Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Radio Engineering

Implementation and performance evaluation framework for LDPC codes

Bachelor thesis

Author: Lukáš Růžička

Supervisor: Prof. Ing. Jan Sýkora, CSc.

Prague 2014

Abstract

Low-density parity-check (LDPC) codes have become quite popular these days. They have good performance which is close to Shannon limits and can be used in satellite communication with excellent results. This thesis presents elementary tools for work with them. Mainly we show how to generate pseudo-random sparse parity-check matrix **H**, which define code. We also describe two methods to make algebraically constructed matrix **H**. Next there is repeat-accumulate (RA) LDPC code which can encode source data to codeword more easily. Further we examine error-correcting decoding for transferred messages through AWGN channel especially with Sum-product algorithm (SPA), which is mainly used with log-likelihood ratio (LLR) information. Then we also present two methods for analyzing qualities of code with Bit-error rate (BER) test and Extrinsic information transfer (EXIT) charts. In the last part we compared some special cases of codes to show differences for various parameters. All methods are implemented in MATLAB codes.

Keywords: LDPC, sparse matrix, pseudo-random matrix, AWGN channel, RA LDPC, SPA decoding, BER, EXIT chart

Abstrakt

Low-denstiy parity-check (LDPC, Řídké paritu kontrolující) kódy jsou v těchto dnech celkem populární. Mají dobrý výkon, který je blízko Shannovýmu limitu a mohou být s výbornými vysledky využity v satelitní komunikaci. V této práci uvádíme základní nástroje pro práci s nimi. Hlavně ukazujeme způsob, jak generovat pseudo náhodné řidké kontrolní matice **H**, které definují kód. Dále popisujeme dvě metody algebraické tvorby matice **H**. V další částí je repeat-accumulate (RA) LDPC kód, který jednodušeji kóduje zdrojová data na kódová slova. Zkoumali jsme také chyby opravující dekódování pro přenesené zprávy přes AWGN kanál, hlavně dekódování s Sum-product algoritmem (SPA), který je hlavně použit s informacemi v podobě log-likelihood ratio (LLR). Dále jsme také ukázali dvě metody pro analýzu kvalit kódů s Bit-error rate (BER) testem a Extrinsic information transfer (EXIT) grafem. V poslední části porovnáváme některé speciální případy kódů pro ukázání rozdílů pro různé parametry. Všechny metody jsou implementovány do MATLABových kódů.

Klíčová slova: LDPC, řídké matice, pseudo náhodné matice, AWGN kanál, RA LDPC, SPA dekódování, BER, EXIT graf

Proclamation

I claim that this thesis was created autonomously according to the rules established by CTU in Prague and all used references are cited in the bibliography.

Prague, 22nd May 2014 ....

Lukáš Růžička

Acknowledgments

I would like to thank my supervisor Prof. Ing. Jan Sýkora, CSc. for his time and giving me valuable advices and suggestions.

Contents

List of Figures

List of Tables

List of Algorithms

List of Implemented Tools

Abbreviations

AWGN	Additive White Gaussian Noise
BF	Bif-Flip Decoding
BPSK	Binary Phase-shift Keying
BEC	Binary Erasure Channel
BER	Bit-Error Rate
BSC	Binary Symmetric Channel
CN	Check Node
CND	Check Node Decoder
DVB-S2	Digital Video Broadcasting - Satellite, second generation
EXIT	Extrinsic Information Transfer Chart
FG	Factor Graph
JCD	Joint Channel Decoding
LDPC	Low-Density Parity-Check
LLR	Log-Likelihood ratio
SCD	Separated Channel Decoding
SNR	Signal-To-Noise Ratio
SPA	Sum-Product Algorithm
QC	Quasi-Cyclic Code
RA	Repeat-Accumulate Code
VN	Variable Node
VND	Variable Node Decoder

Chapter 1

General principles

1.1 Introduction

The goal of this thesis is to make useful codes for work with Low-density parity-check (abbreviated LDPC) codes. We have implemented methods in MATLAB which does not produce the fastest programs (than e.g. in C++), but whose codes are more illustrative. Converting codes from MATLAB to C++ is not too hard, so if it necessary, they can be easily converted. All codes are implemented by us are in Appendix A.

Low-density parity-check codes are a linear block codes from group of forward error correction codes. They were introduced by Robert G. Gallager in 1962 [3], but for a long time they had been forgotten for their computational cost. They were re-discovered by David J. C. MacKay in 1999 [7] and after that their popularity has grown up. Especially for their performance which is close to the Shannon limit (theoretical upper bound of ability to transfer symbols in noisy channel) [8]. That is the reason why they are used for satellite communication in space [2]. They are mainly known for usage with standard DVB-S2 (Digital Video Broadcasting - Satellite, second generation) [9], which is used with current digital television. Also it can be used for microwave wireless communication, for example with Wi-Fi routers.

The main task of this work is to study and implement method for decoding codes (in Chapter 2). The best decoders for LDPC are over Factor graph. In our case, we will use message passing Sum-Product algorithm. Then we will study in Chapter 3 methods of generating codes, mainly how to generate its parity-check matrix $\mathbf{H}$ which defines code. And in Chapter 4 we will show methods of analyzing qualities of codes.

The second goal of this thesis is utilize knowledge gained during study on CTU. We especially talk about subjects with basics of theory of Probability, Information and Digital communication. We want to prove understanding of technical papers and main principles and ability to use them properly to implement in codes. Then simplify implemented methods for better explanation in pseudo-codes algorithms.

1.2 Finite field algebra

Before we start with LDPC codes, we must introduce algebra over a finite field, or also known like Galois field, which is denoted by $GF(p^k)$, where p is a prime number and k is a positive integer [8]. It is field with finite number of elements, exactly with p^k elements (from 0 to $p^k - 1$) which is defined by modulo operator $(\bmod\ p^k)$.

LDPC codes can be used with any of $GF(p^k)$, but in this thesis we will use codes only over $GF(2)$, so we will use binary alphabet. It is more common to work with binary data. With $GF(2)$ addition and subtraction are the same ($(1 + 1) \bmod 2 = (1 - 1) \bmod 2 = 0$) and we can replace this with binary operation XOR. It is similar with multiplication and division ($(1 \times 1) \bmod 2 = (1/1) \bmod 2 = 1$) which can be represent by binary operation AND. In table 1.1 is summary of elementary operation over $GF(2)$.

$+$	0	1
0	0	1
1	1	0

$\cdot$	0	1
0	0	0
1	0	1

Tab. 1.1 – Operation with $GF(2)$

1.3 Parity-check matrix

Performance and properties of codes are determined by their parity-check matrices **H**. So good construction of matrix **H** is essential for good code. From the name of Low-density parity-check code we need for code sparse parity-check matrix **H**. Sparse means that are few ones towards all possible elements of matrix. In figure 1.1 we have example of LDPC parity-check matrix **H** with dimension $n = 10000$, $m = 7500$ and parameters $w_c = 3$, $w_r = 4$ (meaning of w_c, w_r is explained in subsection 1.3.4), where we have $7.5 \cdot 10^7$ elements of matrix, but only $3 \cdot 10^4$ ones, which are only 0.04% of whole matrix. (Dots in figure are bigger than they should be, but smaller we could not see.)

Also from sparse matrix we will gain good factor graph(subsection 1.3.2), which is essential for decoding.

Another important condition for good code is big dimensions of matrix. We will gain better results for bigger matrices with longer codewords. For DVB-S2 LDPC codes we have $n = 64800$ symbols [9].

1.3.1 Construction

Construction of parity-check matrix **H** has a big role of quality of code. It determines the way to source data s encode with code (making of generator matrix **G**), performance of code and mainly storing of matrix **H** (with **G**). There are two main groups of construction - pseudo-random and algebraic. Pseudo-random can have better performance than algebraic, but good algebraic can have the same results, but for coder and decoder they will be more useful, because they can generate matrix **H** anytime they need it and they do not need to remember big matrices like with pseudo-random.

Construction of codes is subject of Chapter 3.

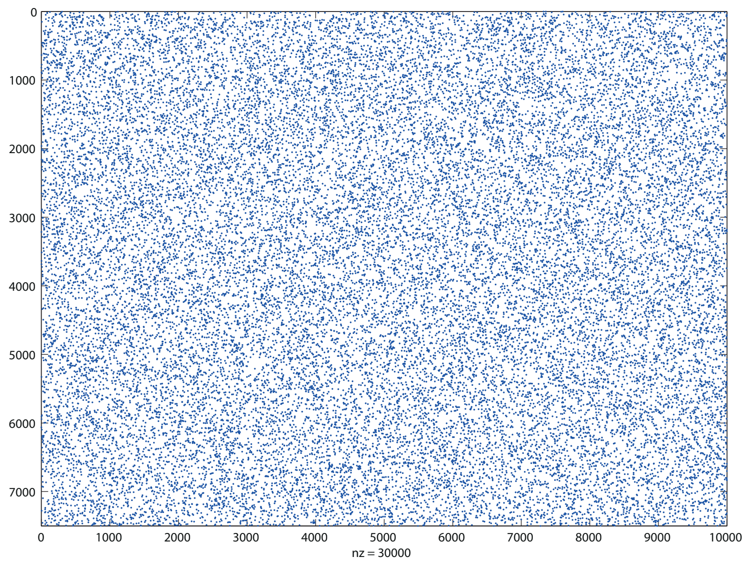

Fig. 1.1 – Example of parity-check matrix **H**, $n = 10000$, $m = 7500$, $w_c = 3$, $w_r = 4$

1.3.2 Factor graph

Parity check matrix can be transferred into graph, to be exact, into factor graph (FG) (for this usage also known like Tanner graph). It is a bipartite graph which is composite from edges and nodes (variable and check nodes). It shows how variable nodes are connected to check nodes via edges.

We can split parity check matrix **H** into n columns and m rows. The columns represent variable nodes (abbreviated VN) of graph, in this thesis denoted by red circles, where j-th variable node represent j-th bit of incoming code word. Rows represent check nodes (CN) of graph, here denoted by blue squares, where i-th check node check parity of all incoming messages. It means that check node control $VN(1) \oplus VN(2) \oplus ... \oplus VN(n) = 0$, where $VN(1)...VN(n)$ are all connected variable nodes to concrete check node.

Edges are connection between variable nodes and check nodes. If $\mathbf{H}_{ij}$ is non-zero element, then is edge between i-th check node and j-th variable node.

In figure 1.2 is factor graph for parity check matrix

$$\mathbf{H} = \begin{bmatrix} 0&0&1&1&0&0&0&0&0&0&0&0&0&1&1 \\ 1&0&0&0&1&0&0&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&1&0&1&0&0&0&1 \\ 0&0&0&0&0&0&0&0&0&0&1&0&1&0&0 \\ 1&0&0&0&0&0&0&1&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&1&0&0&1&0&0&0&0&0 \\ 0&1&0&0&1&0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0&1&1&0 \\ 0&0&0&0&0&1&1&0&0&0&0&1&0&0&0 \\ 0&0&1&0&0&0&0&1&1&0&0&0&0&0&0 \end{bmatrix}, \qquad (1.1)$$

where we have $n = 15$ columns, which make 15 variable nodes and $m = 10$

rows, which make 10 check nodes. In the first column we have ones on positions $i \in \{2, 5\}$, which give us edges from first variable node to the second and fifth check nodes.

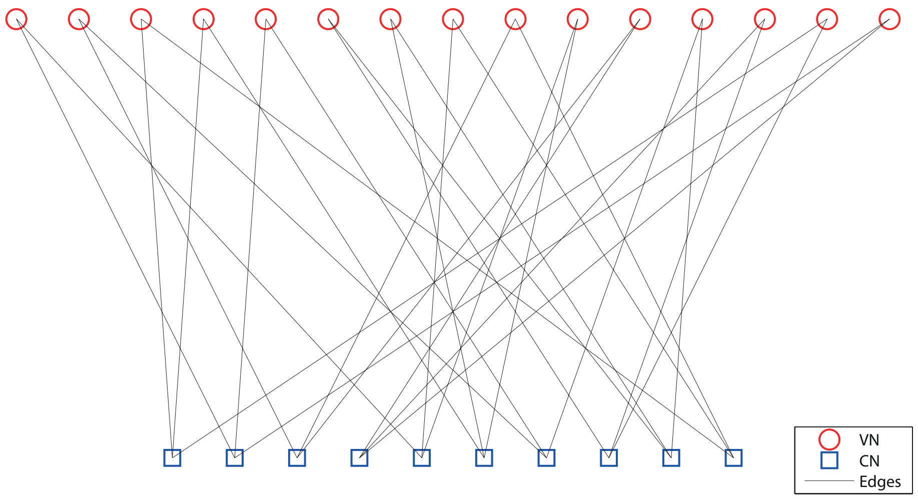

Fig. 1.2 – Factor graph for matrix (1.1), with 15 variable nodes with weight 2 and 10 check nodes weight 3

The main meaning of factor graph is not the better illustration of **H**, but good usage with message passing decoders. The graph here is like a diagram for decoder, where we give functions to nodes [6], which are executed during passing messages. We divide decoders by functions, which is used in nodes, like Sum-product algorithm used in this thesis, in section 2.1.

Example: Figure 1.3 shows example of message passing algorithm, where illustration of one path from the second variable node to others. From this example we can see strength of LDPC codes. There is showed only one path, which connects nine variable nodes. If there is error in the 14th variable node, it means VN number 4 from path, then it can receive good information from four other variable nodes from both sides of this path. There is bigger factor graph for bigger matrix **H**. Then path can be longer, so then variable node can receive more information from others. □

The MATLAB code for drawing factor graph from **H** is in A.1 on page 51.

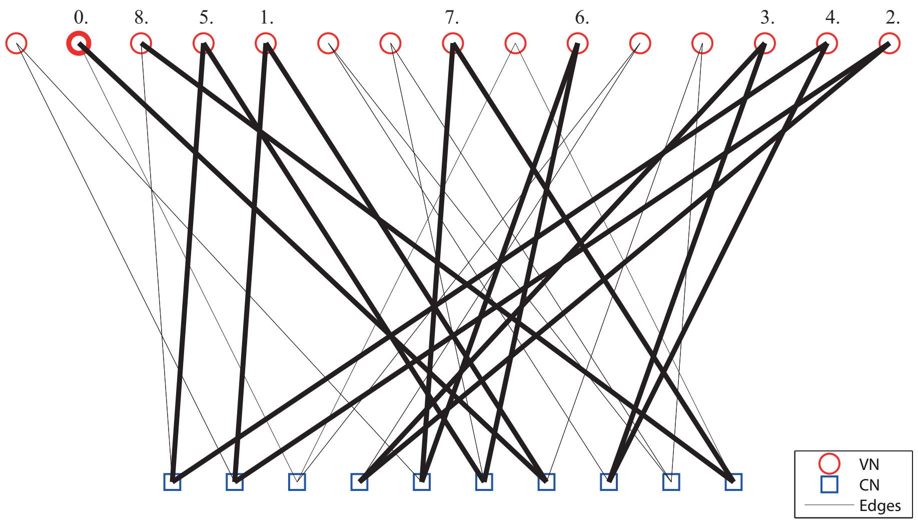

Fig. 1.3 – Example of message passing algorithm

1.3.3 Cycles and girth

In the previous part we have shown factor graph and message passing. Figure 1.3 shows one path of message passing through graph. If we continue from step 8, then we close this path to cycle. Cycle is path which ends in same node like from it start. Possibly length of cycles is $2k$, where $k \in \mathbb{N}$ and possibly minimal length is 4-cycle. Let girth is length of minimal cycle of graph.

Small cycles can decrease performance of code, mainly 4-cycles. In [5] show that cycles of length 6 and more did not affect performance of code. Four-cycles are from four edges which connect two variable nodes to two check nodes. That causes bad correcting of errors in both variable nodes in 4-cycle. In parity check matrix we can recognize 4-cycles, if ones make square in matrix. In matrix

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{1.2}$$

we have two 4-cycles. Blue ones make four cycle in figure 1.4 a) and 4-cycle from square with red ones is in figure 1.4 b).

Finding larger girth of graph than four is not easy task. But we said that big problem makes just 4-cycles and bigger did not affect code performance too much. It is easier to avoid certain girth of matrix **H** from construction than try find it in already existing matrix.

The MATLAB code of tool for finding 4-cycles in already existing matrix **H** is in A.2 on page 51.

1.3.4 Degree and edge distribution

One of the methods how to analyze parity-check matrix **H** is detection of its weight of rows and columns and for irregular matrix degree distribution.

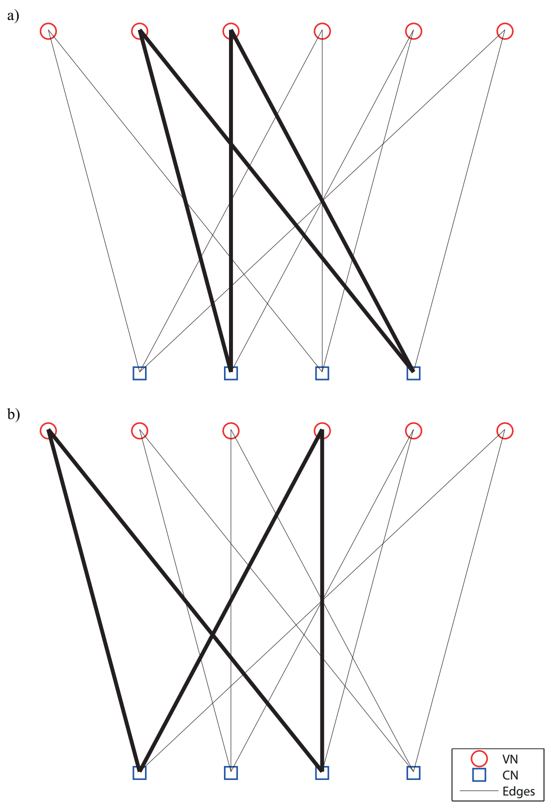

Fig. 1.4 – Example of 4-cycles from matrix **H**(1.2)

The weight of columns w_c is the number of connected check nodes to one variable node which is represented by number of ones in a column. The same with weight of row w_r which represent the number of connected variable nodes to one check node, represented by count of ones in a row of matrix **H**. We can divide matrices by weights on regular or irregular codes, where regulars codes have constant w_r, w_c for every rows and columns and irregular not.

For regular codes we have

$$nw_c = mw_r \tag{1.3}$$

where this represent total amount of edges (total amount of ones in **H**). That give us $m = n\frac{w_c}{w_r}$. From that and from code rate

$$r = \frac{k}{n}, \tag{1.4}$$

which determine ratio between length of data word and code word, we have

$$\frac{k}{n} = \frac{n-m}{n} = 1 - \frac{m}{n} = 1 - \frac{w_c}{w_r}, \tag{1.5}$$

where we assume $w_c \leq w_r$. From this equation we gain design rate r' which can be lower than actual rate of code r, because parity-check equations of **H** might not be all independent [1].

For irregular codes it is similar, but with degree distributions. We can not describe irregular code with just one number of weight. We can use to description degree distribution or edge degree [5].

Degree distribution of column is description towards amount of all columns, which have weight i. Matrix $\mathbf{H}$ with dimension $m \times n$ have for columns with weight i degree distribution

$$v_i = \frac{\text{number of columns with weight } i}{n} \tag{1.6}$$

and for rows with weight i have

$$h_i = \frac{\text{number of rows with weight } i}{m}. \tag{1.7}$$

Both must fulfill

$$\sum_i v_i = 1 \quad \sum_i h_i = 1. \tag{1.8}$$

From that all we get design rate

$$r' = 1 - \frac{\sum\limits_i v_i i}{\sum\limits_i h_i i}. \tag{1.9}$$

Edge degree is ratio of i edges from one node towards all edges in factor graph from parity-check matrix $\mathbf{H}$. For i edges coming from variable node we have

$$\lambda_i = \frac{i \times \text{number of varible nodes with } i \text{ edges}}{\text{total amount of all edges}} \tag{1.10}$$

and for i edges from check node we have

$$\rho_i = \frac{i \times \text{number of check nodes with } i \text{ edges}}{\text{total amount of all edges}} \tag{1.11}$$

and both also must fulfill

$$\sum_i \lambda_i = 1 \quad \sum_i \rho_i = 1. \tag{1.12}$$

We can get degree distribution from edge with

$$v_i = \frac{\lambda_i / i}{\sum\limits_j \lambda_j / j} \tag{1.13}$$

$$h_i = \frac{\rho_i / i}{\sum\limits_j \rho_j / j}. \tag{1.14}$$

and after input it into (1.9) we get design code rate

$$r' = 1 - \frac{\sum\limits_i \rho_i / i}{\sum\limits_i \lambda_i / i}. \tag{1.15}$$

Irregular code can have better performance than regular code with same rate. For improving is enough to has irregular column weight with regular row weight. [5]

Example: Let we have matrix

$$\mathbf{H} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{1.16}$$

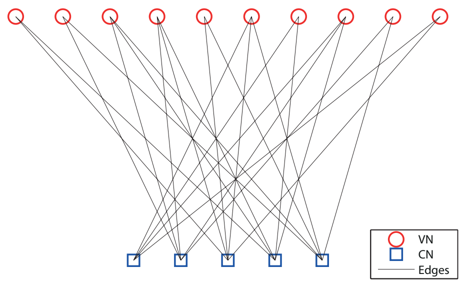

Fig. 1.5 – Factor graph of matrix **H** (1.16)

for irregular code. This matrix have degree distribution $v_2 = 6/10 = 0.6$, $v_3 = 4/10 = 0.4$, $h_4 = 1/5 = 0.2$ and $h_5 = 4/5 = 0.8$. For edge degree we will get $\lambda_2 = 2 \cdot 6/24 = 0.5$ and $\lambda_3 = 3 \cdot 4/24 = 0.5$ for edges coming from variable nodes and $\rho_4 = 4 \cdot 1/24 = 0.1\overline{6}$ and $\rho_5 = 5 \cdot 4/24 = 0.8\overline{3}$ for edges from check nodes. □

The MATLAB code for analysis for weights and degrees is in A.3 on page 52.

1.4 Encoding

In this thesis all types of encoding are systematic. It means that we have k-bits of data word s which we encode to codeword c with length of n bits where first k bits are the same bits like s. Then codeword c can be split to two parts, the first k bits are source data word s and next m bits are parity check bits p. Then we have length of code c expressed by $n = k + m$. That leads to the easiest decoding - we just take from corrected codeword c' first k bits and gain data word s'.

How we said, the code is defined by its parity check matrix **H**, so its encoding is derived from it. It is linear coding and we can use equation

$$\mathbf{G} \cdot \mathbf{H}^T = 0, \tag{1.17}$$

where **G** is generator matrix, which encode source data word s to codeword c with

$$c = \mathbf{G}^{\mathbf{T}} s. \tag{1.18}$$

This works good with for example Hamming codes. But there is problem with LDPC code, because **H** is normally really large and sparse matrix **H** may not generate sparse matrix $\mathbf{G}$. Without sparse $\mathbf{G}$ can be really hard to compute codeword c, because it will cost many additions during multiply the vector s with matrix $\mathbf{G}$.

This leads us to better method for encoding with pseudo-random LDPC codes. In subsection 3.1.1 we introduce Repeat-accumulate code which make new parity check matrix $\mathbf{H}$.

$$\mathbf{H} = [\mathbf{H_s}|\mathbf{H_p}] = \left[\begin{array}{cccccccccc|ccccc} 0&0&0&0&1&1&0&0&1&1&1&0&0&0&0 \\ 1&1&0&1&0&1&0&0&0&0&1&1&0&0&0 \\ 0&1&1&0&0&0&0&1&1&0&0&1&1&0&0 \\ 1&0&1&0&1&0&1&0&0&0&0&0&1&1&0 \\ 0&0&0&1&0&0&1&1&0&1&0&0&0&1&1 \end{array}\right].$$

With this new matrix $\mathbf{H}$, where $\mathbf{H_S}$ represent pseudo-random LDPC parity check matrix and $\mathbf{H_P}$ parity check part, we can also systematicly encode data bits s to codeword c.

For LDPC codes created by Algebraic construction there exists of specific way to encode data. Section 3.2 handle with this subject.

For the method, how to encode two message for one relay there is section 3.3.

1.5 BPSK and AWGN channel

In this thesis we used only binary data, so best modulation for it is BPSK (binary phase-shift keying). This linear modulation has only two constellation points $q_n \in \{-1, 1\}$ for n-th symbol. In this work we use mapping rule $0 \to 1$ and $1 \to -1$ for binary data input [10].

LDPC can correct transmitted codes with errors created during transmission. Good model for real communication is AWGN (additive white Gaussian noise) channel. The output of AWGN channel can be described by

$$y(t) = x(t) + n(t), \tag{1.19}$$

where $x(t)$ are sent symbols with BPSK modulation and $n(t)$ is additive white Gaussian noise. We control this channel by changing SNR (signal-to-noise ratio) or N_0(noise power spectral density). Fig.1.6 is comparing the same data input with output from AWGN channel with different SNR. Obviously in higher SNR it is easier to determine which received symbol belongs to the constellation point.

If we want to determine which symbol we received from AWGN channel, we use likelihood function

$$\Lambda(x) = p(y|x) = ce^{-\frac{||y-x||^2}{2N_0}}. \tag{1.20}$$

This function states probability of likeness of received symbol with every possible sent symbol. Then we assume that we received x_i, if the $\Lambda(x_i)$ is the biggest value of all $\Lambda(x)$.

All MATLAB codes for BPSK and AWGN operation is in A.4 on page 52.

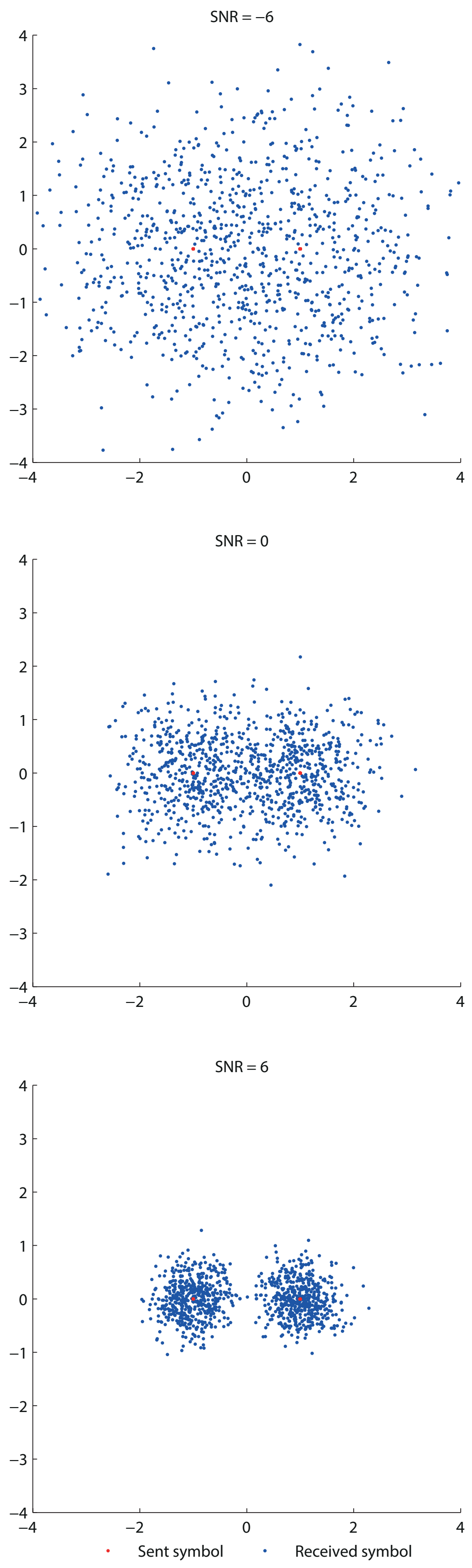

Fig. 1.6 – Constellation space observation of BPSK for 1000 symbols in AWGN Channel for various SNR

1.6 Decoding

LDPC codes are well known for their error-correction ability. The best results we will gain from soft-decision decoding (that mean using soft information), because this use more information than from hard-decision (where we have only binary information for symbol). That is the reason why we are using in this thesis Sum-product algorithm for decoding, which we introduce in Section 2.1, because it works well with soft information. We will systematic decode our codewords after code's error-correction decoding. This means that first k-th symbols of received and corrected word are our decoded data word. Chapter 2 deals with decoding.

Chapter 2

Error-correction decoding

We mostly simulated codes over AWGN channel, so we did not receive exactly bits, but just only likelihood functions described by (1.20) on page 9. We had probability of occurrence of symbol, so we would have lost many valuable information if we had reduced it only on binary bit (0 or 1). It means that is better to use soft decision decoding than hard decisions. In this chapter we implemented for Soft decision decoding Sum-Product algorithm with Probability domain in section 2.1.1.1 or LLR in section 2.1.1.2. For Hard decision decoding we have two methods - Hard decision SPA in section 2.1.2 and Bit-flip decoding in section 2.2.

Soft decision is good for noisy channels, like AWGN (from section 1.5), where we examine information from likelihood function (1.20). There is not strict decision which symbol is receives, it is easier for decoder for deciding use the most information which its can. This means to use exactly soft information from likelihood function.

Hard decision is good for binary symmetric channel (BSC), where only errors are made by flip binary symbol (1 to 0 or 0 to 1) [8]. Also this can be used for binary erasure channel (BEC), where you send 0, 1 and receive 1, 0 and e for error. This means that you received 1, it is certainly 1, but when you receive symbol e, then it means that you lost that symbol during transmission. We did not implement this model of channel, but it is easy to made it.

Hard decision can also be used for noisy channels, but with worse results than with soft decision. For BPSK we assume that received symbol is 0 for $\Lambda(0) \geq \Lambda(1)$, otherwise symbol is 1 for $\Lambda(1) > \Lambda(0)$. If $\Lambda(0) = \Lambda(1)$, then it does not matter which symbol we guess, because we really cannot say anything about it.

For examples in this chapter we use LDPC code with parity check matrix $\mathbf{H}$ with $m = 4$, $n = 6$, $w_c = 2$, $w_r = 3$, without 4-cycles. To be accurate, this one

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

with its factor graph in figure 2.1.

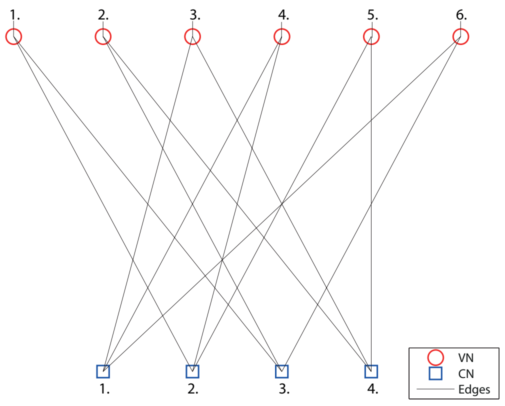

Fig. 2.1 – Factor graph for **H** ($m = 4$, $n = 6$, $w_c = 2$, $w_r = 3$)

2.1 Sum-Product algorithm

The most often used decoder for LDPC codes over factor graph is the sum-product algorithm (abbreviated SPA). Many authors have many names for this algorithm - SPA like in this thesis, the belief propagation algorithm or message passing algorithm.

We are using flooding schedules for decoding. This means that we send all messages from variable nodes to check nodes in one time and vice versa from check nodes to variable nodes. We divide SPA to three categories by type of input data. The first one is soft decision with message in probability form. Then we have reduced form with log-likelihood ratio, which is easier to compute than probability form. At least we have hard decision which reduced information only on binary message. Used methods are described in [11].

2.1.1 Soft decision

We assume that we received transmitted code over AWGN channel and the most information we gain from soft decision, where we use soft information from likelihood function (1.20). We can reduce soft information to hard decision, but then we will gain worse results than from soft decision.

We firstly show decoding with probability domain, which is more illustrative, but can be harder to compute. Then we show SPA with log-likelihood ratio, which is better for computing.

2.1.1.1 Probability domain

For SPA with probability domain we use input message with probabilities from likelihood function for AWGN channel (1.20), but we must multiply every i-th symbol with constant $k_i = \frac{1}{\Lambda_i(0)+\Lambda_i(1)}$. Then we have $p_i(0) = k_i\Lambda_i(0)$ and $p_i(1) = k_i\Lambda_i(1)$, which give us $p_i(0) + p_i(1) = 1$.

The main point of sum-product algorithm is method how check nodes and variable nodes exchange incoming messages with out coming message. Especially the one outcome message from node is composite from all other incoming messages. So from that ideally after few iteration every node is somehow connected to all others. From this follows that every message is influenced by all observation from channel.

Check nodes control that XOR of all incoming messages is zero (like we will more show in hard decision in Fig. 2.5). So probability of correct check node with two inputs is

$$\begin{aligned}\Pr[c_1 \oplus c_2 = 0] &= q_1(1)q_2(1) + q_1(0)q_2(0) \\ &= q_1(1)q_2(1) + (1 - q_1(1))(1 - q_2(1)) \\ &= \frac{1}{2}[1 + (1 - 2q_1(1))(1 - 2q_2(1))] = q. \qquad (2.2)\end{aligned}$$

It is probability that $c_1 = c_2$, so $(1+1) \bmod 2 = (0+0) \bmod 2 = 0$. Probability of check node with error is $\Pr[c_1 \oplus c_2 = 1] = 1 - q$.

For three incoming messages we need even number of 1's for correct check node. We have

$$\begin{aligned}\Pr[(c_1 \oplus c_2) \oplus c_3 = 0] &= (1 - q)q_3(1) + qq_3(0) \\ &= \frac{1}{2}[1 + (1 - 2(1 - q))(1 - 2q_3(1))] \\ &= \frac{1}{2}[1 + (1 - 2q_1(1))(1 - 2q_2(1))(1 - 2q_3(1))] \qquad (2.3)\end{aligned}$$

which we can generalize it to

$$\Pr[(c_1 \oplus ... \oplus c_n = 0] = \frac{1}{2} + \frac{1}{2}\prod_{i=1}^{n}(1 - 2q_i(1)). \qquad (2.4)$$

Therefore, out coming message from check node to variable node is

$$r_{ji}(0) = \frac{1}{2} + \frac{1}{2} \prod_{i' \in V_j \neq i} (1 - 2q_{i'j}(1)) \qquad (2.5)$$

$$r_{ji}(1) = 1 - r_{ji}(0) \qquad (2.6)$$

where V_j is the set of all variable nodes connected to j-th check node (in figure 2.2 a) is shown example of exchange messages in check node).

Variable node receives messages from connected check nodes and observation x_i from channel. From that we have for message from variable node to check node

$$q_{ij}(0) = k_{ij}x_i(0) \prod_{j' \in C_i \neq j} r_{j'i}(0) \qquad (2.7)$$

$$q_{ij}(1) = k_{ij}x_i(1) \prod_{j' \in C_i \neq j} r_{j'i}(1), \qquad (2.8)$$

where C_i is the set of all check nodes connected to i-th variable node. The constant k_{ij} is there for fulfillment condition $q_{ij}(0) + q_{ij}(1) = 1$. (in figure 2.2 b) is shown example of exchange messages in variable node).

For output of algorithm we count effective probabilities of every code symbol

$$Q_i(0) = k_i x_i(0) \prod_{j \in C_i} r_{ji}(0) \qquad (2.9)$$

$$Q_i(1) = k_i x_i(1) \prod_{j \in C_i} r_{ji}(1), \tag{2.10}$$

where we count with all incoming messages from check nodes and observation (shown in figure 2.2 c)). After that we can gain received code word by $c_i = 1$ if $Q_i(1) > Q_i(0)$, or $c_i = 0$ if $Q_i(0) \geq Q_i(1)$.

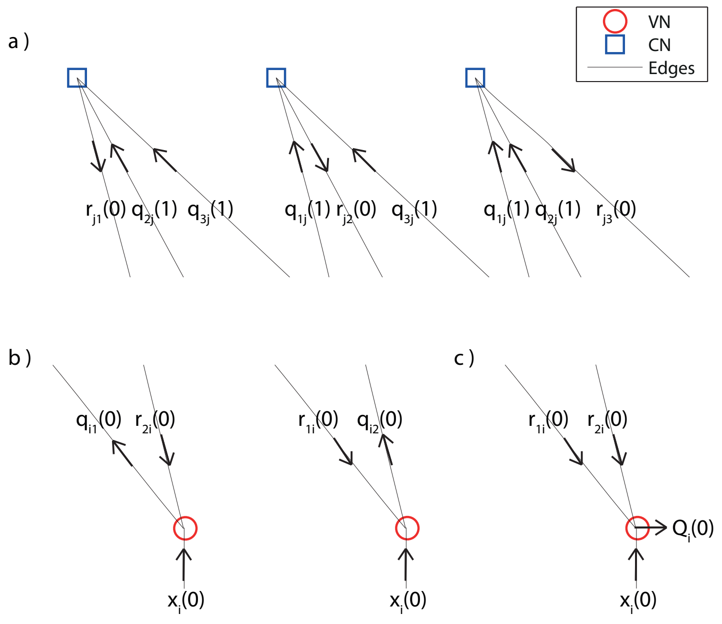

Fig. 2.2 – Exchange messages in soft decision SPA in probability domain

Algorithm 2.1 Soft decision SPA in probability domain

INPUT: x - input probabilities of 0, 1 of symbols, max - maximal iteration, $\mathbf{H}$ - parity-check matrix

OUTPUT: c - corrected codeword

Step 1: Make factor graph from $\mathbf{H}$. Set $counter = 1$, $q_{ij}(0) = x_i(0)$, $q_{ij}(1) = x_i(1)$, where $q_{ij}(0)$ is edge connected from i-th variable node to j-th check node with probability of 0 symbol. The same with $q_{ij}(1)$.

Step 2: If $counter \leq max$, then go to Step 3. Otherwise, Stop with information about reach maximal iteration.

Step 3: Count $r_{ji}(0) = \frac{1}{2} + \frac{1}{2} \prod\limits_{i' \in V_j \neq i} (1 - 2q_{i'j}(1))$, $r_{ji}(1) = 1 - r_{ji}(0)$, where V_j is the set of all variable nodes connected to j-th check node. $r_{ji}(0)$ represent edge connected from j-th check node to i-th variable node and send probability that i-th variable node represent symbol 0. The same with $r_{ji}(1)$.

Step 4: Count $q'_{ij}(0) = x_i(0) \prod\limits_{j' \in C_i \neq j} r_{j'i}(0)$, $q'_{ij}(1) = x_i(1) \prod\limits_{j' \in C_i \neq j} r_{j'i}(1)$, where C_i is the set of all check nodes connected to i-th variable node. Then count $q_{ij}(0) = \frac{q'_{ij}(0)}{q'_{ij}(0)+q'_{ij}(1)}$ and $q_{ij}(1) = \frac{q'_{ij}(1)}{q'_{ij}(0)+q'_{ij}(1)}$ for $q_{ij}(0) + q_{ij}(1) = 1$.

Step 5: Count $Q_i(0) = k_i x_i(0) \prod\limits_{j \in C_i} r_{ji}(0)$, $Q_i(1) = k_i x_i(1) \prod\limits_{j \in C_i} r_{ji}(1)$, where k_i is constant to ensure $Q_i(0) + Q_i(1) = 1$. Q_i represent effective probability of 0 and 1 at i-th symbol from all incoming messages to i-th variable node. Then we estimate $c_i = 1$ if $Q_i(1) > Q_i(0)$, or $c_i = 0$ if $Q_i(0) \geq Q_i(1)$.

Step 6: Count $c \cdot \mathbf{H}^{\mathbf{T}} \bmod 2$. If the result is zero vector, then Stop. Otherwise, set $counter = counter + 1$ and go to Step 2.

The MATLAB code for it is in A.5.1.1 on page 54.

2.1.1.2 Log-likelihood ratio

The principle of SPA for log-likelihood ratio (LLR) is similar to SPA for probability domain. Just we do not use separate probabilities for 0 and 1 symbol, but their ration. For easier counting we use this ratio in logarithmically domain. Exactly we use for i-th symbol

$$l_i = \ln \frac{\Lambda_i(0)}{\Lambda_i(1)}. \tag{2.11}$$

Then it is easy from l_i to determine which symbol is more likely received. Especially when $l_i = \pm\infty$, then we can certainly determine it. It is 0 if $l_i > 0$ or 1 if $l_i < 0$. If $l_i = 0$, then we can't say about it anything and we can choose if it is 0 or 1.

Log-likelihood ratio sum-product algorithm is the same like with probabilities, but we substitute LLR to probabilities and join separated equation for $p_i(0)$ and $p_i(1)$ to one with l_i.

Variable node sends message to check nodes exactly like in probability domain. We use $q'_{ij}(0)$ from (2.7), $q'_{ij}(1)$ from (2.8) and observation x_i to have

$$\begin{aligned} q_{ij} = \ln \frac{q'_{ij}(0)}{q'_{ij}(1)} &= \ln \left[\frac{x_i(0)}{x_i(1)} \prod_{j' \in C_i \neq j} \frac{r_{j'i}(0)}{r_{j'i}(1)} \right] \\ &= x_i + \sum_{j' \in C_i \neq j} r_{j'i}. \end{aligned} \tag{2.12}$$

For message coming from check nodes to variable node we use $r'_{ji}(0)$ from (2.5) and $r'_{ji}(1)$ from (2.6). Then we get

$$\begin{aligned} r_{ji} = \ln \frac{r'_{ji}(0)}{r'_{ji}(1)} &= \ln \frac{\frac{1}{2} + \frac{1}{2} \prod\limits_{i' \in V_j \neq i} (1 - 2q'_{i'j}(1))}{\frac{1}{2} - \frac{1}{2} \prod\limits_{i' \in V_j \neq i} (1 - 2q'_{i'j}(1))} \\ &= \ln \frac{1 + \prod\limits_{i' \in V_j \neq i} \tanh(\frac{q_{i'j}}{2})}{1 - \prod\limits_{i' \in V_j \neq i} \tanh(\frac{q_{i'j}}{2})}. \end{aligned} \tag{2.13}$$

We get (2.13) from (2.12), where

$$\begin{aligned} e^{q_{ij}} &= \frac{1 - q'_{ij}(1)}{q'_{ij}(1)} \\ q'_{ij}(1) &= \frac{1}{1 + e^{q_{ij}}} \end{aligned}$$

and then

$$1 - 2q'_{ij}(1) = \frac{e^{q_{ij}} - 1}{e^{q_{ij}} + 1} = \tanh(\frac{q_{ij}}{2}).$$

At last we turn equation (2.9),(2.10) to

$$Q_i = \ln \frac{Q'_i(0)}{Q'_i(1)} = l_i + \sum_{j \in C_i} r_{ji}. \tag{2.14}$$

If $Q_i > 0$,the $c_i = 0$, otherwise $c_i = 1$.

Algorithm 2.2 Soft decision SPA with log-likelihood ratio

INPUT: l - input LLR of symbols, max - maximal iteration, $\mathbf{H}$ - parity-check matrix
OUTPUT: c - corrected codeword

Step 1: Make factor graph from $\mathbf{H}$. Set $counter = 1$, $q_{ij} = l_i$. Where q_{ij} is edge connected from i-th variable node to j-th check node with LLR information.

Step 2: If $counter \leq max$, then go to Step 3. Otherwise, Stop with information about reach maximal iteration.

Step 3: Count $r_{ji} = \ln \frac{1+\prod\limits_{i' \in V_j \neq i} \tanh(\frac{q_{i'j}}{2})}{1-\prod\limits_{i' \in V_j \neq i} \tanh(\frac{q_{i'j}}{2})}$, where V_j is the set of all variable nodes connected to j-th check node. r_{ji} represent edge connected from j-th check node to i-th variable node and sending LLR information.

Step 4: Count $q_{ij} = x_i + \sum\limits_{j' \in C_i \neq j} r_{j'i}$, where C_j is the set of all check nodes connected to i-th variable node.

Step 5: Count $Q_i = x_i + \sum\limits_{j \in C_i} r_{ji}$, where Q_i represent effective LLR of i-th symbol from all incoming messages to i-th variable node. Then we estimate $c_i = 0$ if $Q_i \geq 0$, or $c_i = 1$ if $Q_i < 0$.

Step 6: Count $c \cdot \mathbf{H}^{\mathbf{T}} \bmod 2$. If result is zero vector or all $Q_i = \pm\infty$, then Stop. Otherwise, set $counter = counter + 1$ and go to Step 2.

The MATLAB code for it is in A.5.1.2 on page 55.

Example: In figure 2.4 is example of soft decision sum-product algorithm with log-likelihood ratio. We coded image (600 x 451 px) where we coded every row with RA LDPC code with rate $r = 0.5$. $\mathbf{H}$ has $m = 600$, $n = 1200$ and base matrix $\mathbf{H_s}$ has $w_{c'} = 3$, $w_{r'} = 3$, so whole $\mathbf{H}$ has $w_r = 5$, $w_{c1} = 3$ with $v_{w_{c1}} = 0.5$ and $w_{c2} = 2$ with $v_{w_{c2}} = 0.5$. Matrix $\mathbf{H}$ is in Fig. 2.3. Code is send over AWGN channel with SNR = - 0.3 dB.

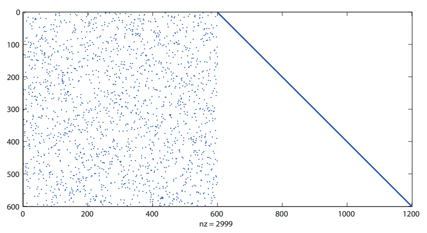

Fig. 2.3 – Parity check matrix **H** for example of SPA LLR

In section a) of the figure 2.4 we can see how we coded image and added parity check bits. Then we have in section b) of figure, what we received from AWGN channel with SNR = - 0.3 dB. There is strong noise. We corrected image with SPA LLR. In section c) is progress of decoder. It is noticeable how errors are correcting from one iteration to next. Correction of the whole image is complete in the twenty-first iteration, but the most of all rows are corrected in the eighth iteration. □

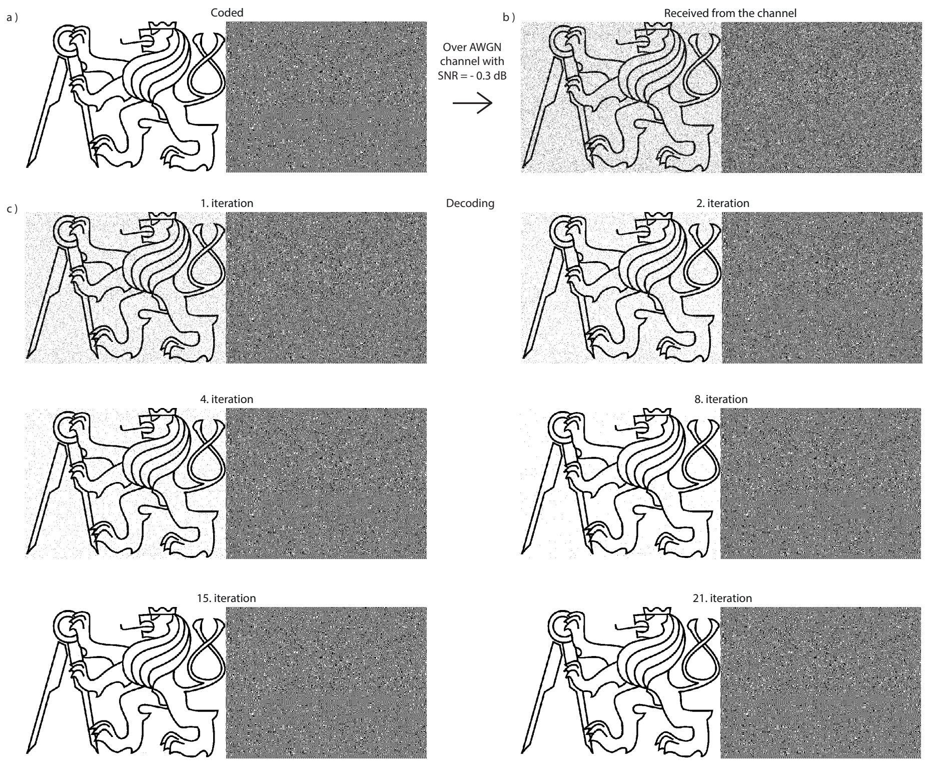

Fig. 2.4 – Example of soft decision SPA with LLR

2.1.2 Hard decision

Sum-product algorithm works very well with soft information, but we can approximate it to hard decision. How it was said in the introduction to this chapter, we can reduce output of likelihood function (1.20) to 0's or 1's with rules that we received symbol is 0 for $\Lambda(0) \geq \Lambda(1)$, otherwise symbol is 1 for $\Lambda(1) > \Lambda(0)$. Or hard decision is good for binary symmetric channel (BSC) or other channels, which use only binary information.

The main idea of hard decision SPA is to count every check nodes and then check nodes which do not meet condition of correct check node (that mean zero value, Fig. 2.5 a)) return to variable nodes opposite values than that come in. We assume that there is one error bit, so check node send back opinion of correcting it to change symbol to opposite value, but can not determine which variable nodes have error, so send it to every one. We can see an example in figure 2.5 b), c), d), where we change the first, second and at last message to variable node. Then we have different inputs from check nodes and from observation to one variable node. The most often incoming value is our decided value.

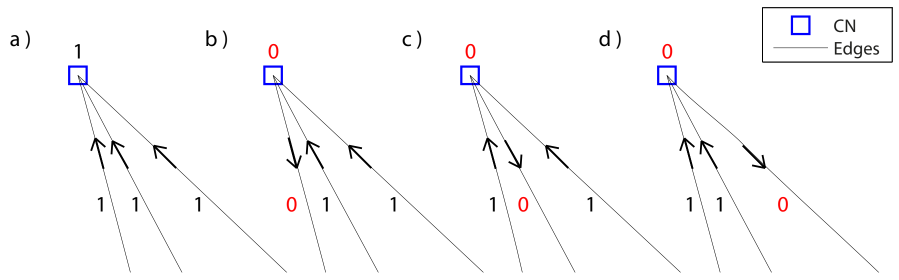

Fig. 2.5 – Exchange messages in check nodes in hard decision SPA

Algorithm 2.3 Hard decision SPA

INPUT: x - input codeword, max - maximal iteration, $\mathbf{H}$ - parity-check matrix
OUTPUT: c - corrected codeword

```
Step 1: Make factor graph from $\mathbf{H}$. Set $c = x$, $counter = 0$.
Step 2: If $counter \leq max$, then go to Step 3. Otherwise, Stop with information
        about reach maximal iteration.
Step 3: Count $CN = c \cdot \mathbf{H^T} \bmod 2$, where $CN$ is vector of check nodes. If $CN$
        is zero vector, then Stop. Otherwise, go to Step 4.
Step 4: For every elements of $CN$. Let $i$-th element of $CN$ is zero, then $i$-th
        $CN$ return to variable nodes what it received from them. Otherwise, if
        $i$-th $CN$ is one, then $i$-th $CN$ send opposite value to variable nodes than
        what it received.
Step 5: For every elements of $c$. Let $j$-th element of $c$ is the most often value
        which $j$-th variable node received from every connected check nodes and
        observation $x$.
Step 6: Set $counter = counter + 1$ and go to Step 2.
```

The MATLAB code for it is in A.5.1.3 on page 57.

Example: Again we use $\mathbf{H}$ (from (2.1)) and code word $c_0 = \begin{bmatrix}1 & 1 & 1 & 1 & 0 & 0\end{bmatrix}$. Let send c_0 over BSC and receive $c = \begin{bmatrix}1 & 1 & 0 & 1 & 0 & 0\end{bmatrix}$ with one error on position three.

Let send received code c from variable nodes to check nodes (Fig. 2.6 a)). We can see that they are two bad check nodes, the first and fourth. It means that the first and fourth check nodes send back opposite values than which they received(Fig. 2.6 b)). Then to third variable nodes come two 1's from check nodes and 0 from observation c. This means that we correct the third symbol of code to 1 and then we gain $c' = \begin{bmatrix}1 & 1 & 1 & 1 & 0 & 0\end{bmatrix}$. That is what we sent. $\square$

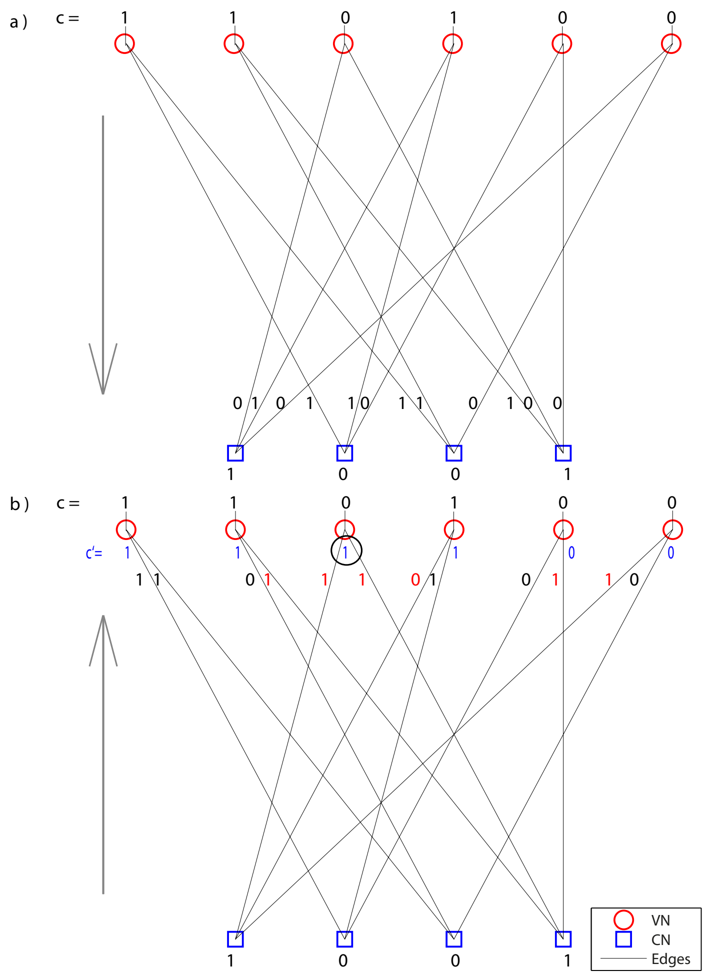

Fig. 2.6 – Example of hard decision SPA

2.2 Bit-flip decoding

How it is described in [4], the main idea of bit-flip decoding (abbreviated BF) is to correct variable node or nodes which are connected to the the greatest number of bad check nodes. Error correction is with bit flip (change $0 \rightarrow 1$ or $1 \rightarrow 0$).

Algorithm 2.4 Bit-flip decoding

INPUT: x - input codeword, max - maximal iteration, $\mathbf{H}$ - parity-check matrix
OUTPUT: c - corrected codeword

```
Step 1: Set $counter = 0$, $c = x$.
Step 2: If $counter \leq max$, then go to Step 3. Otherwise, Stop with information
        about reach maximal iteration.
Step 3: Count $CN = c \cdot \mathbf{H}^{\mathbf{T}}$ mod 2, where $CN$ is vector of values of check
        nodes. If the $CN$ is zero vector, then Stop. Otherwise, go to Step 4.
Step 4: For every bit of $c$, which represent variable node, count the number
        of connected check nodes, which are not have zero value. Change bit (or
        bits) of $c$ with the greatest number of errors from 0 to 1 or from 1 to 0
        and then set $counter = counter + 1$. Then go to Step 2.
```

The MATLAB code for it is in A.5.2 on page 58.

Example: Let have $\mathbf{H}$ (from (2.1)) and code word $c_0 = \begin{bmatrix}1 & 1 & 1 & 1 & 0 & 0\end{bmatrix}$. Let send c_0 over BSC and receive $c = \begin{bmatrix}1 & 0 & 1 & 1 & 0 & 0\end{bmatrix}$ with one error on position two.

Then we send received c over factor graph from $\mathbf{H}$ and count values in check nodes (Fig. 2.7 a)). We can see that we have the third and fourth check nodes with errors. It means that only the fourth variable node is not connect to check node with error. Then the first, third, fifth and sixth are connected to one check node with error and only the second variable node is connected with two check nodes with error (Fig. 2.7 b)). So we bit flip this check node, so we change 0 to 1 and we gain $c' = \begin{bmatrix}1 & 1 & 1 & 1 & 0 & 0\end{bmatrix}$. Again we send it to check nodes and we can see, that is right code word and we got what we sent (Fig. 2.7 c)). $\square$

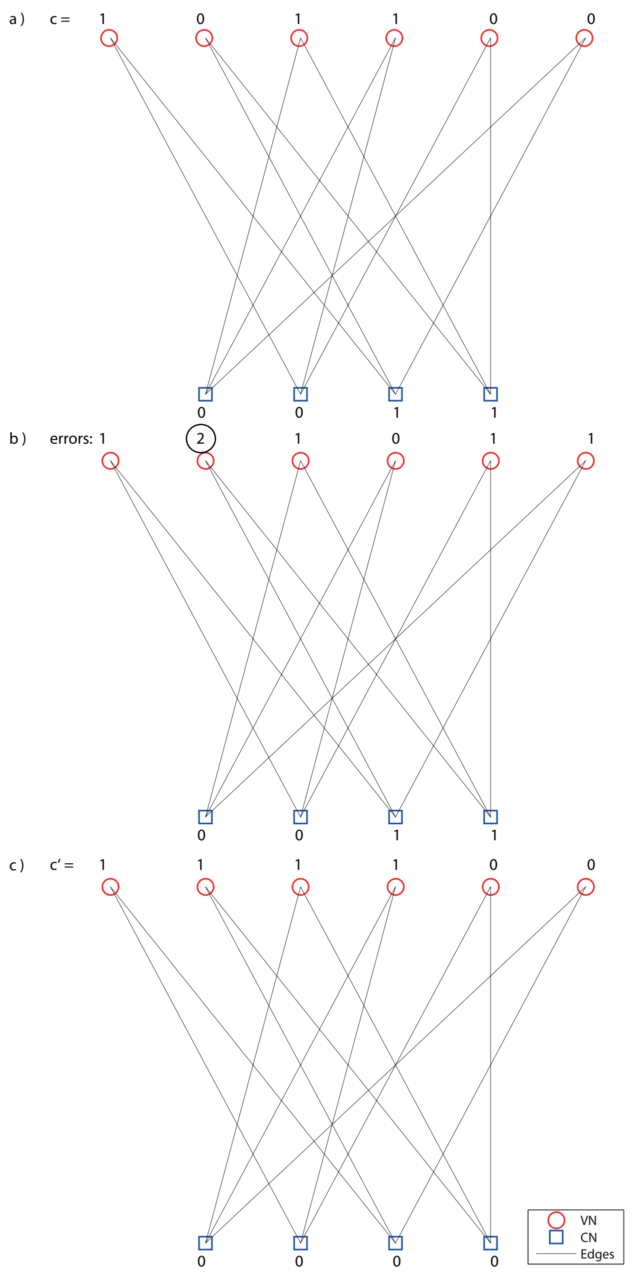

Fig. 2.7 – Example of BF decoding

Chapter 3

Type of codes

LDPC code is defined by its parity-check matrix $\mathbf{H}$. Matrix determine performance of code (how many errors can be corrected (how bad channel can be), how many iteration need to be correct, a rate etc.), coder, decoder etc. An important attribute is construction of parity-check matrix $\mathbf{H}$ and then construction generating matrix $\mathbf{G}$ from this matrix $\mathbf{H}$. There are two types of construction - pseudo-random construction in section 3.1 or algebraic construction in section 3.2. Encoder and decoder for code with pseudo-random constructed $\mathbf{H}$ need to remember really big $\mathbf{H}$ and also it is really hard to make $\mathbf{G}$ with general way from solve equation (1.17) (and sparse matrix $\mathbf{H}$ may not produce sparse matrix $\mathbf{G}$ and then encoding isn't to simple). On the other hand pseudo-random can have better performances than algebraic.

3.1 Pseudo-random construction

Pseudo-random construction was introduced by Gallager [3]. This construction makes good parity-check matrix $\mathbf{H}$, but not generally with easy encoding. Matrix is constructed by pseudo-random placing 1's to zero matrix and with that we connected variable nodes with check nodes. Pseudo is because we control condition of w_c and w_r, it means that we control count of 1's in every row and column. Also control presence of of 4-cycles, or bigger cycles.

With pseudo-random $\mathbf{H}$ is not easy to find way to encoding. Matrix can be really large and can be really hard to solve $\mathbf{G}{\cdot}\mathbf{H}^T = 0$ (1.17). Then we can use other option - use this $\mathbf{H}$ like base matrix for repeat-accumulate low-density parity check code, which we introduce in subsection 3.1.1.

For construction we used algorithm described in [1], which we modified. They make binary random column by column and then new column control with previous columns for meets conditions. As they describe it, they can make only regular $\mathbf{H}$, where w_c is same for every column and w_r is same for every row.

We are making it row by row, but mainly we don't making binary row, but row with indices of 1's in row, so we change this

$$\mathbf{H}_i = \left[0\,0\,1\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\right]$$

into this

$$\mathbf{H}'_i = \begin{bmatrix} 3 & 10 & 22 & 27 \end{bmatrix}.$$

Then it is easy to control number of 1's in every row (number of indices in row is equal to number of 1's in standard row) and in every column (number of repetition of index i represent weight of i-th column). Also this type of description of matrix is more savers than normal description (you have matrix $m \times w_r$, than $m \times n$, where $w_r \ll n$). After know that all, it is easy to make matrix.

Algorithm 3.1 Pseudo-random construction of H

```
INPUT: $n$ - columns, $w_r$- weight of row (must be regular), $w_{ci}$ - $i$-th weight of column, $v_i$- degree distribution of $w_{ci}$, $max$ - maximal loops in one cycle
OUTPUT: H - parity check matrix

Step 1: Sort $w_{ci'} < ... < w_{ci''}$. Count number of rows $m = n\frac{\sum_i w_{ci} v_i}{w_r}$ ($m$ must be integer). For regular code it is only $m = n\frac{w_c}{w_r}$.
Step 2: Count real number of column with weight $w_{ci}$, $w_{cni} \doteq v_i \cdot n$. Where $\sum_i w_{cni} = n$.
Step 3: Permute indices of all columns, $col = \sigma(1, 2, ..., n)$. Then we make vector of column's indices of all 1's. $\mathbf{H}' = [\text{repeat-}w_{ci'}\text{-times}(col[1...w_{cni'}])\ \ \text{repeat-}w_{ci''}\text{-times}(col[(w_{cni'} + 1)...(w_{cni'} + w_{cni'})]\ \ ...\ \ \text{repeat-}w_{ci''''}\text{-times}(col[(w_{cni'} + ...w_{cni'''} + 1)...(w_{cni'} + ... + w_{cni''''})])$. Then we reshape this vector to matrix $m \times w_r$.
Step 4: Set $a = 1$, where $a$ represent actually constructed $a$-th row.
Step 5: If $a \leq m$, then set $help = 0$ and go to Step 3.($help$ stop from endless loops) Otherwise, go to Step 10.
Step 6: If $help < max$, then permute elements of $a$-th to $m$-th row of $\mathbf{H}'$. Otherwise set $a = a - 1$ and go to Step 5.
Step 7: If in the $a$-th row are two or more identical indices, then set $help = help + 1$ and go to Step 6. Otherwise, go to Step 8.
Step 8: If in the $a$-th row and any of 1-st to $(a-1)$-th row are more than one identical indices, then set $help = help + 1$ and go to Step 6. Otherwise, go to Step 9.
Step 9: Set $a = a + 1$ and go to Step 5.
Step 10: Transfer matrix of indices $\mathbf{H}'$ to standard binary matrix $\mathbf{H}$. Then Stop.
```

The MATLAB code for it is in A.6.1 on page 61.

Algorithm 3.1 can make irregular-columns matrix with different w_r for every row. In [5] show, that irregular-rows did not change performance of code more than the irregular-columns. From that we can use only irregular-columns.

Step 8 in Algorithm 3.1 eliminate 4-cycles from matrix, which are described in subsection 1.3.3. Because with

$$\mathbf{H}' = \begin{bmatrix} 1 & 3 & 7 & 10 \\ 2 & 5 & 8 & 9 \\ 1 & 2 & 4 & 7 \end{bmatrix}$$

we have

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

where is in the first and seventh columns and the first and third rows 4-cycle.

Example: We want regular LDPC parity check matrix $\mathbf{H}$, with $n = 10$, $m = 5$, $w_r = 4$, $w_c = 2$. We will make it with Algorithm 3.1. First we will make matrix of all 1's elements indices

$$\mathbf{H}' = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{bmatrix}$$

and reshape it to matrix 5×4 $(m \times w_r)$.

$$\mathbf{H}' = \begin{bmatrix} 1 & 3 & 6 & 8 \\ 1 & 4 & 6 & 9 \\ 2 & 4 & 7 & 9 \\ 2 & 5 & 7 & 10 \\ 3 & 5 & 8 & 10 \end{bmatrix},$$

which can be transformed into

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}.$$

Now we can see, that matrix $\mathbf{H}$ fulfill conditions of w_r, w_c, but after only reshaping is full of 4-cycles, which corrupt performance of code. So then we permute full matrix and then we will control row by row. After the first permutation we get

$$\mathbf{H}' = \begin{bmatrix} \mathbf{6} & \mathbf{4} & \mathbf{5} & \mathbf{5} \\ 7 & 9 & 1 & 8 \\ 3 & 4 & 10 & 8 \\ 7 & 9 & 2 & 6 \\ 3 & 2 & 10 & 1 \end{bmatrix}$$

and we now only control the first row where we have now two same elements. Then we permute it again and get

$$\mathbf{H}' = \begin{bmatrix} \mathbf{9} & \mathbf{10} & \mathbf{5} & \mathbf{6} \\ 1 & 7 & 3 & 2 \\ 7 & 8 & 3 & 2 \\ 10 & 6 & 1 & 8 \\ 5 & 4 & 4 & 9 \end{bmatrix}.$$

We have the complete the first row, so we now permute the second to fifth rows of $\mathbf{H}'$,thus

$$\mathbf{H}' = \begin{bmatrix} 9 & 10 & 5 & 6 \\ \mathbf{10} & \mathbf{1} & \mathbf{8} & \mathbf{7} \\ 6 & 4 & 3 & 8 \\ 5 & 4 & 2 & 3 \\ 1 & 7 & 2 & 9 \end{bmatrix}.$$

In the second row there are not any duplicates and between the first and second rows there are only one same index, so second row is good, then we can make the third row.

$$\mathbf{H}' = \begin{bmatrix} 9 & 10 & 5 & 6 \\ 10 & 1 & 8 & 7 \\ \mathbf{3} & \mathbf{5} & \mathbf{1} & \mathbf{6} \\ 4 & 7 & 3 & 8 \\ 2 & 2 & 9 & 4 \end{bmatrix} \tag{3.1}$$

The third row does not have two the same indices, but it has 4-cycle with first row. We again permutes the third to the fifth row of $\mathbf{H}'$. Then we will make the fourth and the fifth row. After all iteration, we will gain final matrix

$$\mathbf{H}' = \begin{bmatrix} 5 & 6 & 9 & 10 \\ 1 & 2 & 4 & 6 \\ 2 & 3 & 8 & 9 \\ 1 & 3 & 5 & 7 \\ 4 & 7 & 8 & 10 \end{bmatrix}, \tag{3.2}$$

where the second row is different than in $\mathbf{H}'$ from (3.1) because during making matrix occur endless loop, so it was necessary to come back to good rows and change them. We transform (3.2) to standard binary matrix

$$\mathbf{H} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{3.3}$$

and we can see that matrix $\mathbf{H}$ fulfill n, m, w_r, w_c and it is 4-cycle free. □

This method can generate good LDPC codes from $\mathbf{H}$, but problem of pseudo-random matrix is problematic conversion to generating matrix $\mathbf{G}$. For big matrix $\mathbf{H}$ it is really hard to solve equation (1.17), so we can use repeat-accumulate code.

3.1.1 Repeat-accumulate code

Big problem with code from pseudo-random matrix $\mathbf{H}$ is its problematic encoding. Because we assume big dimension of matrix, so then it will be really hard to solve $\mathbf{G}{\cdot}\mathbf{H}^T = 0$ (1.17). But we can use repeat-accumulate codes (abbreviated RA) [5] (or Staircase codes in [8]).

RA LDPC code make encoding really fast without hard computation. Parity check matrix $\mathbf{H}$ for repeat-accumulate low density parity check code is composite from two submatrices - $\mathbf{H_s}$ and $\mathbf{H_p}$. $\mathbf{H_s}$ is base submatrix of $\mathbf{H}$, it is parity

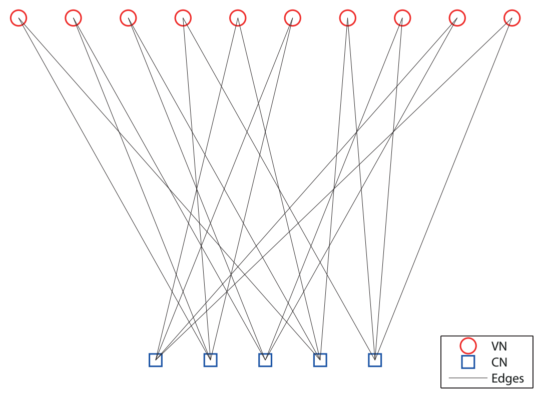

Fig. 3.1 – Factor graph of matrix **H** from (3.3)

check matrix for LDPC code, which control source bits s of code c. For example matrix $\mathbf{H}$ from section (3.1). $\mathbf{H_p}$ is submatrix, which controls parity bits p of code c. It is submatrix of dimension $m \times m$ (m from $\mathbf{H_s}$) with ones on main diagonal and ones on subdiagonal immediately below main diagonal.

In (3.4) is example of RA LDPC parity check matrix $\mathbf{H}$, where $\mathbf{H_s}$ is from (3.3) and in figure 3.2 its factor graph.

$$\mathbf{H} = [\mathbf{H_s}|\mathbf{H_p}] = \left[\begin{array}{cccccccccc|ccccc} 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{array}\right] \tag{3.4}$$

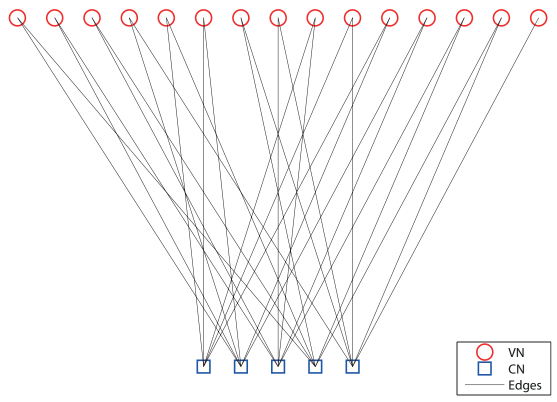

Fig. 3.2 – Factor graph of matrix **H** from (3.4)

Encoding is systematic and it goes in two steps described in Algorithm 3.2. Gained code have first k data bits and then m parity bits.

```
Algorithm 3.2 Repeat-accumulate encoding
INPUT: $s$ - data, $\mathbf{H} = [\mathbf{H_s}|\mathbf{H_p}]$- parity check matrix of RA LDPC
OUTPUT: $c = [s|p]$- codeword

Step 1: Compute $v = \mathbf{H_s}s$.

Step 2: Compute parity bits $p$ by
```

$$\begin{aligned}
p_1 &= & & \sum_{i=1}^{k} \mathbf{H_s}_{1i} s_i &&= & & v_1 \\
p_2 &= p_1 &+& \sum_{i=1}^{k} \mathbf{H_s}_{2i} s_i &&= p_1 &+& v_2 \\
p_3 &= p_2 &+& \sum_{i=1}^{k} \mathbf{H_s}_{3i} s_i &&= p_2 &+& v_3 \\
&\vdots \\
p_m &= p_{m-1} &+& \sum_{i=1}^{k} \mathbf{H_s}_{mi} s_i &&= p_{m-1} &+& v_m.
\end{aligned} \tag{3.5}$$

The MATLAB codes for it is in A.6.2 on page 62.

When $\mathbf{H_s}$ is sparse matrix (by definition of LDPC), then computation of v is just only addition (or XOR) of few elements and (3.5) is then only accumulation of p.

On the same principle work also LDPC codes for DVB-S2 (Digital Video Broadcasting - Satellite - Second Generation), which is then only defined by different $\mathbf{H_s}$[9].

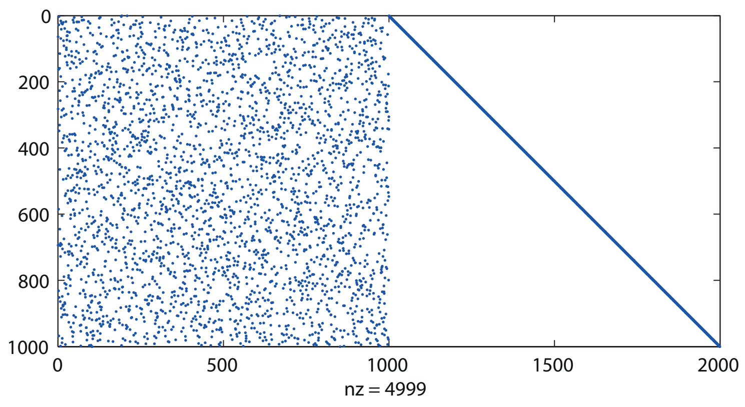

Fig. 3.3 – RA LDPC code's parity check matrix $\mathbf{H}$ ($n = 2000$, $m = 1000$, $R = \frac{1}{2}$), where $\mathbf{H_s}$ is pseudo-random with $w_r = 3$, $w_c = 3$, with 4999 ones

3.2 Algebraic construction

As it is shown in the previous section, generate parity check matrix $\mathbf{H}$ by pseudo-random construction can be very tricky. It takes time to generate matrix, but the main problem is their remake and storage. If you make $\mathbf{H}$ from pseudo-random construction, then you must save it, because there is not another way to make it same again. Together with that there is also a problem with storage of codes. We must store the whole random matrices or their compressed form, as from Section 3.1, where we storage only indices of 1's. Although, that matrix of indices can be really big for matrix $\mathbf{H}$ with $n = 64800$, $m = 32400$.

On the other hand, with parity check matrix made by algebraic construction we gain an easy way to store and make matrix. Algebraic construction means that we have some algorithm, which from inputs can make parity check matrix. That algorithm, program code, can be and it is a lot of easier for storage than random matrix. And with good algorithm we can have plenty of matrices $\mathbf{H}$ for various inputs. For Quasi-cyclic code from subsection 3.2.1 we can gain many codes only with choosing w_r, w_c and parameter for the size of matrix. Or matrix with only few allowed configuration, where you can choose rate r of code and length of codeword n, like in construction from Subsection 3.2.2.

3.2.1 Quasi-cyclic code

This method was shown in [1, 5] and it is really easy to compute and store. The method which makes quasi-cyclic (QC) LDPC codes is only repeating and cycle shifting of predetermined matrix. In our case we will use matrix, which is only cyclically shifted identity matrix, exactly

$$\mathbf{J}_{p\times p} \triangleq \begin{bmatrix} 0 & 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 \\ 1 & 0 & 0 & 0 & \dots & 0 & 0 \end{bmatrix}. \tag{3.6}$$

Then this method made parity check matrix $\mathbf{H}$ from given w_r, w_c and chosen $p > (w_c - 1)(w_r - 1)$, which determine dimension of matrix $\mathbf{J}$.

After all was given, we only count this matrix

$$\mathbf{H} = \begin{bmatrix} \mathbf{J}^0 & \mathbf{J}^0 & \mathbf{J}^0 & \dots & \mathbf{J}^0 \\ \mathbf{J}^0 & \mathbf{J}^1 & \mathbf{J}^2 & \dots & \mathbf{J}^{w_r-1} \\ \mathbf{J}^0 & \mathbf{J}^2 & \mathbf{J}^4 & \dots & \mathbf{J}^{2(w_r-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{J}^0 & \mathbf{J}^{(w_c-1)} & \mathbf{J}^{2(w_c-1)} & \dots & \mathbf{J}^{(w_r-1)(w_c-1)} \end{bmatrix}, \tag{3.7}$$

where l-th power of $\mathbf{J}$ represent cyclical shifting rows by l mod p positions to the right, where $\mathbf{J}^0$ is identity matrix $\mathbf{I}_{p\times p}$.

Matrix $\mathbf{H}$ from (3.7) has exactly $n = w_r \cdot p$ rows and $m = w_c \cdot p$ columns. Also in every row is exactly w_r ones and exactly w_c ones in every column. It can be proofed that this construction guarantees no present of 4-cycles.

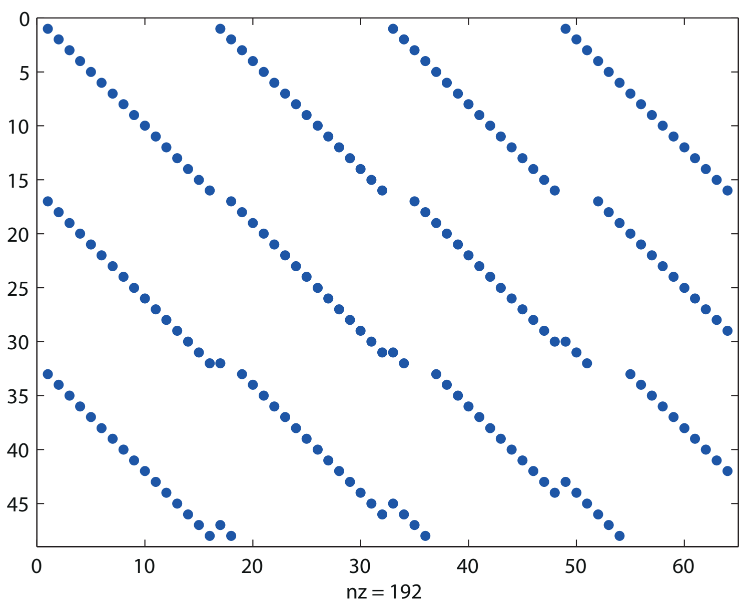

Fig. 3.4 – Example of QC LDPC code's parity check matrix $\mathbf{H}$, with $w_c = 4$, $w_r = 3$, $p = 16$

Matrix from this method can be also used like base matrix $\mathbf{H_s}$ in repeat-accumulate codes from Subsection 3.1.1.

The MATLAB code for encode is in A.6.4.1 on page 64.

3.2.2 AR4JA LDPC code

The second type of algebraic construction is type, where you only choose wanted parameters, like rate, code length or source data length and rest, like w_r, w_c is given by method. Good example of this type is AR4JA LDPC code from Orange Book [2], which is invented for near-Earth and deep space applications. This construction can make nine codes with different rates r or source block length k. Exactly values of supported values are in table 3.1.

Source block length k	Code block length n		
	rate $r = 1/2$	rate $r = 2/3$	rate $r = 4/5$
1024	2048	1536	1280
4096	8192	6144	5120
16384	32768	24576	20480

Tab. 3.1 – Codeblock length k for supported code rates r

Parity check matrix $\mathbf{H}$ is made by following matrices. For code with rate $r = 1/2$ is

$$\mathbf{H}_{1/2} = \begin{bmatrix} \mathbf{0}_M & \mathbf{0}_M & \mathbf{I}_M & \mathbf{0}_M & \mathbf{I}_M \oplus \mathbf{\Pi_1} \\ \mathbf{I}_M & \mathbf{I}_M & \mathbf{0}_M & \mathbf{I}_M & \mathbf{\Pi}_2 \oplus \Pi_3 \oplus \Pi_4 \\ \mathbf{I}_M & \Pi_5 \oplus \Pi_6 & \mathbf{0}_M & \Pi_7 \oplus \Pi_8 & \mathbf{I}_M \end{bmatrix}, \qquad (3.8)$$

where $\mathbf{I}_M$ and $\mathbf{0}_M$ are $M \times M$ identity and zero matrices. Π_k is permutation matrix with ones in row i and column $\pi_k(i)$ for $i \in \{0, 1, ..., M-1\}$ and

$$\pi_k(i) = \frac{M}{4}((\theta_k + \lfloor 4i/M \rfloor) \bmod 4) + (\phi_k(\lfloor 4i/M \rfloor, M) + i) \bmod \tfrac{M}{4} \tag{3.9}$$

where functions θ_k and $\phi_k(j, M)$ are defined in Tables 3.3 and 3.4. Value of size of submatrices M are from Tab. 3.2.

Source block length k	Submatrix size M		
	rate $r = 1/2$	rate $r = 2/3$	rate $r = 4/5$
1024	512	256	128
4096	2048	1024	512
16384	8192	4096	2048

Tab. 3.2 – Sizes of submatrix M for supported codes

For codes with rate $r = 2/3$ we have matrix

$$\mathbf{H}_{2/3} = \left[\begin{array}{cc|c} \mathbf{0}_M & \mathbf{0}_M & \\ \Pi_9 \oplus \Pi_{10} \oplus \Pi_{11} & \mathbf{I}_M & \mathbf{H}_{1/2} \\ \mathbf{I}_M & \Pi_{12} \oplus \Pi_{13} \oplus \Pi_{14} & \end{array}\right]. \tag{3.10}$$

And finally for code with rate $r = 4/5$, which come from matrix for rate $r = 3/4$, we have

$$\mathbf{H}_{3/4} = \left[\begin{array}{cc|c} \mathbf{0}_M & \mathbf{0}_M & \\ \Pi_{15} \oplus \Pi_{16} \oplus \Pi_{17} & \mathbf{I}_M & \mathbf{H}_{2/3} \\ \mathbf{I}_M & \Pi_{18} \oplus \Pi_{19} \oplus \Pi_{20} & \end{array}\right]$$

$$\mathbf{H}_{4/5} = \left[\begin{array}{cc|c} \mathbf{0}_M & \mathbf{0}_M & \\ \Pi_{21} \oplus \Pi_{22} \oplus \Pi_{23} & \mathbf{I}_M & \mathbf{H}_{3/4} \\ \mathbf{I}_M & \Pi_{24} \oplus \Pi_{25} \oplus \Pi_{26} & \end{array}\right]. \tag{3.11}$$

Matrices constructed by this method are constructed bigger than is needed. In Fig. 3.5 we have $\mathbf{H}$ for code with rate $r = 4/5$ and source length $k = 4096$. That configuration gives us code with length $n = 5120$ (from table 3.1), but matrix is 1536×5632. From that we will use only columns 1 through 5120 from matrix $\mathbf{H}$, it means that columns 5121 through 5632 are not transmitted. That corresponds that last M columns are unused ($5632 - 5120 = 512 = M$). In Fig. 3.6 is really used part of matrix $\mathbf{H}$.

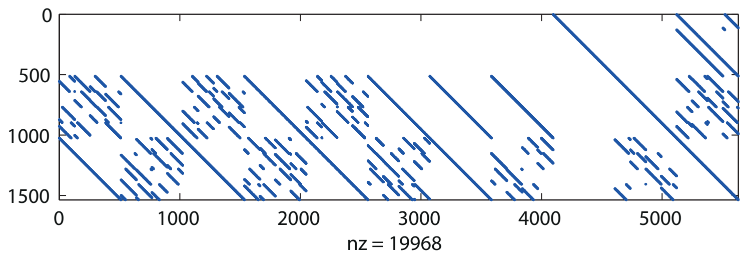

Fig. 3.5 – $\mathbf{H}$ from AR4JA LDPC code for $r = 4/5$, $k = 4096$

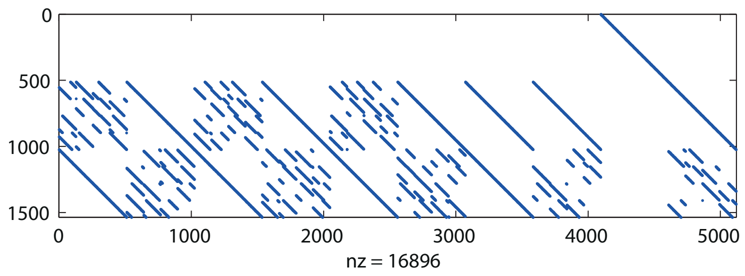

Fig. 3.6 – Used part of **H** from AR4JA LDPC code for $r = 4/5$, $k = 4096$

k	θ_k	$\phi_k(0, M)$ $M = 2^7 ... 2^{13}$	$\phi_k(1, M)$ $M = 2^7 ... 2^{13}$
1	3	1 59 16 160 108 226 1148	0 0 0 0 0 0 0
2	0	22 18 103 241 126 618 2032	27 32 53 182 375 767 1822
3	1	0 52 105 185 238 404 249	30 21 74 249 436 227 203
4	2	26 23 0 251 481 32 1807	28 36 45 65 350 247 882
5	2	0 11 50 209 96 912 485	7 30 47 70 260 284 1989
6	3	10 7 29 103 28 950 1044	1 29 0 141 84 370 957
7	0	5 22 115 90 59 534 717	8 44 59 237 318 482 1705
8	1	18 25 30 184 225 63 873	20 29 102 77 382 273 1083
9	0	3 27 92 248 323 971 364	26 39 25 55 169 886 1072
10	1	22 30 78 12 28 304 1926	24 14 3 12 213 634 354
11	2	3 43 70 111 386 409 1241	4 22 88 227 67 762 1942
12	0	8 14 66 66 305 708 1769	12 15 65 42 313 184 446
13	2	25 46 39 173 34 719 532	23 48 62 52 242 696 1456
14	3	25 62 84 42 510 176 768	15 55 68 243 188 413 1940
15	0	2 44 79 157 147 743 1138	15 39 91 179 1 854 1660
16	1	27 12 70 174 199 759 965	22 11 70 250 306 544 1661
17	2	7 38 29 104 347 674 141	31 1 115 247 397 864 587
18	0	7 47 32 144 391 958 1527	3 50 31 164 80 82 708
19	1	15 1 45 43 165 984 505	29 40 121 17 33 1009 1466
20	2	10 52 113 181 414 11 1312	21 62 45 31 7 437 433
21	0	4 61 86 250 97 413 1840	2 27 56 149 447 36 1345
22	1	19 10 1 202 158 925 709	5 38 54 105 336 562 867
23	2	7 55 42 68 86 687 1427	11 40 108 183 424 816 1551
24	1	9 7 118 177 168 752 989	26 15 14 153 134 452 2041
25	2	26 12 33 170 506 867 1925	9 11 30 177 152 290 1383
26	3	17 2 126 89 489 323 270	17 18 116 19 492 778 1790

Tab. 3.3 – Description of $\phi_k(0, M)$ and $\phi_k(1, M)$

For systematic encoding we will make generating matrix **G** from parity check matrix **H**. We will divide matrix **H** into two submatrix. To **P** which contains last $3M$ columns of **H** and into **Q**, which contains first KM columns, where for $r = 1/2$ is $K = 2$, for $r = 2/3$ is $K = 4$ and for $r = 4/5$ is $K = 8$. Exactly

$$\mathbf{H} = [\mathbf{Q}_{3M \times KM} | \, \mathbf{P}_{3M \times 3M}] \, . \tag{3.12}$$

k	θ_k	$\phi_k(2,M)$ $M=2^7...2^{13}$	$\phi_k(3,M)$ $M=2^7...2^{13}$
1	3	0 0 0 0 0 0 0	0 0 0 0 0 0 0
2	0	12 46 8 35 219 254 318	13 44 35 162 312 285 1189
3	1	30 45 119 167 16 790 494	19 51 97 7 503 554 458
4	2	18 27 89 214 263 642 1467	14 12 112 31 388 809 460
5	2	10 48 31 84 415 248 757	15 15 64 164 48 185 1039
6	3	16 37 122 206 403 899 1085	20 12 93 11 7 49 1000
7	0	13 41 1 122 184 328 1630	17 4 99 237 185 101 1265
8	1	9 13 69 67 279 518 64	4 7 94 125 328 82 1223
9	0	7 9 92 147 198 477 689	4 2 103 133 254 898 874
10	1	15 49 47 54 307 404 1300	11 30 91 99 202 627 1292
11	2	16 36 11 23 432 698 148	17 53 3 105 285 154 1491
12	0	18 10 31 93 240 160 777	20 23 6 17 11 65 631
13	2	4 11 19 20 454 497 1431	8 29 39 97 168 81 464
14	3	23 18 66 197 294 100 659	22 37 113 91 127 823 461
15	0	5 54 49 46 479 518 352	19 42 92 211 8 50 844
16	1	3 40 81 162 289 92 1177	15 48 119 128 437 413 392
17	2	29 27 96 101 373 464 836	5 4 74 82 475 462 922
18	0	11 35 38 76 104 592 1572	21 10 73 115 85 175 256
19	1	4 25 83 78 141 198 348	17 18 116 248 419 715 1986
20	2	8 46 42 253 270 856 1040	9 56 31 62 459 537 19
21	0	2 24 58 124 439 235 779	20 9 127 26 468 722 266
22	1	11 33 24 143 333 134 476	18 11 98 140 209 37 471
23	2	11 18 25 63 339 542 191	31 23 23 121 311 488 1166
24	1	3 37 92 41 14 545 1393	13 8 38 12 211 179 1300
25	2	15 35 38 214 277 777 1752	2 7 18 41 510 430 1033
26	3	13 21 120 70 412 483 1627	18 24 62 249 320 264 1606

Tab. 3.4 – Description of $\phi_k(2,M)$ and $\phi_k(3,M)$

After that we can compute

$$\mathbf{W} = (\mathbf{P}^{-1}\mathbf{Q})^T \tag{3.13}$$

and then we can create generating matrix by

$$\mathbf{G} = \begin{bmatrix}\mathbf{I}_{MK} & \mathbf{W}\end{bmatrix}, \tag{3.14}$$

where $\mathbf{I}_{MK}$ is $MK \times MK$ identity matrix and dense matrix $\mathbf{W}$.

Example: In figure 3.7 and 3.8 we can see difference between $\mathbf{H}$ and $\mathbf{G}$ for code with $r = 1/2$, $k = 4096$. There $\mathbf{H}$ is really sparse with 30 720 ones from all 62 914 560 positions (dimension of matrix is 6144×10240). In matrix $\mathbf{G}$ we have 12 679 168 ones from all 41 943 040 positions ($\mathbf{G}$ has dimension 4096×10240). In figures 3.9 and 3.10 are matrices without unused columns.

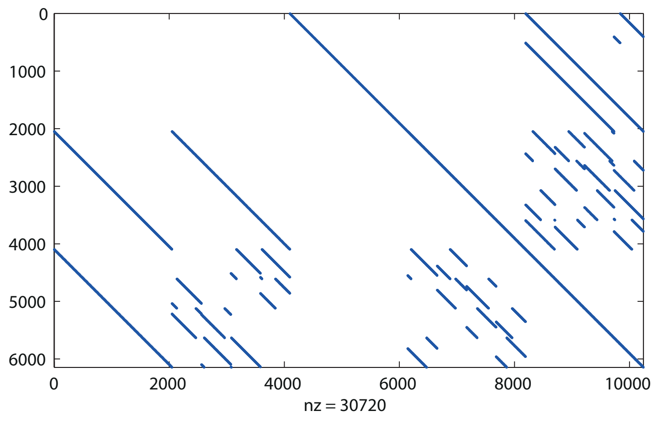

Fig. 3.7 – Example of **H** from AR4JA LDPC code for $r = 1/2$, $k = 4096$

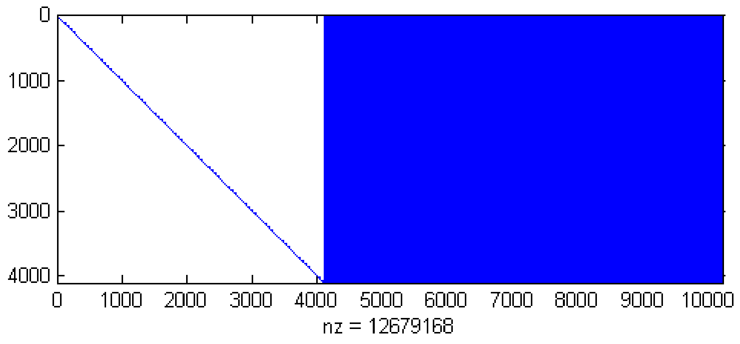

Fig. 3.8 – Example of **G** from AR4JA LDPC code for $r = 1/2$, $k = 4096$

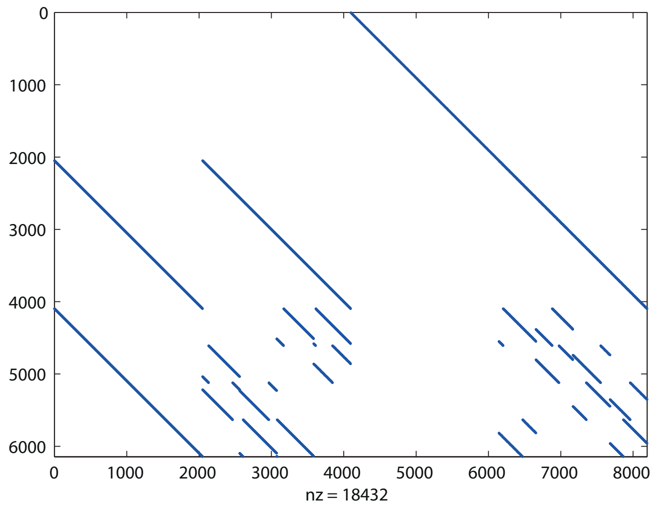

Fig. 3.9 – Example of used part of **H** from AR4JA LDPC code for $r = 1/2$, $k = 4096$

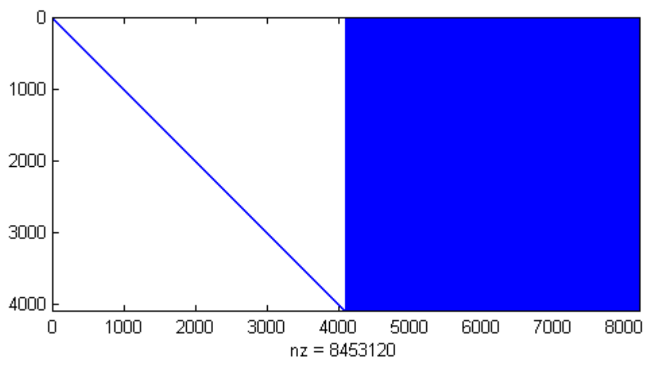

Fig. 3.10 – Example of used part of **G** from AR4JA LDPC code for $r = 1/2$, $k = 4096$

Used parts of matrices for code $r = 4/5$, $k = 1024$ are in figures 3.11 and 3.12 only for illustration. □

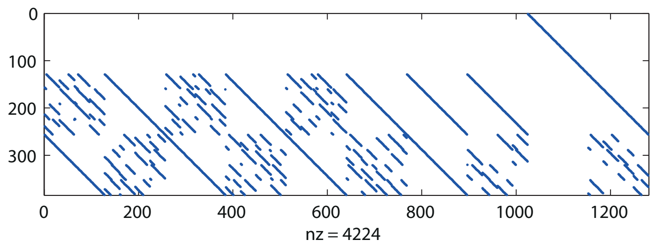

Fig. 3.11 – Example of used part of **H** from AR4JA LDPC code for $r = 4/5$, $k = 1024$

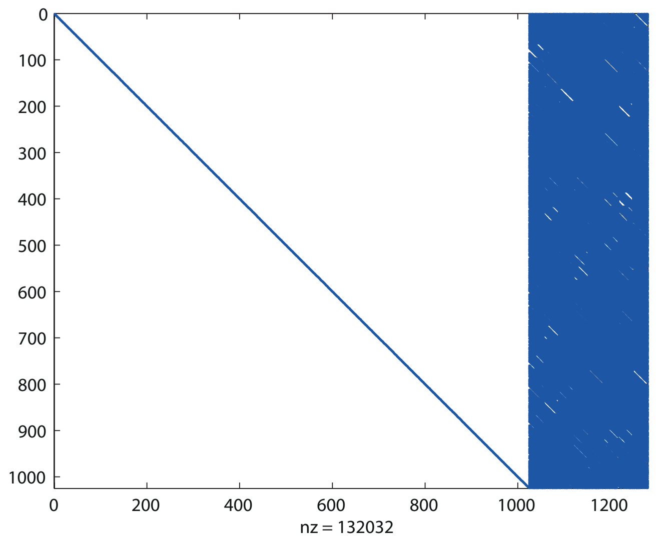

Fig. 3.12 – Example of used part of **G** from AR4JA LDPC code for $r = 4/5$, $k = 1024$

3.3 Joint channel in two-way relay system

We normally take one data input and code them and send them over channel. But Wübben shows in [14] that we can code two data inputs with one code and send it over channel. Just we do not use one BPSK for modulation, but two BPSK with different fading coefficients h_A, h_B.

Let us assume that we have two binary inputs b_A, b_B, both have k bits. We will code both with the same LDPC code, which give us code words c_A a c_B. Then we modulate code word c_A with BPSK and gain channel symbols x_A with fading coefficient h_A, same with c_B, which give us channel symbols x_B with fading coefficient h_B. Then we send simultaneously those x_A and x_B to relay and we get

$$y_R = h_A x_A + h_B x_B + n_R, \tag{3.15}$$

where n_R is noise (AWGN) on relay. This is called Multiple access (MAC) stage and it is illustrated in Fig. 3.13 a).

We received from channel observation x_R, where x_R corresponds modulated relay codeword c_R, which is XOR of c_A, c_B, so $c_R = c_{A \oplus B} = c_A \oplus c_B$. From that we can estimate for sources A, B

$$\begin{aligned} y_A &= h_A x_A + n_A & (3.16) \\ y_B &= h_B x_B + n_B, & (3.17) \end{aligned}$$

where n_A, n_B is noise. This Broadcast (BC) stage is shows in Fig. 3.13 b). Then we can estimate the information $b'_{R,A}$ and $b'_{R,B}$ from y_A, y_B. After knowing one source from Multiple access stage, we can gain $b'_B = b'_{R,A} \oplus b_A$, or $b'_A = b'_{R,B} \oplus b_B$, which is similar to $c'_A = c'_R \oplus c_B$ or $c'_B = c'_R \oplus c_A$.

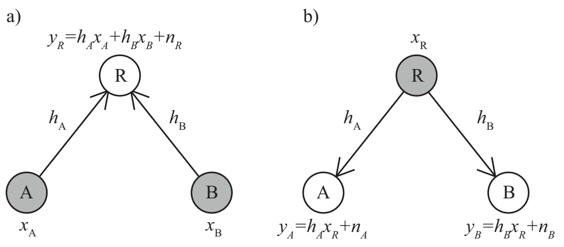

Fig. 3.13 – Connection of two sources with one relay

In Tab. 3.5 are relationships between codewords symbols c_A, c_B and their modulation and then modulation for relay x_R.

i	c_A	c_B	$c_{A\oplus B}$	c_{AB}	x_A	x_B	$x_R = x_{AB}$
0	0	0	0	0	1	1	$h_A + h_B$
1	1	0	1	1	-1	1	$-h_A + h_B$
2	0	1	1	D	1	-1	$h_A - h_B$
3	1	1	0	1+D	-1	-1	$-h_A - h_B$

Tab. 3.5 – Relationship between code words c_A,c_B and x_{AB}

Example: In figure 3.14 we show example of constellation space observation of sent symbol from relay x_R and observation of received symbol y_R. For this example we use $h_A = 1$ for source A and $h_B = \exp(j\frac{\pi}{2})$ for source B. We can see in figure that for $c_B = 0$ we have $x_B(0) = j$, so sent symbols are in upper half of graph. Same for $x_B(1) = -j$, where symbols are in lower part of graph. It is similar with x_A.

The MATLAB code for encode is in A.6.6.1 on page 69.

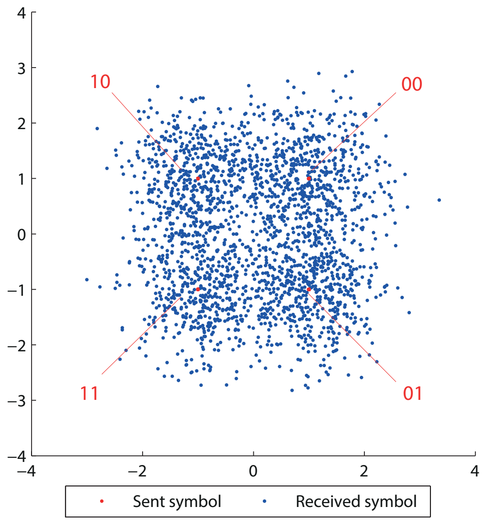

Fig. 3.14 – Constellation space observation from channel with SNR = 0.8 dB, with constellation point for $c_A c_B$

3.3.1 Separated channel decoding

As we can see in diagram in Fig. 3.15, the main point of Separated channel decoding (SCD) is to separate decoding parallely for only one code extra. Input to Sum-product algorithms are probabilities from likelihood function (1.20), exactly

$$P_i\left\{y_R|x_R(i)\right\} = \frac{1}{C}\Lambda(i) \tag{3.18}$$

for $i \in \{0,..,3\}$ from table 3.5 and C is constant, which guarantee $\sum_i P_i = 1$. Inputs to SPA for source A are

$$\begin{aligned}\Pr\left\{c_A = 0|y_R\right\} &= \Pr\left\{c_{AB} = 0|y_R\right\} + \Pr\left\{c_{AB} = D|y_R\right\} \\ &= P_0 + P_2 \qquad (3.19)\\ \Pr\left\{c_A = 1|y_R\right\} &= \Pr\left\{c_{AB} = 1|y_R\right\} + \Pr\left\{c_{AB} = 1 + D|y_R\right\} \\ &= P_1 + P_{3,} \qquad (3.20)\end{aligned}$$

which are based on Tab. 3.5, where $c_A = 0$ for $c_{AB} = 0$, $c_{AB} = D$. Also $c_A = 1$ for $c_{AB} = 1$ or $c_{AB} = 1 + D$. This we can reduce it to LLR, with

$$l_A = \ln\left(\frac{\Pr\left\{c_A = 0|y_R\right\}}{\Pr\left\{c_A = 1|y_R\right\}}\right) = \ln\left(\frac{P_0 + P_2}{P_1 + P_3}\right). \tag{3.21}$$

Similar it is with source B, where we have

$$\begin{aligned}\Pr\{c_B = 0|y_R\} &= \Pr\{c_{AB} = 0|y_R\} + \Pr\{c_{AB} = 1|y_R\} \\ &= P_0 + P_1 \quad (3.22)\\ \Pr\{c_B = 1|y_R\} &= \Pr\{c_{AB} = D|y_R\} + \Pr\{c_{AB} = 1 + D|y_R\} \\ &= P_2 + P_{3,} \quad (3.23)\end{aligned}$$

which give us

$$l_B = \ln\left(\frac{P_0 + P_1}{P_2 + P_3}\right). \quad (3.24)$$

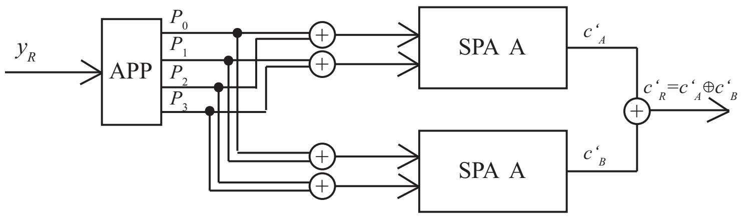

Fig. 3.15 – Block diagram of Separated channel decoding

Now we have all needed information for correction of received codewords. We will put l_A to Sum-product algorithm(Alg. 2.2) which give us corrected c'_A. Same with l_B which give us c'_B. After that, we gain codeword for relay $c'_R = c'_A \oplus c'_B$.

The MATLAB code for it is in A.6.6.2 on page 69.

3.3.2 Joint channel decoding

If we want only get c'_R, then we just want XOR from sources. We don't need execute two SPA and then from their output codewords count their XOR. We will count one SPA, which will directly give us c'_R. In Fig. 3.16 is scheme of Joint channel decoding (JCD). Inputs to SPA are

$$\begin{aligned}\Pr\{c_{A\oplus B} = 0|y_R\} &= \Pr\{c_{AB} = 0|y_R\} + \Pr\{c_{AB} = 1 + D|y_R\} \\ &= P_0 + P_3 \quad (3.25)\\ \Pr\{c_{A\oplus B} = 1|y_R\} &= \Pr\{c_{AB} = 1|y_R\} + \Pr\{c_{AB} = D|y_R\} \\ &= P_1 + P_{2,} \quad (3.26)\end{aligned}$$

where P_i are from (3.18). LLR of that we gain from

$$l_{A\oplus B} = \ln\left(\frac{P_0 + P_3}{P_1 + P_2}\right). \quad (3.27)$$

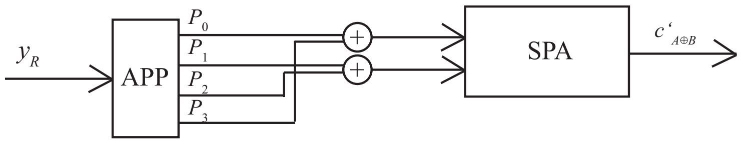

Fig. 3.16 – Block diagram of Joint channel decoding

That log-likelihood ratio is put to SPA, which give us $c'_{A \oplus B}$ which corresponds to $c'_R = c'_A \oplus c'_B$.

The MATLAB code for it is in A.6.6.3 on page 69.

Chapter 4

Analytic tools

In this chapter we would like to introduce two methods to analyze and compare codes and decoding methods. All used methods are functions of signal-to-noise ratio (SNR), which represent noise from channel, which is source of errors. With this analysis we want to show how channel can be bad for receiving correct information. In this thesis we used two types of analysis - analysis from simulation and analysis from matrix.

The analysis from simulation has really good results, because we get testing results from real decoding for channel model. On the other hand we need very much repeated simulation, ideally infinite repetition, because we use model with random noise and we do not get good statistic results from only one experiment. We implement bit error test (section 4.1), which is good a example.

The analysis from matrix makes test just from properties of matrix without simulation of real decoding. Our results are just only from structure of matrix and so this analysis can only make some bounds or determine area for deeper analysis. It is good, because it takes lower time than with simulations. In this thesis we implement EXIT chart (section 4.2) from [13].

4.1 Bit error rate

This is a well know analytic tool which is used for testing constellation, error-correction ability of codes or for everything, where we have bits which are affected by errors.

For our purpose we will use bit error rate (BER) for examine error-correction of LDPC codes for different signal-to-noise ratio. The main idea of analysis is easy - encode source data s, send them over AWGN channel with BPSK modulation, decoding them and then compare it with s and count how many errors left and divide it by all bits. This process will be repeated a few times to get good statistic data for different signal-to-noise ratio. We must repeat, because we send them over AWGN channel, which is model of random noise, so for one SNR we can have better or worse properties than are average for given SNR. We can choose a number of repetition or number of errors, which we want to reach for good results.

We can choose if we will making BER for all transmitted bits(with redundant/parity bits) or only for information bits. We choose for this thesis BER for information bits, so value of BER corresponds probability of error in one bit of decoded word. It is similar with the frame error rate (FER), which control correctest of full frame(transmitted word) than only bit. It depends if we need to receive all bits corrected or if we need almost corrected bits.

Algorithm 4.1 Bit error rate

```
INPUT: $encoder$, $decoder$, $rep$ - number of repetition, $SNR$ - range of examined SNR
Step 1: Set $i = 1$.
Step 2: Set $errors(i) = 0$, $counter = 1$.
Step 3: Make random source word $s$ with length $k$ and encode it with $encoder$ into codeword $c$. Then modulate it with BPSK into $x$.
Step 4: Send $x$ over AWGN channel with $SNR(i)$ and get $y = x + n$, where $n$ is noise.
Step 5: Demodulate $y$ and correct and decode it with $decoder$ into $s'$.
Step 6: Count $errors(i) = errors(i) + (s - s') \bmod 2$, represent remaining errors after decoding.
Step_7: If $counter \leq rep$, then set $counter = counter + 1$ and go to Step 3. Otherwise go to Step 8.
Step 8: Set $BER(i) = \frac{errors(i)}{k \cdot rep}$.
Step 9: If $SNR(i)$ isn't last, then set $i = i + 1$ and go to Step 2. Otherwise Stop.
```

Example: In figure 4.1 is BER for code for example from section 2.1.1.2, where we used RA LDPC code with rate 0.5 (with pseudo-random $\mathbf{H_s}$, where whole $\mathbf{H}$ has $m = 600$, $n = 1200$, $w_r = 5$, $w_{c1} = 3$ with $v_{w_{c1}} = 0.5$ and $w_{c2} = 2$ with $v_{w_{c2}} = 0.5$).

We compare three types of sum product algorithm decoders - LLR, probability domain and hard decision with received message from channel without error correction. All of them have maximum 30 iterations and we sent and decoded 250 messages. We sent it over AWGN channel with SNR from -5 dB to 6 dB, with step 1/3 dB.

For soft decision decoding with LLR and probability domain we have same results. LLR is only modified method of probability domain and we expected the same results. We can see that BER for decoded message with soft SPA is same with transmitted symbols without correction to SNR = -3.66 dB. After SNR = -3.66 dB soft SPA have rapidly better results than direct symbols from channel. After SNR = 1 dB soft SPA decoded messages without remaining errors. For SNR between -0.33 dB and 0.66 dB we have also BER = 0, but for SNR = 0.66 we have BER = $1.33 \cdot 10^{-5}$ and for SNR = 1 is BER = $6.66 \cdot 10^{-6}$, so we assume, that saver is use this code for SNR over 1 dB.

We also use SPA with hard decision, which shows that reduce soft information to hard can injure decoding. To SNR = 0.66 we have worse BER after SPA hard decision than before. After SNR = 0.66 dB we have slowly better BER after decoding than before decoding.

The MATLAB code of this example is in A.7.1 on page 70. □

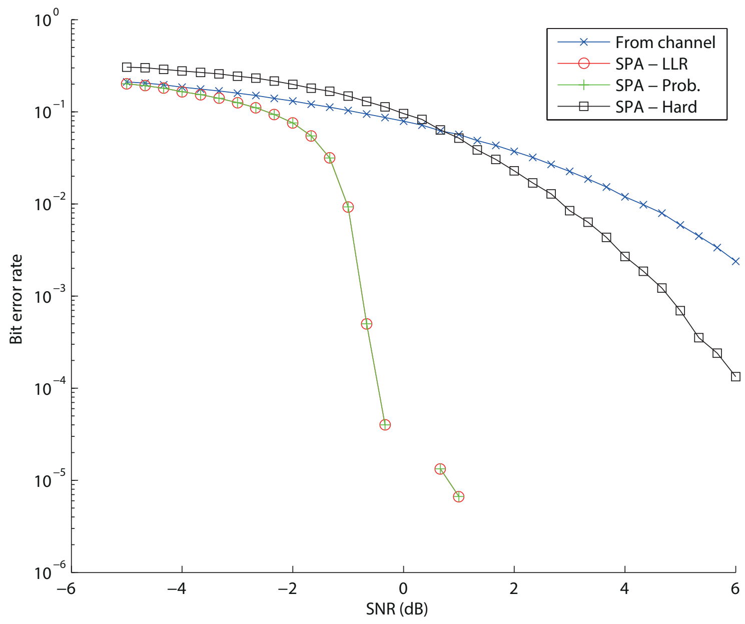

Fig. 4.1 – BER for code from example in section 2.1.1.2

4.2 Extrinsic information transfer chart

Another good tool to examine a code is making extrinsic information transfer (EXIT) chart. It is show how two decoders exchange their information. In our case of LDPC codes it exchanges between variable node decoder (VND) and check node decoder (CND). To be exact, into variable node decoder coming I_{ch} - average channel information and I_{av} - average a priori information, where $I_{av} = I_{ec}$ and from VND coming I_{ev} - average extrinsic information. Into check node decoder come in I_{ac} - average a priori information, where $I_{ac} = I_{ev}$ and from CND is output message I_{ec} - average extrinsic information. Figure 4.2 shows the scheme of decoders and exchange of information.

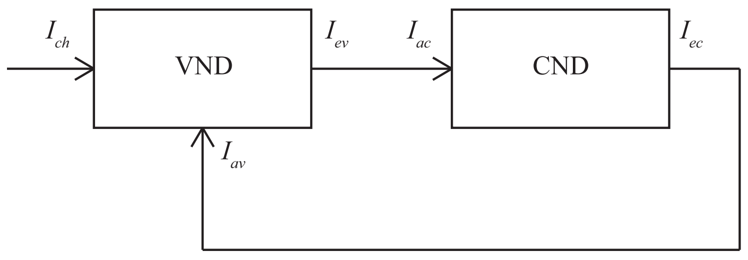

Fig. 4.2 – Scheme of two decoders of LDPC codes for EXIT chart

This thesis uses the method intended for design of LDPC codes by Stephan ten Brink [13]. Another method is described in his other work [12]. But we can use it also for analysis from given parity check matrix **H**. Just matrix **H** must have regular weight of rows and can have irregular weight of columns. Also we assume that code will be used for AWGN channel with soft decision with LLR.

EXIT chart is composed from curve of the Inner VND and curve from Outer CND and curve of transferring information between both decoders, which interpret whole decoder of code.

For curve of the variable node decoder we will come out from SPA LLR decoder in subsection 2.1.2 on page 19, exactly from (2.14), which we will rewrite like

$$L_{i,out} = L_{ch} + \sum_{j \neq i} L_{j,in}. \tag{4.1}$$

We used BPSK over AWGN channel with noise variance σ_n^2. That will give is modified (2.11)

$$L_{ch} = \ln \frac{p(y \mid x = +1)}{p(y \mid x = -1)} = \frac{2}{\sigma_n^2} y, \tag{4.2}$$

from which we can get variance of L_{ch} conditioned on x

$$\sigma_{ch}^2 = \frac{4}{\sigma_n^2}. \tag{4.3}$$

We get for compute EXIT function for regular LDPC with matrix **H** with weight of columns w_c

$$I_{ev}(I_{av}, w_c, SNR, \sigma_n^2) = J\left(\sqrt{(w_c - 1)[J^{-1}(I_{ac})]^2 + \sigma_{ch}^2}\right), \tag{4.4}$$

or for irregular-columns matrix **H** with size $m \times n$, edge degrees w_{ci}, $\lambda_{w_{ci}}$, where $i = 1, ...D$, we get

$$I_{ev}(I_{av}, SNR, \sigma_n^2) = \sum_{i=1}^{D} b_i \cdot I_{ev}(I_{ac}, w_{ci}, SNR, \sigma_n^2), \tag{4.5}$$

where

$$b_i = \frac{n\lambda_i w_{ci}}{n \sum_{j=1}^{D} \lambda_j w_{cj}} \tag{4.6}$$

and $\sum_{i=1}^{D} b_i = 1$. And J and J^{-1} are approximation described by

$$J(\sigma) \approx \begin{cases} a_{J,1}\sigma^3 + b_{J,1}\sigma^2 + c_{J,1}\sigma, & 0 \leq \sigma \leq 1.6363 \\ 1 - e^{a_{J,2}\sigma^3 + b_{J,2}\sigma^2 + c_{J,2}\sigma + d_{J,2}}, & 1.6363 < \sigma < 10 \\ 1, & 10 \leq \sigma \end{cases} \tag{4.7}$$

where

$$\begin{aligned} &a_{J,1} = -0.0421061, \quad b_{J,1} = 0.209252, \quad c_{J,1} = -0.00640081 \\ &a_{J,2} = 0.00181491, \quad b_{J,2} = -0.142675, \quad c_{J,2} = -0.0822054, \quad d_{J,2} = 0.0549608. \end{aligned}$$

For inverse function J^{-1} we get

$$J^{-1}(I) \approx \begin{cases} a_{\sigma,1}I^2 + b_{\sigma,1}I + c_{\sigma,1}\sqrt{I}, & 0 \le I \le 0.3646 \\ -a_{\sigma.2}\ln[b_{\sigma,2}(1-I)] - c_{\sigma,2}I, & 0.3646 < I < 1 \end{cases} \tag{4.8}$$

where

$$a_{\sigma,1} = 1.09542, \quad b_{\sigma,1} = 0.214217, \quad c_{\sigma,1} = 2.33727$$
$$a_{\sigma,2} = 0.706692, \quad b_{\sigma,2} = 0.386013, \quad c_{\sigma,2} = -1.75017.$$

Then we make curve of VND only with putting $I_{av} \in \langle 0,1 \rangle$ into (4.4) or (4.5).

For EXIT curve of check node decoder it is similar. Again we come from SPA LLR, exactly from (2.13)

$$L_{i,out} = \ln \frac{1 - \prod\limits_{j \neq i} \frac{1-e^{L_{j,in}}}{1+e^{L_{j,in}}}}{1 + \prod\limits_{j \neq i} \frac{1-e^{L_{j,in}}}{1+e^{L_{j,in}}}}. \tag{4.9}$$

After adjustments and approximations we get

$$I_{ec}(I_{ac}, w_c) \approx 1 - J\left(\sqrt{w_c - 1} \cdot J^{-1}(1 - I_A)\right). \tag{4.10}$$

The curve of CND we again gain after inputting $I_{ac} \in \langle 0,1 \rangle$. CND is independent on SNR of channel.

For curve of decoder we start with $I_{av} = 0$ and put it in (4.4) for regular or in (4.5) for irregular matrix. We get from it $I_{ev} = I_{ac}$ and put it into (4.10) which give us $I_{ec} = I_{av}$ and so on.

Decoder path ends in place where curves of CND and VND cross. This means that decoder did not reach right codeword. The number of stairs of decoder path can respond to needed number of iterations of decoder.

The MATLAB code for EXIT chart is in A.7.2 on page 71.

Example: In figure 4.3 we have EXIT chart for code from example from section 2.1.1.2, where we used RA LDPC code with rate 0.5 (with pseudo-random $\mathbf{H_s}$, where whole $\mathbf{H}$ has $m = 600$, $n = 1200$, $w_r = 5$, $w_{c1} = 3$ with $\lambda_{w_{c1}} = 0.601$ and $w_{c2} = 2$ with $\lambda_{w_{c2}} = 0.399$).

We analyze SPA with LLR for SNR=[-5,-3,-1,1,3,5]. From the chart we can see that below SNR = 1 dB we cannot reach right codewords and over it we can have good decoding. This corresponds with results from BER analysis in previous part. Just this analysis took few seconds and BER took few hours. □

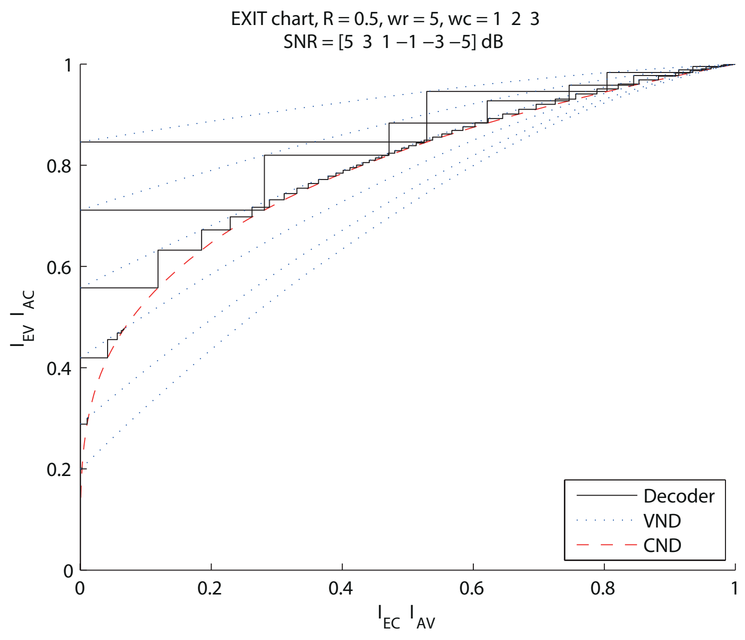

Fig. 4.3 – EXIT chart for code from example in section 2.1.1.2

4.2.1 Design tool

This method was intended for determine performance of code before parity-check matrix **H** construction. Out implementation for analyzing can be used for it. We only directly give values of length of code n, weight of row w_r, which must be regular and weight of columns w_{ci} and their degree distribution $v_{w_{ci}}$. We get edge distribution for computation from degree distribution with

$$\lambda_i = \frac{w_{ci} v_{w_{ci}} n}{m w_r}. \tag{4.11}$$

Rest of it is same like in previous part.

The MATLAB code for design with EXIT chart is in A.7.2.1 on page 73.

Chapter 5

Comparison of codes

We want to show that matrices with almost same properties but with different construction, size or degree distribution can affect better or worse performance. Also we want to show the comparison between all three types of SPA.

5.1 $r = 1/2$, $n = 1200$

We compare the codes with rate $r = 1/2$ and codeblock length $n = 1200$, where we have length of source data bits $k = 600$ and length of parity bits $m = 600$.

We tested RA LDPC codes, where we have different construction of $\mathbf{H_s}$, but same rate r and codeblock length n. In figure 5.1 we have comparison of three codes. Code in column a) has regular $\mathbf{H_s}$made by pseudo-random construction with $w_c = 3$, $w_r = 3$. This code was used in Examples in subsection 2.1.1.2 and 4.1. Next code in column b) has irregular-column $\mathbf{H_s}$ constructed also with pseudo-random construction, but with degree distribution of columns $v_2 = 0.2$, $v_3 = 0.6$ and $v_4 = 0.2$. Last code in column c) have quasi-cyclic constructed matrix $\mathbf{H_s}$ with $w_c = 3$, $w_r = 3$ and $p = 200$.

We made EXIT charts of them and BER test, with 250 repetition and decoders have 30 iterations for SNR from -5 dB to 6 dB, with step 1/3 dB. From EXIT charts we can see that all codes have here almost same results, only irregular code have little better performance.

Last part of figure has BER tests. For pseudo-random constructions we have almost same results. Regular code has BER $= 0$ for SNR > 1 dB and irregular for SNR > 2 dB. But both have rapid decrease of BER for SNR larger -1,66 dB. With QC code we have BER $= 0$ after SNR $= 3.66$ dB. Interesting is that the BER curve decrease here slowly and not so fast, like with pseudo-random matrices.

It is show, that in this case, pseudo-random codes have better results.

5.2 $r = 0.53$, $w_c = 2, 3, 4$

We made this example to show how dimension of matrix $\mathbf{H}$ (with length of codeword) improve performance of code.

We made two parity check matrices $\mathbf{H}$. Both are for RA LDPC and have rate of code $r = 0.53$ and their $\mathbf{H_s}$have $w_r = 4$, $w_c \in \{3, 4\}$ with almost same

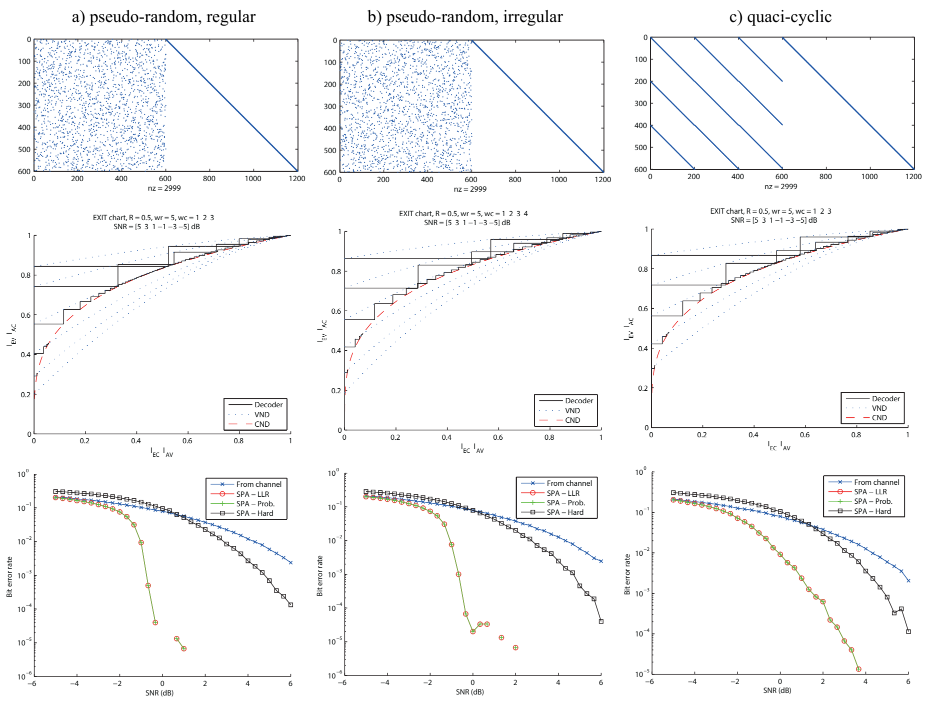

Fig. 5.1 – Comparison of codes with $r = 1/2$, $n = 1200$

degree distribution. For code with length of data bits $k = 500$ we have degree distribution $v_3 = 0.52$ and $v_4 = 0.48$. For code with $k = 5000$ we have $v_3 = 0.5$ and $v_4 = 0.5$.

In figure 5.2 we have again analysis of both codes. Illustration of matrices, EXIT charts and BER tests for AWGN channel with SNR from -5 dB to 6 dB, with step 1/3 dB. We analyze SPA decoders - with LLR, in probability domain and with hard decision. Every decoder have maximal iteration equal to 30.

In section a) we have analysis for code with length of data bits $k = 500$ with parity-check matrix with dimension $m = 435$ and $n = 935$. From EXIT chart we know that code can be decoded without errors for AWGN channel with SNR above 1 dB. But from BER test, with 250 repetition, we know that code works great with soft decisions SPA decoders for AWGN with SNR > 0.33 dB. We have similar results for hard decision SPA like in the previous example. It has worse results than data directly from channel for SNR < 1 dB. After SNR $= 1$ dB hard decision has better results.

Column b) shows results for matrix $\mathbf{H}$ with $k = 5000$. Matrix has dimension $n = 9375$, $m = 4375$. EXIT chart of this code is almost similar like with the previous code. We can see that it should work with AWGN channel with SNR > 1 dB. But from BER test, with 125 repetition, we can see that soft decision SPA can correct words without error for SNR $>$ - 0.66 dB. Hard decision SPA have similar properties like with the second code.

Both BER test proved that code with long codeblock have better performance than code with similar properties, but with shorter length. Soft SPA decoder can decode codeword from code with $n = 935$ without error for AWGN channel with SNR $>$ 0.33 dB, but from code with $n = 9375$ it works for SNR $>$ - 0.66 dB. That is different 1 dB for ten times larger n.

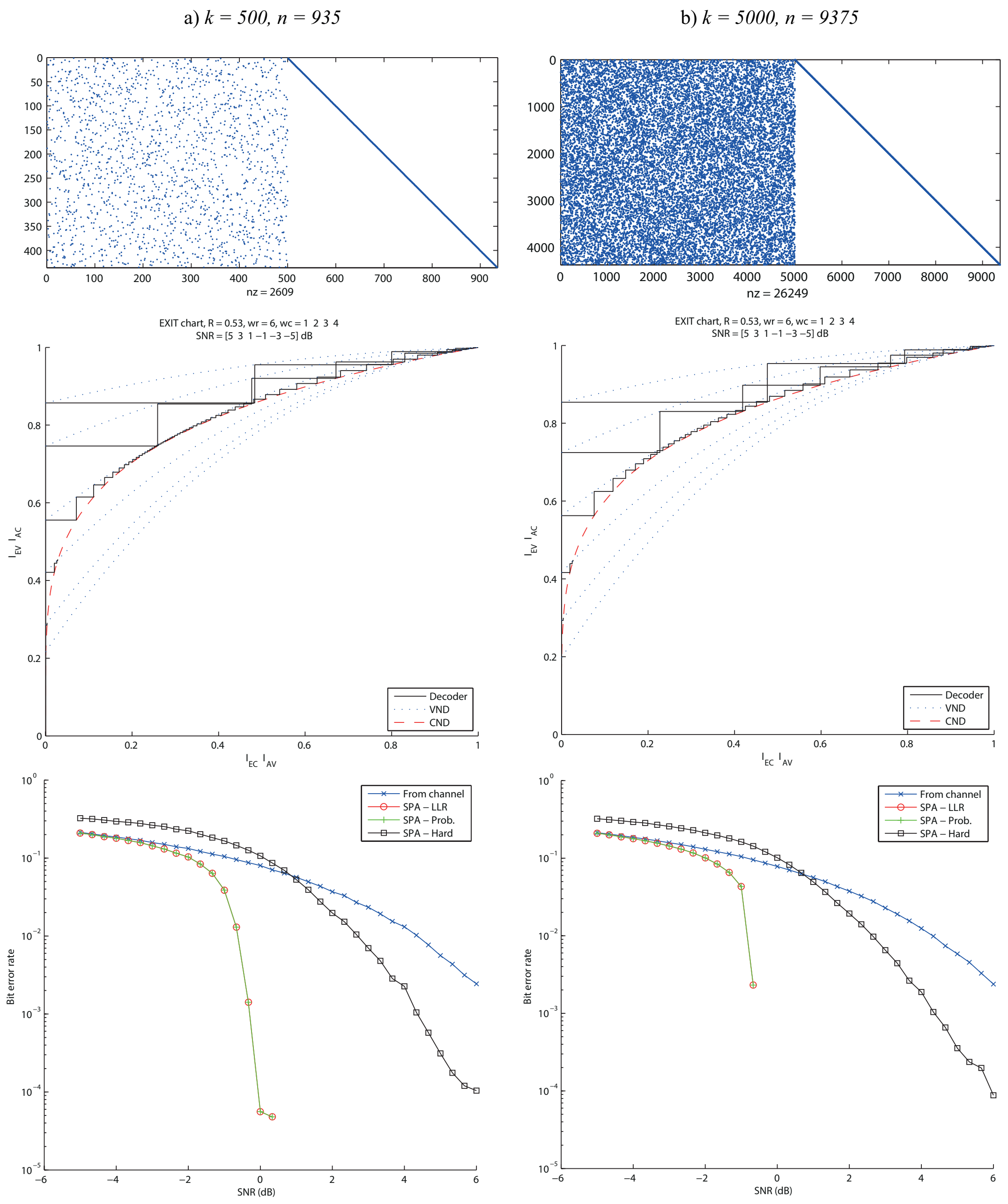

Fig. 5.2 – Comparison of codes with $r = 0.53$, $w_c = 2, 3, 4$

Chapter 6

Conclusion

In the previous chapters we wanted to show what we learned about LDPC codes. A large part of process of writing a thesis consists of making research and reading books and papers. It was not easy to find good sources which examine what we need to find. It is quite a popular topic these days, so there are plenty of works dealing with it, but many of them are either useless for our purposes or over our bachelor's knowledge.

We implemented many tools in MATLAB codes. Especially a big set of decoding tools with three variety of Sum-product algorithm. Then we implemented a good and quite fast method for making pseudo-random parity-check matrices **H** without 4-cycles. For better encoding we implemented repeat-accumulate code, which expands LDPC codes. This method is also used with DVB-S2 LDPC codes. We also implemented two methods for algebraic construction of matrix **H**. At last there are codes for BER test and for EXIT charts, which will be used for determine qualities of codes. All codes are made by us and none of them is copied.

We hope that this thesis will be used as an introduction into LDPC codes and our MATLAB's codes will help with future works with digital communication.

Appendix A

MATLAB source codes

A.1 Making factor graph from parity check matrix

```
function LDPC_draw_FG(H)

%{
  Draw FG from parity check matrix H
%}

[m,n]=size(H);
VN=[-(n-1):2:(n-1)]; % variable node
CN=[-(m-1):2:(m-1)]; % check node
figure
axis([-n n -6 6])
hold on;
plot(VN,5.*ones(n,1),'Marker','o','MarkerEdgeColor','red','MarkerFaceColor',
  'none','MarkerSize',12,'LineStyle','none','LineWidth',1.2);
plot(CN,-5.*ones(m,1),'Marker','s','MarkerEdgeColor','blue','MarkerFaceColor',
  'none','MarkerSize',12,'LineStyle','none','LineWidth',1.2);
set(gca,'XTick',[],'YTick',[]); % remove numbers
set(gca, 'Units', 'normalized', 'Position', [0,0,1,1]);
[ED(:,1),ED(:,2)]=find(H); % edges
for i=1:size(ED,1) % draw edges
  plot([VN(ED(i,2)) CN(ED(i,1))],[5 -5],'black', 'LineWidth',.1);
end
legend('VN','CN', 'Edges', 'Location','SouthEast');
```

A.2 Finding 4-cycles of factor graph

```
function [free4]=LDPC_4cycle(H)
```

```
%{
  Finding 4 cycles of factor graph from parity-check matrix H
  free4 - 1 - without 4-cycles
              0 - with 4-cycle
%}

[m,n]=size(H);
free4=1; %index for controlling
for i=1:(m-1) % Control row by row
  len_i=sum(H(i,:)); % number of 1's in i row
  for j=(i+1):m
    len_j=sum(H(j,:)); % number of 1's in i row
    len_ij=len_i+len_j;
  len_ij_sum=sum(mod(H(i,:)+H(j,:),2)); % number of 1's after i+j row
    if ~((len_ij-len_ij_sum)/2==0 || (len_ij-len_ij_sum)/2==1)
    % there is /2 because if there are same element,
        then it will missing in both rows
      disp(['Factor graph has 4-cycle between ', num2str(i),'-th and '
        ,num2str(j),'-th row.']);
      free4=0;
    end
  end
end
if free4==1
  disp('Factor graph is without 4-cycles');
end
```

A.3 Degree, edge distribution of H

```
function [R, wcd, wrd, wce, wre] = LDPC_H_analysis(H)

[m,n]=size(H);
R=(n-m)/n; % Rate
wc=sum(H,1); % weigth of every column
wr=sum(H,2)'; % weigth of every row
wcd=unique(wc); % edges from VN
wrd=unique(wr); % edges from CN
wcd(2,:)=histc(wc,wcd(1,:))./length(wc); % Degree distribution of VN
wrd(2,:)=histc(wr,wrd(1,:))./length(wr); % Degree distribution of CN
edg=sum(sum(H));
wce=unique(wc); % edges from VN
wre=unique(wr); % edges from CN
wce(2,:)=wce(1,:).*histc(wc,wce(1,:))./edg; % Degree of edge distrib. of VN
wre(2,:)=wre(1,:).*histc(wr,wre(1,:))./edg; % Degree of edge distrib. of CN
```

A.4 BPSK and AWGN

```
function [y]=bpsk(x,h)

%{
  BPSK (0 -> 1, 1 -> -1)
  y - output
  x - input
  h - fading coefficients
%}

y=h.*ones(1,length(x));
y(x==1)=-h;
```

```
function [y,sigma2]=LDPC_AWGN(code, M, SNRdB, No)

%{
  AWGN channel model
  y - word with noise
  sigma2 - Standard deviation^2
  code - input word
  M - number of constellation points
  SNRdB - SNR in dB
  No - noise power spectral density
%}

if isempty(No)
  Es=1;
  Eb=Es/log2(M);
  SNR=10.^(SNRdB/10);
  No=Eb/SNR; %standard deviation
end
N=sqrt(No/2)*(randn(1,length(code))+1i*randn(1,length(code)));
y=code+N;
sigma2=var(N);
```

```
function [p]=prob_awgn(x,S,sigma2)

%{
  Give likelihood function (probability) for AWGN channel
  p - output probability
  x - bpsk output
  S - constellation symbols for [0, 1, 2, 3, ...]
  sigma2 - standart deviation of noise
%}

for i=1:length(S)
  p(i,:)=exp(-((abs(x-S(i)).^2)/2/sigma2));
end
sump=repmat(1./sum(p),length(S) ,1);
p=p.*sump;
```

A.5 Error-correction decoding

A.5.1 Sum-Product Algorithm

A.5.1.1 Soft decision - Probability

```
function [c, p, counter]=SPA_soft(x, H, iter)

%{
  Soft Decision of sum product algorithm
  c - decoded word (binary)
  p - probability
  counter - number of iteration
  x - coded word (probability)
  H - parity check matrix
  iter - maximal iteration
%}

[m,n]=size(H);
counter=1;
q(:,:,1)=H.*repmat(x(1,:),m,1); % From VN to CN
q(:,:,2)=H.*repmat(x(2,:),m,1);
r=zeros(m,n,2); p=zeros(2,n);
while counter<=iter

  % From VN to CN %
  for j=1:m
    ind=find(H(j,:));
    k=0;
    for i=ind
      k=k+1;
      r(j,i,1)=1/2+(1/2)*prod(1-2.*q(j,ind([1:(k-1) (k+1):length(ind)]),2));
      r(j,i,2)=1-r(j,i,1);
    end
  end

  % From CN to VN %
  for j=1:n
    ind=find(H(:,j));
    k=0;
    for i=ind'
      k=k+1;
      q(i,j,1)=x(1,j)*prod(r(ind([1:(k-1) (k+1):length(ind)]) , j , 1));
      q(i,j,2)=x(2,j)*prod(r(ind([1:(k-1) (k+1):length(ind)]) , j , 2));
      sum=q(i,j,1)+q(i,j,2);
      q(i,j,1)=q(i,j,1)/sum;
      q(i,j,2)=q(i,j,2)/sum;
    end
  end

  % Soft decision of VN %
  for j=1:n
    ind=find(H(:,j));
    for i=ind'
      p(1,j)=x(1,j)*prod(r( ind, j , 1));
      p(2,j)=x(2,j)*prod(r( ind, j , 2));
      sum=p(1,j)+p(2,j);
      p(1,j)=p(1,j)/sum;
      p(2,j)=p(2,j)/sum;
    end
  end
  c=round(p(2,:)); % binary code of VN
  if mod(H*c',2)==0
    break;
  end

  if counter==iter
    disp('Reach maximal iteration without codeword');
  else
    counter=counter+1;
  end
end
```

A.5.1.2 Soft decision - Log-likelihood ratio

```
function [c,l,counter]=SPA_llr(x, H, iter, ending)

%{
  Soft Decision of sum product algorithm from LLR (log likelyhood ratio)
  c - decoded word (binary)
  l - LLR
  counter - number of iteration
  x - coded word LLR)
  H - parity check matrix
  iter - maximal iteration
  ending - what is goal of SPA
    - infin - end after reach LLR=+-infinity
        - code - end after reach codeword
%}

[m,n]=size(H);
q=H.*repmat(x,m,1);
counter=1;
r=zeros(m,n);
l=zeros(1,n);
while counter<=iter

  % From VN to CN %
  for j=1:m
    ind=find(H(j,:));
    k=0;
    for i=ind
      k=k+1;
      r(j,i)=log((1+prod(tanh(0.5.*q(j,ind([1:(k-1)
        (k+1):length(ind)]))))))./(1-prod(tanh(0.5.*
      q(j,ind([1:(k-1) (k+1):length(ind)]))))));
    end
  end

  % From CN to VN %
  for j=1:n
    ind=find(H(:,j));
    k=0;
    for i=ind'
      k=k+1;
      q(i,j)=x(j)+sum(r(ind([1:(k-1) (k+1):length(ind)]) ,j));
    end
  end

  % Soft decision of VN %
  for j=1:n
    ind=find(H(:,j));
    for i=ind'
      l(j)=x(j)+sum(r(ind,j));
    end
  end

  % End of algorithm %
  switch ending
    case 'infin'
    %%% End with infinity LLR %%%
    if sum(isinf(abs(l)))==n % if all element of l is infinity
      c=zeros(1,n);
      c(l==Inf)=0;
      % binary code of VN
      c(l==-Inf)=1;
      if mod(H*c',2)==0
        break;
      end
    end
    if counter==iter
      c=ones(1,n);
      c(l>0)=0;
      if mod(H*c',2)==0
        disp('Codeword without infinity LLR');
        break;
      else
        disp('Reach maximal iteration');
        break;
      end
    else
      counter=counter+1;
    end

    case 'code'
    %%% End with codeword from codebook %%%
    c=ones(1,n);
    c(l>0)=0;
    if mod(H*c',2)==0
```

```
      break;
    end
    if counter==iter
      disp('Reach maximal iteration');
      break;
    else
      counter=counter+1;
    end
  end
end
```

A.5.1.3 Hard decision

```
function [c, counter]=SPA_hard(x, H, iter)

%{
  Hard Decision sum-product algorithm
  c - decoded word (binary)
  counter - number of iteration
  x - coded word (binary)
  H - parity check matrix
  iter - maximal iteration
%}

c=x;
counter=0;
if sum(mod(H*c',2))~=0 % if there is error
  [m,n]=size(H);
  Cin_or=H; % original matrix H
  Cin_or(Cin_or==0)=NaN; % NaN is ignored by mode
  while counter<iter
    Cin=Cin_or;
    CN=mod(H*c',2);
    for i=1:m
      if CN(i)==0
        Cin(i,Cin(i,:)==1)=c(find(H(i,:)));
      else
        Cin(i,Cin(i,:)==1)=~c(find(H(i,:)));
      end
    end
    c=mode([x;Cin]);
    if mod(H*c',2)==0
      break;
    end
    counter=counter+1;
    if counter==iter
      disp('Reach maximal iteration');
    end
  end
end
```

A.5.2 Bit-flip decoding

```
function [c,counter]=BITFLIP(x, H, iter)

%{
  Bit flipping algorithm
  c - decoded word / coded word
  counter - end step
  H - parity check matrix
  iter - maximal iteration
%}

c=x;
counter=0;
while counter < iter
  CN=mod(c*H',2);
  if CN==0 % code from code book
    break
  end
  M=H; % matrix with bad CN
  for j=1:length(CN)
    M(j,:)=CN(j).*M(j,:);
  end
  VNerr=sum(M); % sum of bad CN
  flip=find(VNerr==max(VNerr));  c(flip)=mod(c(flip)+1,2);
  counter=counter+1;
  if counter==iter
    disp('Reach maximal itteration');
  end
end
```

A.5.3 Decoding framework

This function cover all decoding methods and make it easier to toggle between them.

```
function [c,counter,soft]=LDPC_er_cor(H, x, iter, msg_type, decod_type, sigma2)

%{
  Correction of errors from received message with factor graph from H
  c - corrected word
  counter - counter of iteration
  soft - value of soft decision
  H - parity-check matrix
  x - received word
  iter - maximal iteration
  msg_type
    - type of received word
    - prob - probability of 0, 1
    - llr - log likelyhood ratio
    - bin - binary msg, hard decision
    - bpsk - raw data from awgn channel with bpsk modulation
  decod_type - type of decoding
    - hspa - hard decision SPA
    - sspa - soft decision SPA
    - llrspa_inf - LLR SPA, ended with infinity LLR or reach max iteration
    - llrspa_code - LLR SPA, ended after finding code word
    - bf - bit flopping
  sigma2 - standart deviation of AWGN
%}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[m,n]=size(H); % m - number of CN, n - number of VN

switch msg_type
case 'prob'
%%%%%%%%%%%%%%%%%%%%%%%%%%% Probability %%%%%%%%%
  switch decod_type
  case 'bf'
    %%%%%%%%% Bit flopping %%%%%%%%%
    [c,counter]=BITFLIP(round(x(2,:)),H, iter);

  case 'hspa'
    %%%%%%%%% Hard Decision SPA %%%%%%%%%
    [c,counter]=SPA_hard(round(x(2,:)),H, iter);

  case 'sspa'
    %%%%%%%%% Soft SPA %%%%%%%%%
    [c, soft, counter]=SPA_soft(x, H, iter);

  case 'llrspa_inf'
    %%%%%%%%% Log likelyhood ration SPA, end inf %%%%%%%%%
    l=log(x(1,:)./x(2,:));
    [c, soft, counter]=SPA_llr(l, H, iter, 'infin');

  case 'llrspa_code'
    %%%%%%%%% Log likelyhood ration SPA, end code %%%%%%%%%
    l=log(x(1,:)./x(2,:));
    [c, soft, counter]=SPA_llr(l, H, iter, 'code');
  end
case 'llr'
%%%%%%%%%%%%%%%%%%%%%%%%%%% Log Likely Hood message %%%%%%%%%
  switch decod_type
  case 'llrspa_inf'
    %%%%%%%%% SPA for LLR, end inf %%%%%%%%%
    [c, soft, counter]=SPA_llr(x, H, iter, 'infin');

  case 'llrspa_code'
    %%%%%%%%% SPA for LLR, end code %%%%%%%%%
    [c, soft, counter]=SPA_llr(x, H, iter, 'code');

  case 'hspa'
    %%%%%%%%% Hard Decision SPA %%%%%%%%%
    bin=ones(1,n);
    bin(x>0)=0;
    [c,counter]=SPA_hard(bin, H, iter);
  end
case 'bin'
%%%%%%%%%%%%%%%%%%%%%%%%% Binnary message %%%%%%%%%
  switch decod_type
  case 'bf'
    %%%%%%%%% Bit flopping %%%%%%%%%
    [c,counter]=BITFLIP(x, H, iter);

  case 'hspa'
    %%%%%%%%% Hard Decision SPA %%%%%%%%%
    [c,counter]=SPA_hard(x, H, iter);
  end
case 'bpsk'
%%%%%%%%%%%%%%%%%%%%%%%%% BPSK message %%%%%%%%%
  % BPSK: 0-> 1, 1 -> -1
  p(1,:)=exp(-((abs(1-x).^2)/2/sigma2));
  p(2,:)=exp(-((abs(-1-x).^2)/2/sigma2));
  switch decod_type
  case 'bf'
    %%%%%%%%% Bit flopping %%%%%%%%%
    sump=p(1,:)+p(2,:);
    x=round(p(2,:)./sump);
    [c,counter]=BITFLIP(x,H, iter);

  case 'hspa'
    %%%%%%%%% Hard Decision SPA %%%%%%%%%
    sump=p(1,:)+p(2,:);
    x=round(p(2,:)./sump);
    [c,counter]=SPA_hard(x,H, iter);

  case 'sspa'
    %%%%%%%%% Soft SPA %%%%%%%%%
    sump=p(1,:)+p(2,:);
    p(1,:)=p(1,:)./sump;
    p(2,:)=p(2,:)./sump;
    [c, soft, counter]=SPA_soft(p, H, iter);
  case 'llrspa_inf'
    %%%%%%%%% Log likelyhood ration SPA, end inf %%%%%%%%%
    l=log(p(1,:)./p(2,:));
    [c, soft, counter]=SPA_llr(l, H, iter, 'infin');

  case 'llrspa_code'
    %%%%%%%%% Log likelyhood ration SPA, end code %%%%%%%%%
    l=log(p(1,:)./p(2,:));
    [c, soft, counter]=SPA_llr(l, H, iter, 'code');
  end
end
```

A.6 Parity-check matrix construction

A.6.1 Pseudo-random construction

```
function [FH]=LDPC_gen_ser_H(n,wc,wr)

%{
  Function to make LDCP control matrix regular or irregular column H with
       regular
    wr (only position of 1 in every row) by make series and then mix it
  wc - ones in column
  wc[2 ; :] - wc(1,:) - weigth of ones, wc(2,:) - degree distribution
  wr - ones in row
  m - number of rows
  n - number of columns
%}

if 1==size(wc,1) && 1==size(wc,2) % for regular H, with one element wc
  m=n*wc/wr;
  FH = repmat(1:n,wc,1); % Make k-times indexes from 1 to n and then
      reshape it to right size
else %% for irregular H
  wc=(sortrows(wc'))';
  m=n*sum(wc(1,:).*wc(2,:))/wr;
  if ~( m == floor(m))
    m
    error('Number of m (CN) is not integer')
  end
  wcn=wc; % wcn - count of column with wc(1,:)
  for i=1:length(wc(1,:))-1
    wcn(2,i)=round(wc(2,i)*n);
  end
  wcn(2,end)=n-sum(wcn(2,1:(end-1)));
  col=randperm(n);
  FH=[];
  j=1;
  for i=1:length(wcn(1,:))
    FH = [FH repmat(col(j:j+wcn(2,i)-1),1,wcn(1,i))];
    % Make k-times indexes from 1 to n and then reshape it to right size
    j=j+wcn(2,i);
  end
end
FH = reshape(FH,m,wr);
a=1;
while a<=m % for rows
  go=1;
  cycle_help=0; % Help to endless cycles
  while go==1
    cycle_help=cycle_help+0.2;
    vec = randperm(size(FH(a:m,:),1)*size(FH(a:m,:),2)); % Vector for
        permutatiton
    i2=1;
    FH2=zeros(m-a+1, wr);
    FH=FH'; % transposition for right reading by FH(1), FH(2), ...
    for i=(1+(a-1)*wr):(m*wr)
      FH2(vec(i2))=FH(i); % This shake elements
      i2=i2+1;
    end
    FH=FH';
    FH(a:m,:) = FH2; % This give shaked matrix to target matrix
    go_if=1;
    if length(unique(FH(a,:)))==length(FH(a,:))
    % Control that there is only unique indexes in every row
      if a==1
        go=0;
      end
      for i=1:(a-1) % Eliminate 4 cycles
        len=length([FH(i,:) FH(a,:)]);
        len_un=length(unique([FH(i,:) FH(a,:)]));
        if ~(len_un<=len && len_un>=(len-1))
        % Can be only one same index in every row
          go_if=0;
          break;
        end
      end
      if go_if==1
        go=0;
      end
    end
    if cycle_help>=ceil(0.1*m)
    % If there is loop, this will return to previous rows and do it again
      a=a-1;
      cycle_help=0;
    end
  end
  a=a+1;
end
FH=sort(FH,2);
```

A.6.2 Repeat-accumulate code

```
function [H]=LDPC_gen_RA_H(Hs)

%{
        Generate repeat accumulate parity check matrix H
        m - number of parity bits
        Hs - base LDPC code
%}

[m,~]=size(Hs);
Hp=eye(m)+diag(ones(m-1,1),-1);
H=[Hs Hp];
```

For better parity check matrix **H** without any 4-cycles is good to add into code A.6.1 condition, which eliminate any same elements in two near rows and then here is not any 4-cycles with elements from $\mathbf{H_p}$.

```
...
if length(unique([FH(a-1,:) FH(a,:)]))~=2*wr % Elimate 4 cycles with Hp
  go_if=0;
else
  for i=1:(a-2) % Eliminate 4 cycles in random matrix
    len_un=length(unique([FH(i,:) FH(a,:)]));
    if ~(len_un<=2*wr && len_un>=2*wr-1)
    % Can be only one same index in every row
      go_if=0;
      break;
    end
  end
end
...
```

```
function [c]=LDPC_encod_RA(d,H)

%{
  Encode message d (length n-m) to code c with RA LDPC code
  c - coded message
  d - data message
  H - RA LDPC parity check matrix
%}

[m,n]=size(H); % m - parity bit, n - lenght of code
k=n-m;
Hs=H(:,1:k);
v=mod(Hs*d',2);
p(1)=v(1);
for i=2:m
  p(i)=mod(p(i-1)+v(i),2);
end
c=[d p];
```

A.6.3 Conversion between indices and binary matrix

```
function [H]=LDPC_IndToMat(FH)

%{
  Convert matrix with indices to matrix with ones
  FH - matrix with indices
  H - normal matrix with ones
%}

m=size(FH,1);
```

```
n=max(max(FH));
H=zeros(m,n);
for i=1:m
  for j=1:size(FH,2)
    if FH(i,j)~=0
    %For irregular matrix, where are zeros in FH for empty elements
      H(i,FH(i,j))=1;
    end
  end
end
```

```
function [FH]=LDPC_MatToInd(H)

%{
  Convert matrix with ones to matrix with indices
  FH - matrix with indices
  H - normal matrix with ones
%}

wr=max(sum(H,2));
FH=zeros(size(H,1),wr);
for i=1:size(H,1)
  FH(i,:)=find(H(i,:));
end
```

A.6.4 Algebraic construction

A.6.4.1 Quasi-cyclic codes

```
function [H]=LDPC_gen_QC_H(wc,wr,r)

%{
  Generate H for LDPC by quasi-cycle
  wc - weight of column
  wr - weight of row
  r - number, which will be added to p=(wc-1)*(wr-1)+r, for p>(1-j)*(1-k)
%}

p=(wc-1)*(wr-1)+r; % p>(1-j)*(1-k)
J=eye(p);
J=circshift(J,-1);
H=[];
for i=0:(wc-1)
  H2=[];
  for j=0:(wr-1)
    H2=[H2 J^(i*j)];
  end
  H=mod([H;H2],2);
end
```

A.6.5 AR4JA LDPC code

```
function [Hused,H] = LDPC_gen_AR4JA_H (r, k)

%{
  Parity check matrix from AR4JA
  Hused - used part of H
  r - rate = [1/2, 2/3, 4/5]
  k - block length = [1024, 4096, 16384]
  M - submatrix size = [128, 256, 512, 1024, 2048, 4196, 8192]
%}

M=value_M(k,r);
switch r
  case 1/2
    H=[zeros(M) zeros(M) eye(M) zeros(M) mod(eye(M)+permM(1,M),2);
      eye(M) eye(M) zeros(M) eye(M) mod(permM(2,M)+permM(3,M)+permM(4,M),2);
      eye(M) mod(permM(5,M)+permM(6,M),2) zeros(M) mod(permM(7,M)+permM(8,M),2)
        eye(M)];
  case 2/3
    H=[zeros(M) zeros(M) zeros(M) zeros(M) eye(M) zeros(M) mod(eye(M)+permM(1,M),2);
      mod(permM(9,M)+permM(10,M)+permM(11,M),2) eye(M) eye(M) eye(M)
        zeros(M) eye(M) mod(permM(2,M)+permM(3,M)+permM(4,M),2);
      eye(M) mod(permM(12,M)+permM(13,M)+permM(14,M),2) eye(M)
        mod(permM(5,M)+permM(6,M),2) zeros(M) mod(permM(7,M)+permM(8,M),2) eye(M)];
  case 4/5
    H=[zeros(M) zeros(M) zeros(M) zeros(M) zeros(M) zeros(M) zeros(M) zeros(M)
        eye(M) zeros(M) mod(eye(M)+permM(1,M),2);
      mod(permM(21,M)+permM(22,M)+permM(23,M),2) eye(M)
        mod(permM(15,M)+permM(16,M)+permM(17,M),2) eye(M)
        mod(permM(9,M)+permM(10,M)+permM(11,M),2) eye(M) eye(M) eye(M)
        zeros(M) eye(M) mod(permM(2,M)+permM(3,M)+permM(4,M),2);
      eye(M) mod(permM(24,M)+permM(25,M)+permM(26,M),2) eye(M)
        mod(permM(18,M)+permM(19,M)+permM(20,M),2) eye(M)
        mod(permM(12,M)+permM(13,M)+permM(14,M),2) eye(M)
        mod(permM(5,M)+permM(6,M),2) zeros(M)
        mod(permM(7,M)+permM(8,M),2) eye(M)];
end
Hused=H(:,1:(end-M));

function [x] = value_M (k,r)
tab = [512 256 128; 2048 1024 512; 8192 4096 2048];
if k==1024, i=1; end
if k==4096, i=2; end
if k==16384, i=3; end
if r==1/2, j=1; end
if r==2/3, j=2; end
if r==4/5, j=3; end
x=tab(i,j);

function [P] = permM (k,M)
P=zeros(M,M);
for i = 0:(M-1)
```

```
  P(i+1,pi_k(M,i,k)+1)=1;
end

function [x] = pi_k (M, i, k)
x=M/4*(mod(theta(k)+floor(4*i/M),4))+mod(fi(k,floor(4*i/M),M)+i,M/4);

function x=theta(k)
% tab - table of values
tab = [3 0 1 2 2 3 0 1 0 1 2 0 2 3 0 1 2 0 1 2 0 1 2 1 2 3];
x=tab(k);

function x=fi(k,i,M)
%{
  fi_k(i=0..3, M)
  i=floor(4*i/M)
  M = [128, 256, 512, 1024, 2048 4196, 8192];
%}

switch i
  case 0
    tab = [1 59 16 160 108 226 1148;
      22 18 103 241 126 618 2032;
      0 52 105 185 238 404 249;
      26 23 0 251 481 32 1807;
      0 11 50 209 96 912 485;
      10 7 29 103 28 950 1044;
      5 22 115 90 59 534 717;
      18 25 30 184 225 63 873;
      3 27 92 248 323 971 364;
      22 30 78 12 28 304 1926;
      3 43 70 111 386 409 1241;
      8 14 66 66 305 708 1769;
      25 46 39 173 34 719 532;
      25 62 84 42 510 176 768;
      2 44 79 157 147 743 1138;
      27 12 70 174 199 759 965;
      7 38 29 104 347 674 141;
      7 47 32 144 391 958 1527;
      15 1 45 43 165 984 505;
      10 52 113 181 414 11 1312;
      4 61 86 250 97 413 1840;
      19 10 1 202 158 925 709;
      7 55 42 68 86 687 1427;
      9 7 118 177 168 752 989;
      26 12 33 170 506 867 1925;
      17 2 126 89 489 323 270];
  case 1
    tab = [0 0 0 0 0 0 0;
      27 32 53 182 375 767 1822;
      30 21 74 249 436 227 203;
      28 36 45 65 350 247 882;
      7 30 47 70 260 284 1989;
      1 29 0 141 84 370 957;
      8 44 59 237 318 482 1705;
      20 29 102 77 382 273 1083;
      26 39 25 55 169 886 1072;
      24 14 3 12 213 634 354;
      4 22 88 227 67 762 1942;
      12 15 65 42 313 184 446;
      23 48 62 52 242 696 1456;
      15 55 68 243 188 413 1940;
      15 39 91 179 1 854 1660;
      22 11 70 250 306 544 1661;
      31 1 115 247 397 864 587;
      3 50 31 164 80 82 708;
      29 40 121 17 33 1009 1466;
      21 62 45 31 7 437 433;
      2 27 56 149 447 36 1345;
      5 38 54 105 336 562 867;
      11 40 108 183 424 816 1551;
      26 15 14 153 134 452 2041;
      9 11 30 177 152 290 1383;
      17 18 116 19 492 778 1790];
  case 2
    tab = [0 0 0 0 0 0 0;
      12 46 8 35 219 254 318;
      30 45 119 167 16 790 494;
      18 27 89 214 263 642 1467;
      10 48 31 84 415 248 757;
      16 37 122 206 403 899 1085;
      13 41 1 122 184 328 1630;
      9 13 69 67 279 518 64;
      7 9 92 147 198 477 689;
      15 49 47 54 307 404 1300;
      16 36 11 23 432 698 148;
      18 10 31 93 240 160 777;
      4 11 19 20 454 497 1431;
      23 18 66 197 294 100 659;
      5 54 49 46 479 518 352;
      3 40 81 162 289 92 1177;
      29 27 96 101 373 464 836;
      11 35 38 76 104 592 1572;
      4 25 83 78 141 198 348;
      8 46 42 253 270 856 1040;
      2 24 58 124 439 235 779;
      11 33 24 143 333 134 476;
      11 18 25 63 339 542 191;
      3 37 92 41 14 545 1393;
      15 35 38 214 277 777 1752;
      13 21 120 70 412 483 1627];
  case 3
    tab = [0 0 0 0 0 0 0;
      13 44 35 162 312 285 1189;
      19 51 97 7 503 554 458;
      14 12 112 31 388 809 460;
      15 15 64 164 48 185 1039;
      20 12 93 11 7 49 1000;
      17 4 99 237 185 101 1265;
```

```
     4 7 94 125 328 82 1223;
     4 2 103 133 254 898 874;
     11 30 91 99 202 627 1292;
     17 53 3 105 285 154 1491;
     20 23 6 17 11 65 631;
     8 29 39 97 168 81 464;
     22 37 113 91 127 823 461;
     19 42 92 211 8 50 844;
     15 48 119 128 437 413 392;
     5 4 74 82 475 462 922;
     21 10 73 115 85 175 256;
     17 18 116 248 419 715 1986;
     9 56 31 62 459 537 19;
     20 9 127 26 468 722 266;
     18 11 98 140 209 37 471;
     31 23 23 121 311 488 1166;
     13 8 38 12 211 179 1300;
     2 7 18 41 510 430 1033;
     18 24 62 249 320 264 1606];
end
x=tab(k,log2(M)-6);
```

```
function [Gused,G] = LDPC_gen_AR4JA_G (H,r,k)

%{
  Generate generating matrix G from AR4JA
  Gused - used part of G
  H - Parity check matrix
  r - rate = [1/2, 2/3, 4/5]
  k - block length = [1024, 4096, 16384]
  M - submatrix size = [128,256,512,1024,2048,4196,8192]
%}

if r==1/2, K=2; % K/(K+2)
elseif r==2/3, K=4;
elseif r==4/5, K=8;
end
M=value_M(k,r);
P=gf(H(:,end-(3*M-1):end));
Pinv=inv(P);
Pinv=double(Pinv.x);
Q=H(:,1:K*M);
W=(mod(Pinv*Q,2))';
G=[eye(M*K) W];
Gused=G(:,1:M*(K+2));

function [x] = value_M (k,r)
tab = [512 256 128; 2048 1024 512; 8192 4096 2048];
if k==1024, i=1; end
if k==4096, i=2; end
if k==16384, i=3; end
if r==1/2, j=1; end
if r==2/3, j=2; end
if r==4/5, j=3; end
x=tab(i,j);
```

A.6.6 Joint channel in two-way relay system

A.6.6.1 Joint channel encoding

```
function [y]=JC_encode(Ca, ha, Cb, hb)

%{
  Joint channel encoding
  Ca - code a
  Cb - code b
  ha - fading coefficients for a
  hb - fading coefficients for b
%}

y=bpsk(Ca,ha)+bpsk(Cb,hb);
```

A.6.6.2 Separated channel decoding

```
function [Cr, Ca, Cb] = SC_Decode(y , ha, hb, H, sigma2, iter)

%{
  Separated Channel Decoding
  Ca, Cb, Cr - output codewords for a,b,r
  y - input
  ha, hb - fading coefficients for a, b
  sigma2 - standart deviation of noice
  iter - maximal iteration
%}

S = [ha+hb; -ha+hb; ha-hb; -ha-hb]; % [Ca Cb] = [0 0; 1 0; 0 1; 1 1]
p=prob_awgn(y,S,sigma2);
Pa=[p(1,:)+p(3,:);p(2,:)+p(4,:)];
la=log(Pa(1,:)./Pa(2,:));
Pb=[p(1,:)+p(2,:);p(3,:)+p(4,:)];
lb=log(Pb(1,:)./Pb(2,:));
Ca=LDPC_er_cor(H,la, iter,'llr','llrspa_code', sigma2);
Cb=LDPC_er_cor(H,lb, iter,'llr','llrspa_code', sigma2);
Cr=mod(Ca+Cb,2);
```

A.6.6.3 Joint channel decoding

```
function [Cr] = JC_Decode(y , ha, hb, H, sigma2, iter)

%{
  Joint Channel Decoding
  Ca, Cb, Cr - output codewords for a,b,r
  y - input ha, hb - fading coefficients for a, b
  sigma2 - standart deviation of noice
  iter - maximal iteration
%}

S = [ha+hb; -ha+hb; ha-hb; -ha-hb]; % [Ca Cb] = [0 0; 1 0; 0 1; 1 1]
p=prob_awgn(y,S,sigma2);
P=[p(1,:)+p(4,:);p(2,:)+p(3,:)];
l=log(P(1,:)./P(2,:));
Cr=LDPC_er_cor(H,l, iter,'llr','llrspa_code', sigma2);
```

A.7 Analytic tool

A.7.1 Example of bit error rate

```
k=600; wc=3; wr=3; iter=30; % maximal iteration for decoder
rep=250; % repetition
SNR=[-5:1/3:6];
H=LDPC_gen_stair_H(k,wc,wr);
[m,n]=size(H);
uncod=randi([0 1],rep,k);
cod=[];
for i=1:rep
  cod(i,:)=LDPC_encod_RA(uncod(i,:),H); % codeword
  cod_tr(i,:)=bpsk(cod(i,:),1);
end
sum_er_channel=nan(rep,length(SNR));
sum_er_de_llr=nan(rep,length(SNR));
sum_er_de_prob=nan(rep,length(SNR));
sum_er_de_hard=nan(rep,length(SNR));
parfor i=1:length(SNR)
  for j=1:rep
    [x,sigma2]=LDPC_AWGN(cod_tr(j,:), 2, SNR(i),[]);
    p0=exp(-((abs(1-x).^2)/2/sigma2));
    p1=exp(-((abs(-1-x).^2)/2/sigma2));
    l=log(p0./p1);
    trans=ones(1,n);
    trans(l>0)=0;
    sum_er_channel(j,i)=sum(mod(uncod(j,:)-LDPC_decod_sys(trans,k),2));
    decod_llr=SPA_llr(l, H, iter, 'code');
    sum_er_de_llr(j,i)=sum(mod(uncod(j,:)-LDPC_decod_sys(decod_llr,k),2));
    sump=p0+p1;
    x=round(p1./sump);
    decod_hard=SPA_hard(x,H, iter);
    sum_er_de_hard(j,i)=sum(mod(uncod(j,:)-LDPC_decod_sys(decod_hard,k),2));
    p0=p0./sump;
    p1=p1./sump;
    decod_prob=SPA_soft([p0;p1], H, iter);
    sum_er_de_prob(j,i)=sum(mod(uncod(j,:)-LDPC_decod_sys(decod_prob,k),2));
  end
end

% count error on one bit
sum_er(1,:)=sum(sum_er_channel,1)./(k*rep);
sum_er(2,:)=sum(sum_er_de_llr,1)./(k*rep);
sum_er(3,:)=sum(sum_er_de_prob,1)./(k*rep);
sum_er(4,:)=sum(sum_er_de_hard,1)./(k*rep);

figure(11)
hold on;
plot(SNR,sum_er(1,:),'bx-');
plot(SNR,sum_er(2,:),'ro-');
plot(SNR,sum_er(3,:),'g+-');
plot(SNR,sum_er(4,:),'ks-');
legend('From channel', 'SPA - LLR ', 'SPA - Prob.', 'SPA - Hard');
set(gca,'YScale','log');
ylabel('Bit error rate');
xlabel('SNR (dB)');
```

A.7.2 EXIT chart

```
function EXIT_chart_multi(H,SNRdB)

%{
  Make EXIT chart for many SNR
  Only for regular wr, that mean only one wc for every CN
%}

%%%%%%%%%% Start - info from H
[m,n]=size(H);
R=(n-m)/n;
wc=sum(H,1); % weigth of every column
wr=sum(H,2)'; % weigth of every row
edg=sum(sum(H));
dvi=unique(wc); % edges from VN
dci=unique(wr); % edges from CN
dvi(2,:)=dvi(1,:).*histc(wc,dvi(1,:))./edg; % Degree distribution of VN
dci(2,:)=dci(1,:).*histc(wr,dci(1,:))./edg; % Degree distribution of CN
dc=ceil(sum(dci(1,:).*dci(2,:))); % degree of CN
dv=sum(dvi(1,:).*dvi(2,:)); % degree of VN
b=(n.*dvi(1,:).*dvi(2,:))/(n*dv); % b for dv (VND)
%%%%%%%%%% EXIT Curves
SNRdB=sort(SNRdB,'descend');
steps=[0.0001:(0.9999-0.0001)/49:0.9999];
nsteps=length(steps);
IecIav=zeros(2,nsteps);
```

```
IevIac=zeros(2,nsteps);
%%%%% Graph
figure
hold on;
axis([0 1 0 1]);
xlabel('I_{EC}␣I_{AV}');
ylabel('I_{EV}␣I_{AC}');
set(gca,'XTick',[0:0.2:1]);
set(gca,'YTick',[0:0.2:1]);
title({['EXIT␣chart,␣R␣=␣' num2str(R,2) ',
␣␣wr␣=␣' num2str(dc) ',␣wc␣=␣' num2str(dvi(1,:))] ;
  ['SNR␣=␣[' num2str(SNRdB) ']␣dB']});
jsnr=1;
for iSNRdB=SNRdB
  [y,sigma2]=LDPC_AWGN(ones(1,n),2,iSNRdB, []); % BPSK 1 -> WORD 0
  sigma2ch=4/sigma2;
  j=0;
  for i=steps
    j=j+1;
    IevIac(1,j)=i;
    IevIac(2,j)=IeVNDdvi(b,IevIac(1,j),dvi,sigma2ch);
    if jsnr==1 % CND is indenpendent of AWGN, it same everytime
      IecIav(1,j)=i;
      IecIav(2,j)=IeCND(IecIav(1,j),dc);
    end
  end
  %%%%% Decoder path
  j=1;
  decoder=[steps(1);steps(1)];
  breaker=0;
  while decoder(1,end)<=0.99999 % For almost 1 (0.999999)
    j=j+1;
    decoder(:,j)=[decoder(1,j-1);
    IeVNDdvi(b,decoder(1,j-1),dvi,sigma2ch)];
    if decoder(2,end)>=0.99999
      break;
    end
    j=j+1;
    decoder(:,j)=[IeCND(decoder(2,j-1),dc);decoder(2,j-1)];
    if decoder(:,j)==decoder(:,j-2)
      breaker=breaker+1;
    else
    breaker=0;
    end
    if breaker==50
      break;
    end
  end
%%%%% Graph
  plot(decoder(1,:),decoder(2,:), 'k-'); % Path of Decoder
  plot(IevIac(1,:),IevIac(2,:), ':' ); % From y
  if jsnr==1
    plot(IecIav(2,:),IecIav(1,:), 'r--'); % From x
    legend('Decoder','VND','CND','Location','SouthEast');
  end
  jsnr=jsnr+1;
end
hold off;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [I]=IeVND(IA,dv,sigma2ch)
I=J(sqrt((dv-1)*(inv_J(IA))^2+sigma2ch));

function [I]=IeVNDdvi(b,IA,dvi,sigma2ch)
I=0;
for i=1:length(b)
  I=I+b(i)*IeVND(IA,dvi(1,i),sigma2ch);
end

function [I]=IeCND(IA,dc)
I=1-J(sqrt(dc-1)*inv_J(1-IA));

function [I]=IaCND(IE,dc)
I=1-J(inv_J(1-IE)/sqrt(dc-1));

function [I]=J(sigma)
if 0<=sigma && sigma<=1.6363
  I=-0.0421061*sigma^3+0.209252*sigma^2-0.00640081*sigma;
elseif 1.6363<sigma && sigma<10
  I=1-exp(0.00181491*sigma^3-0.142675*sigma^2-0.0822054*sigma+0.0549608);
elseif sigma>10
  I=1;
end

function [sig]=inv_J(I)
if 0<=I && I<=0.3646
  sig=1.09542*I^2+0.214217*I+2.337727*sqrt(I);
elseif 0.3646<I && I<1
  sig=-0.706692*log(0.386013*(1-I))+1.75017*I;
end
sig=real(sig);
```

A.7.2.1 Design with EXIT chart

```
function EXIT_chart_design(n,wc,wr,SNRdB)

%{
  Make EXIT chart for many SNR
  Only for regular wr, that mean only one wc for every CN
  wc[2 ; :] - ones in column, irregular columns
    wc(1,:) - weigth of ones,
    wc(2,:) - degree distribution
  wc - ones in column, regular columns
  wr - ones in row
  n - number of columns
%}

%%%%%%%%%% Start - parameters
if 1==size(wc,1) && 1==size(wc,2) % for regular H, with one element wc
  m=n*wc/wr;
else %% for irregular H
  m=n*sum(wc(1,:).*wc(2,:))/wr;
end
if ~( m == floor(m))
  error(['Number of m (CN) is not integer. m = ',num2str(m)])
end
R=(n-m)/n;
edg=m*wr;
dvi(1,:)=wc(1,:); % edges from VN
dvi(2,:)=wc(1,:).*wc(2,:).*n./edg; % Edge distribution of VN
dc=wr; % degree of CN
dv=sum(dvi(1,:).*dvi(2,:)); % degree of VN
b=(n.*dvi(1,:).*dvi(2,:))/(n*dv); % b for dv (VND)

%%%%%%%%%% EXIT Curves
...
```

Rest of code is same like in previous part.

Appendix B

Contents of attached CD

\MATLAB Source Codes
: There are all implemented MATLAB's codes with names and form like in Appendix A. All codes have comments, which describe their inputs and purpose.

\Thesis
: In this folder is our thesis in version for print and for computer.

Bibliography

[1] Ezio Biglieri. *Coding for wireless channels.* Springer, 2005. 6, 24, 30

[2] Orange Book. Experimental specifications, low density parity check codes for use in near-earth and deep space applications ccsds 131.1-o-2. *CCSDS, Sept*, 2007. 1, 31

[3] Robert G Gallager. Low-density parity-check codes. *Information Theory, IRE Transactions on*, 8(1):21–28, 1962. 1, 24

[4] Yuan Jiang. *A practical guide to error-control coding using Matlab.* Artech House, 2010. 21

[5] Sarah J Johnson. *Iterative error correction: turbo, low-density parity-check and repeat-accumulate codes.* Cambridge University Press, 2009. 5, 7, 8, 25, 27, 30

[6] H-A Loeliger. An introduction to factor graphs. *Signal Processing Magazine, IEEE*, 21(1):28–41, 2004. 4

[7] David JC MacKay. Good error-correcting codes based on very sparse matrices. *Information Theory, IEEE Transactions on*, 45(2):399–431, 1999. 1

[8] David JC MacKay. *Information theory, inference and learning algorithms.* Cambridge university press, 2003. 1, 2, 12, 27

[9] Alberto Morello and Vittoria Mignone. Dvb-s2: The second generation standard for satellite broad-band services. *Proceedings of the IEEE*, 94(1): 210–227, 2006. 1, 2, 29

[10] Jan Sýkora and České vysoké učení technické v Praze. *Teorie digitální komunikace.* Vydavatelství ČVUT, 2002. 9

[11] Tuan Ta. A tutorial on low density parity-check codes. 13

[12] Stephan Ten Brink. Convergence behavior of iteratively decoded parallel concatenated codes. *Communications, IEEE Transactions on*, 49(10):1727–1737, 2001. 44

[13] Stephan ten Brink, Gerhard Kramer, and Alexei Ashikhmin. Design of low-density parity-check codes for modulation and detection. *Communications, IEEE Transactions on*, 52(4):670–678, 2004. 41, 44

[14] D Wubben and Yidong Lang. Generalized sum-product algorithm for joint channel decoding and physical-layer network coding in two-way relay systems. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–5. IEEE, 2010. 37