Kötter, 2007). In this thesis, however, we will restrict ourselves to only consider message-passing decoders and bit-flipping decoders.

The message-passing decoders are in general better at correcting errors and were the main type of decoders considered as LDPC codes were rediscovered in the 1990's. However, more recently bit-flipping decoders have received much more attention because of their low complexity which is beneficial for efficient hardware implementations. To clarify, when we talk about complexity in the context of decoders we refer to arithmetic complexity of the decoders, not asymptotic complexity. In addition, the decoders used are in essence linear-time algorithms by choice, and the error-correcting performance and arithmetic complexity is then improved given the constraint that the decoding must be linear-time. The main focus for message-passing decoders has been to decrease complexity without loosing too much in terms of error-correction performance. The focus for bit-flipping decoders has been the opposite: improve the error-correction performance while keeping the complexity low. Currently there is essentially a continuum of decoders all making a trade-off between complexity and error-correction performance.

To be precise, we are assuming an MBRAM machine model for the computations: a random access machine (RAM) with addition, subtraction, multiplication, division and bitwise Boolean instructions, where we assume that all operations take $O(\log n)$-time where n is the size of the largest input operand or output involved in the instructions. In practice, however, we assume that instructions take $O(1)$-time with fixed-width inputs and outputs (van Leeuwen, 1990).

We will look at the decoders roughly in order of complexity. We begin by considering bit-flipping decoders in the next section because of their relative simplicity and to get a first idea of how decoding can be done. We will then move on to message-passing decoders which require some background in graphical models. We will review the most important aspects of *factor graphs* to present the *sum-product* algorithm, which is also known as *belief propagation*. However, the message-passing decoders can be intuitively understood even without the factor graph framework.

4.1 Naïve decoding

Before looking at the practical algorithms, we will note that we can perform exact maximum-likelihood decoding of LDPC codes with the caveat that it has exponential time complexity. For completeness we state here the simple algorithm. If we assume the BSC, maximum-likelihood decoding for a code C simply consists of finding the codeword $\mathbf{x}'$ which is nearest to the received word $\mathbf{y}$. We then hope that the decoded codeword $\mathbf{x}'$ is the same as the sent codeword $\mathbf{x}$. The decoded codeword is chosen to be

$$\mathbf{x}' = \arg\min_{\mathbf{z} \in C} w(\mathbf{z} - \mathbf{y}).$$

Here we clearly have an exponential number of candidates. Trying to be more clever and beginning the search with the words closest to the received word is also of no help, as we want that the code has a minimum distance which increases linearly with the block length. This way we will still end up looking at an exponential number of candidates.

4.2 Bit-flipping decoding

Bit-flipping decoders take a straightforward approach to decoding. Given the received word $\mathbf{y}$ one can start flipping bits in $\mathbf{y}$ according to some appropriate order and criterion in the hope that one will eventually arrive at the original codeword $\mathbf{x}$. In a sense we wish perform a local greedy search in promising directions around the received word. This approach leads at its simplest to low-complexity decoders, which however lack in error-correcting performance compared for example to the sum-product decoder. Adding more information or noise to the decisions of a bit-flipping decoder can already lead to much better performance while keeping the complexity of the decoder fairly low.

4.2.1 Gallager's bit-flipping decoder

Gallager's bit-flipping decoder (Gallager, 1962) is the first and perhaps the simplest bit-flipping decoder for LDPC codes. It proceeds as follows: for each variable node count the number of adjacent check nodes that are unsatisfied or, equivalently, how many parity-check equations in which the variable is involved are unsatisfied; if the number of unsatisfied checks is greater than or equal to some chosen constant K, flip the value of the bit. The algorithm proceeds for a fixed number of rounds through all variables, or until all parity-check equations are satisfied.

There are two main considerations to bit-flipping decoders. The first is how to decide whether a bit should be flipped or not. The second is in which order bits are considered, and if multiple bits are considered at the same time. The second aspect will be referred to as a *schedule* in the following.

4.2.2 Weighted bit-flipping decoders

According to Kou et al. (2001), the essence of the weighted bit-flipping decoder was already introduced by Kolesnik (1971), not long after Gallager's initial work on LDPC codes. Kou et al. (2001) then reintroduced the decoder, and it has been modified and improved in the subsequent years. The weighted bit-flipping decoder refers to the most basic version. However, we will collect all modifications under the same term, and present the decoder below in a general form which includes most modifications. The main difference from Gallager's bit-flipping decoder is that we take *soft* channel values into account. In other words, we assume transmission for example over the BAWGNC such that the channel outputs are real-valued. This allows us to take into account which variable values we can be fairly certain about—those that have large $|y_i|$—and those which could likely have been either -1 or 1—those that have small $|y_i|$. This is in contrast to the BSC, where we only know that *any* bit could have been flipped with the same probability ϵ.

The quantity used for deciding if a bit should be flipped is in its most general form the following:

$$E_i = \sum_{a \in N(i)} (2s_a - 1)w_a - \alpha v_i$$

where w_a and v_i for all $i = 1, 2, \ldots, n$ and for all $a = 1, 2, \ldots, m$ depend on the algorithm. The syndromes s_a are calculated after hard thresholding of the channel values. The parameter $\alpha \geq 0$ is a free parameter chosen separately for best error correcting performance. The sum essentially calculates the number of checksums adjacent to a variable node i that are unsatisfied, weighted by the quantity w_a which is a measure of the reliability of the check node. The term αv_i adjusts for the reliability of the received message at the variable node. In general, a bit is flipped if the quantity E_i is high enough.

We begin by noting that Gallager's bit-flipping decoder fits into this framework, only it does not make use of the weighting. If we set $w_a \equiv 1$ and $\alpha v_i \equiv 0$ and we flip a bit if E_i is larger than some predetermined threshold we essentially get Gallager's bit-flipping decoder. This demonstrates that Gallager's algorithm indeed is perhaps the simplest bit-flipping decoder and sets the stage for the changes and improvements that have been proposed in newer bit-flipping decoders.

Modified weighted bit-flipping decoder

In the modified weighted bit-flipping decoder (Zhang and Fossorier, 2004) the terms are as follows:

$$w_a = \min_{i \in N(a)} |y_i|, \tag{4.1}$$

$$v_i = |y_i|. \tag{4.2}$$

Each term w_a now weights the parity of a check node by the least reliable adjacent variable node of the check node a. This means that if at least one of the variable nodes incident to a check node has a channel value that is close to zero, we will trust the parity of the check node less in making a decision on whether to flip a bit or not. If, however, all variable nodes adjacent to a check node have highly reliable channel values we can trust the value of the check node more and it influences E_i more. The parameter α is left as a free parameter.

Gradient-descent bit-flipping decoder

The bit-flipping decoder can also be presented as a gradient-descent optimizer (Wadayama et al., 2007). In that case

$$w_a = 1,$$
$$v_i = x_i y_i.$$

As $x_i \in \{-1, 1\}$, the term v_i is similar to the form proposed by Zhang and Fossorier (2004). When the value of x_i has not yet been flipped, the signs of x_i and y_i are the same so $x_i y_i = |y_i|$. The difference between the two formulations is when x_i does not have the same sign as the channel value y_i. In that case the E_i is penalized as $-\alpha x_i y_i > 0$. The gradient-descent formulation has a slightly more meaningful interpretation. It maximizes the correlation between the received values $\mathbf{y}$ and the decoded bits $\mathbf{x}'$, penalized by unsatisfied checksums (see appendix A.1 for a derivation of the terms). The free parameter α was not included in the original formulation of the gradient-descent bit-flipping decoder, so $\alpha = 1$ in this case.

Bootstrapped weighted bit-flipping decoder

Bootstrapped weighted bit-flipping (Nouh and Banihashemi, 2002) is a modification of the normal weighted bit-flipping decoder by Kou et al. (2001). The actual decoding steps are identical to the WBF but the initialization of the variable values are modified. A free parameter $\gamma > 0$ is chosen as a threshold. All variable nodes which have a channel value $y_j < \gamma$ are deemed *unreliable*. All other variable nodes are *reliable*. A check node is reliable if all its adjacent variable nodes are reliable and unreliable otherwise. Let $N_r(j)$ be the reliable check node neighbors of a variable node j. The unreliable variable nodes are then re-initialized with the following values

$$y_i \coloneqq y_i + \sum_{a \in N_r(i)} \min_{j \in N(a) \setminus i} |y_j| \prod_{j \in N(a) \setminus i} \operatorname{sgn}(y_j),$$

where $\operatorname{sgn}(x)$ is the sign function which takes the value $\frac{x}{|x|}$ if $|x| > 0$, and 0 otherwise. We can see that the term $\prod_{j \in N(a) \setminus i} \operatorname{sgn}(y_j)$ is essentially a term saying what value the variable node i *should* have according to the check node a. This is again weighted by the smallest reliability of a variable node adjacent to the check node. The unreliable check nodes can be thought of as having 0 as the smallest channel value of an adjacent variable node and are thus excluded from the sum. The channel value of the variable node i is then adjusted according to the reliable check nodes. We will see later that this corresponds to one iteration of *min-sum* decoding, a more powerful message-passing decoder. The authors found that performing this bootstrapping step improved the performance compared to only using bit-flipping, while requiring little additional complexity as the step is only performed on the first iteration. Kou et al. (2001) proposed a similar scheme which they termed hybrid decoding, where one begins with the more powerful sum-product decoder, which we will see in Section 4.3, for a small fixed number of iterations and then continues with a simpler bit-flipping decoder.

4.2.3 Schedules for bit-flipping decoders

The most common *schedule* for actually flipping a bit is the following (as presented by Kou et al. (2001)):

1. If all checksums are satisfied, stop.
2. For all variable nodes, calculate E_i.
3. Flip the value of the variable node for which E_i is highest. That is, flip the value of the node $i = \arg\max_{i \in \{1,2,\ldots,n\}} E_i$.
4. Return to 1.

An alternative schedule is to instead of flipping only the node with the highest E_i flip all variable nodes which have E_i greater than some threshold θ (Wu et al., 2007). The two schedules presented here are convenient to perform in parallel, as one can compute E_i independently for each variable node and then flip each variable node above the threshold independently. We can note here that if $\theta = 0$ and if w_a and v_i are set as in (4.1) and (4.2) the decoder is called a majority-logic decoder (Kou et al., 2001).

A sequential schedule is such that only one variable is considered at a time, and flipped immediately if it is to be flipped. One then continues to the next variable node. This can be done by cycling through all variable nodes for some number of iterations. A more efficient way of doing this is to only ever consider those variables which have at least one unsatisfied checksum with the help of appropriate data structures. This speeds up decoding in a sequential implementation assuming that variables with zero unsatisfied checks should never flipped (Vanek and Farkas, 2009).

The sequential schedule is beneficial in speeding up the convergence of the decoding process. The reason for the improved convergence is that the variable bits are essentially using new information much more often. In a sequential schedule, the variable bits can use the updated values immediately after a single new variable bit has been processed. Compare this with the common approach of comparing all variable bits at a time, where new information is received only after all variable bits have been processed once.

Sequential scheduling for a bit-flipping decoder is called *shuffled* bit-flipping by Zhang et al. (2007). Shuffling refers to a schedule initially proposed for message-passing decoders (Zhang and Fossorier, 2002). Although the sequential schedule is beneficial from the point of view of the absolute number of iterations, it is not possible to properly parallelize a decoder with a sequential schedule. For this reason the parallel schedule, where all bits are considered at a time and some bits are then flipped, is preferred for high-throughput implementations. However, one can also take an intermediate schedule between the two extremes. One can process the variable bits in groups of bits, such that the bits are processed independently in parallel within the groups but sequentially between the groups. This approach was described by Ismail et al. (2013) for bit-flipping decoders. Once again, the approach was first presented for message-passing decoders where it is referred to as parallel shuffled decoding (Zhang and Fossorier, 2002), or more recently and commonly as layered decoding (Hocevar, 2004). Layered decoding was also independently presented by Mansour and Shanbhag (2002), who called it Turbo Decoding Message Passing, due to the relation to the schedule used in the original Turbo decoder (Berrou et al., 1993).

4.2.4 Stochastic bit-flipping decoders

While the deterministic bit-flipping decoders presented above can perform reasonably well, they can often get stuck in local optima where there are still unsatisfied checks and no way to escape from the local optimum. To this end, stochastic bit-flipping decoders add some amount of noise to the decisions to escape local optima and to improve the error-correcting performance of the decoder.

Miladinovic and Fossorier (2005) presented a modified version of Gallager's original bit-flipping decoder. Instead of flipping a bit with certainty when the number of bad checks exceeds some threshold, the decoder only flips these variable nodes with some probability $p < 1$. This does not lead to a significant increase in error-correcting performance, but can lead to faster convergence times and can take the decoding process away from local optima and cycles.

In the same vein, Zhou et al. (2007) presented another bit-flipping decoder only slightly modified from Gallager's original algorithm, but already with more promising results. While strictly deterministic, the algorithm is close to the idea of a stochastic bit-flipping decoder. The algorithm is modified as follows: instead of setting a fixed threshold K such that a variable bit is flipped only if the number of bad checks adjacent to it is at least K, we define a sequence of thresholds $(K_0, K_1, \ldots, K_{T-1})$. A variable bit is then flipped on iteration t if the number of bad checks is at least at the threshold $K_{t \bmod T}$. As an example, the work suggested a sequence $(3, 2)$ meaning every second iteration the threshold is 3 and every second it is 2. In relation to stochastic decoding, this is on average the same as flipping bits with at least 3 bad checks with certainty, and bits with 2 bad checks with probability 0.5. The authors noted a clear improvement compared to Gallager's bit-flipping decoder, but with performance still far from the sum-product and min-sum decoders.

A more recent, and successful, attempt at a stochastic bit-flipping decoder was presented by Sundararajan et al. (2014). They proposed a stochastic version of the gradient-descent bit-flipping decoder where a random, normally distributed term q_i is added to the decision quantity as follows

$$E_i = \sum_{a \in N(i)} (2s_a - 1)w_a - \alpha v_i + q_i,$$

where

$$q_i \sim \mathcal{N}(0, \eta^2)$$

for some variance η^2.

The decoder was presented in both a single-bit and a multi-bit version, meaning that either the bit i with the lowest E_i is flipped, or all bits below a certain threshold are flipped. The authors call the decoder a *noisy gradient-descent bit-flipping* decoder.

As originally presented by Wadayama et al. (2007), one can also flip a fixed number of bits at a time. This was also called a multi-bit version. This type of multi-bit bit-flipping decoder, just like the threshold based multi-bit decoder, tends to converge faster toward the optimum. However, it is prone to oscillations as it approaches the local maximum. The threshold based multi-bit decoder can also be prone to this but to a much smaller degree (Sundararajan et al., 2014). Sundararajan et al. also reported that it can be beneficial to employ a so-called mode-switching strategy, where the decoder begins decoding with the multi-bit decoder, but changes to the single-bit version after some steps to ensure proper convergence. The noisy gradient-descent bit-flipping decoder performed in some cases even comparably to the more complex min-sum decoder which we will present later.

4.2.5 Stochastic bit-flipping decoder with hard channel values

In this thesis we propose a variation of a stochastic bit-flipping decoder which only operates on hard values. That is, it operates on channel values from the BSC or channel values from the BAWGNC quantized to one bit. The idea behind the decoder is similar to the stochastic gradient-descent bit-flipping decoder. The main differences are that this decoder does not use soft channel values and that the probability for flipping a bit is parameterized differently.

In the current variation, which we will from now on call the stochastic bit-flipping decoder, as opposed to the stochastic *gradient-descent* bit-flipping decoder, the decision to flip a bit is based on three factors: (i) the degree of the variable node, if the code is not regular; (ii) the number of unsatisfied checks; and (iii) the channel value of the bit we are considering. Let us denote the degree of the variable node by d, the number of adjacent unsatisfied checks by b and the XOR of the channel value and the current value of the bit by e. The value of e is then 1 if the channel value and the current value differ, and 0 otherwise. Let us denote the degree of a variable node by d. We then decide to flip the current bit with probability some $p_{e,b,d}$. We will assume that we never flip a bit if it has 0 adjacent unsatisfied checks as it will rarely be beneficial. Thus we define $p_{e,b,d}$ for all $e = 0, 1$ and $b = 1, 2, \ldots, d$ and for each variable node degree d occurring in the code. In general, the probabilities $p_{e,b,d}$ should be optimized individually for best error-correcting performance. In practice, performing this optimization can be difficult to perform if the number of probabilities to consider is high. In addition, while for example degree distributions of irregular codes can be optimized fairly efficiently using density evolution for message-passing decoders, no such convenient tools for analyzing the performance of bit-flipping decoders exists yet. For this reason, evaluating the performance of a set of parameters simply requires running the decoder which can be time-consuming. To avoid this we parameterize the flip probabilities. The probability of flipping a bit is

$$\begin{aligned} p_{e,b,d} &= \min\left\{\exp\left(-2\frac{d-2b+\theta(2e-1)}{T}\right), 1\right\}, \\ \theta &= \frac{T}{2}\log\left(\frac{1-p}{p}\right), \end{aligned}$$

where $T > 0$ and $p > 0$ are free parameters. The above parameterization leads to flipping a bit with higher probability if the variable node has several unsatisfied checks as opposed to fewer. In addition, if the current value of the bit differs from the channel value, the bit is flipped with higher probability, resulting in general to the decoder trusting the channel value more than the current value of the bit. Table 4.1 shows an example of the flip probabilities using the above parameterization with the values $T = 0.8$ and $p = 0.12$.

Table 4.1: Flip probabilities for the stochastic bit-flipping decoder when $T = 0.8$ and $p = 0.12$. The values are shown for a variable node of degree 3 with b unsatisfied adjacent checks and e. The value of e is 1 if the current value of the variable is not equal to the channel value. When $b = 0$ a bit is never flipped.

	$e = 1$	$e = 0$
$b = 1$	0.602	0.011
2	1.000	1.000
3	1.000	1.000

While similar to the stochastic gradient-descent bit-flipping decoder, the inspiration for the decoder comes from an algorithm for the satisfiability (SAT) problem presented by Alava et al. (2008). The proposed decoding algorithm can be run with various schedules. In its simplest form it can be run by sequentially considering each bit in the word and flipping the bit if necessary. A main insight in the work by Alava et al. (2008) was that of *focusing* the search in promising directions. They call this algorithm *focused metropolis search* or FMS for SAT. In the current stochastic decoding algorithm this can be implemented by only ever considering such variable nodes which have at least one adjacent check which is unsatisfied. If the variable node has zero unsatisfied checks, the bit is never flipped and so such variable nodes need not be considered. This can speed up the convergence of the decoder, but only applies straightforwardly to a serial implementation. The idea is essentially the same as that presented by Vanek and Farkas (2009). The same downsides apply, namely that of no known convenient way of parallelizing the implementation while using focusing. In Chapter 5 we only consider a sequential schedule using an implementation for the GPU. In a situation where one is constrained to serial execution, the idea of focusing can be beneficial but for most practical decoding purposes, the implementation must be parallel in some way to achieve high enough throughputs.

4.3 Message-passing decoding

Message-passing decoding is, despite the improvements in bit-flipping decoders, still the preferred choice of decoder in hardware implementations as is evident from the comparatively larger amount of hardware implementations published using message-passing decoders. A message-passing decoder is also recommended in the DVB-T standard (ETSI, 2013a). The sum-product decoder, which is a type of message-passing decoder, is still the decoder that in linear time comes closest to an optimal decoder in terms of error-correcting performance.

The class of message-passing decoders operate on the simple principle that messages are sent between the nodes of the Tanner graph, where the messages represent some kind of belief in what the values of the variable bits are. The messages are passed from nodes to their neighboring nodes, where new values are calculated, and new messages are again passed on. Operating in this way, it is possible to construct good decoders for LDPC codes.

Once again, Gallager presented already in 1962 a version of a message-passing decoder for the BSC which is in essence the same as those in use today. We will, however, begin by looking at the sum-product algorithm, which is more generally an algorithm for performing inference on graphical models. The graphical model in the case of decoding of linear codes is roughly the Tanner graph of a code, with some minor additions.

A message-passing decoder most similar to its current form was initially presented for decoding of Turbo codes (Berrou et al., 1993). It was called *iterative* decoding, as it performs several rounds of decoding. The decoding algorithm was then found to be largely identical (McEliece et al., 1998) to what is called belief-propagation in AI, as presented by Pearl (1982). Belief-propagation is the same algorithm as the sum-product algorithm. Hagenauer and Papke (1994) presented the so-called Viterbi algorithm (Viterbi, 1967) for turbo codes and later presented the sum-product algorithm again in a more familiar form (Hagenauer et al., 1996) for general block codes. Wiberg (1996); Wiberg et al. (1995) give a good summary of the connections between the different proposed decoding algorithms at the time when LDPC codes were being rediscovered. Kschischang et al. (2001) present another thorough overview of the various forms of the same algorithm.

Bahl et al. (1974) proposed an optimal decoding algorithm for general linear block codes but noted that it is impractical because of the exponential time and space complexity of the algorithm. Later work has found the same: optimal decoding is NP-hard when the so-called factor graph describing a linear block code has cycles (Lauritzen and Spiegelhalter, 1988). Pearl (1982) also assumed an acyclic graph for belief propagation, the case when belief propagation is optimal. However, it is good to remember here that all the following message-passing algorithms are essentially suboptimal, but still perform well despite the existence of cycles in the graph.

4.3.1 Exact inference on factor graphs and the sum-product algorithm

We will now give a short introduction to factor graphs, where we derive the general sum-product algorithm for factor graphs that are trees, after which we consider the special case of decoding of linear block codes with factor graphs. This will largely follow Kschischang et al. (2001) and Richardson and Urbanke (2008). Another comprehensive work on probabilistic methods in relation to coding is by MacKay (2003).

First we introduce some notation. A function

$$g(x_1, x_2, \ldots, x_n)$$

is written equivalently as

$$g(\mathbf{x}),$$

where $\mathbf{x}$ is the vector of variables $(x_1, x_2, \ldots, x_n)$. We will write the sum over the variables in $\mathbf{x}$ as

$$\sum_{\mathbf{x}} g(\mathbf{x}) = \sum_{x_1, x_2, \ldots, x_n \in S} g(x_1, x_2, \ldots, x_n),$$

where S is the domain of each x_i. The domain is assumed to be the same for all x_i. That is, $\mathbf{x} \in S^n$ where n is the length of $\mathbf{x}$. For example, for binary codes $S = \{0, 1\}$ or $\{-1, 1\}$. We write the *marginal* of $g(\mathbf{x})$ with respect to some variable x_i which appears in $\mathbf{x}$ as

$$\sum_{\mathbf{x} \backslash x_i} g(\mathbf{x}) = \sum_{x_1, x_2, \ldots x_{i-1}, x_{i+1}, \ldots, x_n \in S} g(x_1, x_2, \ldots, x_n),$$

where with slight abuse of notation we mean by $\mathbf{x} \backslash x_i$ the vector $\mathbf{x}$ with x_i removed. We will sometimes write

$$\sum_{\mathbf{x} \backslash x_i} g(\mathbf{x}) = \sum_{\mathbf{x} \backslash x_i} g(x_i, \mathbf{x} \setminus x_i)$$

to make it explicit that the variable with respect to which we are marginalizing appears in $\mathbf{x}$.

Suppose then that we have a function g on the variables $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ which we assume has the *factorization*

$$g(\mathbf{x}) = \prod_{a=1}^{m} g_a(\mathbf{x}_a). \tag{4.3}$$

The $\mathbf{x}_a$ are sub-vectors of $\mathbf{x}$. We can represent the factorization (4.3) of g with the help of a *factor graph.* The factor graph is a bipartite graph consisting of *factor* nodes and *variable* nodes. The factor graph has one factor node for each factor in (4.3), and one variable node for each variable in $\mathbf{x}$. We draw the factor nodes as squares and the variable nodes as circles. A factor node a corresponding to a factor $g_a(\mathbf{x}_a)$ is connected to a variable node i corresponding to a variable x_i by an edge if and only if x_i appears in $\mathbf{x}_a$. In other words, the neighboring variable nodes of a factor node a in the factor graph correspond exactly to the variables which appear in $\mathbf{x}_a$.

Example 5 (Factorization of a function)

The factor graph of the function

$$g(\mathbf{x}) = g(x_1, x_2, x_3, x_4, x_5) = g_1(x_1)g_2(x_2)g_3(x_1, x_2, x_3)g_4(x_3, x_4)g_5(x_3, x_5) \tag{4.4}$$

is shown in Figure 4.1. Note that we have drawn the factor graph laid out as a tree with x_1 chosen as the root.

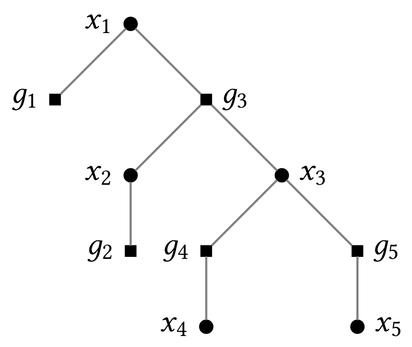


Figure 4.1: The factor graph of the function g in (4.4) drawn as a rooted tree with x_1 as the root.

The general problem we wish to solve in order to perform decoding is to compute the marginal of g with respect to some variable x_i. More precisely, we wish to compute

$$\sum_{\mathbf{x}\setminus x_i} g(\mathbf{x}). \tag{4.5}$$

We will eventually show that so-called *maximum a posteriori* decoding can be formulated as computing a marginal as above. In the case of maximum a posteriori decoding of binary linear block codes each variable takes one of two values and naïvely performing the summation would require summing over 2^{n-1} terms. We will for the case of decoding assume that the variables take the values in $\{-1, 1\}$.

In the case that the factor graph corresponding to g is a *tree*—a connected acyclic graph—we can do the marginalization more efficiently. To see the relation between the marginalization of g with respect to x_i and the factor graph we can draw the factor graph as a *rooted* tree with the variable node i as the root as we have done in Figure 4.1. Figure 4.2 shows the factor graph of a generic function g as a rooted tree. We will use the same notation in the derivation beginning in (4.6) as in Figure 4.2. For a rooted tree with root i, we say that the factor or variable node j is a *child* of the factor or variable node k if k is adjacent to j on the unique path from the root i to j. The node k is then called the *parent* of j. The children of the root node i are then exactly the neighbors of i. The children of a node which is not the root node are the neighbors of the node excluding its parent node. A node j is a *descendant* of a node k if k *lies* on the unique path from the root i to j. A *leaf* node is a node which has degree one. We will denote the vector containing all variable nodes which are descendants of the node i by $\mathbf{z}_i$. If i is a variable node we include x_i in $\mathbf{z}_i$ as well. We will occasionally refer to variable nodes i and factor nodes a by the corresponding variables x_i or factors g_a, respectively, to ease the reading.

Example 6 (Root, parents, children, descendants and leafs of a tree)

In the factor graph shown in Figure 4.1 the variable node x_1 is the root of the tree. The children of the root are the factor nodes g_1 and g_2. The variable node x_5 is a descendant, but not a child, of the variable node x_3. The factor node g_3 is the parent of the variable node x_3. Finally, the variable node x_5 is a leaf node.

To begin with, we consider each child a of the root i as corresponding to a single factor G_a. A factor G_a is a function of the parent variable x_i and all descendant variables $\mathbf{z}_a$. The function g may then also be written as

$$g(\mathbf{x}) = \prod_{a\in N(i)} G_a(x_i, \mathbf{z}_a). \tag{4.6}$$

The factorization (4.6) is in general *not* the same factorization as that in (4.3), but each factor G_a will consist of several factors appearing in (4.3). Only in the case that all factors in (4.3) are functions of the variable x_i with respect to which we are marginalizing are the two factorizations (4.6) and (4.3) equal. Since the factor graph is a tree, the variables in $\mathbf{z}_a$ for all $a \in N(i)$ form a *partition* of the variables in $\mathbf{x} \setminus x_i$, meaning they are pairwise disjoint and contain together all the variables in $\mathbf{x} \setminus x_i$. We can thus rewrite the marginalization using the distributive law as

$$\begin{aligned}\sum_{\mathbf{x}\setminus x_i} g(x_i, \mathbf{x} \setminus x_i) \\ &= \sum_{\mathbf{x}\setminus x_i} \prod_{a\in N(i)} G_a(x_i, \mathbf{z}_a) \\ &= \prod_{a\in N(i)} \sum_{\mathbf{z}_a} G_a(x_i, \mathbf{z}_a). \qquad (4.7)\end{aligned}$$

The result is that the marginal of g with respect to x_i is the product of marginals of each of the factors G_a.

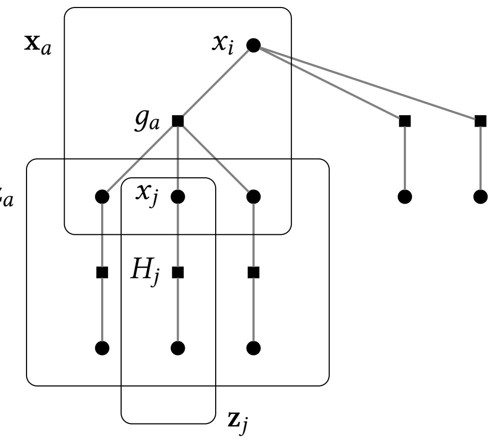


Figure 4.2: The factor graph of a generic function g that factorizes into multiple factors. The variable node neighbors of a are denoted by $\mathbf{x}_a$, the descendant variable nodes of the check node a are denoted by $\mathbf{z}_a$, and the descendants of the variable node j including j itself is denoted by $\mathbf{z}_j$.

Next, let us consider the marginal $\sum_{\mathbf{z}_a} G_a(x_i, \mathbf{z}_a)$ corresponding to one of the children a of the root i. Without loss of generality, G_a can be factorized further. The factor G_a then has a factor which is called the *kernel*. The kernel $H(\mathbf{x}_a) = H(x_i, \mathbf{x}_a \setminus x_i)$ of G_a is the only function in the factorization of G_a which depends on the parent x_i. In addition, it depends on the child variables of a, but not on the descendant variables of a. The kernel is also exactly the factor corresponding to the child a of the root i in the factorization (4.3), so we can write it as $g_a(\mathbf{x}_a)$. In addition, G_a may consist of another set of factors, one for each child variable node j of a. The factor corresponding to the child j is denoted by H_j. Each H_j is a function of the variable x_j—which corresponds to the child j of a—and all descendant variable nodes $\mathbf{z}_j$ of j. The factorization of G_a can then be written as

$$\begin{aligned} G_a(x_i, \mathbf{z}_a) &= H(x_i, \mathbf{x}_a \setminus x_i) \prod_{j \in N(a) \setminus i} H_j(x_j, \mathbf{z}_j) \\ &= g_a(\mathbf{x}_a) \prod_{j \in N(a) \setminus i} H_j(x_j, \mathbf{z}_j). \end{aligned}$$

The marginal

$$\sum_{\mathbf{z}_a} G_a(x_i, \mathbf{z}_a)$$

which we are now considering can then be rewritten as

$$\sum_{\mathbf{z}_a} g_a(\mathbf{x}_a) \prod_{j \in N(a) \setminus i} H_j(x_j, \mathbf{z}_j).$$

We can then use the distributive law again to rewrite the marginal. Doing so we get

$$\begin{aligned} \sum_{\mathbf{z}_a} G_a(x_i, \mathbf{z}_a \setminus x_i) &= \sum_{\mathbf{z}_a} g_a(\mathbf{x}_a) \prod_{j \in N(a) \setminus i} H_j(x_j, \mathbf{z}_j) \\ &= \sum_{\mathbf{x}_a \setminus x_i} g_a(x_i, \mathbf{x}_a \setminus x_i) \prod_{j \in N(a) \setminus i} \sum_{\mathbf{z}_j} H_j(x_j, \mathbf{z}_j). \end{aligned} \tag{4.8}$$

Now we see that the terms $\sum_{\mathbf{z}_j} H_j(x_j, \mathbf{z}_j)$ are of the same form as the form we started with for marginalizing g in (4.5), so we can apply the above recursively to each smaller marginal. Applying this until we reach the leaf nodes we reduce the sums so that each sum is always at most over a number of variables equal to the maximum degree of the graph.

Example 7 (Rearranging the marginal of a function)

We may wish to compute the marginal of $g(\mathbf{x})$ *in* (4.4) *in Example 5 with respect to the variable* x_1. *In that case we can rearrange the sum with the help of the distributive law to get:*

$$\sum_{\mathbf{x} \setminus x_1} g(\mathbf{x}) = g_1(x_1) \sum_{x_2, x_3} g_3(x_1, x_2, x_3) g_2(x_2) \sum_{x_4} g_4(x_3, x_4) \sum_{x_5} g_5(x_3, x_5) \tag{4.9}$$

The factor graph in Figure 4.1 is drawn with x_1 *as the root.*

The decomposition we have just shown in (4.7) and (4.8) gives rise to a convenient way to compute the marginals by sending messages between nodes, namely the sum-product algorithm. We first consider the single marginal of g with respect to x_i that we have been considering thus far. The intuition behind the sum-product algorithm is that since we assumed that the factor graph is a (rooted) tree we can begin at the leaf nodes by sending suitable messages from the leaf nodes to their parents. Once a non-leaf node has received all messages from its children it will compute a new message and pass it to its parent node. Finally, at the root node we simply take the product of the incoming messages.

In the general sum-product algorithm, we send *functions* $f_{a\to i}(x)$ from a factor node a to a variable node i, and functions $v_{i\to a}(x)$ from variable nodes i to factor nodes a. In practice the functions are *tables* of the functions evaluated at the values in the domain of x_i. Let us look at how we can compute a marginal by only passing messages in the following example.

Example 8 (Computing the marginal)

The decomposition of (4.4) *is shown in* (4.9)*. We now compute the marginal of* (4.4) *with respect to* x_1 *by instead passing messages towards the root in the corresponding rooted tree in Figure 4.1. We begin at the leaf variable nodes* x_4 *and* x_5*. At these nodes we will simply send the constant function with value 1 to their respective parent nodes. At the factor nodes* g_4 *and* g_5 *we have now received all messages from their respective child nodes. Then, at the factor nodes* g_4 *and* g_5 *we will compute the marginals* $\sum_{x_4} g_4(x_3, x_4)$ *and* $\sum_{x_5} g_5(x_3, x_5)$*, respectively. That is, we have now at the factor node* g_4 *computed the marginal of the product of the kernel at* g_4 *and the incoming messages from the children of* g_4 *with respect to* x_3*. In this case the message from the single child node was the constant function with value 1. At the factor node corresponding to* g_5 *we have done the corresponding operations.*

The newly computed marginals at g_3 *and* g_4 *are then sent to the common parent node* x_3*. The variable node* x_3 *has now received messages from all its children. At* x_3 *we then compute the pointwise product of the messages from its children, that is*

$$\sum_{x_4} g_4(x_3, x_4) \sum_{x_5} g_5(x_3, x_5).$$

We can see that we have now computed the rightmost two sums in the decomposition in (4.9).

We continue with the leaf node g_2*. In this case the node is a factor node, so we simply send the function itself to its parent. The variable node* x_2 *then receives* g_2 *and computes the pointwise product of the messages coming from its child nodes. Since it only has one child, it simply passes* g_2 *on to its parent* g_3*. Now* g_3 *has received all its incoming messages. At* g_3 *we then again take the product of the incoming messages* *from its child nodes and the kernel and marginalize the product with respect to the parent node x_1. At g_1 we again simply send the function itself since it is a factor node and leaf node. Finally, at x_1 we only need to take the pointwise product of the two incoming messages from g_1 and g_3 and we will have computed the marginal* (4.4).

Following the intuition shown in Example 8, we see that the correct way to send a message from a leaf node that is a variable node to its parent node is to send the constant function with value 1. At a leaf node that is a factor node we instead send the factor corresponding to the leaf node. At a non-leaf, non-root variable node j we perform a pointwise multiplication of the incoming functions from the child factors and send the function to the parent factor a. More precisely, we send

$$v_{j\to a}(x_j) = \prod_{b\in N(j)\setminus a} f_{b\to j}(x_j) \tag{4.10}$$

from j to a. At a non-leaf factor node a we take the pointwise product of the kernel g_a and the incoming messages $v_{k\to a}$, all of which are functions of x_j, the parent variable of a. We then marginalize the product with respect to x_j. More precisely, we send

$$f_{a\to j}(x_j) = \sum_{\mathbf{x}_a\setminus x_j} g_a(\mathbf{x}_a \setminus x_j) \prod_{k\in N(a)\setminus j} v_{k\to a}(x_j).$$

from a to j. To finish the marginalization we take at the root node i the pointwise product of all incoming messages. We get

$$\sum_{\mathbf{x}\setminus x_i} g(\mathbf{x}) = \prod_{b\in N(i)} f_{b\to i}(x_i). \tag{4.11}$$

We can thus using the rules in (4.10) and (4.11) compute the marginal of a generic function $g(\mathbf{x})$ with respect to a variable x_i.

In general we often want to compute the marginal of g with respect to *all* variables in $\mathbf{x}$ in turn. In such a case we could naïvely perform the above message-passing rules separately for each variable. A better way to do it is to *reuse* messages, since many of them will be identical for marginals of g with respect to different variables. To do this, we will again follow the rules in (4.10) and (4.11) but we will not have a designated root node. We will begin by passing messages from each leaf node to its only neighbor. At a non-leaf node we send messages to *all* its neighbors. More specifically, at a variable node i we will send a message to a neighboring factor node a only when all incoming messages from the factor nodes $b \in N(i) \setminus a$ have arrived at i. The same holds for messages sent from non-leaf factor nodes to variable nodes. Note, however, that we do not need to send messages to leaf nodes that are factor nodes since we only perform final marginalization at variable nodes. Proceeding this way we can send at most one message in each direction of an edge after which we can perform the final marginalization step (4.11) at each variable node separately.

4.3.2 The sum-product algorithm for decoding

Let us return to the actual problem at hand, namely that of decoding of LDPC codes. What we wish to compute, for each bit in a received word, is the *maximum a posteriori* (MAP) estimate. That is, we want to choose

$$\begin{aligned}\hat{\mathrm{x}}_i &= \arg\max_{x_i} p(x_i|\mathrm{y}) \\ &= \arg\max_{x_i} \sum_{\mathrm{x}\setminus x_i} p(\mathrm{x}|\mathrm{y}) \\ &= \arg\max_{x_i} \sum_{\mathrm{x}\setminus x_i} p(\mathrm{x})p(\mathrm{y}|\mathrm{x}) \qquad (4.12)\end{aligned}$$

as the value for the ith bit. The vector y contains the channel values $(y_1, y_2, \ldots, y_n)$. The posterior probability $p(\mathrm{x}|\mathrm{y})$ follows from rewriting the joint probability $p(\mathrm{x},\mathrm{y})$ in two ways in terms of conditional probabilities as

$$p(\mathrm{x},\mathrm{y}) = p(\mathrm{x}|\mathrm{y})p(\mathrm{y}) = p(\mathrm{y}|\mathrm{x})p(\mathrm{y}).$$

Rearranging the second equality we get the posterior probability:

$$p(\mathrm{x}|\mathrm{y}) = \frac{p(\mathrm{y}|\mathrm{x})p(\mathrm{x})}{p(\mathrm{y})}.$$

This is also known as Bayes' theorem. The important parts of the posterior are (i) the *prior* $p(\mathrm{x})$, and (ii) the *likelihood* $p(\mathrm{y}|\mathrm{x})$. The denominator $p(\mathrm{y})$ is a constant with the given channel values y and serves as a normalization term. When maximizing (or minimizing) we can ignore constant scaling factors, so $p(\mathrm{y})$ is left out in the MAP estimate in (4.12). Thus we only need to compute the prior and the likelihood.

For the prior of a sent word we assume that each codeword is equally likely to have been sent and a non-codeword is never sent. In that case the prior is equal to the *characteristic function* of a code C, again ignoring a constant scaling factor. We will use Iverson's bracket notation in the following. An expression $[P]$ is 1 if the proposition P is true and 0 otherwise. The characteristic function of a code C is defined for all $\mathbf{x} \in \{0,1\}^n$ by

$$\begin{aligned}\chi_C(\mathbf{x}) &= \prod_{a=1}^{m} [\textstyle\prod_{i\in N(a)} x_i = 1] \\ &\propto p(\mathbf{x}),\end{aligned}$$

The function $\chi(\mathbf{x})$ is then 1 if and only if the modulo-2 sum of each parity-check is 0. We see that the Tanner graph of a code is exactly the factor graph of the characteristic function of a code, with the convention of square nodes for factors and circular nodes for variables. The likelihood is

$$p(\mathbf{y}|\mathbf{x}) = \prod_{i=1}^{n} p(y_i|x_i)$$

since we assume that the noise of each received channel value y_i is independent. Note that the likelihood is a function of $\mathbf{x}$ only since the channel values $\mathbf{y}$ are given. Finally, the posterior probabilities are given by

$$\begin{aligned}p(\mathbf{x}|\mathbf{y}) &\propto p(\mathbf{x})p(\mathbf{y}|\mathbf{x}) \\ &= \prod_{a=1}^{m} [\textstyle\prod_{i\in N(a)} x_i = 1] \cdot \prod_{i=1}^{n} p(y_i|x_i).\end{aligned}$$

The decision for a single variable i is then

$$\hat{x}_i = \arg\max_{x_i} \sum_{\mathbf{x}\setminus x_i} \prod_{a=1}^{m} [\textstyle\prod_{j\in N(a)} x_j = 1] \cdot \prod_{j=1}^{n} p(y_j|x_j).$$

This way we choose for each bit the value that is most likely given the channel values and we minimize the bit error rate. Notice that we now have two sets of factors: one set corresponding to the characteristic function and one set corresponding to the likelihoods of the channel values. Thus the factor graph needs to be slightly modified from the plain Tanner graph. We add a factor node representing the factor $p(y_i|x_i)$ for all $i = 1, 2, \ldots, n$. We will call the factors $p(y_i|x_i)$ the *channel factors* and the factors $[\prod_{i\in N(a)} x_i = 1]$ the *check factors*. The factor graph used in decoding of the code in (2.1) is shown in Figure 4.3.

We now know that if the factor graph of the code is a tree we can perform MAP decoding using the message-passing rules presented in (4.10) and (4.11). Following the message-passing rules, the variable-to-check messages for decoding are the same as the generic variable-to-factor messages as stated in (4.10), namely

$$v_{i\to a}(x_i) = \prod_{b\in N(i)\setminus a} f_{b\to i}(x_i). \tag{4.13}$$

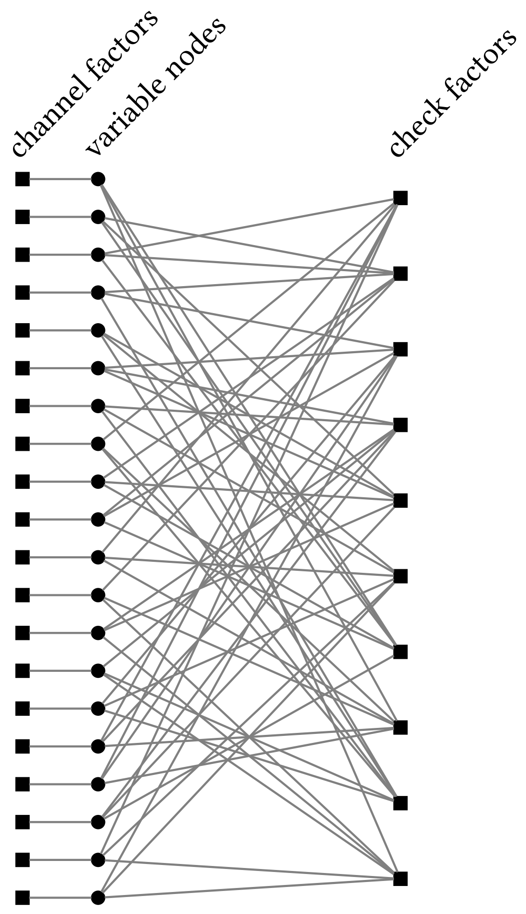


Figure 4.3: The factor graph used in decoding of the code in (2.1).

Note that since the channel factors are leaf nodes, the messages in (4.13) are only sent from variable nodes to *check factors*. For the factor-to-variable messages we make a distinction between the channel factors and check factors, and state separate message-passing rules for them. The check-to-variable messages are

$$f_{a\to i}(x_i) = \sum_{\mathbf{x}_a \setminus x_i} \left[\textstyle\prod_{j\in N(a)} x_j = 1\right] \textstyle\prod_{j\in N(a)\setminus i} v_{j\to a}(x_j). \tag{4.14}$$

We will denote a message going from a channel factor to a variable node i simply by $f_i(x_i)$ as each channel factor is connected to only one variable node. We send the message

$$f_i(x_i) = p(y_i|x_i)$$

to a variable node i from the channel factor adjacent to i.

Unfortunately in the case of LDPC codes, the factor graphs are in general not trees. The cycles in a factor graph which is not a tree prevent us from directly applying the sum-product update rules meant for trees in (4.10)–(4.11). To do exact inference on factor graphs with cycles one can for example use the junction tree algorithm, which produces a new acyclic graph on which we can again use the sum-product algorithm. This, however, is not practical in general for graphs with cycles (Lauritzen and Spiegelhalter, 1988). Another option is to ignore the fact that the sum-product algorithm should be applied only when the graph is acyclic, and apply the sum-product rules to a cyclic factor graph nonetheless. This is, in fact, the approach taken in decoding and has been shown to lead to good results in practice, assuming that the graph does not contain many short cycles. We have just concluded that the sum-product update rules cannot be applied directly to a cyclic graph. What we can do, however, is to first *initialize* all messages to some value, after which all incoming messages at each node are defined and we can apply the usual sum-product update rules. This way of applying the sum-product to a factor graph with cycles is sometimes called *loopy belief propagation*. If the Tanner graph has many short cycles, a message sent along an edge on a short cycle will more often use information that has been sent along the same edge earlier, meaning that the message uses less extrinsic information compared to messages that are sent along edges that do not belong to short cycles.

To use the sum-product algorithm in decoding of a code whose factor graph has cycles we initialize the messages as follows. First, we set the outgoing messages at each check factor to zero. Second, we set the outgoing message at each channel factor to $p(y_i|x_i)$. We do not need to initialize the outgoing messages at the variable nodes. At this point we have defined all incoming messages at each variable node. We can then proceed with the normal sum-product update rules. At each variable node we use the update rule (4.13) to send messages to the check factors. Following that we update the check-to-variable messages using (4.14). We then alternate between sending all check-to-variable messages and all variable-to-check messages for a fixed number of rounds, where each round consists of updating all check-to-variable and all variable-to-check messages once. Once we have updated the messages for a fixed number of rounds we perform the final marginalization step as in (4.11). Finally, as an estimate $\mathbf{x}'$ of the sent codeword we set

$$x_i' = \operatorname*{arg\,max}_{x_i \in \{1,-1\}} \prod_{b \in N(i)} f_{b \to i}(x_i), \quad \forall i = 1, 2, \ldots, n. \tag{4.15}$$

Note that we denote the estimate of the sent codeword by $\mathbf{x}'$ to differentiate it from the true MAP estimate $\hat{\mathbf{x}}$ as the two will generally not be the same when the factor graph has cycles. Alternatively, we can calculate (4.15) in each round when we update the variable-to-check messages. If the estimate $\mathbf{x}'$ forms a codeword we can stop the decoding process early and return $\mathbf{x}'$ as the final estimate.

In the case of decoding of binary linear codes we can further simplify the sum-product algorithm. Instead of sending functions as messages, it is enough to send a single scalar as a message and this will be equivalent to sending functions. To arrive at this we will use the *log-ratios* of the function values that we are sending as messages. More precisely, we will at each channel factor initialize the outgoing message to the logarithm of the ratio of the two function values as follows

$$f_i = \ln\left(\frac{p(x_i = 1|y_i)}{p(x_i = -1|y_i)}\right)$$

such that f_i is now a single scalar. Using log-ratios also for the variable-to-check and check-to-variable messages we get what is called the tanh-rule for the sum-product algorithm. The tanh-rule modifies the check-to-variable messages so that a message sent from a check factor a to a variable node i is given by

$$f_{a\to i} = 2\tanh^{-1}\left(\prod_{j\in N(a)\setminus i} \tanh\left(\frac{v_{j\to a}}{2}\right)\right). \tag{4.16}$$

A message sent from a variable node i to a check factor a is given by

$$v_{i\to a} = f_i + \sum_{b\in N(i)\setminus a} f_{b\to i}. \tag{4.17}$$

The derivation of the tanh-rule is shown in Appendix A.2. We can also factor out the signs in the check-to-variable (4.16) updates to get

$$f_{a\to i} = 2\prod_{j\in N(a)\setminus i} \operatorname{sgn}(v_{j\to a}) \cdot \tanh^{-1}\left(\prod_{j\in N(a)\setminus i} \tanh\left(\frac{|v_{j\to a}|}{2}\right)\right). \tag{4.18}$$

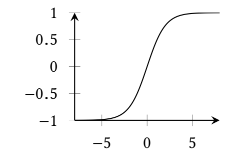


Figure 4.4: The tanh function.

The form of the tanh-rule with the signs factored out makes it especially convenient to interpret the update rules. In the check-to-variable messages (4.18), the product of signs carries the message of which sign the receiving variable node *should* have according to the check factor sending the message as we are leaving out the sign of the message from the receiving variable node. The second part of the check-to-variable rule in (4.18) can be thought of as carrying the *reliability* of the adjacent variable nodes, again excluding the one we are sending to. Excluding the values of the nodes we are sending messages to can be thought of as avoiding using the values between two nodes too often, and instead relying on *extrinsic* information coming from the other adjacent nodes. The variable-to-check updates (4.17) can be seen as the check nodes voting on what value the variable node should have, weighted by the message magnitudes or reliabilities.

When using log-ratios as messages the estimate $\mathbf{x}'$ of the transmitted codeword is set to

$$x_i' = \begin{cases} 1, & \text{if } v_{i\to a} = f_i + \sum_{b\in N(i)} f_{b\to i} > 0 \\ -1, & \text{otherwise,} \end{cases} \tag{4.19}$$

for all $i = 1, 2, \ldots, n$. In (4.19) we use all the incoming messages at the variable node to estimate the transmitted word. Here it is good to remember that even the exact MAP solution does not necessarily give a codeword as the estimate. This is because we are minimizing the bit error rate for each bit *individually*.

Using log-ratios is better in practice as it avoids underflow of floating point values when many small values are multiplied. There are other variations of calculating the messages which also simplify the original sum-product update rules. Chen et al. (2005) present a few formulations of the sum-product decoder for linear codes.

A similar formulation was derived by Gallager (1963) completely independently of the factor graph framework, but also using log-ratios as messages. Only the check-to-variable messages are different from the tanh-rule in (4.16). The update rule is

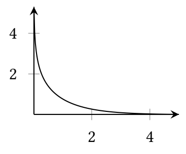


Figure 4.5: The function $g(x) = \ln\left(\frac{e^x+1}{e^x-1}\right)$.

$$f_{a\to i} = \prod_{j\in N(a)\setminus i} \operatorname{sgn}(v_{j\to a}) \cdot g\left(\prod_{j\in N(a)\setminus i} g\left(\frac{|v_{j\to a}|}{2}\right)\right),$$

where $g(x) = \ln\left(\frac{e^x+1}{e^x-1}\right)$ and is defined for $x > 0$. This function is an *involution*, meaning that it satisfies $g(g(x)) = x$. Although the evaluation of $g(x)$ requires more work than the tanh and $\tanh^{-1}$ functions, it is convenient if evaluated with the help of a look-up table, as one only needs one look-up table. This form of the check node update rule also follows easily from the derivation for the tanh-rule.

4.3.3 Implementing the sum-product decoder

While research on bit-flipping decoders has focused on improving the error-correcting performance of the decoders, the focus has been on reducing complexity for message-passing decoders. For example, the full sum-product decoder requires multiplication of floating-point values and the evaluation of the tanh and $\tanh^{-1}$ functions, which are all costly operations in hardware compared to for example simple XOR operations. For this reason approximations have to be made when implementing these decoders. An additional source of complexity is that messages are generally stored for each *edge* of the Tanner graph, as opposed to storing values only for each variable *node* with bit-flipping decoders.

Quantized symbol alphabet

The first approximation one is forced to make is the use of finite-precision values for the messages. While an exact implementation would require infinite precision, in practice the precision is limited to only 32 or 64 bits at most as a consequence of the binary32 and binary64 floating point specifications (IEEE, 2008). However, it turns out that one does not necessarily need more than 4–8 bits of precision for the messages, as verified by a large number of works on message-passing decoders, for example by He et al. (2003); Xiao et al. (2008); Zhang et al. (2001); Zhao et al. (2005). Hardware implementations also generally use far fewer than 32 or 64 bits for the message values. Using fewer message bits already reduces the complexity of the decoders significantly.

The min-sum decoder

The min-sum decoder is essentially a lower-complexity version of the sum-product decoder. In some sense it can be seen as the sum-product decoder with a different set of operators (Richardson and Urbanke, 2008). Alternatively, it can be seen as an approximation to the sum-product decoder (Fossorier et al., 1999). The min-sum decoder performs (approximate) *blockwise* MAP decoding, as opposed to (approximate) bitwise MAP decoding in the case of the sum-product decoder. Since the min-sum algorithm can be seen as an approximation to the sum-product algorithm, which is the same as belief propagation, it is also sometimes called BP-based decoding. It was presented in the context of decoding by Chung (2000); Fossorier et al. (1999); Wiberg (1996), but special cases such as the Viterbi algorithm had been presented earlier. The update rules are:

$$\begin{aligned} v_{i\to a} &= f_i + \sum_{b\in N(i)\setminus a} f_{b\to i}, \\ f_{a\to i} &= 2 \prod_{j\in N(a)\setminus i} \operatorname{sgn}(v_{j\to a}) \cdot \min_{j\in N(a)\setminus i} |v_{j\to a}|. \end{aligned}$$

The only difference to the sum-product decoder is that the magnitudes of the check-to-variable message are determined only by the magnitude of the smallest incoming message. The variable-to-check messages are identical to the sum-product updates in (4.17). The final estimate for the received word is as given in (4.19).

The offset and normalized min-sum decoders

The min-sum decoder is less complex than the sum-product decoder, but it comes with a cost in error-correcting performance. Some work has been put in to improve the performance of the min-sum decoder. Chen et al. (2005); Chen and Fossorier (2002a,b) presented two improvements to the min-sum decoder, both of which work to minimize the effect of overestimating the message values. Yazdani et al. (2004) presented essentially the same modifications but for the sum-product decoder. Both modifications apply to the check-to-variable updates. The two modifications are termed *offset* and *normalized* BP-based decoding. The *normalization* modification is

$$f_{a\to i} = \alpha \prod_{j\in N(a)\setminus i} \operatorname{sgn}(v_{j\to a}) \cdot \min_{j\in N(a)\setminus i} |v_{j\to a}|. \tag{4.20}$$

The messages are simply scaled by a factor α, which is set to a value which results in the best decoding performance. The *offset* modification is equally simple:

$$f_{a\to i} = \prod_{j\in N(a)\setminus i} \operatorname{sgn}(v_{j\to a}) \cdot \max\left\{ \min_{j\in N(a)\setminus i} |v_{j\to a}| - \beta, 0 \right\}.$$

In effect, values smaller than β are set to zero, while all larger values are shifted down by β.

λ-min decoder

Boutillon et al. (2003) presented an alternative modification to the min-sum decoder: the λ-min decoder. Once again, the modification is simple but can have a significant impact on the performance. This of course comes at a cost in arithmetic complexity. The modification consists of using the λ smallest messages for calculating the magnitude of the message instead of only the smallest, as in the min-sum algorithm, or all messages, as in the sum-product decoder. Let $\lambda > 1$ and let $N_\lambda(j)$ be the set of indices of the λ smallest incoming messages to node j. The check-to-variable updates are then

$$f_{a\to i} = \prod_{j\in N_\lambda(a)\setminus i} \operatorname{sgn}(v_{j\to a}) \cdot \prod_{j\in N_\lambda(a)\setminus i} |v_{j\to a}|. \tag{4.21}$$

Successive relaxation

The idea of *successive relaxation* in message-passing decoders was first considered for LDPC codes in the context of analog decoders by Hemati and Banihashemi (2006) and later more thoroughly investigated in terms of decoding performance by Xiao et al. (2008). The idea behind successive relaxation comes from the fact that while on cycle free graphs the sum-product decoder is exact, on graphs with cycles it is not guaranteed to be optimal. The values of the messages—the beliefs—are overestimated due to dependencies between messages caused by cycles, leading to suboptimal performance. The offset and normalization modifications to the min-sum decoder in (4.20) and (4.21) essentially attempt to solve the same problem. The idea behind successive relaxation is to only gradually change the message values. The update rules for the sum-product decoder with log-ratio messages are

$$f_{a\to i} := f_{a\to i} + \beta\left(2\prod_{j\in N(a)\setminus i}\operatorname{sgn}(v_{j\to a})\cdot\tanh^{-1}\left(\prod_{j\in N(a)\setminus i}\tanh\left(\frac{|v_{j\to a}|}{2}\right)\right) - f_{a\to i}\right),$$

$$v_{i\to a} := v_{i\to a} + \beta\left(f_i + \sum_{b\in N(i)\setminus a} f_{b\to i} - v_{i\to a}\right).$$

Here β is a free parameter chosen to be between 0 and 1. It is easy to see that when $\beta = 1$ this corresponds to the usual sum-product tanh update rules, which is also called *successive substitution*. When $\beta < 1$ the messages are only adjusted by a fraction β towards the usual new values. This works to alleviate the overestimation of the messages. The result is that for some of the codes tested even the min-sum decoder with successive relaxation outperforms the standard sum-product decoder. The sum-product decoder with successive relaxation performs at least as well as the min-sum decoder with successive relaxation and better than all decoders with successive substitution.

4.3.4 Binary message-passing decoder

A *differential decoding with binary message-passing*, or DD-BMP, decoder was presented by Mobini et al. (2009), based on the ideas of successive relaxation applied to a binary message-passing decoder. This approach takes the message-passing algorithms to the lower end of the complexity spectrum, with one version of the decoder being similar in complexity to some of the bit-flipping decoders. The DD-BMP operates by only sending binary messages along the edges, while the variable nodes have a memory with one or more bits.

The DD-BMP introduces a *memory* $c_{i\to a}$ for each edge going from a variable node i to a check node a. The memory has multiple bits to represent its value, while the messages sent consist of only a single bit. The messages take values from $\{1,-1\}$. The check-to-variable update is

$$f_{a\to i} = \prod_{j\in N(a)\setminus i} v_{j\to a}.$$

At the variable nodes, a memory is updated as

$$c_{i\to a} = c_{i\to a} + w \cdot \sum_{b\in N(i)\setminus a} f_{b\to i},$$

where w is now a free weight parameter. Having computed the memory, the variable-to-check update is then

$$v_{i\to a} = \operatorname{sgn}\left(c_{i\to a}\right).$$

The *memory* c is initially set to the quantized value of the channel output. The memory serves as the state in which the variable is at each iteration of the decoder. The final decision for a variable node i is

$$x_i' = \begin{cases} 1, & \text{if } \sum_{a\in N(i)} \operatorname{sgn}\left(f_{a\to i}\right) + \operatorname{sgn}\left(y_j\right) > 0, \\ 0, & \text{otherwise,} \end{cases}$$

for all $i = 1, 2, \ldots, n$. That is, the decision is the sign of the sum of the signs of the memories and the sign of the channel value. The DD-BMP decoder significantly reduces the complexity of a hardware implementation as messages require only a single bit, and updates consist of only incrementing or decrementing the memory by one.

A further reduced complexity decoder was also proposed, the MDD-BMP, where the M stands for modified. In the MDD-BMP decoder a single memory c_i is kept for each variable node, instead of one for each edge of the Tanner graph. This simplification resulted in a larger or smaller loss in performance depending on the code. In general, it did not perform much worse than the standard DD-BMP and can be a viable alternative for some applications. The MDD-BMP variable memories are updated as

$$c_i = c_i + w \cdot \sum_{a\in N(i)} f_{a\to i}.$$

Cushon et al. (2014) presented a hardware simulation of both the DD-BMP decoder and the MDD-BMP decoder. They also presented a further modification to the decoder by Mobini et al. (2009). They call this modification the *improved differential binary* decoding algorithm. Cushon et al. note that the standard DD-BMP decoders are sensitive to *trapping sets*, a concept similar to stopping sets. To reduce the effects of trapping sets two modifications were proposed: *degeneration* and *relaunching*. Degeneration consists of performing the following update for to the variable node memories:

$$c_i = c_i + w \cdot \sum_{a \in N(i)} f_{a \to i} - d \cdot \operatorname{sgn}(c_i).$$

Here the free parameter is d added to improve performance. The arbitrary constant g determines the amount of degeneration that occurs. If the sum of the incoming messages is less than $\frac{g}{d}$ the memories move towards zero. The purpose of the degeneration is to avoid message values staying constant in trapping sets.

The second modification the authors proposed was relaunching. It simply consists of starting the decoder again. However, as the decoder is deterministic, the decoder is started with slightly changed initial memories. A full run of the decoding algorithm is called a *phase*, and a phase is indexed by p. The decoder is relaunched on phase p such that the memories are initialized to

$$c_i = \operatorname{sgn}(y_i) \cdot \max\left(\frac{1 - \operatorname{sgn}(y_i)}{2}, |y_i| - F(p,i)\right),$$

where $F(p,i)$ is a non-negative function which depends on the phase p and the variable node j. In the work by Cushon et al. the decoder was run for 6 phases, with 45 iterations in each phase. They found that the largest improvement came from adding degeneration to the decoder, while relaunching still slightly improved the performance.

4.3.5 Message-passing schedules

Some of the schedules already presented for bit-flipping decoders apply equally well to message-passing decoders, and were in some cases initially devised for message-passing decoders. The standard decoding schedule consists of first updating the variable to check messages, after which all the check-to-variable messages are updated. This is repeated until decoding is successful or a maximum number of iterations is reached. One iteration with a message-passing algorithm refers to updating all the variable-to-check and check-to-variable messages once. Processing the messages in turn like this is called a *flooding schedule*.

Layered, or group shuffled, decoding was already introduced in one form in the context of bit-flipping decoders in the previous sections. The initial idea was, however, presented in the context of message-passing decoders. Updating the messages on both sides of the Tanner graph in smaller groups is at least as beneficial for message-passing decoders as it is for bit-flipping decoders for the same reason: variable and check nodes can utilize updated values much earlier instead of relying on old messages.

Informed dynamic scheduling

Elidan (2006) presented a scheme for improving the convergence of general belief propagation algorithms and Vila Casado et al. (2007) presented and analyzed the improvements in the context of LDPC code decoding. They call the general method of using a more sophisticated schedule for a message-passing algorithm *informed dynamic scheduling*. The idea is to process and propagate messages which matter the most first. To arrive at a good schedule, they use *residual belief propagation*, which essentially involves calculating the absolute value of the change in the message values from one iteration to the next. The edges are then held in a priority queue according to their residuals and the message with the highest residual is first calculated, propagated, and reinserted into the priority queue in the appropriate position.

The authors find that informed dynamic scheduling improves on the performance of the sum-product algorithm while requiring fewer iterations (an iteration in this case is defined as processing a number of messages equal to the number of edges in the Tanner graph). The downside of this approach is again limited parallelizability. The straightforward implementation is inherently sequential. The authors also mention a parallel implementation of informed dynamic scheduling, similar to multi-bit bit-flipping decoders or bit-flipping decoders with a threshold. Instead of only processing the message with the highest residual, one processes the p messages with the highest residuals. They result is a negligible loss in performance, but the authors do not mention how large p was chosen to be. While the method showed promising results in sequential decoding, it is unclear whether such a schedule can efficiently be implemented in hardware. The need for a priority queue also increases complexity.

4.4 Turbo codes

We will for completeness give a brief overview of Turbo codes. The reason for presenting them at this point is that they rely on the same message-passing tools which we have just presented for LDPC codes. Turbo codes are *convolutional* codes. This means that, unlike LDPC codes, they operate on continuous *streams* of data instead of blocks of data, but in practice one uses finite streams of data also for Turbo codes. Turbo codes are defined by a linear feedback system.

The decoding of Turbo codes can be done with what is also essentially the sum-product decoder. In the simplest case the factor graph one gets for the maximum a posteriori estimation of the coded bits is a tree. In this case the sum product is also exact. In general, however, Turbo codes consist of a concatenation of individual Turbo codes either in parallel or serially. In this case the factor graph will not be a tree but it will, similarly to the factor graph of LDPC codes, consist of several sets of nodes such that one can do message-passing where one updates the messages from one set of nodes at a time. This resembles the conventional schedule for LDPC message-passing decoders where all variable-to-check messages are updated at once, and then all the check-to-variable messages. A thorough description of Turbo codes is given by Richardson and Urbanke (2008), and Berrou et al. (2005) gives a more general overview of the field of Turbo codes.

4.5 Summary

There has been a wealth of work on decoding of LDPC codes in the last 10–15 years. The main types of decoders are the message-passing and the bit-flipping decoders. Despite the recent increased interest in bit-flipping decoders, message-passing decoders often still come out ahead, at least in terms of error-correcting performance. This is perhaps a natural consequence of the message-passing decoders in general using more information than the bit-flipping decoders. Still, they can be made low-complexity as is for example the case with the MDD-BMP. Bit-flipping decoders, on the other hand, have seen improvements in the error-correcting performance without too large costs in arithmetic complexity. However, they still have not been able to achieve low enough bit-error rates compared to message-passing decoders. On the other hand, there may be situations where one can be more relaxed in terms of error-correction and get a corresponding increase in throughput.

We have not covered here another notable class of message-passing decoders: stochastic message-passing decoders (Gaudet and Rapley, 2003; Gross et al., 2005; Huang et al., 2013; Naderi et al., 2011; Noorshams and Iyengar, 2014; Tehrani et al., 2006, 2008, 2010, 2011). In the standard sum-product decoder messages are soft values, represented generally by more than one bit. Stochastic message-passing decoders operate using probabilities (as opposed to log-ratios as with the tanh-rule) and send instead *streams* of bits which are Bernoulli distributed according to the probabilities concerning a particular edge over which the stream is sent. We have not covered stochastic message-passing decoders here as there exists an extensive amount of work about them and the general principles are the same as in the sum-product decoder. However, they are potentially important as they present yet another way to reduce the complexity of decoding algorithms without too much loss in error-correcting performance.

Another aspect that will become more clear in the following chapter is that some degree or form of parallelism is often necessary to achieve high enough decoding throughputs. Both message-passing decoders and bit-flipping decoders benefit from sequential schedules as information is propagated faster to other nodes. In the case of for example informed dynamic scheduling the absolute number of operations per bit needed for decoding is clearly reduced. The increased number of operations per bit in a parallel implementation is often, however, offset by being able to process many or all bits in parallel.

Chapter 5

Decoder implementations

In the two previous chapters we have reviewed algorithms for encoding and decoding of LDPC codes. In practice, the algorithms must be implemented either in software for simulating the behavior of codes or algorithms, or in hardware for use in devices. Conveniently, simulating the behavior can be done *exactly*. That is, software encoders and decoders work for encoding and decoding real pieces of data, but at much lower throughputs than what can be achieved with hardware implementations. When implementing encoders and decoders in hardware it is important to consider not only the asymptotic complexity of the algorithms, but also the complexity of the implementation for example in terms of how much wiring is required on the chip, or what power consumption it has. Simply put, *constants matter*. This is especially true for decoders as essentially all decoders for LDPC codes are linear-time by design.

LDPC codes have in the recent years been incorporated as part of various wired and wireless standards, some of which are the DVB (ETSI, 2012, 2013a,b), WiMAX (IEEE, 2009), WiFi (IEEE, 2012b) and Ethernet (IEEE, 2012a) standards. The various standards have specified LDPC codes, but also put requirements on the throughput and error-correcting performance of the decoders. This clearly limits the actual decoders one can implement. For example, the DVB standards set the requirements that for each type of code given in the standard and a given signal-to-noise ratio, the decoder must achieve a bit-error rate of at most 10^{-7}.

To examine the error-correcting performance of decoders simulations on a conventional CPU is often sufficient. For examining the throughput of a decoder, the ideal situation would be to have an actual hardware implementation of the decoder. In practice, however, this is not feasible for testing various designs quickly, so one can instead simulate the hardware and this way get approximate results not only of how the decoding algorithm behaves but also of how the hardware design would behave when implemented in terms of throughput and power consumption. In this chapter, we will review some existing simulations of specialized hardware decoders. These are presented in Section 5.1. Recent advances in the programmability of graphics processing units (GPUs) have meant that they can be used to implement decoders with much higher throughputs than what is possible with ordinary CPUs. A fair number of works have been published on implementing message-passing decoders on GPUs and we review them in Section 5.2. We will then present an implementation of a decoder for GPUs by the author in Section 5.3.

5.1 Hardware simulations

Essentially all practical implementations of decoders employ some type of parallelism to achieve a high enough throughput. There are two main ways of parallelizing a decoding algorithm. The first is to process the bits of a word in parallel, which can be done for example with the flooding schedule of the message-passing decoders. We will call such a parallel implementation *bit-parallel*. Alternatively, one can consider several received words simultaneously and perform the same operations on each word. This can easily be done since each word is independent of each other. We will call such an implementation *block-parallel*. An implementation can of course be both bit- and block-parallel.

A recent and promising work on decoder implementations is by Cushon et al. (2014). They present a simulated hardware implementation of a binary message-passing decoder, the MDD-BMP as presented in Section 4.3.4. They also implement their proposed improvement to the MDD-BMP decoder, the improved differential binary decoding algorithm. The implementation presented is bit-parallel and is applied to finite geometry LDPC codes, as specified by the Ethernet standard. The IDB achieves at best a throughput of 170 Gb/s in their implementation which is higher than the throughput of 10 Gb/s required by the targeted Ethernet standard. The decoder, as implemented, is often close to the error-correcting performance of the offset min-sum decoder, but does not surpass it. The improved differential binary decoder surpasses in some situations the performance of the regular min-sum decoder. However, especially the MDD-BMP decoder exhibits high error floors with certain codes. Codes with low variable node degrees are particularly troublesome for both binary message-passing decoders implemented, confirming the observations of Mobini et al. (2009) which show the same problem.

In the work by Mohsenin et al. (2010) an improved *split-row* min-sum decoder is implemented as a hardware simulation. The split-row min-sum further simplifies the check-to-variable updates of the min-sum decoder by using messages from only a subset of the variable nodes adjacent to a check node sending a message. This reduces complexity in a hardware implementation by requiring less wiring, and increases throughput. The decoder achieves a peak throughput of 90 Gb/s. Results for the implementation are only presented for a bit-error rate down to 10^{-7} so the presence of error floors is not known unlike in the work of Cushon et al. (2014). Schläfer et al. (2012) present another high-throughput decoder design. Their design implements a min-sum decoder with 9 iterations, where their contribution consists of a fully *unrolled* decoder. Unrolling means that each of the 9 iterations are executed on physically different parts of the decoder allowing multiple words to be decoded simultaneously in different iterations resulting in a reported increase in area and energy efficiency compared with earlier works. The fully unrolled decoder, however, comes with the downside of being less flexible. For example, the number of iterations can not be adjusted for different levels of noise and a fixed number of iterations will always be performed. They achieve a peak throughput of 160 Gb/s.

Naderi et al. (2011) present a hardware design of a stochastic message-passing decoder. They provide designs for a short code from the Ethernet standard—the same code used in the work by Cushon et al. (2014) and Mohsenin et al. (2010). They also show a design for a longer code with block length 32768. The design for the short Ethernet code achieves a peak throughput of 170 Gb/s and the design for the longer code achieves a peak throughput of 480 Gb/s. The work shows that stochastic message-passing decoders can be competitive in terms of throughput.

The three hardware decoder designs mentioned above all reach similar maximum throughputs. However, choosing the *best* decoder can be difficult. There are many factors to take into account, some of which are error-correcting performance, latency, throughput, power consumption and chip size. As an example, the fully unrolled decoder cannot be stopped early because of the fixed number of iterations which may lead to higher power consumption compared to decoders which can stop earlier when noise levels are low. On the other hand, a fully unrolled decoder can achieve higher throughput and, ignoring early stopping, can in itself be more energy efficient in normal operation compared to non-unrolled decoders.

5.2 GPU implementations

Using graphics processing units (GPUs) for other uses than their original purpose is becoming more common. Scientific computation can often benefit from being implemented on GPUs, and especially *highly parallel* work is well suited for GPUs. Compared to common CPUs, GPUs often contain more processors, each of which contains several cores. As a downside the cores typically run at lower frequencies than CPUs. However, the higher number of cores generally makes up for the lower frequency, assuming that the problem one wishes to compute is sufficiently parallelizable. We will now give a brief overview of existing decoder implementations using GPUs. Compared to specialized hardware implementations as presented in the previous section, GPUs generally achieve decoding throughputs on the order of 100–1000 Mb/s. While slower than the specialized implementations, this is still 1–2 orders of magnitude faster than implementations on CPUs, which can achieve throughputs on the order of 10 Mb/s (Grönroos et al., 2012).

Wang et al. (2013) presented an implementation of a normalized min-sum decoder on a GPU. They use rate-$\frac{1}{2}$ codes from the WiMAX and WiFi standards with block lengths 2304 and 1944, respectively. They use one or four NVIDIA GTX TITAN GPUs for decoding. They report a peak throughput of approximately 315 Mb/s using one GPU and 10 iterations of the min-sum decoder. Using four GPUs and 10 iterations they report a throughput of 1.25 Gb/s. Their implementation is both bit- and block-parallel. This is the highest throughput reported for a GPU implementation although the hardware used is also the most powerful. Some earlier work using GPUs for decoding has been presented by the same authors (Wang et al., 2011a,b).

Abburi (2011) reports a throughput of 160 Mb/s using 5 iterations of the layered min-sum decoder on a single NVIDIA GeForce 9800 GTX+. Kang and Moon (2012) implemented a sum-product decoder on a NVIDIA GTX 480, reaching similar maximum throughputs using 10 iterations. Other works are by Chang et al. (2011); Grönroos et al. (2012); Martínez-Zaldívar et al. (2011). Some of the earliest work on using GPUs for decoding of LDPC codes was done by Falcão et al. (2008).

5.3 A GPU implementation of a stochastic-bit flipping decoder

Programming for a GPU is slightly different from programming for a CPU. Before we present the implementation of a bit-flipping decoder for GPUs, we will give a brief overview of the architecture of a GPU and the important aspects of programming for a GPU. We will in the following only consider NVIDIA's so-called *Compute Unified Device Architecture*, or simply CUDA, platform as the decoder was implemented for CUDA-devices. The decoder was finally run on two types of CUDA-devices, of which the first is the NVIDIA Tesla M2090, which is based on the Fermi microarchitecture, and the second is the NVIDIA Tesla K40, which is based on the Kepler microarchitecture. The two devices have different *compute capabilities*. The compute capability version specifies what kind of features and hardware is available on the device. Essentially, it specifies the microarchitecture of the device. The NVIDIA Tesla M2090 has compute capability 2.0 and the NVIDIA Tesla K40 has compute capability 3.5. Details of the two devices are shown in Table 5.1. Programming for CUDA devices is done using the CUDA *software platform* which provides a programming interface based on the C language. Code written using CUDA C is then compiled to an intermediate form

Table 5.1: Specifications of the two CUDA-devices used in this work. (NVIDIA, 2011, 2013)

Model	NVIDIA Tesla M2090	NVIDIA Tesla K40
Compute capability	2.0	3.5
Multiprocessors	16	15
Cores	512	2880
Processor core clock [GHz]	1.3	0.745
Global memory [GB]	6	12
Memory clock [GHz]	1.85	3.0
Memory bandwidth [GB/s]	176	288

of assembly language which is generic for all CUDA devices and uses the Parallel Thread Execution (PTX) Instruction Set Architecture (ISA). PTX assembly is then compiled to an architecture-specific binary based on the compute capability of a device.

5.3.1 Architecture and programming of CUDA devices

A GPU generally contains multiple processors. Each processor is called a *streaming multiprocessor*. Each streaming multiprocessor, or just multiprocessor, contains multiple cores and is capable of executing a number of threads simultaneously. In simple terms, each thread executes the same instruction, resulting in a multiprocessor essentially acting like a vector machine. This is called *single-instruction, multiple-data* (SIMD) parallelism. Execution on a GPU is divided into threads which all run the same *kernel*. A kernel is simply a function which can run on different threads and is aware of in which thread it is running. This allows one to perform work in parallel on for example different pieces of data. A *warp* is a unit of 32 threads which are executed simultaneously on a single multiprocessor. If some threads in a warp take a different execution path than other threads in the warp, each group of threads with the same execution path will be executed in parallel while threads with different execution paths are executed serially. This is called thread divergence. Because of this behavior, it is important to try to have minimal thread divergence within warps to achieve high performance. Ideally, all threads within a warp should execute the same sequence of instructions.

An important aspect resulting from the microarchitecture of CUDA devices is that of arithmetic latency. An arithmetic instruction generally takes between 10–20 clock cycles to complete (NVIDIA, 2014). Arithmetic instructions are, however, pipelined. This means that multiple instructions of the same type can be executed simultaneously by being in different stages of the pipeline, assuming that the instructions operate on *independent* data. Put simply, the *latency* of arithmetic instructions on a GPU is high, but the *throughput* can be higher than only one instruction per 10–20 clock cycles. If not enough independent instructions are available to be executed simultaneously, the pipeline *stalls* until new instructions can be executed. The pipelining is done automatically, but requires that a sufficient number of independent instructions are available to be executed. Ignoring memory accesses, to fully utilize a multiprocessor one then needs to ensure that the multiprocessor can execute a sufficient number of independent instructions. One way of achieving this is by running a number of threads equal to 10–20 times the number of cores available on the multiprocessor. Alternatively, one can increase the number of independent instructions *within* a thread. The typical latency of instructions performed on devices with compute capability 2.0 is 22 while with 3.5 it is 11. Although hiding arithmetic latency can be important for achieving maximal performance, hiding memory latencies can be more important as we will see below.

The memory hierarchy of a GPU is as follows: all multiprocessors share access to the *global memory*, which acts similarly to RAM for CPUs and is relatively slow to access. Each multiprocessor has a *shared memory* common for all threads within that multiprocessor, and all multiprocessors typically share one or two levels of cache for global memory accesses. Additionally, each thread has access to a maximum number of registers. For example, GPUs with CUDA compute capability 2.0, such as the NVIDIA Tesla M2090, the maximum number of registers per thread is 63. In addition, there is a maximum number of registers a multiprocessor has in total, which is 32 K for compute capability 2.0. For the NVIDIA Tesla K40 with compute capability 3.5 the maximum number of registers per thread is 255 and the total number of registers in a multiprocessor is 64 K. Each thread can also access a portion of the global memory which is local to each thread. This is called *local memory*. Access to local memory is, however, as slow as accesses to global memory. The memories are, in decreasing order of size and latency: global and local memory, shared memory and registers. Global memory latencies are the most noticeable. For compute capability 2.0 the global memory latency is 400–800 cycles and for compute capability 3.5 the latency is 200–400 cycles (NVIDIA, 2014).

Because of the high latency in accessing global memory, the most important aspect of achieving high performance is generally to avoid or hide the global memory latency. In the case of the current decoder implementation avoiding the latency is not possible as the received words need to be stored in global memory because of their size. Hiding the latency can be done by ensuring that enough memory requests are made to global memory so that the memory bandwidth is saturated. This is done in practice by ensuring that enough threads are running or by increasing the number of memory requests within a thread.

When designing a kernel, data that is accessed frequently and is small enough should be stored in shared memory. While slower than registers, the shared memory is still significantly faster than global memory to access. For both compute capability 2.0 and 3.5 the amount of shared memory per multiprocessor is 48 KB. Finally, registers are used for the remaining variables. If a thread requires more registers than are available, variables need to be stored in local memory causing what is called *register spilling*. When register spilling occurs, accessing spilled variables from local memory is significantly slower than accessing registers. The local memory makes use of a cache which can help reduce access times, but generally register spilling will have a detrimental effect on the performance of a kernel. While the CUDA compiler automatically optimizes the use of registers and local memory for performance, manually reducing the number of variables used in a kernel may also help in reducing register spilling.

5.3.2 Decoder implementation

For this thesis, a stochastic bit-flipping decoder as described in Section 4.2.5 was implemented and tested on two CUDA devices. The decoder works on hard values, meaning that we assume either that transmission has occurred over the BSC or that soft channel values from the BAWGNC are quantized to 1 bit. We assume here that the channel values are in $\{0,1\}$. The decoder employs a sequential schedule, where the decision to flip is done randomly based on probabilities $p_{e,b,d}$ which depend on if the current value of the bit is the same as the channel value, the number of adjacent unsatisfied checks and the degree of the variable node. The decoder implementation in the current work is block-parallel. That is, the decoder decodes multiple words in parallel as all received words are independent from a decoding perspective, while the decoding of each individual word is done with a sequential schedule. The decoder is made block-parallel on two levels. First, we will formulate the decoder mostly as a Boolean circuit. This is important as it allows the decoder to operate using bitwise Boolean operations on multiple words at a time. At the same time, formulating the decoder as a Boolean circuit avoids thread divergence and allows a high throughput. Second, multiple threads on multiple multiprocessors each decode their own set of words, further increasing the level of parallelism.

To clarify, the term *word* will only be used in the sense of coding theory, meaning the sequence of bits that we wish to decode to a codeword, of coding theory. This should not be confused with the commonly used *word* which refers to a group of bits on which a processor generally performs computations. As the GPUs we are considering have a 32-bit architecture, the latter use of *word* refers to a group of 32 bits on which the GPU operates using an instruction. To avoid confusion with the former use we will avoid using *word* in the latter sense. Additionally, we will often simply use the term *bit* interchangeably with variable node or column of the parity-check matrix since the value of the variable is quantized to a single bit.

Computation

We will, to begin with, only consider the decision to flip a single bit i in a single word. Having done that, realizing the decoder as a block-parallel decoder is straightforward. The decoding takes place in four steps:

1. Draw a value r uniformly at random between 0 and 1 using a random number generator.
2. Count the number of checks adjacent to the current bit i that are unsatisfied.
3. Compare the pseudo-random value r drawn in the first step to the probability of flipping a bit $p_{e,b,d}$ given the degree d of the bit; the number of unsatisfied checks b in the second step; and e, the XOR of the current value and channel value of the bit.
4. Flip the bit if the pseudo-random value is less than the probability of flipping.

In the first step, the value of r is represented by a 32-bit integer taking values between 0 and $2^{32}-1$. The probabilities $p_{e,b,d}$ are also represented as 32-bit integers. The 32-bit integers representing the probabilities are calculated as $p_{e,b,d} \cdot (2^{32}-1)$. The value of r is drawn using a linear-feedback shift register (LFSR). While consecutive numbers drawn with a LFSR are highly correlated, the simplicity of a LFSR makes it ideal for a high-performance implementation.

The second and third steps are as follows. As we are working with binary codes, calculating the value of a checksum a adjacent to the bit i is done simply by XORing the values of the bits adjacent to a. A check is satisfied if the sum is 0 and unsatisfied if it is 1. To count the number of unsatisfied checks a simple incrementer circuit is sufficient. We only considered parity-check matrices with maximum variable node degree 6 for the implementation, so an incrementer with 3 bits suffices. The circuit used for counting the number of unsatisfied checks is shown in Figure 5.1. We denote the number of unsatisfied checks by the bits b_0, b_1 and b_2 such that $b_2b_1b_0$ is the binary representation of the number of unsatisfied checks. That is, b_2 is the most significant bit and b_0 is the least significant bit. The decoder initially sets all three incrementer bits to 0. It then processes each adjacent check in turn and adds the result to the counter bits. When the value of the checksum of each adjacent check have been added to the incrementer, $b_2b_1b_0$ holds the binary representation of the number of unsatisfied adjacent checks.

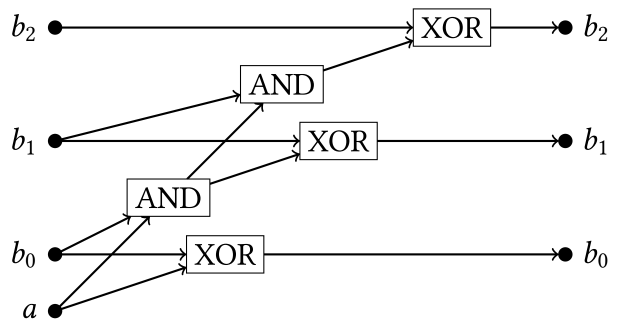


Figure 5.1: A 3-bit incrementer circuit with modular behavior. The circuit adds the bit a to the 3-bit number with a binary representation $b_2b_1b_0$. Each XOR essentially calculates the new value of the corresponding bit, and each AND calculates the carry bit for the next position.

The third step is also implemented as a Boolean circuit. Let us denote by f the decision to flip a bit i, such that f takes the value 1 if the bit i should be flipped and 0 otherwise. As in Section 4.2.5, let e be 1 if the current value of the bit is different from the channel value, and 0 otherwise. That is, e is the XOR of the current value of the bit and the channel value. The probability of flipping a bit with degree d, b unsatisfied checks and e is denoted by $p_{e,b,d}$. Then, let $r_{e,b,d}$ be 1 if the drawn pseudo-random value r is less than the probability $p_{e,b,d}$, and 0 otherwise. The decision to flip a bit with degree d is then given by

$$\begin{aligned} f = {} & (r_{1,1,d} \land (\neg b_2 \land \neg b_1 \land b_0 \land e))\lor \\ & (r_{0,1,d} \land (\neg b_2 \land \neg b_1 \land b_0 \land \neg e))\lor \\ & (r_{1,2,d} \land (\neg b_2 \land b_1 \land \neg b_0 \land e))\lor \\ & (r_{0,2,d} \land (\neg b_2 \land b_1 \land \neg b_0 \land \neg e))\lor \\ & \qquad\qquad \cdots \\ & (r_{1,d,d} \land (\neg b_2 \land b_1 \land \neg b_0 \land e))\lor \\ & (r_{0,d,d} \land (\neg b_2 \land b_1 \land \neg b_0 \land \neg e)). \end{aligned} \tag{5.1}$$

Let us look at the first row of (5.1) in more detail. The value of $r_{1,1,d}$ is true if the bit should be flipped given that it has one unsatisfied check and the value of the bit we are considering is different from the channel value. The expression on the right, $(\neg b_2 \land \neg b_1 \land b_0 \land e)$, checks that the current variable node actually has exactly 1 unsatisfied check and that its value is equal to the channel value. Each row then checks this for different values of the number of unsatisfied checks and e. The number of unsatisfied checks and the value of e will match on exactly one row. Taking the disjunction between each row ensures that if for the matching row the value of $r_{e,b,d}$ is also true, the whole expression will be true. Thus f is true exactly with the probability $p_{e,b,d}$ given b bad checks and e. By computing the XOR of f and the current value of the bit we get the new value of the bit. We should note here that a reason for restricting ourselves to codes with low variable node degree is that the number of rows in (5.1) grows exponentially with the maximum variable node degree.

The above steps apply for a single bit in a single word. The full decoder for a single word then consists of doing the above for each bit sequentially for a fixed number of rounds. The reason for formulating the decoder with the help of Boolean circuits was to allow running the decoder easily as a block-parallel decoder. This can be realized since we can perform each type of Boolean operation, or logic gate, using a single bitwise Boolean instruction on 32-bit integers on the GPU. As a result, the decoder can evaluate a logic gate for 32 independent Boolean circuits simultaneously, each corresponding to an independent received word.

For generality, we will assume that we can perform bitwise Boolean operations on w-bit types. This is because CUDA provides the vector types `uint1`, `uint2` and `uint4`, which consist of 1, 2 and 4 unsigned integers each. That is, the types contain 32, 64 and 128 bits, respectively. We can then define bitwise Boolean operations to work on the vector types as well. In practice, the bitwise Boolean operations on for example a variable of type `uint4` must be performed as 4 separate 32-bit bitwise Boolean operations as there is only support for 32-bit bitwise Boolean operations in CUDA hardware (NVIDIA, 2014). The benefit of using vector types is that there is support in CUDA for memory loads from global memory using vector types. This is beneficial for maximizing the use of global memory bandwidth. The instruction-level parallelism is also increased by using larger vector types as each of the 32-bit bitwise Boolean operations are independent. The parallelization of the decoder then comes partly from using w-bit types in each thread to process w independent words, and partly from running multiple threads on each multiprocessor.

When decoding w words in each thread, the value of $r_{e,b,d}$ needs to be specified for each of the w words being decoded by the thread. This was done simply by using a single LFSR as the pseudo-random number generator in each thread. The pseudo-random value r was then compared to the probabilities to flip $p_{e,b,d}$ and setting a full `uint1`, `uint2` or `uint4` to only ones or zeros depending on the result of the comparison. That is, the same random number generator was used for decoding of all words within a thread.

Memory layout

The two most important memory-related aspects of the decoder implementation are the layout and access of both received and current bits, and the layout and access of the parity-check matrix. We first consider the layout of the words. Generally, the words will be received sequentially at the decoder. Storing bits in the order that they are received means that the bits of a single word will be close to each other in memory, while bits in the same position but in different words will in general be far from each other. We want the opposite to hold for the decoder to be efficiently block-parallel. That is, we want bits in the same position in different words to be close to each other in memory so that each thread in the decoder can access bits in the same position of different words efficiently to perform bitwise Boolean operations.

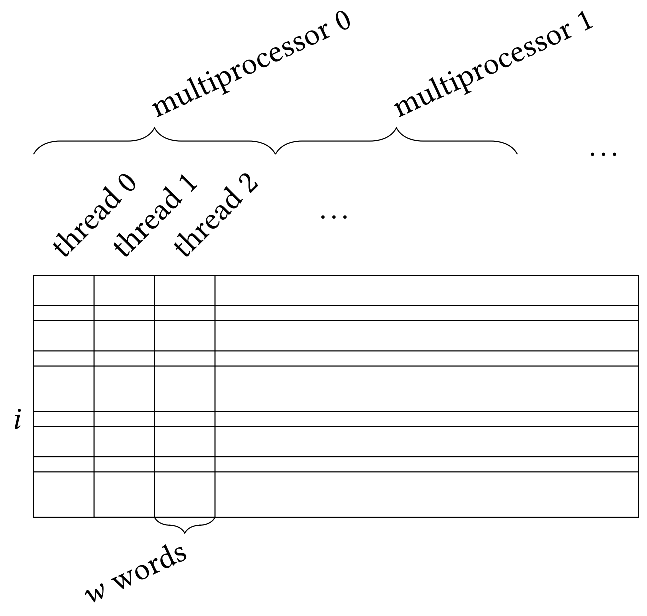


Figure 5.2: The transposed received words. Each column contains one received word and each row contains the bits in the ith position of each received word. A thread accesses w consecutive bits from row i to get the values of the ith bits of w words processed by that thread. In addition, the thread accesses w bits from each row corresponding to the second neighbors of i. The work is split up so that consecutive threads within the same multiprocessor process consecutive sets of w words.

We can think of the received bits as a binary matrix where each row contains one word, and each row is stored sequentially in memory. With this analogy, we wish to take the matrix transpose of the received bits, such that the bits in a given position of each word form a row of the binary matrix. A binary matrix transpose can be done quickly on modern CPUs with vector registers and operations. To be more precise, the matrix transpose was implemented using instructions which are part of the Advanced Vector Extensions (AVX) for the Intel 64 architecture (Intel, 2012). The transpose was realized using the instruction to left-shift 128-bit vectors and the instruction to select most significant bits of each byte in a 128-bit vector. Once the received bits have been transposed, the transposed bits are transferred to the GPU where they are now laid out so that they can be accessed efficiently. Once decoding is finished, the transposed bits are transferred back from the GPU and transposed again so that the bits are in the order in which they were initially received. Figure 5.2 shows the transposed memory layout more clearly. When decoding, a thread is responsible for decoding a fixed set of w sequential words, or columns, in the transposed matrix. A thread can then efficiently access the ith bits of a set of w words for which the thread is responsible by reading consecutive bits in the ith row of the transposed bits. Another benefit of this layout is that for performance reasons a warp of 32 threads should generally access consecutive memory locations in global memory. The transposed memory layout allows this, since we can assign each warp that will be executed a set of 32 consecutive sets of w words.

The storage and access of the parity-check matrix is the second aspect that is important for performance. For this decoder we use a QC-LDPC code because the parity-check matrix can be stored implicitly, reducing memory use and accesses. In addition, the positions of the ones in the parity-check matrix can be calculated quickly from the implicit representation.

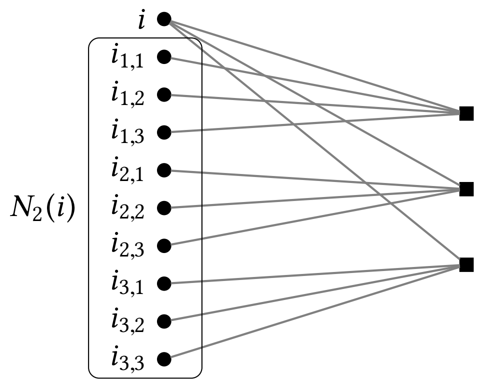


Figure 5.3: The second neighbors of the variable node i, denoted by $N_2(i)$ on a Tanner graph where the variable node has degree 3 and the check nodes degree 4. Note that the node i itself is not included in the set.

When considering a bit in position i of a word we need to know which check nodes are adjacent to i, and which variable nodes are adjacent to each neighbor a of i to calculate the checksums of the neighboring check nodes, and how many of them are unsatisfied. We will call these neighbors of neighbors of i the *second* neighbors of i, and denote the set of second neighbors by $N_2(i)$. We do not include i itself as the second neighbor of i. Figure 5.3 shows the set of second neighbors graphically. Because of the cyclic structure of the sub-matrices of a QC-LDPC code it is enough to store the information of the second neighbors of a single column or variable node within each major column of the parity-check matrix. As before, we let the size of each sub-matrix in the parity-check matrix of a QC-LDPC code be $z \times z$ and by a major column we mean a column in the model matrix of the QC-LDPC code. Thus, a major column specifies the ones in z columns of the full parity-check matrix. Given the second neighbors of a variable node, or column, within the major column, we can calculate the positions of the second neighbors of the other columns within the major column. For simplicity, we choose to store the second neighbors of the leftmost column in each major column. Figure 5.4 shows the parity-check matrix of a QC-LDPC code with the neighbors of one check node highlighted. For generality, we will assume that the code is irregular, in which case we also need to store the degree of the variable node i as well as the degrees of the adjacent check nodes. Since we do not include i itself as a second neighbor, we only need to store the indices of $d - 1$ neighbors of a check node, assuming that it has degree d. Figure 5.5 shows a portion of the memory layout of the parity-check matrix containing the second neighbors of one column within a major column. Note that each portion corresponding to one major column is padded to a maximum length so that calculating the address of the first element in each portion can be calculated by multiplying the index of the major column with the maximum block length, assuming that the major column indices are zero-based.

Let us look more closely at how the memory layout of the parity-check matrix is used by the decoder, and let us for the moment again only consider decoding a single word. Calculating the positions of the second neighbors of the other columns within a major column is especially convenient to do if we consider the columns in order, beginning from the leftmost column. That is, we store in the implicit representation the positions of the second neighbors of the leftmost column within each major column. The implicit representation of the parity check matrix we have presented above is too large to fit into registers, but is small enough to fit in the shared memory of a multiprocessor, so it is stored in the shared memory. Let the first column within a major column have index i. Thus, we first load the indices of the second neighbors of column i from shared memory. Once the indices have been loaded, we load from global memory the

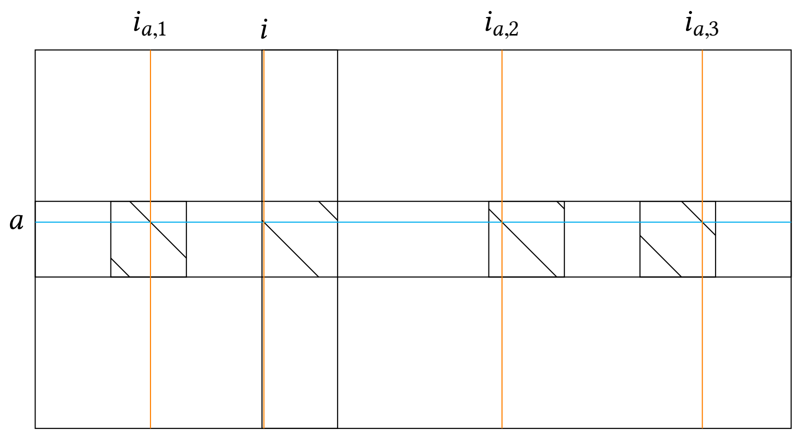


Figure 5.4: The parity-check matrix of a QC-LDPC code where the positions of the nonzero entries are shown on one major row. The diagonal lines show the positions of the ones in the sub-matrices. In addition, the columns whose indices are stored in the implicit representation for the GPU implementation of the stochastic bit-flipping decoder are shown in orange. The column marked with i is the first column in the major column whose second neighbors we wish to store. The blue row marked with a is one of the check node neighbors of i. The three other orange columns marked with $i_{a,1}, i_{a,2}$ and $i_{a,3}$ are the neighbors of the check node a. The indices of the three orange columns are then stored in the implicit representation of the parity-check matrix. This is repeated for other check node neighbors that i may have, and for each major column.

bits corresponding to the column i and its second neighbors. We also load the channel value corresponding to the column i. We then calculate the checksums of the of the neighbors of i, count the number of unsatisfied checks, decide if the ith bit should be flipped, and flip it if needed. We then continue to column $i + 1$. For column $i + 1$ it is enough to increment the index of each second neighbor of column i by one. Because of the cyclic nature of the sub-matrices, we also need to check for each index if it is divisible by the sub-matrix size z. If it is, we subtract z from the index value, effectively ensuring that we follow the cyclic structure of the sub-matrix. For this we again assume that the column indices are zero-based. We can then flip the bit corresponding to the second column of the major column if needed. We can continue in this way, incrementing the indices and subtracting z when needed, for all columns $i, i + 1, \ldots, i + z - 1$ in the major column to get the correct indices of the second neighbors of each column. For each column, we use the calculated indices to load the corresponding bits from global memory, calculate the checksums, count the number of unsatisfied checks and flip the bit if necessary.

$\cdots$	d	d_1	$i_{1,1}$	$i_{1,2}$	$\cdots$	i_{1,d_1-1}	d_2	$i_{2,1}$	$\cdots$	i_{d,d_d-2}	i_{d,d_d-1}	$\cdots$

Figure 5.5: A portion of the memory layout of the implicit representation of a QC-LDPC code corresponding to a single major column. The second neighbors of the first column i in the major column is stored. The degree of the variable node or column is given by d. The degrees of the neighboring check nodes are given by $d_1, d_2, \dots, d_d$, and the position of the jth variable neighbor of the ath check neighbor of i is given by i_{aj}. Note that we do not store the column i itself as a neighbor of its neighbors.

First, the indices loaded and calculated in the above procedure can easily be used within a thread to consider w independent words. Instead of loading a single bit for each column i and its second neighbors, the thread loads w bits, corresponding to w different words, for each column i and its second neighbors. Second, each thread then performs the above calculations independently on their own set of w words. An important practical detail is that the indices of the second neighbors can be loaded without conflicts from shared memory. Since each thread in a warp accesses the same position in shared memory, the result is broadcast to each thread in the warp with a single memory read request (NVIDIA, 2014). Another consideration is checking for divisibility efficiently. If the sub-matrix size z is a power of 2, checking for divisibility by z can be done efficiently using for example a simple bit-masking operation. In the general case, however, checking for divisibility requires slightly more work (Warren, 2012).

To further optimize the decoder kernel, the flip probabilities were generated before compiling the kernel and the values hard-coded. In addition, code was generated to specialize on the degrees of the variable and check nodes to be able to unroll the majority of the loops and so improve performance. This could be done on the level of a major column of the parity-check matrix as the degrees are fixed within a major column.

Altogether the GPU implementation of the stochastic bit-flipping decoder does the following steps:

1. Transpose the received words on the CPU.
2. Copy the transposed words to the GPU. Make an additional copy of the received words on the GPU for determining e.
3. Each thread decodes w words for a fixed number of iterations.
4. Copy the words back from the GPU.
5. Transpose the decoded words back.

In the next chapter we will look at how fast the decoder is in practice.

Chapter 6
Experimental results

In this chapter, we will present an experimental comparison of the error-correcting performance of five decoders implemented in software and run with typical configurations. In addition, we will look at the encoding complexity of codes from a regular and an irregular ensemble of LDPC codes using the method proposed by Richardson and Urbanke (2001b). Finally, we will present experimental results on the decoding throughput of the GPU implementation of the stochastic bit-flipping decoder. To thoroughly examine the performance of decoders one would need to take into account details of hardware implementations of the decoders, and implementing or simulating multiple hardware designs of decoders is beyond the scope of this thesis. For this reason, we will focus only on the error-correcting performance, or bit-error rate, of the decoders and ignore the throughput, latency and other hardware-related details in the decoder comparison. On the other hand, the performance evaluation of the GPU implementation will only focus on the throughput and latency of the decoder, and on how well it utilizes the hardware.

6.1 Comparison of decoders

For this thesis, five different decoders were implemented in software to run on the CPU and were tested with three different codes or ensembles of codes on two different channels. The five decoders implemented were the sum-product decoder, the min-sum decoder, the gradient-descent bit-flipping decoder, Gallager's bit-flipping decoder and the stochastic bit-flipping decoder presented in Section 4.2.5. The implementations of the sum-product and min-sum decoders use 64-bit message values with the assumption that this approximates infinite precision messages well. Most importantly, as we reviewed briefly in Section 4.3.3, even using less than 10 bits for the message values is generally enough to approximate the infinite precision sum-product decoder sufficiently well. Because of this, we consider 64-bit message values to be sufficient for the performance of the sum-product decoder to be a good baseline to which other decoders can be compared. In addition, the channel noise parameter was assumed known to the sum-product and min-sum decoders for initializing the messages from channel factors to variable nodes.

The sum-product and min-sum decoders were both run for 20 iterations. The three bit-flipping decoders were each run for 100 iterations with a sequential schedule. The number of iterations for the decoders were chosen to be typical values found in the literature. In Section 5.1 on simulations of hardware decoders we saw implementations which use approximately 10 iterations with min-sum decoders, and so 20 iterations was chosen to ensure that the error-correcting performance is generally not worse than it would be in a hardware implementation. The higher number of 100 iterations for the bit-flipping decoders was chosen as the bit-flipping decoders are generally less complex and require more iterations for good performance. The number of iterations for the bit-flipping decoders is also typical for the literature. For example, in the work by Sundararajan et al. (2014) where the noisy gradient-descent bit-flipping decoder is presented, the different variations are run with 100–300 iterations. The random number generator used for CPU implementation the stochastic bit-flipping decoder is identical to the LFSR used in the GPU implementation.

The decoders were tested with three types of codes. First, codes from the (3,6)-regular ensemble were used. Second, codes from an irregular ensemble of codes presented by Richardson et al. (2001) were used. The ensemble has maximum variable node degree 4 and has been optimized to have a high threshold in the asymptotic case of infinite block length and an infinite number of iterations with the sum-product decoder. The degree distribution of the ensemble is given by the following polynomials

$$\begin{aligned} L(x) &= 0.54883x^2 + 0.04042x^3 + 0.41075x^4, \\ R(x) &= 0.276153x^5 + 0.723847x^6. \end{aligned}$$

Here, the coefficient of a monomial x^i of the polynomial $L(x)$ gives the fraction of variable nodes with degree i. The polynomial $R(x)$ gives the equivalent fractions for the check nodes. Codes from the two ensembles were drawn by rejecting codes which have duplicate edges in their Tanner graph. Finally, the rate-$\frac{1}{2}$ QC-LDPC code defined in the WiMAX standard (IEEE, 2009) was used. All three codes or code ensembles have rate $\frac{1}{2}$.

A more thorough set of tests was run with the sum-product decoder and the stochastic bit-flipping decoders. The two decoders were tested with block lengths 2^i for $i = 10, 11, \dots, 20$ with the (3,6)-regular ensemble and the irregular ensemble. The decoders were only tested with the block length 2304 with the rate-$\frac{1}{2}$ WiMAX code, which is the longest block length defined in the WiMAX standard. The transmission of a total of 2^{30} bits was simulated for each code, block length, channel and decoder. More precisely, for the two ensembles of codes, the following was repeated $\left\lfloor \frac{2^{30}}{n} \right\rfloor$ times for each block length n: draw a new code, simulate transmission over a channel, and decode the received word. The same was repeated $\left\lfloor \frac{2^{30}}{2304} \right\rfloor$ times for the rate-$\frac{1}{2}$ WiMAX code, with the difference that the same code was used in each repetition. Notice that encoding is not simulated because it is time consuming for codes from the (3,6)-regular ensemble with long block lengths. Instead, it is assumed that the sent codeword is the all-zeros codeword when transmitting over the BSC as it is a codeword of any LDPC code. It can be assumed that the all-zero codeword is sent as, by symmetry, the performance of a code and decoder does not depend on the sent codeword, but only on the noise in the channel (Richardson and Urbanke, 2008). On the BAWGNC, the equivalent assumption is that the all-ones codeword is sent. We will consider encoding complexity separately in Section 6.2.

The three remaining decoders—the min-sum decoder, the gradient-descent bit-flipping decoder, and Gallager's bit-flipping decoder—were tested with a smaller set of tests. They were all run with the block length 2^{14} for the two ensembles of codes, and with the block length 2304 for the rate-$\frac{1}{2}$ WiMAX code. The tests were repeated identically to the tests with the sum-product decoder and the stochastic bit-flipping decoder by simulating the transmission of a total of 2^{30} bits.

Finally, all decoders were tested on both the BSC and the BAWGNC, with the exception of Gallager's bit-flipping decoder and the stochastic bit-flipping decoder as they only work on hard channel values. However, the results of the two bit-flipping decoders on the BSC were translated to the BAWGNC by assuming hard-thresholding of the soft-channel values. The decoders were run with a range of noise levels to see the behavior as a function of the noise as well.

6.1.1 Choosing the decoder parameters

The stochastic bit-flipping decoder has two free parameters for the parameterization of the flip probabilities as presented in Section 4.2.5, and the gradient-descent bit-flipping decoder with a sequential schedule has one free parameter, the threshold θ for flipping a bit. The free parameters were chosen by running a constant-spaced grid search over a set of parameter values, after which the overall best parameter values were chosen. The results of the grid search are shown in Appendix B. It is good to note from the results of the grid searches that the optimal values generally depend on the level of noise. The parameter values were chosen here to give good results at moderate levels of noise, possibly leaving larger error floors for low levels of noise. The chosen parameter values are shown in Tables 6.1 and 6.2. For the comparison here, we only considered fixed parameter values, but further optimization of the error-correcting performance could involve dynamically adjusting the parameter values based on the level of noise or the iteration number. This has already been suggested by for example Ismail et al. (2013) in the context of bit-flipping decoders.

Gallager's bit-flipping decoder has a free parameter which adjusts how many adjacent check nodes must be unsatisfied for a bit to be flipped. For Gallager's bit-flipping decoder this value was simply chosen for the two ensembles of codes and the rate-$\frac{1}{2}$ WiMAX code. For the (3,6)-regular ensemble, the threshold was set to 2, meaning that 2 or more check nodes adjacent to a variable node must be unsatisfied for the bit to be flipped. The threshold was set to 2 for the irregular ensemble and to 3 for the rate-$\frac{1}{2}$ WiMAX code.

Table 6.1: Parameter values chosen for the stochastic bit-flipping decoder.

	(3,6)-regular	Irregular	WiMAX
T	0.8	0.8	0.9
p	0.12	0.08	0.08

Table 6.2: Parameter values chosen for the gradient-descent bit-flipping decoder.

	(3,6)-regular	Irregular	WiMAX
θ (BAWGNC)	-0.8	-0.4	-0.6
θ (BSC)	-0.5	-0.5	-0.5

6.1.2 Results

Figure 6.1 shows the bit-error rate P_b of the sum-product decoder on the BSC as a function of the crossover probability ϵ and the block length n. With the (3,6)-regular ensemble the bit-error rate clearly shows a typical waterfall region around $\epsilon = 0.075$ and an error floor at lower levels of noise. On the other hand, the bit-error rate of the irregular code is far worse than that of the (3,6)-regular ensemble. Additionally, the irregular ensemble doesn't show a clear waterfall region and has a relatively high bit-error rate even with long block lengths and low levels of noise. The bit-error rate of the rate-$\frac{1}{2}$ WiMAX code is again better than the irregular ensemble with block length 2048 and slightly better than the (3,6)-regular ensemble with the block length 2048. However, with the block length 4096 the (3,6)-regular ensemble performs roughly equally to the rate-$\frac{1}{2}$ WiMAX code. Figure 6.2 shows the bit-error rate of the sum-product decoder on the BAWGNC as a function of the signal-to-noise ratio and the block length n. The behavior is largely similar to on the BSC, with the rate-$\frac{1}{2}$ WiMAX code having the lowest bit-error rate compared to short block lengths with the (3,6)-regular ensemble and the irregular ensemble. With longer block lengths, the (3,6)-regular ensemble shows again typical behavior and the irregular ensemble performs worse than the (3,6)-regular ensemble. Finally, the bit-error rate of the stochastic bit-flipping decoder on the BSC is shown in Figure 6.3. The behavior is similar to the sum-product decoder on the BSC but with slightly worse performance in general. It is good to note that the error floors are at similar levels using both decoders, and that the main difference in error-correcting performance with the current implementations is between the thresholds of the decoders, with that of the stochastic bit-flipping decoder being lower.

The comparison of the five decoders on the BSC is shown in Figure 6.4. The results show a clearer picture of how the stochastic bit-flipping decoder compares to the generally better message-passing decoders and to simpler bit-flipping decoders. The sum-product decoder is consistently the best decoder with the (3,6)-regular ensemble, irregular ensemble and the rate-$\frac{1}{2}$ WiMAX code. The stochastic bit-flipping decoder behaves similarly to the sum-product with the difference that the results are shifted to lower levels of noise. With the (3,6)-regular ensemble the gap in error-correcting performance between the two decoders is the smallest. In terms of the crossover probability, the results of the stochastic bit-flipping decoder are shifted to the left from the sum-product decoder by approximately 0.01. The corresponding gaps are 0.02 with the irregular ensemble and the rate-$\frac{1}{2}$ WiMAX code. The two other bit-flipping decoders, Gallager's bit-flipping decoder and the gradient-descent bit-flipping decoder, generally perform the worst. Gallager's bit-flipping decoder only barely improves on transmission without any encoding at low levels of noise, and the gradient-descent bit-flipping decoder performs only slightly better. However, with the (3,6)-regular ensemble the gradient-descent bit-flipping decoder performs clearly better than Gallager's bit-flipping decoder. Interestingly, the min-sum decoder performs badly compared to the sum-product decoder because of the hard channel values on the BSC.

Figure 6.5 shows the same comparison as above but on the BAWGNC. Compared to the BSC, the min-sum decoder now performs nearly as well as the sum-product decoder. The gradient-descent bit-flipping decoder also performs better relative to the stochastic bit-flipping decoder as it can make use of soft channel values. As explained earlier, the results of Gallager's bit-flipping decoder and the stochastic bit-flipping decoder have been translated from the results on the BSC by assuming hard-thresholding of the channel values on the BAWGNC. Thus, the two bit-flipping decoders clearly don't benefit from moving to the BAWGNC and so their performance is comparatively worse. However, the stochastic bit-flipping decoder still performs relatively well. Most importantly, it performs better than

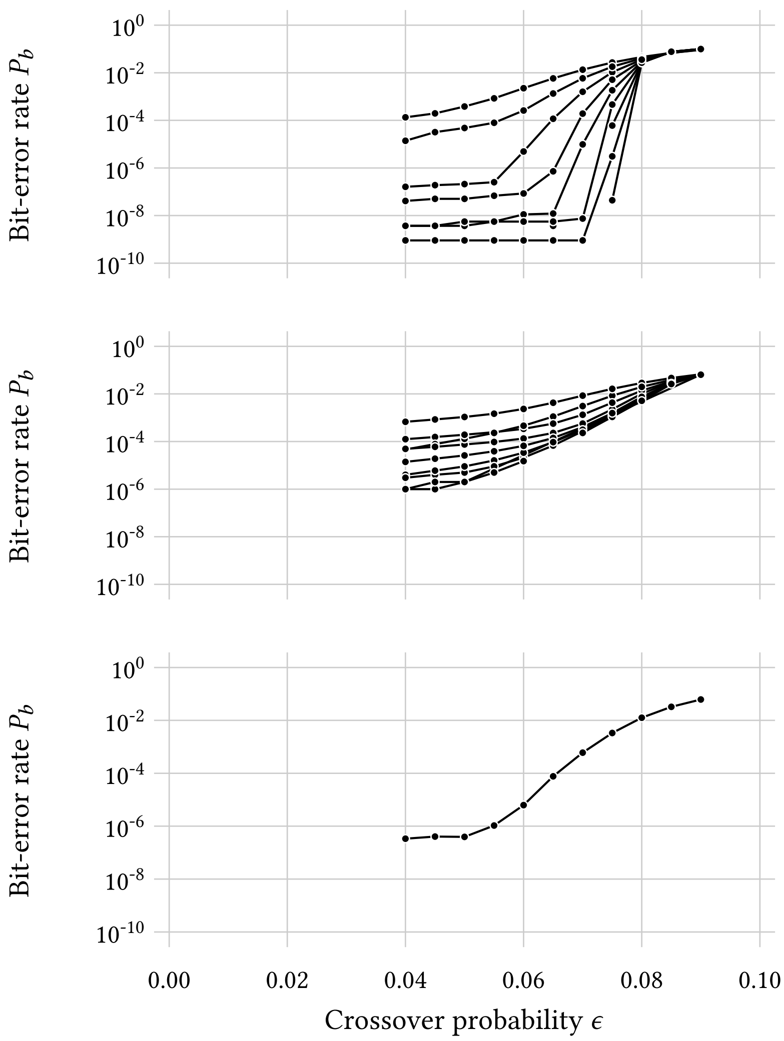


Figure 6.1: Bit-error rate P_b of the sum-product decoder on the BSC as a function of the crossover probability ϵ. The sum-product decoder was run for 20 iterations, using the (3,6)-regular ensemble with block lengths 2^i for $i = 10, 11, \ldots, 20$ (top), the irregular ensemble with block lengths 2^i for $i = 10, 11, \ldots, 20$ (middle), and the rate-$\frac{1}{2}$ WiMAX code with block length 2304 (bottom). The transmission of a total of 2^{30} bits was simulated for each code, block length and crossover probability. For the (3,6)-regular ensemble and the irregular ensemble, the bit-error rate decreases as the block length increases. Note that the vertical axis is logarithmic, and that if no errors occurred for a particular block length and crossover probability that particular result is not plotted.

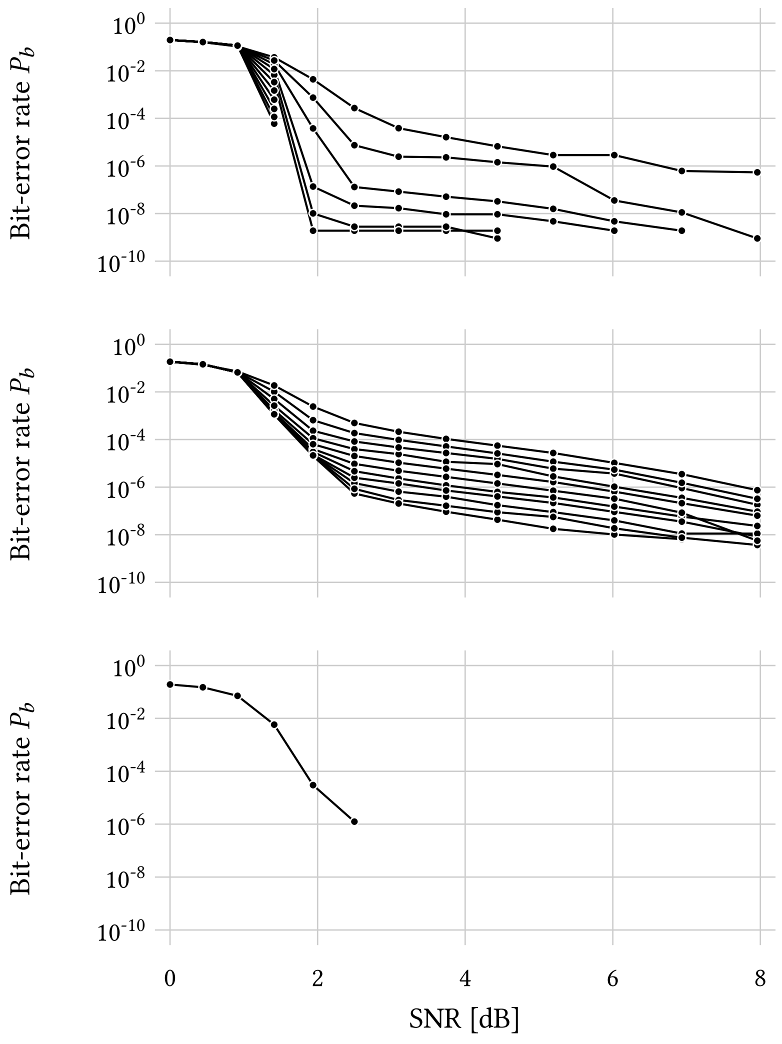


Figure 6.2: Bit-error rate P_b of the sum-product decoder on the BAWGNC as a function of the signal-to-noise ratio (SNR) in dB. The sum-product decoder was run for 20 iterations, using the (3,6)-regular ensemble with block lengths 2^i for $i = 10, 11, \ldots, 20$ (top), the irregular ensemble with block lengths 2^i for $i = 10, 11, \ldots, 20$ (middle), and the rate-$\frac{1}{2}$ WiMAX code with block length 2304 (bottom). The transmission of a total of 2^{30} bits was simulated for each code, block length and crossover probability. For the (3,6)-regular ensemble and the irregular ensemble, the bit-error rate decreases as the block length increases. Note that the vertical axis is logarithmic, and that if no errors occurred for a particular block length and crossover probability that particular result is not plotted.

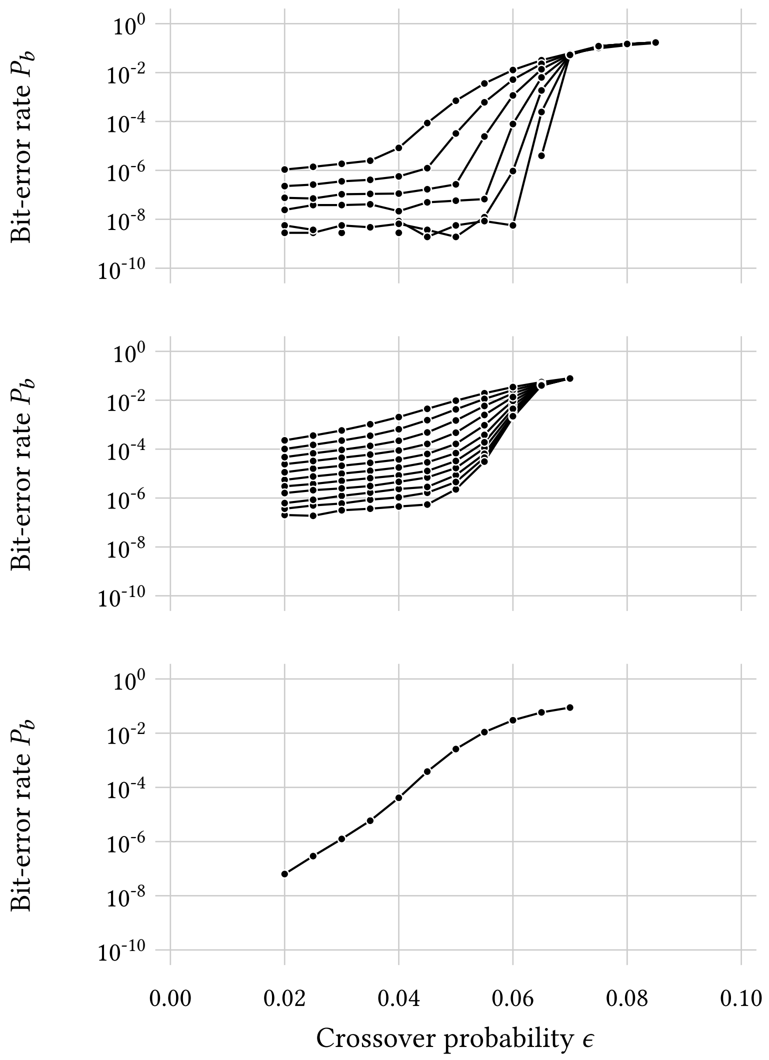


Figure 6.3: Bit-error rate P_b of the stochastic bit-flipping decoder on the BSC as a function of the crossover probability ϵ. The stochastic bit-flipping decoder was run for 100 iterations, using the (3,6)-regular ensemble with block lengths 2^i for $i = 10, 11, \ldots, 20$ (top), the irregular ensemble with block lengths 2^i for $i = 10, 11, \ldots, 20$ (middle), and the rate-$\frac{1}{2}$ WiMAX code with block length 2304 (bottom). The parameter values of the stochastic bit-flipping decoder are shown in Table 6.1. The transmission of a total of 2^{30} bits was simulated for each code, block length and signal-to-noise ratio. For the (3,6)-regular ensemble and the irregular ensemble, the bit-error rate decreases as the block length increases. Note that the vertical axis is logarithmic, and that if no errors occurred for a particular block length and crossover probability that particular result is not plotted.

the gradient-descent bit-flipping decoder showing that despite using only hard values, adding noise to the decoding process can improve the error-correcting performance.

It is interesting to note the performance of the irregular ensemble. As can be seen from the figures, the performance of the irregular ensemble is significantly worse than the (3,6)-regular ensemble on both the BSC and the BAWGNC. At this point it is important to remember that the degree distribution of the irregular ensemble we have used here was optimized in the asymptotic setting. The threshold is indeed better than for the (3,6)-regular ensemble but the threshold alone does not guarantee good performance. On the other hand, this also does not mean that irregular cannot perform well. The rate-$\frac{1}{2}$ WiMAX code is an irregular code and performs better than the (3,6)-regular ensemble at the short block length of approximately 2000 bits. However, at longer block lengths the (3,6)-regular ensemble performs better than the short rate-$\frac{1}{2}$ WiMAX code.

To put the performance of the decoders into perspective, the Shannon limit for codes with rate-$\frac{1}{2}$ on the BSC is approximately $\epsilon = 0.11$. The Shannon limit for rate-$\frac{1}{2}$ codes on the BAWGNC is 0.19 dB or, expressed in terms of the standard deviation of the noise, $\sigma = 0.98$ (Richardson et al., 2001).

6.2 Complexity of approximate lower triangular encoding

The performance of the encoding method by Richardson and Urbanke (2001b) using an approximate lower triangular form of the parity-check matrix was tested using codes from the same (3,6)-regular ensemble and irregular ensemble already used for comparing decoders. The encoding time was recorded for block lengths 2^i for $i = 10, 11, \ldots, 20$ with 2^{24} total encoded bits for each block length. The repetitions for each block length n were made similarly to how the decoders were tested. The following was repeated $\left\lfloor \frac{2^{24}}{n} \right\rfloor$ times for each block length: draw a new code and encode a word using the drawn code.

The encoding tests were run on the following hardware: HP ProLiant BL465c G6 with two 2.6 GHz AMD Opteron 2435 CPUs with six cores each, and 32 GiB of DDR2-800 main memory. The encoder implementation is single-threaded.

6.2.1 Results

Figure 6.6 shows the encoding time in seconds as a function of the block length of the regular and irregular code side by side. In addition to showing the total encoding time, the time taken to perform the greedy approximate lower triangulation, the time taken to perform Gaussian elimination and invert ϕ, and the actual encoding time ignoring the preprocessing steps are shown. It is clear that

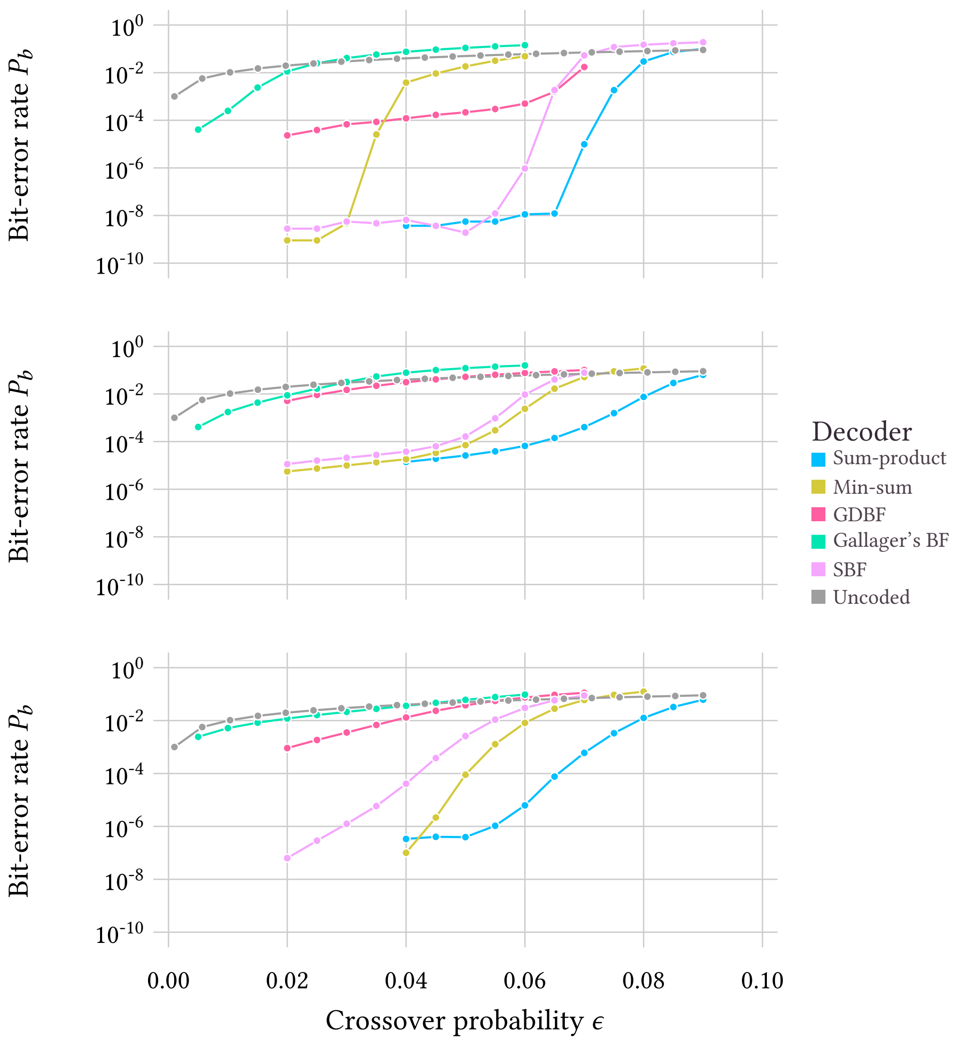


Figure 6.4: Comparison of decoders on the BSC. The bit-error rate P_b as a function of the crossover probability ϵ using the (3,6)-regular ensemble with block length 2^{14} (top), the irregular ensemble with block length 2^{14} (middle), and the rate-$\frac{1}{2}$ WiMAX code with block length 2304 (bottom). Five decoders were compared: the sum-product decoder, the min-sum decoder, the gradient-descent bit-flipping decoder (GDBF), Gallager's bit-flipping decoder (Gallager's BF), and the stochastic bit-flipping decoders. Also shown is uncoded transmission in gray. The sum-product decoder and the min-sum decoder were run with 20 iterations; the rest were run with 100 iterations. The parameter values of the stochastic bit-flipping decoder and the gradient-descent bit-flipping decoder are shown in Table 6.1 and Table 6.2. The transmission of a total of 2^{30} bits was simulated for each code, block length and crossover probability. Note that the vertical axis is logarithmic, and that if no errors occurred for a particular block length and crossover probability that particular result is not plotted.

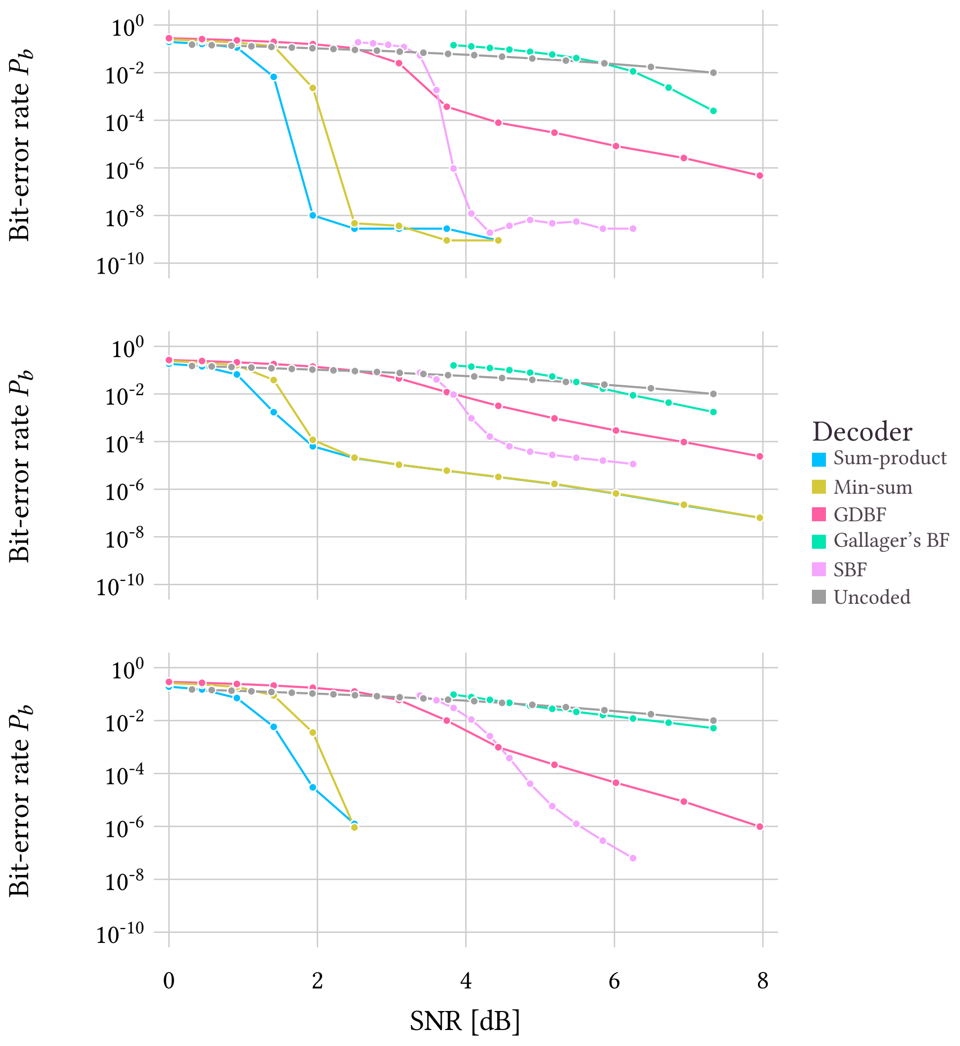


Figure 6.5: Comparison of decoders on the BAWGNC. The bit-error rate P_b as a function of the signal-to-noise ratio (SNR) in dB using the (3,6)-regular ensemble with block length 2^{14} (top), the irregular ensemble with block length 2^{14} (middle), and the rate-$\frac{1}{2}$ WiMAX code with block length 2304 (bottom). Five decoders were compared: the sum-product decoder, the min-sum decoder, the gradient-descent bit-flipping decoder (GDBF), Gallager's bit-flipping decoder (Gallager's BF), and the stochastic bit-flipping decoders. Also shown is uncoded transmission in gray. The sum-product decoder and the min-sum decoder were run with 20 iterations; the rest were run with 100 iterations. The parameter values of the stochastic bit-flipping decoder and the gradient-descent bit-flipping decoder are shown in Table 6.1 and Table 6.2. The transmission of a total of 2^{30} bits was simulated for each code, block length and signal-to-noise ratio. Note that the vertical axis is logarithmic, and that if no errors occurred for a particular block length and crossover probability that particular result is not plotted.

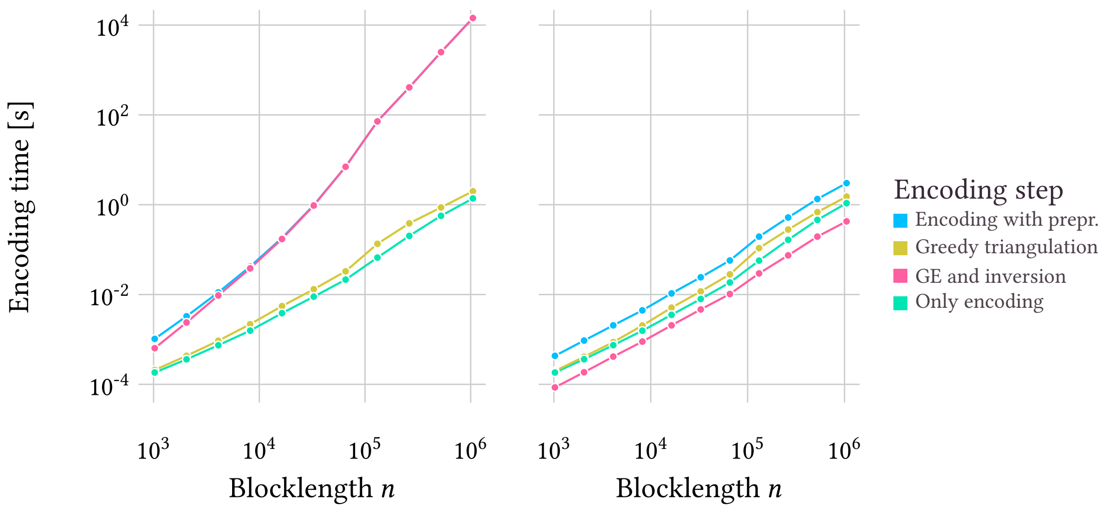


Figure 6.6: Encoding time t as a function of the gap g for the (3,6)-regular ensemble (top) and the irregular ensemble (bottom) using the method by Richardson and Urbanke (2001b). The block length n was 2^i for $i = 10, 11, \ldots, 20$. The graphs show the total encoding time including preprocessing steps, the time to perform greedy approximate upper triangulation of the parity-check matrix, the time to perform Gaussian elimination and invert ϕ, and the time to perform only the actual encoding without preprocessing. The encoding was repeated so that a total of 2^{20} bits were encoded for each block length and ensemble. Both axes are logarithmic.

the main computational cost in the encoding method by Richardson and Urbanke (2001b) comes from performing the Gaussian elimination and inverting ϕ. For the regular code this can clearly be attributed to the linear growth of the gap g as can be seen in Figure 6.7. However, the average gap of codes from the irregular ensemble is consistently small up to the block length 2^{20}. This leads to consistently low total encoding times including preprocessing. In practice, the preprocessing is done beforehand and does not need to be repeated each time a new block is encoded. In the current software implementation, the encoding times excluding preprocessing are similar for both the (3,6)-regular ensemble and the irregular ensemble. It is interesting to note that the degree distribution for the irregular ensemble is not significantly different from the (3,6)-regular case. The two ensembles have similar average variable and check node degrees, but the reason for the small average gap for the irregular ensemble is essentially the larger number variable nodes of degree two.

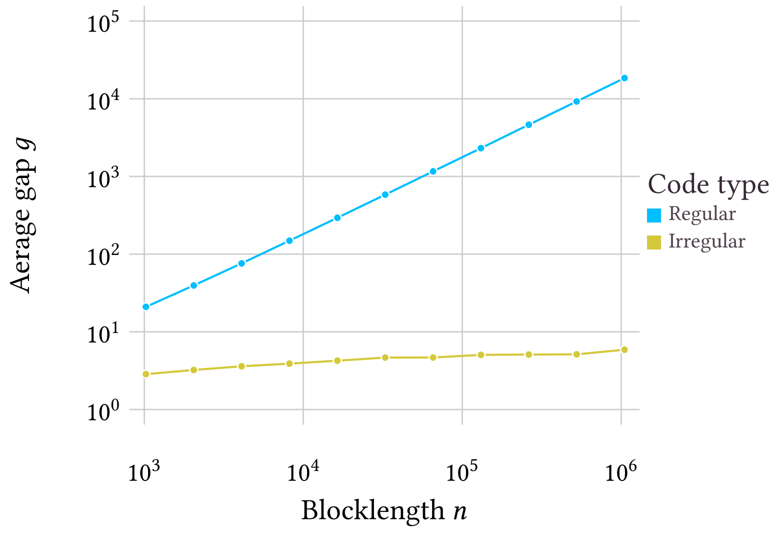


Figure 6.7: Average gap g of the parity-check matrix after greedy approximate lower triangulation as a function of the block length n for the (3,6)-regular ensemble and the irregular ensemble. The block length n was 2^i for $i = 10, 11, \ldots, 20$. Note that the average gap of the irregular code is consistently less than 10 even for the block length 2^{20}. Both axes are logarithmic.

6.3 The stochastic bit-flipping decoder on the GPU

As noted in Section 5.3.1, to fully utilize the GPU a sufficient number of threads needs to be started simultaneously or there needs to be a sufficient number of independent instructions executed in each thread. In practice, this means that a sufficient number of received words need to be decoded simultaneously. The number of threads to use depends in general on the type of GPU used for computations. The implementation for the current work was tested on two CUDA devices: (i) the NVIDIA Tesla M2090 and (ii) the NVIDIA Tesla K40.

The decoder was tested on both devices using two types of codes. The first is the rate-$\frac{1}{2}$ code of block length 1536 defined in the WiMAX standard and the second is a (3,4)-regular QC-LDPC code, also with block length 1536. The (3,4)-regular code was generated by first generating a (3,4)-regular code of block length 24 using the configuration model. Then, for each nonzero entry of the smaller parity-check matrix a random shift value was drawn uniformly at random between 0 and 63. By setting the nonzero entries in the smaller parity-check matrix to the randomly drawn shift values we form the model matrix of the (3,4)-regular QC-LDPC code. By using the block length 1536 with a model matrix with 24 columns the sub-matrix size is $2^6 = 64$ meaning that checking for divisibility by the sub-matrix size can be done quickly.

To reach a high throughput, the decoder was tested using different configurations. The free parameters were the number of threads per multiprocessor core and the vector type used for bitwise Boolean operations. Let w be the number of bits in a vector type, n_A the number of threads used per core and n_C the total number of cores available on the GPU. The total number of words being decoded simultaneously on the GPU is then $n_C n_A w$. The decoder was tested on both GPUs with $n_A = 2^i$ for all $0, 1, \ldots, 5$. Additionally, the decoder was tested using the vector types `uint1`, `uint2` and `uint4` for all bitwise Boolean operations, meaning that each thread was responsible for decoding $w = 32, 64$ or 128 words, respectively, with each vector type. The decoder was run for 100 iterations on both GPUs with all configurations. For each set of parameters, the decoder was run once to decode $n_C n_A w$ words and the execution time of the decoding process was recorded.

6.3.1 Results

The decoding throughput of the decoder on the NVIDIA Tesla M2090 is shown in Figure 6.8 as a function of the code, the number of threads per core and the number of words w decoded on each thread. The decoding throughput clearly increases as the number of threads per core is increased, eventually plateauing at 8 threads per core. In addition, there is a considerable increase in decoding throughput when decoding 64 words per thread instead of 32 words per thread, but decoding 128 words per thread no longer increases the throughput considerably. The peak decoding throughput on the NVIDIA Tesla M2090 was 350 Mb/s with the rate-$\frac{1}{2}$ WiMAX code tested and 700 Mb/s with the (3,4)-regular code. The global memory bandwidth used on the NVIDIA Tesla M2090, shown in Figure 6.9, follows a similar pattern in terms of the number of threads per core and the number of words per thread. With both codes use of the global memory bandwidth reaches a maximum of approximately 100 GB/s, which is 60 % of the maximum theoretical bandwidth of 176 GB/s.

The decoding throughput and use of global memory bandwidth for decoder on the NVIDIA Tesla K40 are shown in Figure 6.10 and Figure 6.11. The main difference to the NVIDIA Tesla M2090 is that the maximum decoding throughput and use of global memory bandwidth are approximately a factor of 2 higher on the NVIDIA Tesla K40. The peak decoding throughput with the NVIDIA Tesla K40 is approximately 700 Mb/s with the rate-$\frac{1}{2}$ WiMAX code and approximately 1200 Mb/s with the (3,4)-regular code. The GPU used in the work by Wang et al. (2013), the NVIDIA GTX TITAN, has similar specifications to the NVIDIA Tesla K40 and is based on the same microarchitecture. Compared to the work by Wang et al., the decoder implemented for this thesis achieves approximately twice the throughput with the rate-$\frac{1}{2}$ WiMAX code using 100 iterations, while the min-sum decoder in the work by Wang et al. uses 10 iterations. However, one has to keep in mind that the message-passing decoders generally perform better than bit-flipping decoders with a smaller number of iterations. The peak use of global memory bandwidth is nearly 200 GB/s with the rate-$\frac{1}{2}$ WiMAX code and approximately 170 GB/s with the (3,4)-regular code. The maximum theoretical bandwidth of the global memory on the NVIDIA Tesla K40 is 288 GB/s, meaning that nearly 70 % of the maximum bandwidth is used with the rate-$\frac{1}{2}$ WiMAX code. Compared to the decoder on the NVIDIA Tesla M2090, using only one thread per core on the NVIDIA Tesla K40 already comes close to the maximum performance. Using the rate-$\frac{1}{2}$ WiMAX code, one thread per core, and decoding 128 words per thread achieves a decoding throughput of nearly 600 MB/s, while the maximum is approximately 700 MB/s. The corresponding values on the NVIDIA Tesla M2090 are 100 MB/s and 400 MB/s. The difference is likely a result of the NVIDIA Tesla K40 having nearly 6 times as many cores as the NVIDIA Tesla M2090, while the global memory bandwidth is only approximately 2 times higher. This means that the global memory bandwidth can be saturated on the NVIDIA Tesla K40 with a smaller number of threads per core compared to the NVIDIA Tesla M2090.

It is important to remember the trade-offs one needs to consider when implementing decoders. Particularly important are the trade-offs between throughput, latency and error-correcting performance. We have seen in the decoder comparison that the current decoder has relatively good error-correcting performance. In addition, it can achieve high throughputs. However, because of the large number of words that need to be decoded simultaneously to achieve high throughputs, the latencies are relatively high. The configuration achieving minimum latency with maximum throughput on the NVIDIA M2090 is to run 8 threads for each core and to decode 64 words in each thread. Using this configuration a total of 400 Mb are decoded simultaneously in one pass of the decoder, and the time taken to decode this amount is approximately 1 s with the rate-$\frac{1}{2}$ WiMAX code and 0.6 s with the (3,4)-regular QC-LDPC code. On the NVIDIA Tesla K40, using the rate-$\frac{1}{2}$ WiMAX code one can run only 2 threads for each core, decode 128 words in each thread, and reach the maximum decoding throughput. Using this configuration approximately 1.1 Gb are decoded simultaneously, which takes 1.5 s to complete. Using the (3,4)-regular QC-LDPC code, one needs to run 4 threads for each core and decode 128 words per thread to reach the maximum decoding throughput. Doing so results in 2.3 Gb being decoded simultaneously with a latency of 1.8 s. In comparison to the current implementation, the decoder presented by Wang et al. has decoding latencies on the order of 1 ms, which is significantly lower than the latencies of the implementation presented here.

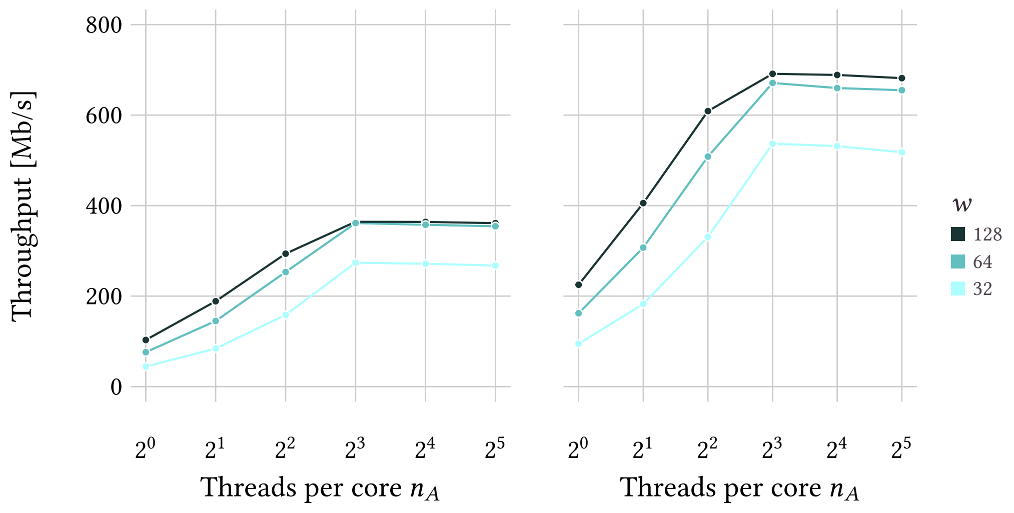


Figure 6.8: Decoding throughput of the stochastic bit-flipping decoder on the NVIDIA M2090 with the rate-$\frac{1}{2}$ WiMAX code of block length 1536 (left) and a (3,4)-regular QC-LDPC code of block length 1536 (right). The number of threads was n_A multiplied by the number of cores available on the GPU. The number of words decoded simultaneously by each thread is given by w. The horizontal axis is logarithmic.

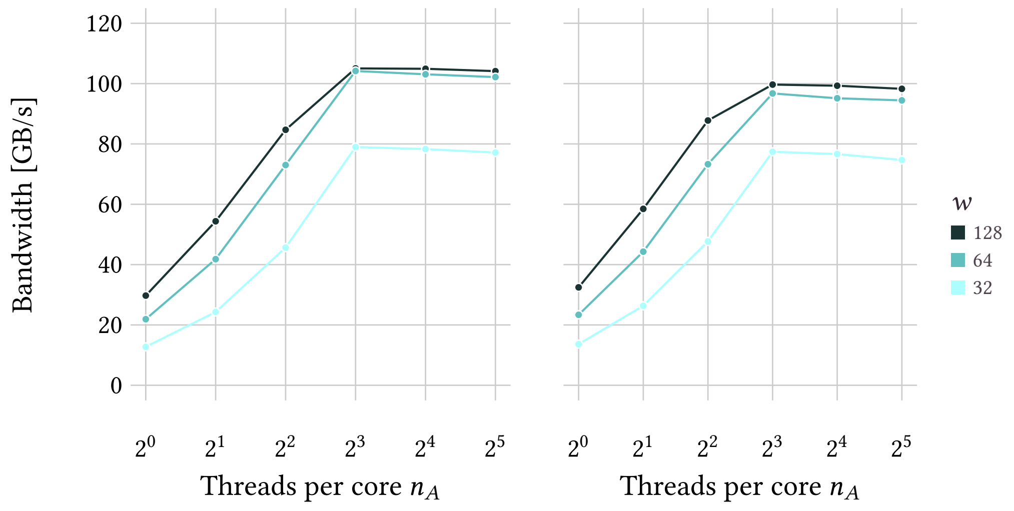


Figure 6.9: Global memory bandwidth used by the stochastic bit-flipping decoder on the NVIDIA M2090 with the rate-$\frac{1}{2}$ WiMAX code of block length 1536 (left) and a (3,4)-regular QC-LDPC code of block length 1536 (right). The number of threads per core on the GPU is denoted by n_A. The number of words decoded simultaneously by each thread is given by w. The horizontal axis is logarithmic.

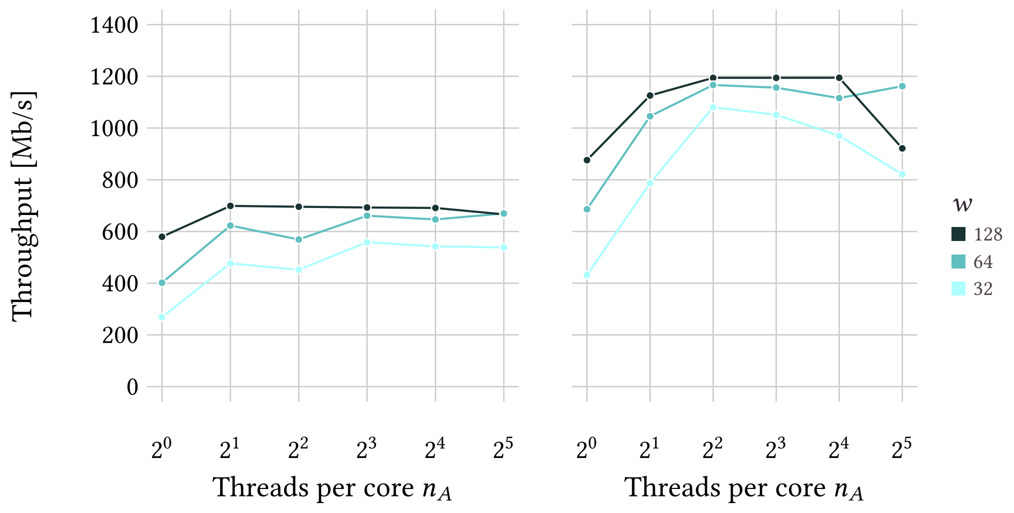


Figure 6.10: Decoding throughput of the stochastic bit-flipping decoder on the NVIDIA K40 with the rate-$\frac{1}{2}$ WiMAX code of block length 1536 (left) and a (3,4)-regular QC-LDPC code of block length 1536 (right). The number of threads was n_A multiplied by the number of cores available on the GPU. The number of words decoded simultaneously by each thread is given by w. The horizontal axis is logarithmic.

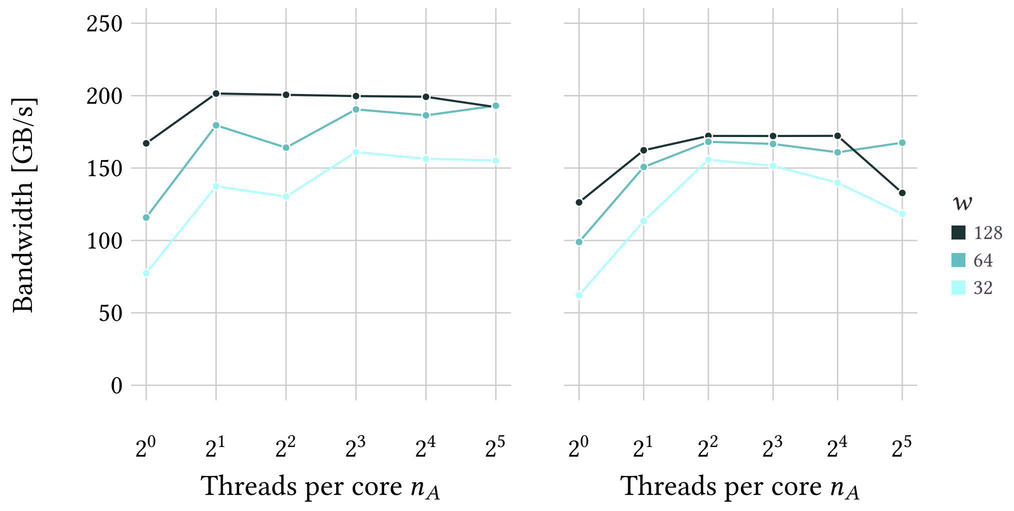


Figure 6.11: Global memory bandwidth used by the stochastic bit-flipping decoder on the NVIDIA K40 with the rate-$\frac{1}{2}$ WiMAX code of block length 1536 (left) and a (3,4)-regular QC-LDPC code of block length 1536 (right). The number of threads per core on the GPU is denoted by n_A. The number of words decoded simultaneously by each thread is given by w. The horizontal axis is logarithmic.

Chapter 7

Conclusion

After their rediscovery in the 1990's, LDPC codes have shown themselves to be viable codes in theory, with constructions and decoders that approach the Shannon limit; and in practice, having been included in various communication standards. The field has matured in the last years, but there is still room for improvements both in encoding and decoding of LDPC codes. To the best of the author's knowledge, encoding can still not be done in linear time for general LDPC codes. Resolving whether or not it can be done would be an important result. Current decoders can still be improved by reducing their complexity to increase throughput, reduce latency, and allow longer block lengths to be used for better error-correction. The GPU implementation of the stochastic bit-flipping decoder presented in this thesis is a high-throughput decoder with relatively good error-correcting performance.

The comparison of decoders in the previous chapter mainly introduces the stochastic bit-flipping decoder as a new decoder. The focus of the stochastic bit-flipping decoder is on low complexity as it uses only hard channel values, and on examining to what extent adding noise to the decoding process helps for the error-correcting performance. The decoder performs well compared to the sum-product decoder considering its simplicity, and the gap in error-correcting performance is especially small on the BSC. An interesting comparison is that between the stochastic bit-flipping decoder and the gradient-descent bit-flipping decoder on the BAWGNC. While the gradient-descent bit-flipping decoder makes use of the soft channel values and the stochastic-bit flipping decoder does not, the stochastic bit-flipping decoder still has better error-correcting performance with the codes tested in this thesis.

The important trade-offs when considering decoders are those between error-correcting performance, throughput and latency. With the GPU implementation of the stochastic bit-flipping decoder it is clear that its simplicity allows for high-throughput implementations, while the decoder comparison shows that the decoder is capable of relatively good error-correcting performance considering its simplicity. The downside of the current GPU implementation is the high decoding latency which is a consequence of the decoder being block-parallel. The current high latency could be reduced by considering bit-parallel implementations of the decoder, meaning that multiple bits in the same codeword are processed in parallel. When using QC-LDPC codes this could be done by processing all columns within a major column of the parity-check matrix in parallel as the second neighbors of the columns in a major column are independent. For example, with the codes used here which have a sub-matrix size of 64×64, the potential gain in latency would be a factor of 64, bringing the latencies to the order of 10–100 ms.

In this work we optimized the free parameters of the stochastic bit-flipping decoder with a simple grid search and chose the parameters depending on the code. To ensure that the stochastic bit-flipping decoder performs as well as possible, the parameters should be further optimized by using a finer grid. Additionally, the grid search shows that the optimal parameters are different at different levels of noise. Adjusting the parameters based on the level of noise in the channel and the current iteration may help further improve the error-correcting performance. The number of iterations used for the decoder is another free parameter which is important for the error-correcting performance. However, changing the number of iterations directly affects throughput. Increasing the number of iterations reduces the bit-error rate, but it needs to be examined more thoroughly what exactly is a suitable number of iterations such that error-correcting performance and throughput are well balanced. Having said that, it should be remembered that an optimal set of parameter is unlikely to exist for every situation. Most importantly though, a thorough comparison of the bit-error rate, throughput and latency should be made with dedicated hardware designs of various decoders. In general, the stochastic bit-flipping decoder should also be compared in complexity and error-correcting performance to more advanced bit-flipping decoders such as the noisy gradient-descent decoder. Additionally, the gradient-descent bit-flipping decoder was optimized with only one free parameter in this work. Future work might involve including more free parameters to the gradient-descent bit-flipping decoder to improve its performance.

In addition to the free parameters of the decoder, code constructions and the block length need to be considered when talking about error-correcting performance. First, the block length is an important factor in the error-correcting performance as increasing it can significantly improve error-correcting performance. Most importantly, increasing the block length lowers the error floor. We saw in Chapter 6 that for example the (3,6)-regular ensemble can produce error floors lower than 10^{-8} already at the block length 2^{14}. Using a low-complexity decoder and long block lengths should be considered as an option for high-throughput decoding with low bit-error rates. However, by increasing the block length one is again giving up on the decoding latency. Second, code constructions should be looked at more closely. The results in this thesis show that the stochastic bit-flipping decoder performs well with the (3,6)-regular ensemble, but performs worse with for example the rate-$\frac{1}{2}$ WiMAX code. Generally, message-passing decoders are considered first when designing good codes which may lead to worse results with bit-flipping decoders. Thus it may be beneficial to examine what types of codes work especially well with bit-flipping decoders in more detail.

As the literature review in this thesis shows, there exists a plethora of decoders for LDPC codes. Future work on the stochastic bit-flipping decoder must show that it can compete also on error-correcting performance, and not only on throughput, for it to be a viable decoder alternative. To date, message-passing decoders are still preferred because of their good error-correcting performance, but bit-flipping decoders have been improving in error-correcting performance. On the other hand, message-passing decoders have been reducing in complexity with the binary message-passing decoder being an extreme example. However, the line between message-passing decoders and bit-flipping decoders is currently being blurred with ideas from message-passing decoders being incorporated into bit-flipping decoders, and vice versa. Future work on decoders will have to weigh the trade-offs between error-correcting performance and complexity thoroughly, and ideally attempt to find decoder designs that are clearly better in both respects.

The current state-of-the-art in encoding is the method by Richardson and Urbanke (2001b) which is based on permuting the rows and columns of the parity-check matrix so that the resulting parity-check matrix is in approximate lower triangular form. While the method has linear time complexity for a useful subset of codes, it does not have linear time complexity for general LDPC codes. Codes which can be encoded in linear time are good enough for inclusion in standards, such as the QC-LDPC codes in the WiMAX standard. However, a method for linear-time encoding of general LDPC codes could allow more freedom in designing codes that work well with different decoders, as well as allowing scaling of the block length to larger values. One reason for the difficulty of encoding LDPC codes is that good codes should in some sense protect all information bits equally well. How well the code does this is likely captured to some extent by the size of the gap when doing greedy upper triangulation. It remains an open question to determine whether alternative encoding methods exist that can circumvent the problem of large gaps for arbitrary LDPC codes.

In conclusion, this work has presented a stochastic bit-flipping decoder which has been shown to be easily parallelizable by decoding multiple words at a time, allowing the decoder to reach high throughputs. In addition, a review of the current state-of-the-art of encoding and decoding of LDPC codes has been given. Experimental results show that the stochastic bit-flipping decoder has relatively good error-correcting performance at low complexity. The prospect of low-complexity encoders and decoders that would allow scaling of the block length significantly is especially exciting as this would allow further reductions in bit-error rates, albeit with a cost in latency. As an extreme example, entire hard-drives or even multiple hard-drives could be encoded with a single block making them more robust to errors. While the current work does not directly allow this, it shows that there is room for improvement in making low-complexity decoders that still have good error-correcting performance. Resolving whether or not general LDPC codes can be encoded in linear time is another important step in the field, and especially for increasing the block lengths significantly.

Bibliography

Abburi, K. (2011). A scalable LDPC decoder on GPU. In *2011 24th International Conference on VLSI Design (VLSI Design)*, pages 183–188.

Alava, M., Ardelius, J., Aurell, E., Kaski, P., Krishnamurthy, S., Orponen, P., and Seitz, S. (2008). Circumspect descent prevails in solving random constraint satisfaction problems. *Proceedings of the National Academy of Sciences*, 105(40):15253–15257.

Amraoui, A., Montanari, A., Richardson, T., and Urbanke, R. (2009). Finite-Length scaling for iteratively decoded LDPC ensembles. *IEEE Transactions on Information Theory*, 55(2):473–498.

Bahl, L., Cocke, J., Jelinek, F., and Raviv, J. (1974). Optimal decoding of linear codes for minimizing symbol error rate (Corresp.). *IEEE Transactions on Information Theory*, 20(2):284–287.

Berlekamp, E. R., McEliece, R. J., and van Tilborg, H. C. A. (1978). On the inherent intractability of certain coding problems. *IEEE Transactions on Information Theory*, 24(3):384–386.

Berrou, C., Glavieux, A., and Thitimajshima, P. (1993). Near shannon limit error-correcting coding and decoding: Turbo-codes. 1. In *Conference Record, IEEE International Conference on Communications*, volume 2, pages 1064–1070 vol.2.

Berrou, C., Pyndiah, R., Adde, P., Douillard, C., and Le Bidan, R. (2005). An overview of turbo codes and their applications. In *The European Conference on Wireless Technology, 2005*, pages 1–9.

Bollobás, B. (2001). *Random Graphs.* Cambridge University Press.

Boutillon, E., Guillou, F., and Danger, J. (2003). lambda-min decoding algorithm of regular and irregular LDPC codes. In *3rd International Symposium on Turbo Codes & Related Topics.*

Burshtein, D. (2009). Iterative approximate linear programming decoding of LDPC codes with linear complexity. *IEEE Transactions on Information Theory*, 55(11):4835–4859.

Burshtein, D. and Goldenberg, I. (2011). Improved linear programming decoding of LDPC codes and bounds on the minimum and fractional distance. *IEEE Transactions on Information Theory*, 57(11):7386–7402.

Chang, C., Chang, Y., Huang, M., and Huang, B. (2011). Accelerating regular LDPC code decoders on GPUs. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 4(3):653–659.

Chen, J., Dholakia, A., Eleftheriou, E., Fossorier, M., and Hu, X. (2005). Reduced-Complexity decoding of LDPC codes. *IEEE Transactions on Communications*, 53(8):1288–1299.

Chen, J. and Fossorier, M. P. (2002a). Density evolution for two improved BP-based decoding algorithms of LDPC codes. *IEEE Communications Letters*, 6(5):208–210.

Chen, J. and Fossorier, M. P. (2002b). Near optimum universal belief propagation based decoding of low-density parity check codes. *IEEE Transactions on Communications*, 50(3):406–414.

Cho, J., Kim, J., and Sung, W. (2010). VLSI implementation of a High-Throughput Soft-Bit-Flipping decoder for geometric LDPC codes. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 57(5):1083–1094.

Chung, S. (2000). *On the construction of some capacity-approaching coding schemes*. PhD thesis, Massachusetts Institute of Technology.

Chung, S., Forney Jr, G. D., Richardson, T. J., and Urbanke, R. (2001a). On the design of low-density parity-check codes within 0.0045 dB of the shannon limit. *IEEE Communications Letters*, 5(2):58–60.

Chung, S., Richardson, T. J., and Urbanke, R. L. (2001b). Analysis of sum-product decoding of low-density parity-check codes using a gaussian approximation. *IEEE Transactions on Information Theory*, 47(2):657–670.

Cushon, K., Hemati, S., Leroux, C., Mannor, S., and Gross, W. (2014). High-Throughput Energy-Efficient LDPC decoders using differential binary message passing. *IEEE Transactions on Signal Processing*, 62(3):619–631.

Di, C., Proietti, D., Telatar, I., Richardson, T., and Urbanke, R. (2002). Finite-length analysis of low-density parity-check codes on the binary erasure channel. *IEEE Transactions on Information Theory*, 48(6):1570–1579.

Elidan, G. (2006). Residual belief propagation: Informed scheduling for asynchronous message passing. In *Proceedings of the Twenty-second Conference on Uncertainty in AI*.

ETSI (2012). Digital video broadcasting (DVB); frame structure channel coding and modulation for a second generation digital transmission system for cable systems (DVB-C2).

ETSI (2013a). Digital video broadcasting (DVB); frame structure channel coding and modulation for a second generation digital terrestrial television broadcasting system (DVB-T2).

ETSI (2013b). Digital video broadcasting (DVB); second generation framing structure, channel coding and modulation systems for broadcasting, interactive services, news gathering and other broadband satellite applications (DVB-S2).

Falcão, G., Sousa, L., and Silva, V. (2008). Massive parallel LDPC decoding on GPU. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 83–90.

Feldman, J. (2003). *Decoding error-correcting codes via linear programming*. PhD thesis, Massachusetts Institute of Technology.

Feldman, J., Wainwright, M., and Karger, D. (2005). Using linear programming to decode binary linear codes. *IEEE Transactions on Information Theory*, 51(3):954–972.

Fossorier, M. P., Mihaljevic, M., and Imai, H. (1999). Reduced complexity iterative decoding of low-density parity check codes based on belief propagation. *IEEE Transactions on Communications*, 47(5):673–680.

Freundlich, S., Burshtein, D., and Litsyn, S. (2007). Approximately lower triangular ensembles of LDPC codes with linear encoding complexity. *IEEE Transactions on Information Theory*, 53(4):1484–1494.

Gallager, R. G. (1962). Low-density parity-check codes. *IRE Transactions on Information Theory*, 8(1):21–28.

Gallager, R. G. (1963). *Low-Density Parity-Check Codes*, volume 21 of *Research Monograph Series*. Cambridge, MA, USA.

Gaudet, V. and Rapley, A. (2003). Iterative decoding using stochastic computation. *Electronics Letters*, 39(3):299.

Goldin, D. and Burshtein, D. (2013). Iterative linear programming decoding of nonbinary LDPC codes with linear complexity. *IEEE Transactions on Information Theory*, 59(1):282–300.

Greenhill, C., McKay, B. D., and Wang, X. (2006). Asymptotic enumeration of sparse 0–1 matrices with irregular row and column sums. *Journal of Combinatorial Theory, Series A*, 113(2):291–324.

Grönroos, S., Nybom, K., and Björkqvist, J. (2012). Efficient GPU and CPU-based LDPC decoders for long codewords. *Analog Integrated Circuits and Signal Processing*, 73(2):583–595.

Gross, W., Gaudet, V., and Milner, A. (2005). Stochastic implementation of LDPC decoders. In *Conference Record of the Thirty-Ninth Asilomar Conference on Signals, Systems and Computers*, pages 713–717.

Hagenauer, J., Offer, E., and Papke, L. (1996). Iterative decoding of binary block and convolutional codes. *IEEE Transactions on Information Theory*, 42(2):429–445.

Hagenauer, J. and Papke, L. (1994). Decoding "Turbo"-codes with the soft output viterbi algorithm (SOVA). In *IEEE International Symposium on Information Theory*, page 164.

He, Y., Li, H., Sun, S., and Li, L. (2003). Threshold-based design of quantized decoder for LDPC codes. In *IEEE International Symposium on Information Theory*, page 149.

Hemati, S. and Banihashemi, A. (2006). Dynamics and performance analysis of analog iterative decoding for low-density parity-check (LDPC) codes. *IEEE Transactions on Communications*, 54(1):61–70.

Hocevar, D. (2004). A reduced complexity decoder architecture via layered decoding of LDPC codes. In *IEEE Workshop on Signal Processing Systems*, pages 107–112.

Huang, K., Gaudet, V., and Salehi, M. (2013). A scaling method for stochastic LDPC decoding over the binary symmetric channel. In *47th Annual Conference on Information Sciences and Systems*, pages 1–5.

IEEE (2008). IEEE standard for Floating-Point arithmetic. *IEEE Std 754-2008*.

IEEE (2009). IEEE standard for air interface for broadband wireless access systems. *IEEE Std 802.16-2012*.

IEEE (2012a). IEEE Standard for Ethernet. *IEEE Std 802.3-2012*.

IEEE (2012b). Part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. *IEEE Std 802.11 2012*.

Intel (2012). Intel architecture, instruction set extensions programming, reference.

Ismail, M., Coon, J., Ahmed, I., Armour, S., and McGeehan, J. (2013). Turbo adaptive threshold bit flipping for LDPC decoding. *IEEE Wireless Communications Letters*, 2(1):118–121.

Kang, S. and Moon, J. (2012). Parallel LDPC decoder implementation on GPU based on unbalanced memory coalescing. In *IEEE International Conference on Communications*, pages 3692–3697.

Kolesnik, V. D. (1971). Probabilistic decoding of majority codes. *Problemy Peredachi Informatsii*, 7(3):3–12.

Kou, Y., Lin, S., and Fossorier, M. (2001). Low-density parity-check codes based on finite geometries: a rediscovery and new results. *IEEE Transactions on Information Theory*, 47(7):2711–2736.

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

Lauritzen, S. L. and Spiegelhalter, D. J. (1988). Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society. Series B (Methodological)*, 50(2):157–224.

Luby, M. G., Mitzenmacher, M., Shokrollahi, A., and Spielman, D. A. (1998). Analysis of low density codes and improved designs using irregular graphs. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, pages 249–258.

Luby, M. G., Mitzenmacher, M., Shokrollahi, M. A., and Spielman, D. A. (2001a). Efficient erasure correcting codes. *IEEE Transactions on Information Theory*, 47(2):569–584.

Luby, M. G., Mitzenmacher, M., Shokrollahi, M. A., and Spielman, D. A. (2001b). Improved low-density parity-check codes using irregular graphs. *IEEE Transactions on Information Theory*, 47(2):585–598.

Luby, M. G., Mitzenmacher, M., Shokrollahi, M. A., Spielman, D. A., and Stemann, V. (1997). Practical loss-resilient codes. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, pages 150–159.

MacKay, D. J. (1999). Good error-correcting codes based on very sparse matrices. *IEEE Transactions on Information Theory*, 45(2):399–431.

MacKay, D. J. (2003). *Information theory, inference, and learning algorithms*, volume 7. Cambridge University Press.

MacKay, D. J. C. (1995). Free energy minimisation algorithm for decoding and cryptanalysis. *Electronics Letters*, 31(6):446–447.

Mansour, M. and Shanbhag, N. (2002). Memory-efficient turbo decoder architectures for LDPC codes. In *IEEE Workshop on Signal Processing Systems*, pages 159–164.

Mao, Y. and Banihashemi, A. (2001). A heuristic search for good low-density parity-check codes at short block lengths. In *IEEE International Conference on Communications, 2001. ICC 2001*, volume 1, pages 41–44.

Martínez-Zaldívar, F. J., Vidal-Maciá, A. M., Gonzalez, A., and Almenar, V. (2011). Tridimensional block multiword LDPC decoding on GPUs. *The Journal of Supercomputing*, 58(3):314–322.

McEliece, R. J., MacKay, D. J. C., and Cheng, J. (1998). Turbo decoding as an instance of pearl's "belief propagation" algorithm. *IEEE Journal on Selected Areas in Communications*, 16(2):140–152.

Miladinovic, N. and Fossorier, M. (2005). Improved bit-flipping decoding of low-density parity-check codes. *IEEE Transactions on Information Theory*, 51(4):1594–1606.

Mobini, N., Banihashemi, A., and Hemati, S. (2009). A differential binary message-passing LDPC decoder. *IEEE Transactions on Communications*, 57(9):2518–2523.

Mohsenin, T., Truong, D., and Baas, B. (2010). A Low-Complexity Message-Passing algorithm for reduced routing congestion in LDPC decoders. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 57(5):1048–1061.

Myung, S., Yang, K., and Kim, J. (2005). Quasi-cyclic LDPC codes for fast encoding. *IEEE Transactions on Information Theory*, 51(8):2894–2901.

Naderi, A., Mannor, S., Sawan, M., and Gross, W. (2011). Delayed stochastic decoding of LDPC codes. *IEEE Transactions on Signal Processing*, 59(11):5617–5626.

Noorshams, N. and Iyengar, A. (2014). A novel stochastic decoding of LDPC codes with quantitative guarantees. *preprint arXiv:1405.6353*.

Nouh, A. and Banihashemi, A. (2002). Bootstrap decoding of low-density parity-check codes. *IEEE Communications Letters*, 6(9):391–393.

NVIDIA (2011). Tesla M2090 dual-slot computing processor module board specification.

NVIDIA (2013). Tesla K40 GPU active accelerator board specification.

NVIDIA (2014). CUDA C Programming Guide v6.5.

Pearl, J. (1982). Reverend Bayes on inference engines: a distributed hierarchical approach. In *Proceedings of the National Conference on Artificial Intelligence*, pages 133–136.

Richardson, T. (2003). Error floors of LDPC codes. In *Proceedings of the annual Allerton conference on communication control and computing*, volume 41, pages 1426–1435.

Richardson, T., Shokrollahi, A., and Urbanke, R. (2000). Design of provably good low-density parity check codes. In *Proceedings of IEEE International Symposium on Information Theory*, page 199.

Richardson, T., Shokrollahi, A., and Urbanke, R. (2002). Finite-length analysis of various low-density parity-check ensembles for the binary erasure channel. In *Proceedings of IEEE International Symposium on Information Theory*.

Richardson, T. and Urbanke, R. (2008). *Modern Coding Theory*. Cambridge University Press.

Richardson, T. J., Shokrollahi, M. A., and Urbanke, R. L. (2001). Design of capacity-approaching irregular low-density parity-check codes. *IEEE Transactions on Information Theory*, 47(2):619–637.

Richardson, T. J. and Urbanke, R. L. (2001a). The capacity of low-density parity-check codes under message-passing decoding. *IEEE Transactions on Information Theory*, 47(2):599–618.

Richardson, T. J. and Urbanke, R. L. (2001b). Efficient encoding of low-density parity-check codes. *IEEE Transactions on Information Theory*, 47(2):638–656.

Schläfer, P., Weis, C., Wehn, N., and Alles, M. (2012). Design space of flexible multigigabit LDPC decoders. *VLSI Design*.

Shannon, C. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423.

Sipser, M. and Spielman, D. A. (1996). Expander codes. *Institute of Electrical and Electronics Engineers. Transactions on Information Theory*, 42(6, part 1):1710–1722. Codes and complexity MR: 1465731.

Sundararajan, G., Winstead, C., and Boutillon, E. (2014). Noisy gradient descent Bit-Flip decoding for LDPC codes. *arXiv:1402.2773 [cs, math]*.

Tanner, R., Sridhara, D., Sridharan, A., Fuja, T., and Costello, D. (2004). LDPC block and convolutional codes based on circulant matrices. *IEEE Transactions on Information Theory*, 50(12):2966–2984.

Tanner, R. M. (1981). A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, 27(5):533–547.

Tehrani, S., Gross, W., and Mannor, S. (2006). Stochastic decoding of LDPC codes. *IEEE Communications Letters*, 10(10):716–718.

Tehrani, S., Mannor, S., and Gross, W. (2008). Fully parallel stochastic LDPC decoders. *IEEE Transactions on Signal Processing*, 56(11):5692–5703.

Tehrani, S., Naderi, A., Kamendje, G., Hemati, S., Mannor, S., and Gross, W. (2010). Majority-Based tracking forecast memories for stochastic LDPC decoding. *IEEE Transactions on Signal Processing*, 58(9):4883–4896.

Tehrani, S. S., Naderi, A., Kamendje, G., Mannor, S., and Gross, W. J. (2011). Tracking forecast memories for stochastic decoding. *Journal of Signal Processing Systems*, 63(1):117–127.

van Leeuwen, J., editor (1990). *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*. MIT Press, Cambridge, MA, USA.

Vanek, M. and Farkas, P. (2009). Fast parallel weighted bit flipping decoding algorithm for LDPC codes. In *Wireless Telecommunications Symposium*, pages 1–4.

Vila Casado, A., Griot, M., and Wesel, R. (2007). Informed dynamic scheduling for Belief-Propagation decoding of LDPC codes. In *IEEE International Conference on Communications*, pages 932–937.

Viterbi, A. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269.

Vontobel, P. O. and Kötter, R. (2007). On low-complexity linear-programming decoding of LDPC codes. *European Transactions on Telecommunications*, 18(5):509–517.

Wadayama, T., Nakamura, K., Yagita, M., Funahashi, Y., Usami, S., and Takumi, I. (2007). Gradient descent bit flipping algorithms for decoding LDPC codes. *arXiv:0711.0261 [cs, math]*. arXiv: 0711.0261.

Wang, G., Wu, M., Sun, Y., and Cavallaro, J. R. (2011a). GPU accelerated scalable parallel decoding of LDPC codes. In *Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers*, pages 2053–2057.

Wang, G., Wu, M., Sun, Y., and Cavallaro, J. R. (2011b). A massively parallel implementation of QC-LDPC decoder on GPU. In *IEEE 9th Symposium on Application Specific Processors*, pages 82–85.

Wang, G., Wu, M., Yin, B., and Cavallaro, J. R. (2013). High throughput low latency LDPC decoding on GPU for SDR systems. In *Proceedings of the IEEE Global Conference on Signal and Information Processing*.

Warren, H. S. (2012). *Hacker's Delight*. Addison-Wesley.

Wiberg, N. (1996). *Codes and decoding on general graphs*. PhD thesis, Linköping University.

Wiberg, N., Loeliger, H., and Kötter, R. (1995). Codes and iterative decoding on general graphs. *European Transactions on Telecommunications*, 6(5):513–525.

Wu, X., Zhao, C., and You, X. (2007). Parallel weighted Bit-Flipping decoding. *IEEE Communications Letters*, 11(8):671–673.

Xiao, H., Tolouei, S., and Banihashemi, A. (2008). Successive relaxation for decoding of LDPC codes. In *24th Biennial Symposium on Communications*, pages 107–110.

Yazdani, M., Hemati, S., and Banihashemi, A. (2004). Improving belief propagation on graphs with cycles. *IEEE Communications Letters*, 8(1):57–59.

Yue, G., Lu, B., and Wang, X. (2007). Analysis and design of Finite-Length LDPC codes. *IEEE Transactions on Vehicular Technology*, 56(3):1321–1332.

Zhang, J. and Fossorier, M. (2002). Shuffled belief propagation decoding. In *Conference Record of the Thirty-Sixth Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 8–15.

Zhang, J. and Fossorier, M. (2004). A modified weighted bit-flipping decoding of low-density parity-check codes. *IEEE Communications Letters*, 8(3):165–167.

Zhang, J., Yedidia, J. S., and Fossorier, M. (2007). Low-Latency decoding of EG LDPC codes. *Journal of Lightwave Technology*, 25(9):2879–2886.

Zhang, T., Wang, Z., and Parhi, K. K. (2001). On finite precision implementation of low density parity check codes decoder. In *IEEE Symposium on Circuits and Systems*, volume 4, pages 202–205.

Zhao, J., Zarkeshvari, F., and Banihashemi, A. (2005). On implementation of Min-Sum algorithm and its modifications for decoding Low-Density Parity-Check (LDPC) codes. *IEEE Transactions on Communications*, 53(4):549–554.

Zhou, X. S., Cockburn, B., and Bates, S. (2007). Improved iterative bit flipping decoding algorithms for LDPC convolutional codes. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 541–544.

Appendix A

Derivations

A.1 Derivation of the gradient-descent bit-flipping decoder

The following is the formulation of the bit-flipping decoder as a gradient-descent decoder, as presented by Wadayama et al. (2007). We assume here that the symbol alphabet is $\{1, -1\}$ and that transmission occurs over the BAWGNC. Maximum-likelihood (or maximum a posteriori) decoding is equivalent to finding the codeword which maximizes the correlation to the received values, where the correlation is

$$\sum_{i=1}^{n} x_i y_i .$$

For the gradient-descent bit-flipping decoder, we then define an objective function

$$f(\mathbf{x}) = \sum_{i=1}^{n} x_i y_i + \sum_{a=1}^{m} \prod_{i \in N(a)} x_i .$$

The first part of the objective function is simply the correlation of the current decoded codeword x and the channel values y. The second part is a term which accounts for satisfied parity-checks. When all parity-checks are satisfied it takes the value m and when none are satisfied it takes the value $-m$. Our goal is then to maximize f or, equivalently, minimize $-f$. To do so we can use the gradient-descent decoder. We first calculate the partial derivative with of f with respect to a variable x_i:

$$\frac{\partial}{\partial x_i} f(x) = y_i + \sum_{a \in N(i)} \prod_{j \in N(a) \setminus i} x_j . \tag{A.1}$$

Note here the strong resemblance to for example the sum-product tanh-rule. The product on the right hand side in (A.1) is similar to the check-to-variable messages, but with the log-ratio messages replaced by x_j. The sum of products and addition of y_i on the right hand side of (A.1) again resembles the final marginalization step (4.19) in the sum-product decoder.

The first-order approximation of f in the x_i-coordinate is

$$f(x_1, \ldots, x_i + s, \ldots, x_n) = f(\mathbf{x}) + s\frac{\partial}{\partial x_i} f(\mathbf{x}),$$

where s is the step length in the ith coordinate. Since we would like to maximize f we would like to choose s so that $s\frac{\partial}{\partial x_i} f(\mathbf{x}) > 0$. This happens if we choose $s > 0$ when $\frac{\partial}{\partial x_i} f(\mathbf{x}) > 0$ and $s < 0$ when $\frac{\partial}{\partial x_i} f(\mathbf{x}) < 0$. Since $x_i \in \{1, -1\}$ we can multiply the gradient by x_i and get that the objective function value is increased, according to the first-order approximation, if we flip the value of x_i when $x_i\frac{\partial}{\partial x_i} f(\mathbf{x}) < 0$. One possible way to choose which bit to flip is to, at each iteration, flip the bit i that has the smallest

$$E_i = x_i \frac{\partial}{\partial x_i} f(\mathbf{x}) = x_i y_i + \sum_{a\in N(i)} \prod_{j\in N(a)} x_j.$$

Alternatively, to keep with the convention that bits are flipped when E_i is large enough, we can define

$$E_i = -x_i \frac{\partial}{\partial x_i} f(\mathbf{x})$$

and flip the bit i that has the largest E_i or all bits i that have E_i greater than some threshold θ.

A.2 Derivation of the tahn-rule for the sum-product decoder

Recall that in the first formulation of the sum-product decoder for decoding of linear block codes presented in Chapter 4 the variable-to-check update rule is

$$v_{i\to a}(x_i) = \prod_{\substack{b\in N(i)\\ b\neq a}} f_{b\to i}(x_i).$$

Also, since we are considering *binary* codes we send in practice the vector $(v_{i\to a}(1), v_{i\to a}(-1))$ from a variable node i to a check node a. The check-to-variable update rule is

$$f_{a\to i}(x_i) = \sum_{\mathbf{x}_a\setminus x_i} [\textstyle\prod_{j\in N(a)} x_j = 1] \prod_{j\in N(a)\setminus i} v_{j\to a}(x_j)$$

and we send the vector $(f_{a\to i}(1), f_{a\to i}(-1))$ from a check node a to a variable node i.

The intuition for being able to use single scalars as messages in the case of binary codes is that a probability distribution over two states can be described by a single scalar due to the constraint that probabilities over all states must sum to 1. To simplify the conventional sum-product rules we introduce the ratios

$$\begin{aligned} f^r_{a\to i} &= \frac{f_{a\to i}(1)}{f_{a\to i}(-1)}, \\ f^r_i &= \frac{f_i(1)}{f_i(-1)} \end{aligned} \tag{A.2}$$

and

$$v^r_{i\to a} = \frac{v_{i\to a}(1)}{v_{i\to a}(-1)} \tag{A.3}$$

as the message values. We will use the superscript r on the messages to differentiate them from the conventional sum-product update rules where each message consist of two values. We can then write the ratio $v^r_{i\to a}$ as a product of ratios of incoming messages $f^r_{b\to i}$ from the neighbors b of i, excluding a. More precisely, we can write

$$v^r_{i\to a} = \frac{v_{i\to a}(1)}{v_{i\to a}(-1)} = \frac{\prod_{b\in N(i)\setminus a} f_{b\to i}(1)}{\prod_{b\in N(i)\setminus a} f_{b\to i}(-1)} = \prod_{b\in N(i)\setminus a} f^r_{b\to i}. \tag{A.4}$$

We can likewise write the ratio $f^r_{a\to i}$ at a check node a using the ratios $v^r_{i\to a}$ of the neighbors j of a, excluding i. Doing so, we get

$$\begin{aligned} f^r_{a\to i} &= \frac{f_{a\to i}(1)}{f_{a\to i}(-1)} \\ &= \frac{\sum_{\mathbf{x}_a\setminus x_i} [\prod_{j\in N(a)} x_j = 1] \prod_{j\in N(a)\setminus i} v_{j\to a}(x_j)}{\sum_{\mathbf{x}_a\setminus x_i} [\prod_{j\in N(a)} x_j = -1] \prod_{j\in N(a)\setminus i} v_{j\to a}(x_j)} \\ &= \frac{\sum_{\mathbf{x}_a\setminus x_i} [\prod_{j\in N(a)} x_j = 1] \prod_{j\in N(a)\setminus i} \frac{v_{j\to a}(x_j)}{v_{j\to a}(-1)}}{\sum_{\mathbf{x}_a\setminus x_i} [\prod_{j\in N(a)} x_j = -1] \prod_{j\in N(a)\setminus i} \frac{v_{j\to a}(x_j)}{v_{j\to a}(-1)}} \\ &= \frac{\sum_{\mathbf{x}_a\setminus x_i} [\prod_{j\in N(a)} x_j = 1] \prod_{j\in N(a)\setminus i} \left(v^r_{j\to a}\right)^{\frac{1+x_j}{2}}}{\sum_{\mathbf{x}_a\setminus x_i} [\prod_{j\in N(a)} x_j = -1] \prod_{j\in N(a)\setminus i} \left(v^r_{j\to a}\right)^{\frac{1+x_j}{2}}} && \text{(A.5)} \\ &= \frac{\prod_{j\in N(a)\setminus i}(v^r_{j\to a}+1) + \prod_{j\in N(a)\setminus i}(v^r_{j\to a}-1)}{\prod_{j\in N(a)\setminus i}(v^r_{j\to a}+1) - \prod_{j\in N(a)\setminus i}(v^r_{j\to a}-1)} && \text{(A.6)} \\ &= \frac{1 + \frac{\prod_{j\in N(a)\setminus i}(v^r_{j\to a}-1)}{\prod_{j\in N(a)\setminus i}(v^r_{j\to a}+1)}}{1 - \frac{\prod_{j\in N(a)\setminus i}(v^r_{j\to a}-1)}{\prod_{j\in N(a)\setminus i}(v^r_{j\to a}+1)}} && \text{(A.7)} \end{aligned}$$

In (A.5) we have simply used the fact that the ratio $\frac{v_{j\to a}(x_j)}{v_{j\to a}(-1)}$ is 1 when $x_k = -1$ and $v^r_{j\to a}$ otherwise. In (A.6) we have done slightly more work. The product $\prod_{j\in N(a)\setminus i}(v^r_{j\to a}+1)$ expands to the sum of the products

$$\prod_{j\in M} v^r_{j\to a}, \quad \forall M \subseteq N(a)\setminus i,$$

where M can be the empty set in which case we define the product to be 1. The product $\prod_{j\in N(a)\setminus i}(v^r_{j\to a}-1)$ expands to the same but with a negative sign for some terms. Each term with $|M|$ terms such that $|N(a)\setminus i| - |M|$ is odd, is negative. Now consider the indicator function $[\prod_{j\in N(a)\setminus i} x_j = -1]$ in the numerator of (A.5). It selects in the sum all such terms where there are an even number of x_j which take the value -1. On the other hand, all terms with an odd number of x_j which take the value -1 are ignored. The numerator in (A.6) now selects exactly the same terms as the indicator function does. That is, we have

$$\prod_{j\in N(a)\setminus i}(v^r_{j\to a}+1) + \prod_{j\in N(a)\setminus i}(v^r_{j\to a}-1) = 2\sum_{\mathbf{x}_a\setminus x_i}[\textstyle\prod_{j\in N(a)} x_j = 1]\prod_{j\in N(a)\setminus i}\left(v^r_{j\to a}\right)^{\frac{1+x_j}{2}}$$

Doing the same analysis for the denominator of (A.5) and (A.6) we get the result in (A.6) as the 2's cancel out. Further rearranging (A.7) we get

$$\frac{f^r_{a\to i}-1}{f^r_{a\to i}+1} = \prod_{j\in N(a)\setminus i}\frac{v^r_{j\to a}-1}{v^r_{j\to a}+1}. \tag{A.8}$$

Instead of using the ratios in (A.2)–(A.3), we can use the log-ratios

$$f^l_{a\to i} = \ln\left(f^r_{a\to i}\right),$$
$$f^l_i = \ln\left(f^r_i\right),$$

and

$$v^l_{i\to a} = \ln\left(v^r_{i\to a}\right).$$

We can then rearrange the results using log-ratios. We first write the left-hand side of (A.8) as

$$\frac{f^r_{a\to i}-1}{f^r_{a\to i}+1} = \frac{e^{f^l_{a\to i}}-1}{e^{f^l_{a\to i}}+1} = \tanh\left(\frac{f^l_{a\to i}}{2}\right), \tag{A.9}$$

where we have simply used the definition of the tanh function. Likewise, we can write the right-hand side of (A.8) as

$$\prod_{j\in N(a)\setminus i} \frac{f^r_{j\to a} - 1}{f^r_{j\to a} + 1} = \prod_{j\in N(a)\setminus i} \tanh\left(\frac{f^l_{j\to a}}{2}\right). \tag{A.10}$$

Rearranging (A.8) with the help of (A.9) and (A.10), we finally get the tanh-rule for the check-to-variable updates. The rule is

$$f^l_{a\to i} = 2\tanh^{-1}\left(\prod_{j\in N(a)\setminus i} \tanh\left(\frac{f^l_{j\to a}}{2}\right)\right). \tag{A.11}$$

If we now initialize the outgoing messages at the channel factors as

$$f^r_i = \ln\left(\frac{f_i(1)}{f_i(-1)}\right)$$

we can send (A.11) as the check-to-variable message at each check node. In addition, rewriting (A.4) with the help of log-ratios we can send

$$v^l_i = \sum_{b\in N(i)\setminus a} f^l_{b\to i}.$$

as the variable-to-check message at each variable node. Thus, in the case of binary linear codes we can send a single scalar over each edge as the message, instead a vector containing two scalars if we directly apply the sum-product decoder.

Appendix B

Parameter searches

The following two sections show plots of the bit-error rate as a function of the two free parameters used in the stochastic bit-flipping decoder and the WBF decoder. In the stochastic bit-flipping decoder, the probability to flip a bit was parameterized using two parameters p and T. The gradient-descent bit-flipping decoder was chosen to have one free parameter, the threshold θ for flipping. The parameter α in the general formulation of the weighted bit-flipping decoder was set to 1 as originally presented by Wadayama et al. (2007). The stochastic bit-flipping decoder was run after transmission over the BSC and the gradient-descent bit-flipping decoder after transmission over the BSC and BAWGNC.

B.1 Parameter searches for the stochastic bit-flipping decoder

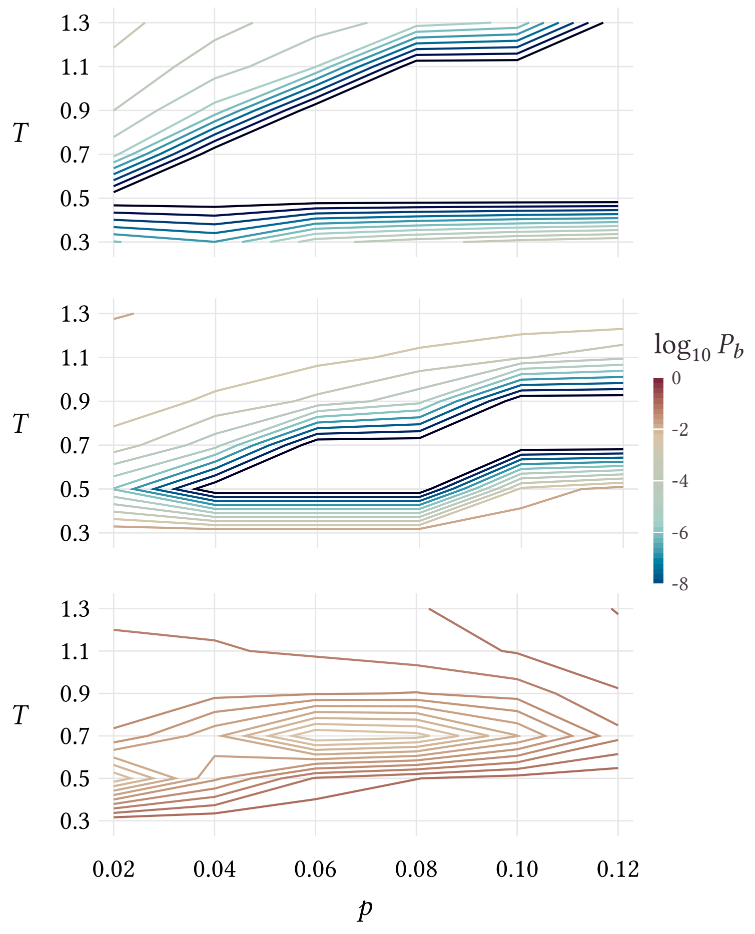


Figure B.1: Parameter search for the stochastic bit-flipping decoder with the (3,6)-regular ensemble and block length 2^{14}. The parameterization uses the parameters T and p. The bit-error rate P_b was evaluated at the grid intersections. The performance was evaluated on the BSC with crossover probability 0.03 (top), 0.05 (middle) and 0.07 (bottom) Note that the color scale is logarithmic.

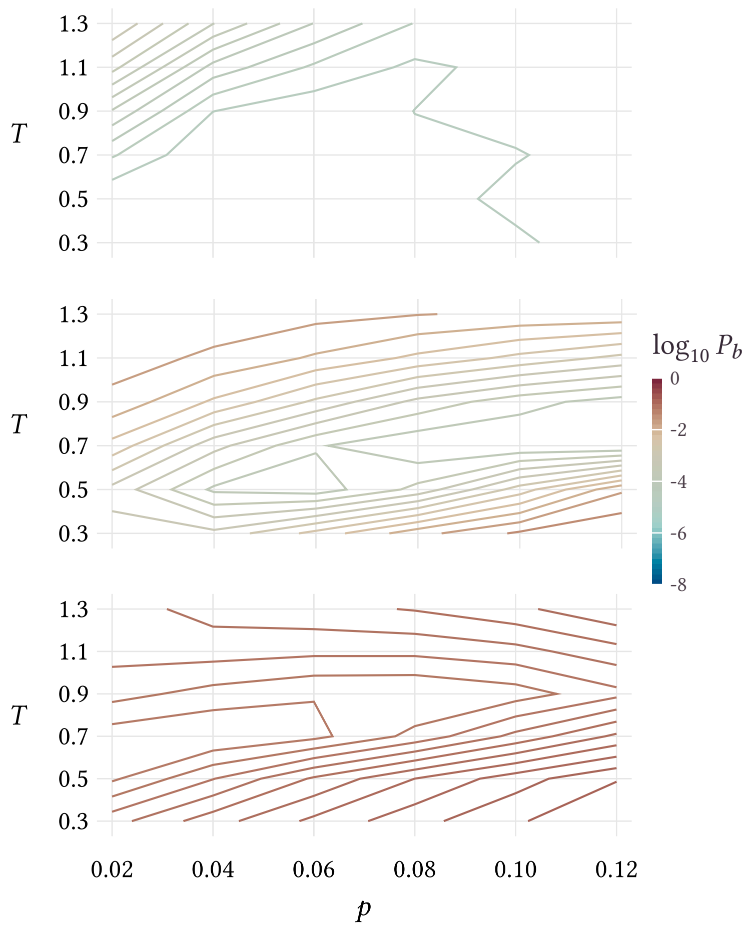


Figure B.2: Parameter search for the stochastic bit-flipping decoder with the irregular ensemble and block length 2^{14}. The parameterization uses the parameters T and p. The bit-error rate P_b was evaluated at the grid intersections. The performance was evaluated on the BSC with crossover probability 0.03 (top), 0.05 (middle) and 0.07 (bottom) Note that the color scale is logarithmic.

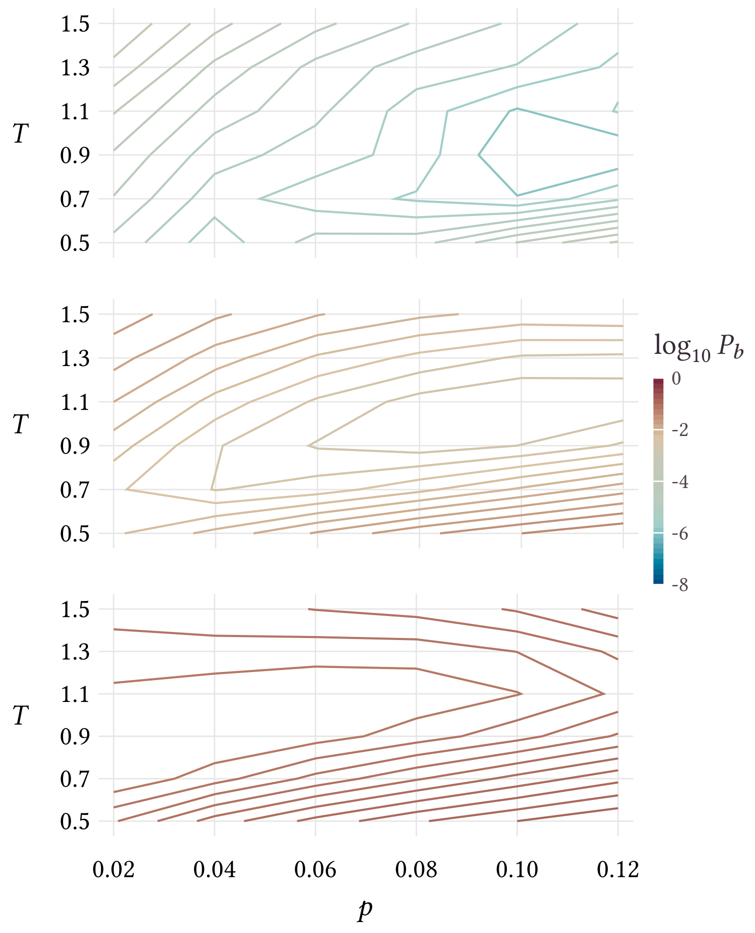


Figure B.3: Parameter search for the stochastic bit-flipping decoder with the rate-$\frac{1}{2}$ WiMAX code and block length 2304. The parameterization uses the parameters T and p. The bit-error rate P_b was evaluated at the grid intersections. The performance was evaluated on the BSC with crossover probability 0.03 (top), 0.05 (middle) and 0.07 (bottom) Note that the color scale is logarithmic.

B.2 Parameter searches for the gradient-descent bit-flipping decoder

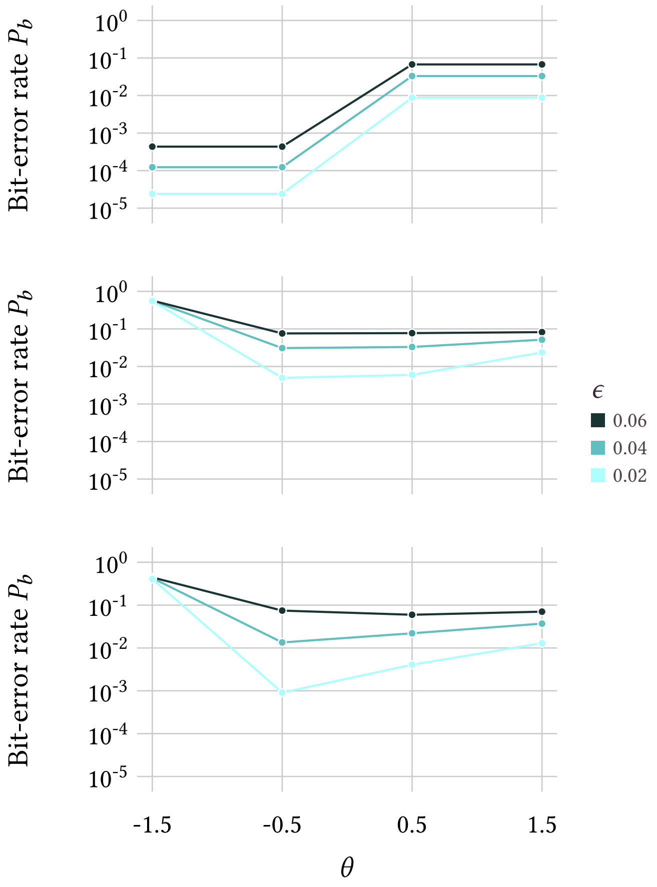


Figure B.4: Parameter search for the gradient-descent bit-flipping decoder with the (3,6)-regular ensemble with block length 2^{14} (top), the irregular ensemble with block length 2^{14}, and the rate-$\frac{1}{2}$ WiMAX code with block length 2304. The threshold θ is the free parameter in the gradient-descent bit-flipping decoder. The bit-error rate P_b was evaluated on the BSC as a function of the crossover probability ϵ. The vertical axis is logarithmic.

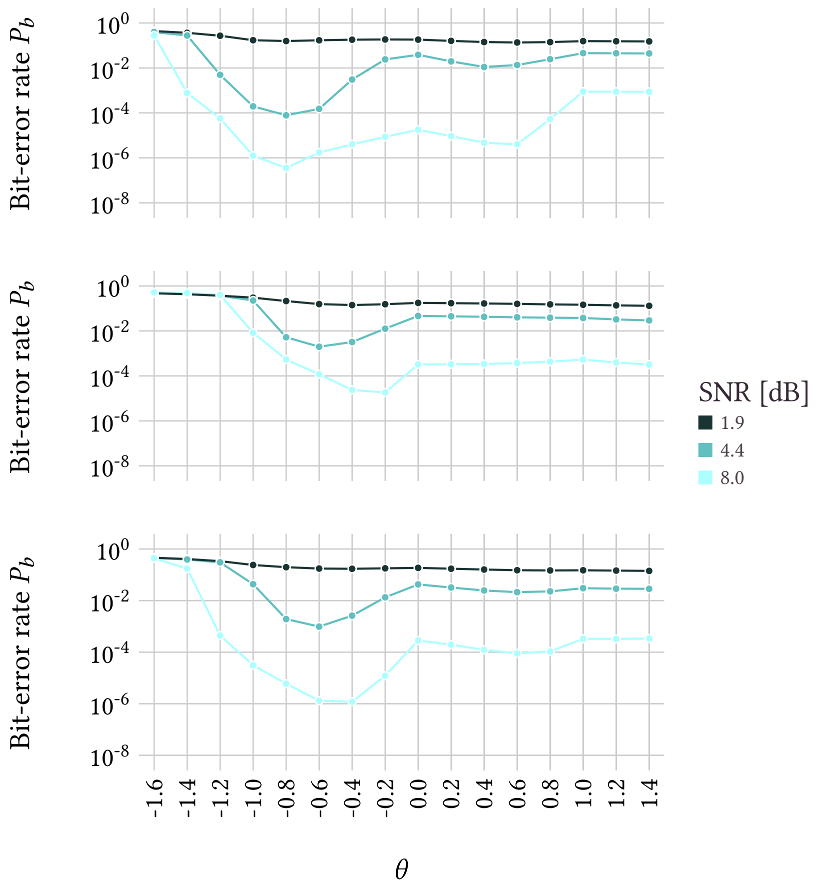


Figure B.5: Parameter search for the gradient-descent bit-flipping decoder with the (3,6)-regular ensemble with block length 2^{14} (top), the irregular ensemble with block length 2^{14}, and the rate-$\frac{1}{2}$ WiMAX code with block length 2304. The threshold θ is the free parameter in the gradient-descent bit-flipping decoder. P_b was evaluated on the BAWGNC as a function of the signal-to-noise ratio (SNR) in dB. The vertical axis is logarithmic.