Master's thesis

Decoding Algorithms for Sparse Network Coding and Their Implementation

$$\begin{bmatrix}
 & 109 & 254 & 82 & 169 & 122 & 231 & 183 & 164 & 13 & 153 & 75 & 212 & 166 \\
0 & & 43 & 231 & 42 & 273 & 9 & 142 & 201 & 154 & 219 & 8 & 199 & 15 \\
0 & 1 & & 127 & 3 & 241 & 177 & 284 & 167 & 198 & \multicolumn{3}{c}{\textbf{Complexity}} & 152 \\
1 & 0 & 0 & & 109 & 91 & 126 & 23 & 198 & 215 & 43 & 213 & 132 & 172 \\
0 & 0 & 0 & 1 & & 193 & 3 & 87 & 244 & 83 & 1 & 209 & 166 & 113 \\
0 & 1 & 0 & 0 & 0 & & 236 & 240 & 65 & 148 & 129 & 126 & 134 & 216 \\
0 & 0 & 1 & 0 & 1 & 0 & & 54 & 165 & 129 & 200 & 153 & 31 & 91 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & & 220 & 3 & 123 & 192 & 241 & 155 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & & 241 & 100 & 251 & 230 & 214 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & & 234 & 3 & 175 & 137 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & & 256 & 72 & 69 \\
0 & \multicolumn{3}{c}{\textbf{Simplicity}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 184 & 188 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & & 212 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 &
\end{bmatrix}$$

Networks and Distributed Systems

Aalborg University, 4. semester spring 2013

School of Information and Communication Technology
Networks and Distributed Systems
Frederik Bajers Vej 7
Telefon +45 99 40 86 00
http://www.sict.aau.dk
E-mail webinfo@es.aau.dk

Title:
Decoding Algorithms for Sparse Network Coding and Their Implementation

Theme:
Master's thesis

Project Period:
P10, Spring semester 2013

Written by:
Chres Wiant Sørensen

Supervisors:
Daniel Enrique Lucani Roetter
Janus Heide

Number of copies: 4

Number of pages: 45

Appended documents: CD-ROM

Finished: 06-06-2013

Abstract:

The development of novel applications for mobile devices demands more and more of existing network technologies.
One of these technologies is Network Coding (NC), which allows devices to transmit linear combinations of original data packets. This provides a number of benefits at the cost of increased complexity due to coding of data packets on both receiver and transmitter.
This project will therefore focus on how this complexity can be reduced on new decoding algorithms, and how these algorithms can be developed and implemented particularly for sparse codes

Preface

This thesis concludes my Master of Science education in Network and Distributed Systems at Aalborg University.

The project deals with development and implementation of decoding algorithms for sparse linear network codes

There will be used the following notation style throughout the report, and vector and matrix indexes will start from zero unless written otherwise.

- a: Scalar
- $\hat{\boldsymbol{a}}$: Vector
- $\hat{\boldsymbol{A}}$: Matrix

Citations will look like this, [1], and will link to the source list on page 43 if the report is read on a computer.

A CD is provided with the report. It contains a digital copy of the report, measurement data, graphs, and multiplication and division tables for the finite field, $GF(2^8)$.
The code will not be provided on the CD, but it will be available online on github:
`https://github.com/chres/kodo`

I would like to thank Associate Professor Daniel Enrique Lucani Roetter, Postdoc Janus Heide, and Postdoc Morten Videbæk Pedersen for all the helpful discussions and comments. I would also like to thank Steinwurf ApS for making Kodo software library available.

Chres Wiant Sørensen

Table of contents

Glossary

NC Network Coding. 4, 5, 9, 11–13

RLNC Random Linear Network Coding. 4, 12, 13

Chapter 1

Introduction

The still increasing number of mobile devices worldwide together with a growing eagerness to have applications constantly connected with more and more network connected devices' demands communication technologies to work both smarter and more efficiently in order not to drain all resources on especially devices with limited resources.

One of such technologies is Network Coding (NC), which allows a device to transmit data to one or multiple devices by generating linear combinations of original data packets. This can be useful to enhance throughput in networks or to transmit packets that are useful to multiple receivers because encoded packets in Network Coding (NC) are combinations of multiple original packets.

Achieving these gains comes at the expense of increased complexity due to encoding and decoding which translates directly into increased energy consumptions on both receiver and transmitter. This is in particular undesired on mobile devices that are often battery powered and have limited processing capabilities. It is therefore essential to reduce encoding and decoding complexity. This was also the main purpose in [1], where different strategies for generating sparse codes were investigated to minimize coding complexity at the cost of additional transmission delay. (Sparse codes are codes with few nonzero values. A matrix is for example said to be sparse in case most of its elements are zero, and it is said to be dense in case most of its elements are nonzero)
Another approach to minimize complexity was presented in [2] where different decoding algorithms were considered to find the least complex algorithm and thereby increase data throughput in decoders for Random Linear Network Coding (RLNC). This is, in contrast to sparse codes, not decreasing complexity on both encoder and decoder.

This report will therefore investigate whether novel decoding algorithms can be designed and implemented particularly for sparse codes, and in that way exploit the benefits of lower complexity during both encoding and decoding.

The algorithms in this report will build on the methods proposed in [2], but also on the algorithm proposed in [3], that in contrast to the algorithms in [2], proposes an algorithm based on column swaps that are particularly intended for sparse codes.

Chapter 2

Pre-Analysis

This section will provide an overview of NC and will be used to investigate existing decoding algorithms. Knowledge of finite fields is required to understand calculations in NC, so the chapter will be organized in the following order.

First, fields of the form $GF(p)$ are introduced because they are mathematically simple to use. In computers, these are not so practical because data representation is performed in bits. Focus will therefore turn to fields of the form $GF(2^n)$ which will typically require more complex operations, but allow you to operate seamlessly with a computer's data representation. After setting up this common ground, attention is turned toward NC where an introduction is given of its basic concepts and usage. This is followed by a description of its potential performance benefits.

Since the goal of this work is to develop and implement the algorithms proposed in this project, it will be followed by a description of Kodo software library, which is a NC library, that will be used for implementation of the algorithms. Kodo already implements a few decoding algorithms that will be investigated in the end of the chapter together with another algorithm.

2.1 Finite fields

To understand the algorithms throughout the report, it is important to have a basic understanding of finite fields. In particular, the fields $GF(2)$ and $GF(2^n)$ which all algorithms and examples in this report will be based on. $GF(2)$ will be preferred over $GF(2^n)$ in examples because of its simpler arithmetics and because the principles are basically the same no matter which field is used. In a few cases, the principles will differ depending on the field, and in those cases $GF(2^8)$ will be used to represent all fields of the form $GF(2^n)$.

The rest of the section is based on [1] and [4], and will give a short description of finite fields and a short explanation of how calculations are performed within the different fields.
Binary numbers will be enclosed by $()_2$ to distinguish binary numbers from decimal numbers.

Basics

A finite field can be represented on the form $GF(p^n)$ where GF stands for "Galois field" which will be refered to as "finite field". p is a prime number called the characteristic of the field, and n is a positive integer called the dimension of the field.
A field of the form given above contains the field elements $\{0, p^n - 1\}$, and is said to have order p^n. In contrast to ordinary arithmetic, a finite field have two basic operations called addition and multiplication, but subtraction and division are however defined as addition and multiplication with the respective inverse element which will be shown below.

Calculations in $\mathbf{GF(p)}$

In $GF(p)$, calculations are simply performed modulus p, so in case $p = 7$, the equation 4 plus 5 could be calculated as $(4 + 5) \mod 7 = 2$, and 4 times 5 evaluates to $(4 \times 5) \mod 7 = 6$.

Choosing $p = 2$ instead, the field becomes the binary field $GF(2)$, which contains the elements $\{0, 1\}$. When calculations are performed modulus 2, addition becomes the binary operation XOR, and multiplication becomes the binary operation AND.

This is illustrated by the addition and multiplication tables below:

+	0	1
0	0	1
1	1	0

Addition table for $GF(2)$

×	0	1
0	0	0
1	0	1

Multiplication table for $GF(2)$

The inverse values for subtraction can now be found in the addition table where values are zero. These values have been marked and copied to table 2.1. The inverse values for division can be found in the multiplication table where values are one. These have also been marked and copied to table 2.1.

w	$-w$	w^{-1}
0	0	-
1	1	1

Table 2.1: Additive and multiplicative inverses in $GF(2)$

From table 2.1, it can be derived that subtraction in the binary field is simply the same as XOR just like addition. Division is a little more complicated as there doesn't exist any inverse for 0. Therefore, it is only possible to divide by 1, which has the inverse 1 meaning that division by one is simply the same as multiplication by one.

Memory issue
As it was mentioned above, p has to be a prime. This means that fields of the type $GF(p)$ will not fit very well into computer memory for all p.
Assume that a finite field of the form $GF(p)$ is to be stored into a variable of 8 bits. Using 8 bit, it should be possible to store 256 values, but because 256 is not a prime this is simply not possible. The nearest prime below 256 therefore has to be used which is 251, but in that case the values $\{251, 252, \ldots 255\}$ are wasted.

In the following, we will analyze the characteristics and operations of extension fields, which allow us to get around this issue while providing operations that are consistent with the properties of a finite field.

Calculations in $\mathbf{GF(2^n)}$

In $GF(2^n)$, each field consumes n bits in memory, and calculations can be performed using polynomial arithmetic, where each bit in the field can be represented as a polynomial of degree $n-1$.

The polynomial is defined as

$$f(x) = \sum_{i=0}^{n} a_i x^i = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0,$$

where a_i is the i'th coefficient (bit) value. As an example, consider the decimal value 10, which can be represented in binary on the left side and on polynomial form on the right side in the equation below:

$$(1010)_2 = x^3 + x$$

Because $GF(2^n)$ is basically just n elements belonging to $GF(2)$ arithmetics of the coefficients are performed modulo 2. To illustrate some calculations, assume that two polynomials are given as below:

$$f(x) = \sum a_i x^i \quad \text{and} \quad g(x) = \sum b_i x^i$$

where a_i and b_i are the bits on the $i'th$ position.

The polynomial addition is then defined as the bitwise xor:

$$f(x) + g(x) = \sum (a_i + b_i) x^i \tag{2.1}$$

and the multiplication is defined as:

$$f(x) \times g(x) = \sum c_i x^i \tag{2.2}$$

where

$$c_k = a_0 b_k + a_1 b_{k-1} + \cdots + a_k b_0$$

In order not to increase the degree of the polynomial when multiplying, the result should be reduced modulo an irreducible polynomial $p(x)$ which is simply the polynomial representation of a prime within the field:

$$p(x) = \sum_{i=0}^{n} x^i$$

To demonstrate the calculations within $GF(2^n)$, the field $GF(2^2)$ will be used as an example, so numbers and tables stay small. The principals for fields with larger n is exactly the same just with a different irreducible polynomial $p(x)$.

The irreducible polynomial used in this example will be:

$$p(x) = x^2 + x + 1$$

Solving the equation 2 plus 3 is simple by using the equation for addition given in equation 2.1, which is just a bitwise xor, to sum the numbers. This can be seen below, where the numbers to the left are on decimal representation, and the numbers to the right are on the binary representation. On the binary representation, the bitwise xor operation are easily performed to obtain the result below the line.

Multiplication requires the polynomial representation. This can be illustrated below by solving 2 times 3:

$$2 = (10)_2 = x \quad \text{and} \quad 3 = (11)_2 = x + 1$$

$$\begin{array}{ll} 2: & (10)_2 \\ 3: & \underline{(11)_2} \\ & (01)_2 \end{array}$$

Now, when the numbers has been represented on polynomial form, multiplication of 2 times 3 can be perform as below:

$$\begin{aligned} 2 \times 3 &= x \times (x+1) \mod p(x) \\ &= x^2 + x \mod p(x) \end{aligned}$$

It can be seen that the value $x^2 + x$ exceeds the field size of $GF(2^2)$ which only allows values in the set $\{0, 1, 2, 3\}$. This means that the modulus operation should be performed which is done with long polynomial division by the irreducible polynomial $p(x)$:

$$\begin{array}{rr|lll} & & 1 & & \\ \hline x^2 + x & & +x^2 & +x & +1 \\ & & +x^2 & +x & \\ \hline & & & & 1 \end{array}$$

The residue is the result, so in $GF(2^2)$, $2 \times 3 = 1$.

These calculations can be time consuming to perform. Therefore, the results of addition and multiplication are usually performed only once for all combinations of numbers within the field and stored in a tables. Such tables are shown for $GF(2^2)$ below:

+	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

Addition table for $GF(2^2)$

×	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

Multiplication table for $GF(2^2)$

As last time, the marked cells are the inverse values that can be used for subtraction and division. As previously, the marked cells are copied to another table to better see of the inverse elements. This is shown in table 2.2.

w	$-w$	w^{-1}
0	0	-
1	1	1
2	2	3
3	3	2

Table 2.2: Polynomial additive and multiplicative inverses in $GF(2^2)$

Now, subtraction and division can be performed by a lookup in the tables. In table 2.2 it can be seen that addition and subtraction is the same, so we get:

$$2 - 3 \mod p(x) = 2 + 3 \mod p(x) = 1$$

and with division, we can see from table 2.2 that the inverse of 3 is 2, so the equation below can be rewritten as:

$$2/3 \mod p(x) = 2 \times 2 \mod p(x)$$

Instead of doing long polynomial division, the result 2 times 2 are simply looked up in the multiplication table yielding the result

$$2 \times 2 \mod p(x) = 3$$

2.2 Network coding

NC is a method that can be applied to generate linear combinations of data packets that can be transmitted over the network instead of the original data itself. To understand how it works, lets first consider the terminology in network coding.
Assume a big file is to be transmitted from one computer to another. Without NC, the file could be split into smaller chunks of data called blocks. These blocks would then be loaded into the computer's memory and split once again into data chunks of a size that can be transmitted over the network. This is all illustrated in figure 2.1.
This same procedure is also performed with NC, but the naming convention is slightly different and a few additional constraints, other than the packet size, are to be considered. This is visualized in figure 2.2
First of all, the blocks are now denoted generations, and the packets are denoted symbols. Each symbol within a generation is chopped into finite field elements. The size of each element depends on the field. $GF(2)$ would for example use 1 bit per element, where $GF(2^8)$ on the other hand would use 8 bits. Apart from the field sizes, another constraint is that all symbols within a generation has to contain the same number of elements. The field elements at the end can be filled with trailing zeros in case where the last generation should exceed the file's data as depicted in figure 2.2. Generations can then be transmitted sequentially by generating linear combinations of the symbols within only one generation, meaning that linear combinations can only be generated from symbols within the same generation. Therefore, this work will only focus on transmitting a single generation as transmitting more generations are simply redoing the same encoding/decoding procedure.

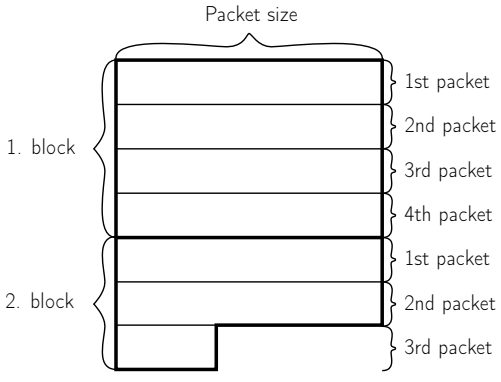

Figure 2.1: Traditional terminology[1]

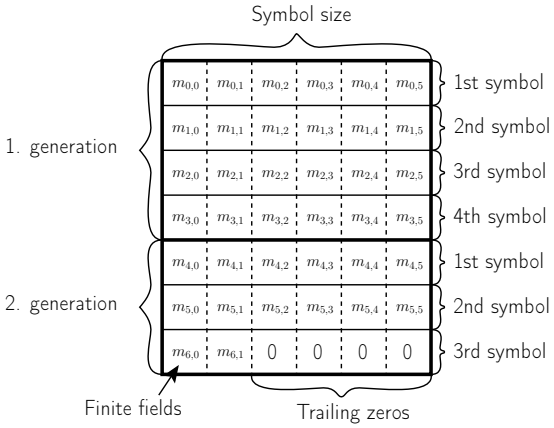

Figure 2.2: NC terminology[1]

2.2.1 Encoding/Decoding

When a generation has been split into symbols and field elements, the encoding process can begin. This is done by combining symbols within a generation. A single encoded symbol can be generated using equation 2.3

$$\hat{\boldsymbol{x}} = \sum_{i=1}^{n} g_i \, \hat{\boldsymbol{m}}_i = \begin{bmatrix} g_1 & g_2 & \cdots & g_n \end{bmatrix} \begin{bmatrix} \hat{\boldsymbol{m}}_1 \\ \hat{\boldsymbol{m}}_2 \\ \vdots \\ \hat{\boldsymbol{m}}_n \end{bmatrix} \tag{2.3}$$

where:

$\hat{\boldsymbol{x}}$ is the encoded symbol which is of the same size as the uncoded symbols.

g_i is a encoding coefficient that is randomly generated from the subset of values within the finite field used. The coefficients form a vector $\hat{\boldsymbol{g}}$ called the encoding vector.

$\hat{\boldsymbol{m}}_i$ is the i'th symbol which corresponds to the i'th row in figure 2.3.

This can be illustrated by an example. Assume that the field $GF(2)$ is chosen and the encoding vector $\begin{bmatrix} 1 & 0 & 0 & 1 & 0 \end{bmatrix}$ is randomly generated. Then the encoded symbol will consist of $\hat{\boldsymbol{m}}_0$ and $\hat{\boldsymbol{m}}_3$ as it can be seen below:

$$\hat{\boldsymbol{x}} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \hat{\boldsymbol{m}}_0 \\ \hat{\boldsymbol{m}}_1 \\ \vdots \\ \hat{\boldsymbol{m}}_{n-1} \end{bmatrix} = \hat{\boldsymbol{m}}_0 + \hat{\boldsymbol{m}}_3$$

Because $GF(2)$ was chosen, addition is simply the same as xor, so $\hat{\boldsymbol{x}}$ is equal to $\hat{\boldsymbol{m}}_0 \oplus \hat{\boldsymbol{m}}_3$. This is however not enough knowledge to decode a generation. For that, at least n encoded packets are required. This is better modelled with the matrix representation in equation 2.4.

$$\begin{bmatrix} x_{0,0} & x_{0,1} & \cdots & x_{0,m-1} \\ x_{1,0} & x_{1,1} & \cdots & x_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n-1,0} & x_{n-1,1} & \cdots & x_{n-1,m-1} \end{bmatrix} = \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,m-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{n-1,0} & g_{n-1,1} & \cdots & g_{n-1,m-1} \end{bmatrix} \begin{bmatrix} m_{0,0} & m_{0,1} & \cdots & m_{0,m-1} \\ m_{1,0} & m_{1,1} & \cdots & m_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ m_{n-1,0} & m_{n-1,1} & \cdots & m_{n-1,m-1} \end{bmatrix} \tag{2.4}$$

where:

$\hat{\boldsymbol{X}}$ is the matrix of encoded symbols. Each row, $\hat{\boldsymbol{x}}_i$ corresponds to an encoded symbol.

$\hat{\boldsymbol{G}}$ is the matrix of coefficients randomly generated. Each row, $\hat{\boldsymbol{g}}_i$, is an encoding vector that describes which symbols were combined to generate a coded symbol, $\hat{\boldsymbol{x}}_i$

$\hat{\boldsymbol{M}}$ is still the generation of original data just represented as finite fields. Each row, $\hat{\boldsymbol{m}}_i$, corresponds to an uncoded symbol.

On the receiver, decoding can start either after n symbols has been received or on-the-fly as symbols are being received. Decoding can be performed using Gauss Jordan elimination.

$$\hat{\boldsymbol{X}} = \hat{\boldsymbol{G}}\hat{\boldsymbol{M}} \Rightarrow \hat{\boldsymbol{M}} = \hat{\boldsymbol{G}}^{-1}\hat{\boldsymbol{X}}$$

To completely solve the system, it is required that the decoder receives n linear independent encoding vectors, $\hat{\boldsymbol{g}}$. This means that $\hat{\boldsymbol{G}}$ has to be invertible, which is true if the determinant of $\hat{\boldsymbol{G}}$ is nonzero, $\det \hat{\boldsymbol{G}} \neq 0$. In case linear dependent encoding vectors are received, the decoder can simply drop them as they provide no information that isn't already known. In other words - they are not innovative.

2.2.2 Benefits of network coding

This section is based on [5] and will present some benefits of NC.

One advantage of NC is that it can be used to obtain optimal throughput in networks. This is illustrated with a butterfly network in 2.3, where two transmitters are collaborating to distribute a file to two receivers in a directed graph where each edge has capacity 1.
It can be seen that T_1 can send directly to R_1 and T_2 directly to R_2, but the relays in the middle are required to send from T_1 to R_2 and T_2 to R_1. This is however not possible, as there are only capacity enough for one packet at a time. This means that only one receivers will be able to fully decode the data.
Considering figure 2.4, that are using NC, it can be seen that this problem can be solved using NC. This is because the relay that receives both symbols generates an encoded symbol from both x_1 and x_2. That way, data can actually be transmitted as encoded symbols have the same size as uncoded symbols no matter how many symbols are combined. The complexity is however increased by combining two symbols.

The decoding process to deduce the missing symbol on R_1 and on R_2 can now be done as follows:

$$x_2 = (x_1 + x_2) - x_1$$
$$x_1 = (x_1 + x_2) - x_2$$

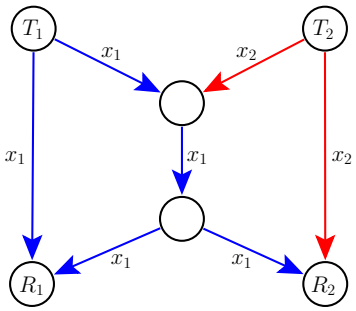

Figure 2.3: Traditional method

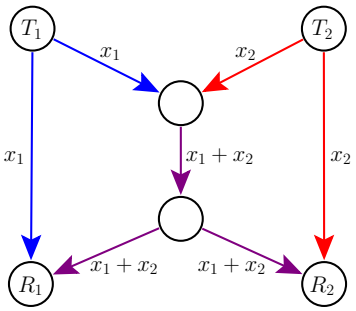

Figure 2.4: Network coding

A second advantage of NC is its ability to provide innovative data in network in which the state of receiving nodes are unknown. Assume that a node broadcasts three uncoded packets x_1, x_2, x_3 on an erasure channel. This is illustrated in figure 2.5. If the transmitter is without NC, it will have to transmit all uncoded symbols once again because R_1 is missing x_1, R_2 is missing x_2 and R_3 is missing x_3. This is however not necessary if the transmitter is using NC, because an encoded symbol $x_1 + x_2 + x_3$ could be generated to provide the missing information for each receiver in only one packet.

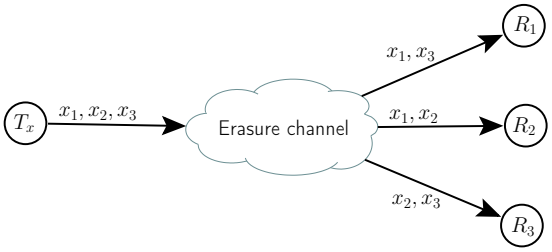

Figure 2.5: Erasure channel

In contrast to many other coding technologies, such as Reed-Solomon, LT codes, Raptor codes, and Tornado codes, a third advantage of NC is that it allows recoding of symbols on intermediate nodes in networks. This is done by either forwarding the symbols received or by combining multiple symbols. This has the disadvantage that it will often increase complexity of sparse codes as the number of nonzero coefficients increase as more symbols are combined. This increased complexity can be counteracted by also decoding symbols as much as possible on intermediate nodes before symbols are combined and forwarded.

2.2.3 Encoding schemes

Generating the encoding vectors coefficients can be done using different schemes. Some of the most popular schemes will be described below.

Random Linear Network Coding (RLNC)

The first and most popular encoding scheme is called RLNC and is also the most simple scheme. With this scheme, each coefficient in the encoding vector $\hat{\boldsymbol{g}}$ is generated randomly from the finite field used by the encoder. It can be shown using psudocode as below, where n is the length of coding vectors:

```
1 for i = 0 to n-1 do
2     ĝ_i = Uniform(0, FieldMax)
```

Algorithm 2.1: Random Linear Network Coding

Sparse Network Coding

Sparse codes are almost equally as simple to generate as in RLNC. Their purpose is to reduce the complexity of the codes generated by increasing the probability of generating zero coefficients. This is done to create a sparser $\hat{\boldsymbol{G}}$ matrix, which will decrease code complexity and ease the work on both encoders and decoders.

An algorithm generating sparse codes can be seen below. This algorithm generates sparse codes that are uniformly distributed, but sparse codes could also be generated from other distributions. This project will however use codes that are uniformly distributed as illustrated in the pseudocode.

```
1 ĝ = 0
2 for i = 0 to n-1 do
3     if Bernoulli(p) then
4         ĝ_i = Uniform(1, FieldMax)
```

Algorithm 2.2: Sparse Network codes

It can be seen that the method is basically the same as for RLNC except that all coefficients are initialized to zero and that a nonzero value is only generated for the i'th coefficient with probability, p. p is in this context a scalar that specifies how sparse coding vectors should be made, and should be within the range $0 < p < 1$.

This scheme has the advantage of lower complexity but increased probability of generating dependent encoding vectors.

The lower complexity is due to the fact that the encoder combines fewer symbols together every time it produces an encoded symbol. This also provides the decoder with symbols that are combined with fewer original symbols which means that fewer symbols has to be subtracted to decode the symbol.

Systematic Network Coding

A third scheme that should be presented is called systematic network coding, which is a scheme for transmitting both original data and encoded data. In NC, systematic coding sends all n packets uncoded at first and after that it start generating encoded symbols.

The idea behind this scheme is that there is no need to encode data from the beginning as the first n packets are guaranteed to be innovative to all receivers. Innovative means that they provide new information. If the decoder(s) have not successfully decoded after the first n symbols was send, the transmitter cannot know which symbols were lost without any feedback from each receiver(s). Therefore, RLNC are usually applied to provide the missing information. This scheme will however not be considered because uncoded symbols don't need to be decoded and secondly because packet loss probabilites are not considered in this work. Hence, there will never be send a coded packet with systematic coding and no packet loss.

2.3 Software libraries

Implementing and testing decoding algorithms for NC requires a lot of code to get started. It has therefore been decided to use some libraries to ease implementations.

Most importantly is the library called Kodo, which is an open source library for network coding developed and maintained by Steinwurf ApS. It is written in C++ using a layered design approach called mixin-layers, that makes it extremely well suite for code reuse because all functionalities are written as layers. These layers can be stacked together to form an encoder or decoder, where functionalities can easily be added or removed simply by including or excluding layers that provides a given functionalities.

Another advantage of using Kodo is that it already implements an encoder and a few different decoding algorithms. The decoding algorithms are described in [2] and perform very well, so they can be used to compare this projects decoding algorithms against. The layered design in Kodo allows algorithms

in Kodo to be replaced by decoding algorithms developed in this project so most functionalities can be reused. This ensures that algorithms are directly comparable as they are simulated within the same code. That way, it can be sure that it is the decoding algorithms that are compared and not other decoder parts such as memory storage which may perform differently as they are identical.

Steinwurf ApS provides an additional library that can be used with Kodo called gauge. Gauge is a benchmark library that can measure execution time of code, and automatically decides how many times a simulation should be run to keep the error below some threshold.

A third library that will be used is a profiling library called Valgrid. Valgrind will be used to get a picture of where time is spent within a program during runtime. This will be used to profile algorithms so that it can be determined to which extend an algorithm can be improved.

2.4 Existing decoding algorithms

This section will discuss a few existing decoding algorithms which includes two algorithms that are already implemented in Kodo and presented in [2] and the algorithm proposed for sparse codes which was proposed in [3].

This will be done with the intention of providing a better understanding of decoding and the different steps in the decoding process.
To keep it simple, it has been chosen to only present existing algorithms for $GF(2)$. Moving to a higher fields requires vectors to be normalized. This step will be presented seperately as it will be accounted for in the algorithms that will be developed.

To decode packets, the receiver will need both the encoded symbols and the coding vectors that tells which original symbols were combined. These are usually transmitted as pairs of the encoded symbol and the coding vector used to generate it. It will therefore be assumed throughout this report that the encoded symbol, $\hat{\boldsymbol{x}}_i$, and the coding vector, $\hat{\boldsymbol{g}}_i$, are always available pairs.

2.4.1 Basic decoder (implemented in Kodo)

The first decoding algorithm that will be presented is the default decoding algorithm used in Kodo. It is presented in [2] from which some of the psudocodes blocks has been copied from as they provide a very good explanation of how the algorithm works. The implementation in Kodo has however matured slightly compared to what was presented in the article. The algorithms have therefore been updated to reflect the current implementation in Kodo.

Introduce variable:
g: coding vector length

Basic Decoding algorithm

The algorithm for the basic decoder is given by the psudocode in algorithm 2.3. The functions called within the psudocode will be explained in the following sections.

The code in algorithm 2.3 is run everytime a new packet is received. A packet is assumed to contain an encoded symbol, $\hat{\boldsymbol{x}}$, and the coding vector, $\hat{\boldsymbol{g}}$.

The algorithm works on-the-fly which means that encoded symbols are solved as they are received. This is a huge advantage as it doesn't have a big delay at the end. Another advantage of decoding on-the-fly is that the rank can be known throughout the decoding process.

Input: $\hat{\boldsymbol{x}}$, $\hat{\boldsymbol{g}}$
1 pivotPosition = ForwardSubstituteToPivot($\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$)
2 **if** *pivotPosition* > 0 **then**
3 ForwardSubstituteFromPivot($\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$,pivotPosition)
4 BackwardSubstitute($\hat{\boldsymbol{x}}$, $\hat{\boldsymbol{g}}$, pivotPosition)
5 InsertPacket($\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$,pivotPosition)
6 rank++
7 **return** *rank*

Algorithm 2.3: Basic Decoder[2]

Example:
To illustrate the algorithm and all its steps with an example, assume that some packets have already been received and inserted into the symbol matrix, $\hat{\boldsymbol{X}}$ and the coefficient matrix, $\hat{\boldsymbol{G}}$ as below:

$$\hat{\boldsymbol{X}} = \begin{bmatrix} ? \end{bmatrix} \qquad \text{and} \qquad \hat{\boldsymbol{G}} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

What is actually stored in the symbol matrix is not interesting for the example as the algorithm only reads from the coefficient matrix, $\hat{\boldsymbol{G}}$, and the coding vector, $\hat{\boldsymbol{g}}$. The only thing to keep in mind is that every operation done that changes either $\hat{\boldsymbol{g}}$ or $\hat{\boldsymbol{G}}$ should be replicated on $\hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{X}}$ respectively. This will appear from the pseudocodes in the following sections.

Now assume a packet is received containing:

$$\hat{\boldsymbol{x}} = \begin{bmatrix} ? \end{bmatrix} \qquad \text{and} \qquad \hat{\boldsymbol{g}} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

This packet has to go through all functions in algorithm 2.3 to be inserted somewhere in the matrices. The packet will therefore work as an example that is spread over the following sections that describes the functionalities of the functions called in algorithm 2.3. The sections are listed in the same order as they are called.

Forward substitute to pivot

The first function that is called in algorithm 2.3 is *ForwardSubstituteToPivot* which is implemented as in algorithm 2.4. The purpose of the algorithm is to eliminate nonzeros in $\hat{\boldsymbol{g}}$ until $\hat{\boldsymbol{g}}$ is either a null vector or has the first nonzero at a pivot position that is not used in $\hat{\boldsymbol{G}}$.

```
   Input: x̂,ĝ
1  pivotIndex = 0                    /* 0 indicates that no povot was found */
2  for i = 1 to g do
3      if ĝ[i] = 1 then
4          if Ĝ[i,i] = 1 then
5              ĝ = ĝ[i] ⊕ Ĝ[i]                  /* substitute into vector */
6              x̂ = x̂[i] ⊕ X̂[i]                  /* substitute into symbol */
7          else
8              pivotPosition = i                   /* pivot element found */
9              break
```

Algorithm 2.4: Forward substitute to pivot[2]

Example continued:
The algorithm can be visualized as in figure 2.6 which proceeds the example. Here it can be seen that the nonzero values in $\hat{\boldsymbol{g}}$ are eliminated until it has pivot on the 4th element which is not 'reserved' in $\hat{\boldsymbol{G}}$.

The coding vector will therefore be changed from $\hat{\boldsymbol{g}} = \begin{bmatrix}1 & 0 & 1 & 0 & 1\end{bmatrix}$ to $\hat{\boldsymbol{g}} = \begin{bmatrix}0 & 0 & 0 & 1 & 1\end{bmatrix}$ as the figure depict.

Symbol data

$$\begin{bmatrix}\mathbf{1} & 0 & 1 & 0 & 1\end{bmatrix} \oplus \begin{bmatrix}\mathbf{1} & 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & \mathbf{1} & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 1\end{bmatrix} \rightarrow \begin{bmatrix}0 & 0 & 1 & 1 & 0\end{bmatrix}$$

$$\begin{bmatrix}0 & 0 & \mathbf{1} & 1 & 0\end{bmatrix} \oplus \begin{bmatrix}0 & 0 & 0 & 1 & 1\end{bmatrix}$$

Figure 2.6: Forward substitute to pivot

Forward substitute from pivot

The second function called by algorithm 2.3 is *ForwardSubstituteFromPivot*. The purpose of this function is to use existing vectors in $\hat{\boldsymbol{G}}$ to eliminate nonzero values in $\hat{\boldsymbol{g}}$ after the pivot index.

```
   Input: x̂, ĝ, pivotIndex
1  for i = pivotIndex+1 to g do
2      if ĝ[i] = 1 then
3          if Ĝ[i,i] = 1 then
4              ĝ[i] = ĝ ⊕ Ĝ[i]          /* substitute into coding vector */
5              x̂[i] = x̂ ⊕ X̂[i]                /* substitute into symbol */
```

Algorithm 2.5: Forward substitute from pivot

Example continued:
Proceeding the example, where the coding vector is now $\hat{\boldsymbol{g}} = \begin{bmatrix}0 & 0 & 0 & 1 & 1\end{bmatrix}$ after it went through

ForwardSubstituteToPivot. What can be seen in figure 2.7 is that the last encoding vector in $\hat{\boldsymbol{G}}$ will be used to eliminate the last nonzero value in $\hat{\boldsymbol{g}}$.

$$\begin{matrix} & & & & & & \text{Symbol data} & & & & \\ \begin{bmatrix}0 & 0 & 0 & 1 & \mathbf{1}\end{bmatrix} & \oplus & \begin{bmatrix}1 & 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & \mathbf{1}\end{bmatrix} & \rightarrow & \begin{bmatrix}0 & 0 & 0 & 1 & 0\end{bmatrix} \end{matrix}$$

Figure 2.7: Forward substitute from pivot

Backward substitute

Now that $\hat{\boldsymbol{g}}$ has been sent through *ForwardSubstituteToPivot* and *ForwardSubstituteFromPivot* it is guaranteed to have a pivot element that can be used to eliminate nonzero values in all rows above.

Input: $\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$,pivotIndex
for *i = pivotPosition-1* **to** *1* **do**
  if $\hat{\boldsymbol{G}}[i, pivotPosition]$ **then**
    $\hat{\boldsymbol{G}}[i] = \hat{\boldsymbol{G}}[i] \oplus \hat{\boldsymbol{g}}[i]$ `/* substitute into coefficient matrix */`
    $\hat{\boldsymbol{X}}[i] = \hat{\boldsymbol{X}}[i] \oplus \hat{\boldsymbol{x}}[i]$ `/* substitute into symbol matrix */`

Algorithm 2.6: Backward substitute[2]

Example continued:
In figure 2.8, it can be seen that the values made boldface in $\hat{\boldsymbol{G}}$ can be eliminated in case they are nonzero. Because coding vectors are only inserted in $\hat{\boldsymbol{G}}$ where they have pivot, it will be safe to only eliminate nonzero values in rows above.
In contrast to *ForwardSubstituteToPivot* and *ForwardSubstituteFromPivot* it will not be $\hat{\boldsymbol{g}}$ that is changed, but instead $\hat{\boldsymbol{G}}$ will be altered.

$$\begin{matrix} & & & & & & \text{symbol data} & & & & \\ \begin{bmatrix}0 & 0 & 0 & \mathbf{1} & 0\end{bmatrix} & \oplus & \begin{bmatrix}1 & 0 & 0 & \mathbf{1} & 0\\ 0 & 0 & 0 & \mathbf{0} & 0\\ 0 & 0 & 1 & \mathbf{0} & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 1\end{bmatrix} & \rightarrow & \begin{bmatrix}1 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 1\end{bmatrix} \end{matrix}$$

Figure 2.8: Backward substitute

Insert packet

The function named *InsertPacket* is the last function that is called in algorithm 2.3. This function is called after all undesired nonzero values has been removed both in $\hat{\boldsymbol{g}}$ and $\hat{\boldsymbol{G}}$, so what is left is basically just to insert on the pivot index.

Input: $\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$,pivotIndex
1 $\hat{\boldsymbol{G}}[i] = \hat{\boldsymbol{g}}$ `/* insert into coefficient matrix */`
2 $\hat{\boldsymbol{X}}[i] = \hat{\boldsymbol{x}}$ `/* insert into symbol matrix */`

Algorithm 2.7: InsertPacket[2]

Example continued:
The $\hat{\boldsymbol{g}}$ vector used in the example can now be inserted into the coefficient matrix, and if operations done on coefficients were also done on the symbols it should also be possible to insert $\hat{\boldsymbol{x}}$ into $\hat{\boldsymbol{X}}$.

$$\begin{bmatrix} 0 & 0 & 0 & \mathbf{1} & 0 \end{bmatrix} \rightarrow \overset{\text{symbol data}}{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}}$$

Figure 2.9: Insert packet

Normalize

The function named *Normalize* was not presented in algorithm 2.3 because the example was based on $GF(2)$.

The purpose of the function is to normalize pivot elements in a given coding vector when using higher field. This process is shown by an example below:

$$\tfrac{1}{25} \times \begin{bmatrix} 103 & 198 & 105 & 115 & 81 & 255 \\ 101 & 101 & 28 & 29 & 192 & 152 \\ 215 & 23 & 128 & 122 & 172 & 121 \\ 28 & 246 & 5 & 153 & 33 & 142 \end{bmatrix} = \tfrac{1}{25} \times \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 25 & 162 & 0 \\ 0 & 0 & 231 & 248 \\ 0 & 0 & 0 & 186 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}$$

$$\begin{bmatrix} 103 & 198 & 105 & 115 & 81 & 255 \\ 218 & 218 & 128 & 94 & 182 & 240 \\ 215 & 23 & 128 & 122 & 172 & 121 \\ 28 & 246 & 5 & 153 & 33 & 142 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 76 & 0 \\ 0 & 0 & 231 & 248 \\ 0 & 0 & 0 & 186 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}$$

Figure 2.10: Normalize example

The inverse of 25 is found by looking it up a table. How this works was explained in 2.1. Both a multiplication and division table has been included on the CD that are attached to the project.

$$\frac{1}{25} \mod p(x) = 222$$

All the nonzero values within the coding vector to normalize can be calculated as below:

$$\frac{1}{25} \times 25 \mod p(x) = 222 \times 25 \mod p(x) = 1$$

$$\frac{1}{25} \times 162 \mod p(x) = 222 \times 162 \mod p(x) = 76$$

2.4.2 Delayed decoder

The delayed decoder is described in [2] and is in general working exactly as the basic decoder described in section 2.4.1. The pseudocode for the delayed decoder can be seen in algorithm 2.8. The only difference between the basic decoder and the delayed decoder is that *ForwardSubstituteFromPivot* and *BackwardSubstitute* has been removed in 2.8. These are replaced by *BackwardSubstituteAll* which is called when the decoder has full rank. The algorithm for *BackwardSubstituteAll* can be seen in algorithm 2.9 and will basically just start from the bottom row in $\hat{\boldsymbol{G}}$ and use the coefficients to eliminate nonzero values upwards in $\hat{\boldsymbol{G}}$.

```
   Input: x̂, ĝ
1  pivotPosition = ForwardSubstituteToPivot(x̂,ĝ)
2  if pivotPosition > 0 then
3      InsertPacket(x̂,ĝ,pivotPosition)
4      rank++
5  if rank = g then
6      BackwardSubstituteAll()
7  return rank
```

Algorithm 2.8: Delayed decoder[2]

Backward substitute all

The pseudocode for *BackwardSubstituteAll* is shown in algorithm 2.9.

The purpose of the backward substitute is to eliminate all nonzero values that are not pivot elements in the coefficient matrix, $\hat{\boldsymbol{G}}$. $\hat{\boldsymbol{G}}$ is assumed to be on upper triangular form.

```
   Input: x̂,ĝ
1  for i = symbols to 1 do
2      BackwardSubstitute(x̂, ĝ, pivotPosition)
```

Algorithm 2.9: Backward substitute all[2]

Example: The algorithm can be illustrated by an example as depicted in figure 2.11 below, where both the coefficient matrix and the symbol matrix are shown. This also illustrates how all operations done on the coefficient matrix also has to be replicated on the symbol matrix. First, the last row in $\hat{\boldsymbol{G}}$ will be used to eliminate nonzero values above in the last column. Then the second last row in $\hat{\boldsymbol{G}}$ is used, and so on.

$$
\begin{array}{cc}
\multicolumn{2}{c}{\text{Symbol data}} & & \multicolumn{2}{c}{\text{Coefficients}} \\
\begin{bmatrix}1&0&1&1&0&1\end{bmatrix} \oplus \begin{bmatrix}0&1&0&1&1&1\\1&0&0&0&1&0\\1&1&1&0&0&0\\1&0&1&1&0&1\end{bmatrix} & = & \begin{bmatrix}0&0&0&1\end{bmatrix} \oplus \begin{bmatrix}1&0&1&1\\0&1&0&0\\0&0&1&1\\0&0&0&1\end{bmatrix}\begin{bmatrix}P_1\\P_2\\P_3\\P_4\end{bmatrix} \\
\begin{bmatrix}1&0&1&1&0&1\end{bmatrix} \oplus \begin{bmatrix}1&1&1&0&1&0\\1&0&0&0&1&0\\0&1&0&1&0&1\\1&0&1&1&0&1\end{bmatrix} & = & \begin{bmatrix}0&0&1&0\end{bmatrix} \oplus \begin{bmatrix}1&0&1&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}P_1\\P_2\\P_3\\P_4\end{bmatrix} \\
\begin{bmatrix}1&0&1&1&1&1\\1&0&0&0&1&0\\0&1&0&1&0&1\\1&0&1&1&0&1\end{bmatrix} & = & \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}P_1\\P_2\\P_3\\P_4\end{bmatrix}
\end{array}
$$

Figure 2.11: Backward substitute all

2.4.3 Sparse decoding algorithm

The sparse decoding algorithm will not be described with pseudocode, but it can be visualized with an example. The objective of the algorithm is to use column and row swaps to rewrite the coefficient matrix to be on upper triangular form. The algorithm can substitute two symbols (xor in $GF(2)$) in case it should get stock. Below is an example of the algorithm.

Simple decoding example

Assume that the encoder is given some original symbols and generates coding vectors that results in the encoded symbols in the left most matrix in figure 2.12 below.

$$
\begin{array}{cc}
\multicolumn{2}{c}{\text{Symbol data}} & & \text{Coefficients} & \text{Original symbols} \\
\begin{bmatrix}\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{0}&\mathbf{0}\\1&0&0&0&1\\0&1&0&1&1\end{bmatrix} & = & \begin{bmatrix}\mathbf{0}&\mathbf{1}&\mathbf{0}\\1&1&0\\1&0&1\end{bmatrix} & \begin{bmatrix}0&1&1&0&1\\\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{0}&\mathbf{0}\\0&0&1&1&0\end{bmatrix}
\end{array}
$$

Figure 2.12: Simple example og column/row swaps

The encoder will then transmit the encoded symbols along with the coding vectors in pairs one packet at a time until the matrices are received by the decoder as shown in figure 2.13.

Below the coefficient matrices can be seen numbers counting nonzero values in each column. These will be used to decide which column to swap. To the right of $\hat{\boldsymbol{G}}$ can be seen a vector that is used to keep track of the original symbols, as column swaps in $\hat{\boldsymbol{G}}$ will require these to know where the symbol originally belonged.

To actually get $\hat{\boldsymbol{G}}$ on upper triangular form, the first pivot element should be found. The algorithm proposes that a column with only one nonzero value can form the pivot element if the whole column is swapped to the column where the pivot element is supposed to be.

The first step is therefore to locate a column with only one nonzero coefficient which in this example is the third column. This column should then be swapped with the first column in $\hat{\boldsymbol{G}}$ as this is where the next pivot element is supposed to be. When columns are swapped it is important also to swap the original symbols in order not to change which original symbols were used to generate a corresponding encoded symbol. The swap of the first column and the third column is performed in the transaction from figure 2.13 to figure 2.14. This is also illustrated in figure 2.14.

Next step is to actually move the nonzero coefficient to the correct row. This is done by finding the nonzero coefficient in the pivot column and swap the row with the nonzero coefficient to the row where the pivot element is supposed to be. This is done in the transaction figure 2.14 and 2.14. Figure 2.15 simply illustrates which rows where swapped. Because the i'th coding vector in $\hat{\boldsymbol{G}}$ was used to generate the i'th encoded symbol, the encoded symbols should be swapped correspondingly when swapping rows.

It can now be seen in figure 2.15 that the first pivot element has been placed, and it can be disregarded. This forms a submatrix which is marked in figure 2.15. This submatrix can be considered alone, so the nonzero coefficients in the first row should therefore not be counted anymore.

In figure 2.16, it can be seen that column two and three could be swapped to form the next pivot element. This time, the element is already on the correct position in $\hat{\boldsymbol{G}}$, so there is no need to swap rows this time.

Also, the last coding vector ends with pivot element on the correct position, so the system can actually be solved.

$$\underset{\text{Symbol data}}{\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}} = \underset{\substack{\text{Coefficients} \\ \\ 2 \quad 2 \quad 1}}{\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}} \underset{\text{Original symbols}}{\begin{bmatrix} P_1 \\ P_2 \\ P_3 \end{bmatrix}}$$

Figure 2.13: Example - step 1

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} & 1 & \boldsymbol{0} \\ \boldsymbol{0} & 1 & \boldsymbol{1} \\ \boldsymbol{1} & 0 & \boldsymbol{1} \\ \boldsymbol{1} & 2 & \boldsymbol{2} \end{bmatrix} \begin{bmatrix} \boldsymbol{P_3} \\ P_2 \\ \boldsymbol{P_1} \end{bmatrix}$$

Figure 2.14: Example - step 2

$$\begin{bmatrix} \boldsymbol{0} & \boldsymbol{1} & \boldsymbol{0} & \boldsymbol{1} & \boldsymbol{1} \\ 1 & 0 & 0 & 0 & 1 \\ \boldsymbol{1} & \boldsymbol{1} & \boldsymbol{1} & \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} = \begin{bmatrix} \boldsymbol{1} & \boldsymbol{0} & \boldsymbol{1} \\ 0 & 1 & 1 \\ \boldsymbol{0} & \boldsymbol{1} & \boldsymbol{0} \\ 2 & 1 & \end{bmatrix} \begin{bmatrix} P_3 \\ P_2 \\ P_1 \end{bmatrix}$$

Figure 2.15: Example - step 3

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{1} & \mathbf{0} \\ 0 & \mathbf{1} & \mathbf{1} \\ 0 & \mathbf{0} & \mathbf{1} \end{bmatrix} \begin{bmatrix} P_3 \\ \boldsymbol{P_1} \\ \boldsymbol{P_2} \end{bmatrix}$$
$$\mathbf{1} \quad \mathbf{2}$$

Figure 2.16: Example - Step 4

Solving the system can now be done by hand as illustrated by the equations below. On the computer, a better approach would be to use the function named *BackwardSubstituteAll* that was presented for the delayed decoder.

$$\begin{aligned} x_3 &= P_2 = 11100 \\ x_2 &= P_1 + P_2 \\ x_1 &= P_1 + P_3 \end{aligned}$$

$$\begin{array}{ll} X_2: & 10001 \\ P_2: & 11100 \\ \hline P_1: & 01101 \end{array} \qquad \begin{array}{ll} X_1: & 01011 \\ P_1: & 01101 \\ \hline P_3: & 00110 \end{array}$$

Getting stuck

The decoding is not always going as well as the example above. Sometimes there will not be any columns with only one nonzero value. This is illustrated by the example below, where it can also be seen that symbols from $\hat{\boldsymbol{G}}$ can be used to eliminate all but one nonzero values in the column where the pivot element is supposed to be.

$$\begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix} \oplus \overset{\text{coefficients}}{\left[\begin{array}{c:ccc} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{array}\right]} \rightarrow \overset{\text{coefficients}}{\left[\begin{array}{cc:cc} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{array}\right]}$$

Even after decoding was stuck, it may be stuck again. This time, all but one nonzero value in the pivot column has to be eliminated again using substitute. This is illustrated in the figure below. This time, the substituted coding vector will however result in a null vector which means that there was two dependent coding vectors in $\hat{\boldsymbol{G}}$.

$$\begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix} \oplus \left[\begin{array}{cc:cc} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{array}\right] \rightarrow \left[\begin{array}{cc:cc} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

After dependent coding vectors has been totally eliminated, the coefficient matrix will be in a state where most of the matrix is already on pivot form, but receiving a new symbol may not be that easy to insert. Assume that a packet is received with the following coding vector:

$$\begin{bmatrix} 1 & 1 & 0 & 1 \end{bmatrix}$$

Inserting that will result in the following coefficient matrix:

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

As it can be seen, the whole triangular form is corrupted. Another issue is that the coding vector couldn't be inserted that easily in a real implementation. This is because columns were swapped in the coefficient matrix during decoding. The coefficients in the coding vectors will therefore have to be swapped accordingly to how columns were swapped throughout the whole decoding of a generation.

Chapter 3

Design

3.1 Evaluation of decoding methods

In this section, the existing decoding algorithms and steps will be evaluated.

The basic decoder has the advantage of using on-the-fly decoding, which means that it decodes a symbol as much as it possible can. This has the advantage over other the delayed decoding method that it does have a significant delay in the end after all coded symbols have been received and need to be coded. The disadvantage of is however that it comes with a cost of using forward and backward substitute every time a symbol is received. This has the disadvantage that forward substitute may introduce more nonzero values in the coding vector, and backward substitute may result in more nonzero values in the coefficient matrix.

The delayed decoding algorithm tries to prevent that more nonzero values may be introduced by simply delaying the backward substitute until the decoder has full rank. That way, nonzero values can be eliminated without introducing new nonzero values. This algorithm will however introduce a delay in the end where all nonzero values in the upper triangle of the coefficient vector should be eliminated.

The sparse decoding algorithm can work both on-the-fly or as delayed decoding. This work will however focus entirely on creating delayed decoders as they are expected to require least finite field operations and therefore should result in higher throughputs. The idea behind using column and row swaps is that they doesn't introduce more nonzero values, but it will however require more bookkeeping. The hope is therefore that the bookkeeping is less significant than the gain of using column/row swaps.

In this project it is therefore intended to develop and implement decoding algorithms that are based on the sparse approach. As mentioned, it will focus on the implementation of a delayed decoder. This means that an approach could be to fill the symbol and coefficient matrices with data, and then attempt to solve it using column/row swaps.

3.2 Creating decoding algorithm

This section will attempt to split the sparse decoding algorithm into steps that can be implemented.

It is know form previously that all operations done on the coefficient matrix should simply be replicated on the symbol matrix. The symbol data can therefore be disregarded, so only the coefficient matrix are considered.

This is done in the decoding example in figure 3.1, where an arbitrary coefficient matrix is given and solved using column/row swaps. Each row of matrices perform first a column swap, then a row swap if necessary, and finally show the resulting matrix.

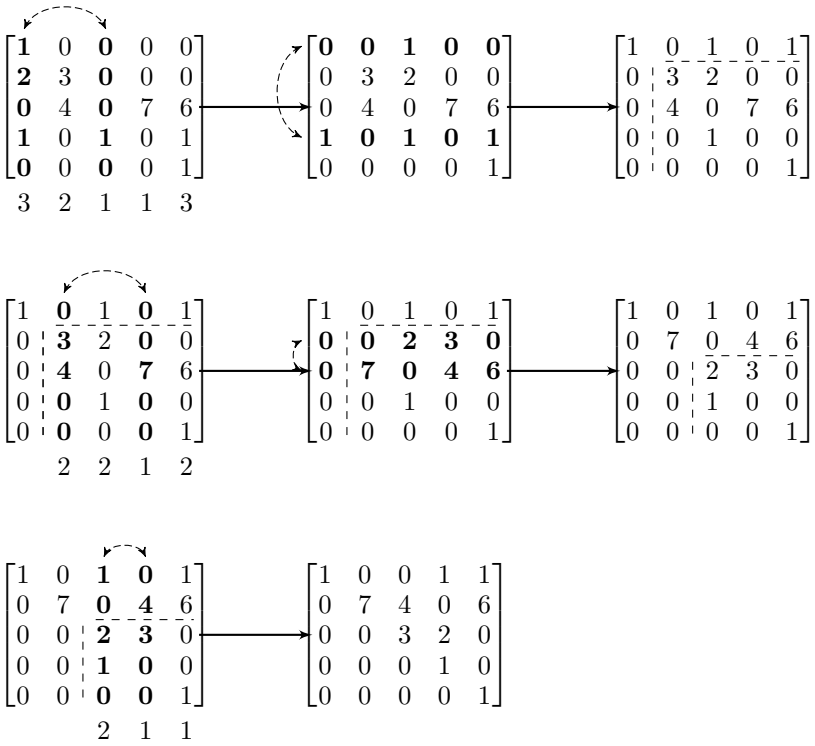

Figure 3.1: Example of decoding in some arbitrary finite field

The following steps can be derived from the figure:

Step 1: Swap columns

- Count elements in each column
- Swap first column that has one non-zero element with left most column in sub-matrix

Step 2: Swap rows

- Swap row containing left most columns non-zero with upper most row

Step 3: Update nonzero counts (Decrement by one for nonzero elements in pivot row)

Step 4: Increment rank by one

Step 5: Redo Steps until there are no columns with one non-zero element

Step 6: Solve remaining sub-matrix using another method or await new linear combinations

The disadvantage of the above method is that it works mostly on column element which are slow in Kodo because memory are aligned according to encoding vectors (rows). This means that iterating through column elements requires lots of random reads in memory which are much slower than sequential reads that can be done on the rows.

This may however not be that bad, because swapping the columns doesn't require any operations to be performed on the encoded symbols like a substitute for example. Substitute has to loop over all finite field elements in the encoded symbols where column swaps has to loop over all finite field in the columns. This column swap may then be faster than for example a substitute because the number of symbols are often smaller than the symbol size. The symbol size is illustrated in figure 2.2 of the original symbol, which has the exact same size as encoded symbols will have.

3.2.1 Finding which columns to swap

As it appear in figure 3.1, the pivot column is swapped with the first column with only one nonzero element. This is easily found by a linear search in a list counting nonzero elements in the columns.

Another option could be to always find the column with only one nonzero element. This can also be done with a linear search, but this time it simply starts from the end in the list counting nonzero elements in the columns.

3.2.2 Finding which row to swap

As it can be seen in the figure 3.1, the objective is to find and swap the single element in the pivot column to the pivot row. This can simply be done by a linear search in the pivot column.

3.2.3 Extending algorithm

The objective is of course to have an algorithm that works even when dependent symbols are received. This section will therefore explain how the algorithm can be extended.

The first problem of getting stuck can simply be solved by first searching for one nonzero element in the list counting nonzero elements. If nothing is found, then simply search for the column with least nonzero elements. When such a column is swapped, it is known to contain multiple nonzero elements. This can easily be solved by finding one of the rows, and substitute with all vectors with nonzero elements in the pivot column.

Performing substitute on a symbol is done with the intention of making an element zero, so it should be remembered to update the list counting nonzero elements. This should be done from the pivot position to the end of the matrix.

3.2.4 Receiving new packets

As it was explained in 2.4.3, it may be a problem to insert symbols into the coefficient matrix after it has already been put on triangular form.
’ This can however be solved by doing the exact same swaps that was performed on the coefficient matrix, so when a new symbol is received, and columns have been swapped, the same exact coefficients should be swapped in the same order. Then the newly received symbol will have its coefficients on the correct places.

The second problem was that the triangular form was destroyed if new coding vectors were just inserted in the end of the coding matrix. This problem can however be solved by the same ideas that was used in the method *ForwardSubstituteToPivot*. Now that the coding coefficients are on triangular form, they can simply be used to eliminate nonzero elements in the newly received coding vector.

These ideas are the fundamental ideas behind the decoding algorithm developed in 3.3.

3.3 Decoding algorithm 1

In this section will the first decoding algorithm be presented. The algorithm is based on the sparse decoding method, where column/row swaps are used. It fills the symbol matrix and the coefficient matrix with data, and then after the matrices are filled up, column and row swaps are used. Substitute will be used in case it gets stuck. When the matrix has been put on triangular form, the algorithm will start to swap coefficients after which nonzero coefficients within the newly received symbol are eliminated. The element is then inserted as the next in the bottom of the matrices.

The algorithm always starts in *Decode symbol* which is the function that is called when a new symbol is received from the encoder.

This algorithm will be referenced to as *Swap Substitute* which will be shortened to *SS*.

```
   Input: x̂, ĝ
1  if rank > 0 then
2      SwapCoefficients(ĝ)
3      ForwardSubstituteToRank(x̂, ĝ)
4  InsertPacket2(x̂, ĝ)
5  if insertIndex = symbols then
6      DecodeSymbols()
```

Algorithm 3.1: Decode symbol

```
   Input: x̂,ĝ
1  Ĝ[i] = ĝ
2  X̂[i] = x̂
3  insertIndex++
4  IncrementNonzeros(ĝ, rank)
```

Algorithm 3.2: InsertPacket2

```
1  for pivotIndex = rank to symbols do
2      j = LastSmallestNonzeroIndex(pivotIndex)
3      if j ≥ symbols then
4          rank = pivotIndex
5          insertIndex = pivotIndex
6          return
7      SwapColumns(pivotIndex, j)
8      i = FirstNonzeroIndex(pivotIndex)
9      SwapRows(pivotIndex, i)
10     if field != GF(2) then
11         Normalize(x̂, ĝ, pivotIndex)
12     ForwardSubstitute(pivotIndex);
13     DecrementNonzeros(ĝ, pivotIndex)
14 BackwardSubstituteAll();
15 ReverseSwaps()
16 rank = symbols
```

Algorithm 3.3: Decode Symbols

```
   Input: x̂,ĝ
1  for i = 0 to rank do
2      if ĝ[i] then
3          if Ĝ[i, i] = 1 then
4              ĝ = ĝ[i] ⊕ Ĝ[i]                    /* substitute into vector */
5              x̂ = x̂[i] ⊕ X̂[i]                    /* substitute into symbol */
```

Algorithm 3.4: Forward substitute to rank

```
   Input: pivotIndex
1  while nonzeroCount[pivotIndex] > 1 do
2      if Ĝ[i, pivotIndex] then
3          decrementNonzeros(Ĝ[i], pivotIndex)      /* subtract vector nonzeros */
4          Ĝ[i] = Ĝ[i] ⊕ Ĝ[pivotIndex]               /* substitute into vector */
5          X̂[i] = X̂[i] ⊕ X̂[pivotIndex]               /* substitute into symbol */
6          incrementNonzeros(Ĝ[i], pivotIndex)          /* add vector nonzeros */
```

Algorithm 3.5: Forward substitute

3.4 Decoding algorithm 2

In this section, the second algorithm will be presented. This algorithm was made to test how fast it would be to only substitute in case the algorithm got stuck. This means that it would not substitute symbols inserted after the matrix has been written to triangular form.

It will however require two lists to count the number of nonzero elements in the columns, because it starts all over and assumes the pivot index to be zero each time the coefficient matrix has been filled.

All the places where functions ends with ”Both” means that both the original list counting nonzero coefficients and the copy of this list should be updated in the exact same way, so the functions ending on ”Both” only have this additional line where the copy of the list counting nonzero coefficients are also updated.

This algorithm will be referenced to as *Swap Restart* which will be shortened to *SR*.

```
   Input: x̂, ĝ
 1 if swapInputSymbols then
 2     SwapCoefficients(ĝ)
 3 InsertPacket2(x̂, ĝ)
 4 if insertIndex = symbols then
 5     PushNonzeros()                          /* Create copy of nonzeros */
 6     DecodeSymbols()
 7     PullNonzeros()                        /* Retrieve copy of nonzeros */
```

Algorithm 3.6: Decode symbol

```
 1 for pivotIndex = 0 to symbols do
 2     j = LastSmallestNonzeroIndex(pivotIndex)
 3     if j ≥ symbols then
 4         insertIndex = pivotIndex
 5         swapInputSymbols = true
 6         return
 7     SwapColumnsBoth(pivotIndex, j)
 8     i = FirstNonzeroIndex(pivotIndex)
 9     SwapRows(pivotIndex, i)
10     if field != GF(2) then
11         NormalizeBoth(x̂, ĝ, pivotIndex)
12     ForwardSubstituteBoth(pivotIndex);
13     DecrementNonzeros(ĝ, pivotIndex)
14 BackwardSubstituteAll();
15 ReverseSwaps()
16 rank = symbols
```

Algorithm 3.7: Decode Symbols

3.5 Increment/Decrement nonzeros

The most called functions in the sparse decoding algorithms are the functions that maintain the list counting nonzero elements in each of $\hat{G}$'s columns. It is therefore essential that these functions work as fast as possible.

The functions are used in pairs where *increment_nonzeros* increase the nonzero count with one if the coding vector contains a nonzero, and the function called *decrement_nonzeros* decrement the nonzero count of a column with one.

The two functions will be tested as part of the two decoding algorithms implemented in this project. They will be referenced to as the *Branching* implementations, because they have an if statement that may cause the processor to guess whether it should increment/decrement the nonzero count by one or whether they should do nothing. A pseudocode for these two function can be seen below.

When the decoding algorithms are testet, they will add ”-B” to the end of the particular algorithm, so the *Swap Substitute* method would be called *Swap Substitute Branching*, which will be shortened to *SS-B*.

```
   Input: ĝ, pivotIndex
1  for i = pivotIndex to symbols do
2      if ĝ[i] then
3          nonzeros[i] = nonzeros[i] + 1
```

Algorithm 3.8: Increment nonzeros

```
   Input: ĝ, pivotIndex
1  for i = pivotIndex to symbols do
2      if ĝ[i] then
3          nonzeros[i] = nonzeros[i] - 1
```

Algorithm 3.9: Decrement nonzeros

During the first simulations, it was seen that the implementations above performed very poorly in $GF(2)$. This can also be seen in section 4.2.

It was therefore decided to try profiling the *SS-B* algorithm to see why the implementations performed that poorly in $GF(2)$. The result for the profiling can be seen figure 3.2, which shows in percentage how much of the time a program spends in a given function. As it can be seen from the figure, it shows that *increment_nonzeros* and *decrement_nonzeros* with branching takes almost 40 percent of the time during decoding. Two new implementations of the functions were therefore implemented.

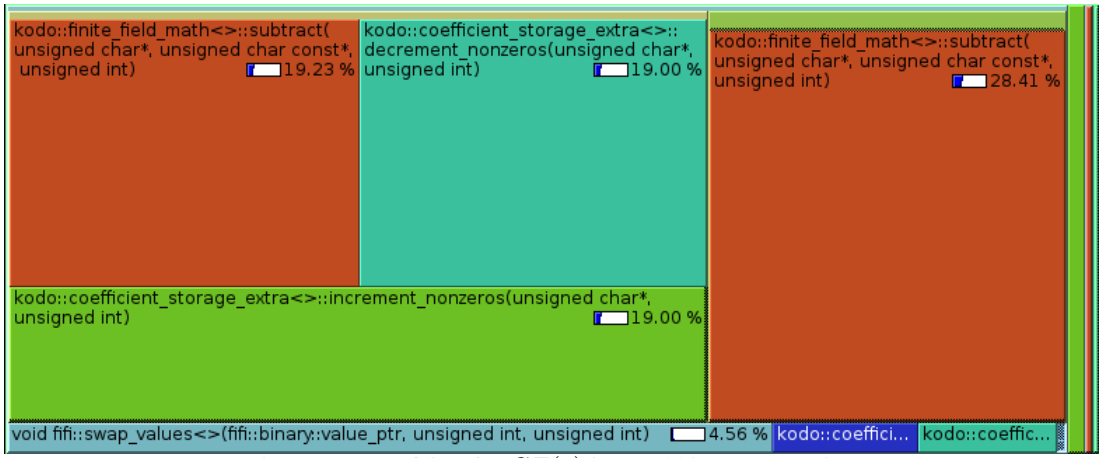

Figure 3.2: Profiling for $GF(2)$ in an old implementation

3.5.1 Optimization - Prevent branching

As described above, the implementations of *increment_nonzeros* and *decrement_nonzeros* took almost 40 percent of the time during decoding. The two replacement functions were therefore implemented. The thought was that these would perform better because they don't create branches which require the computer to guess whether an if statement will be true or false. The throughput improvements can be seen in section 4.2, where these two implementations are just referenced to as *SS* and *SR*, as they will be used by default.

```
   Input: ĝ, pivotIndex
1  for i = pivotIndex to symbols do
2      nonzeros[i] = nonzeros[i] + (ĝ[i] > 0)
```

Algorithm 3.10: Increment nonzeros

```
   Input: ĝ, pivotIndex
1  for i = pivotIndex to symbols do
2      nonzeros[i] = nonzeros[i] - (ĝ[i] > 0)
```

Algorithm 3.11: Decrement nonzeros

It was seen that these implementations did improve the throughput in $GF(2)$, but what if it was not necessary to loop through the coding vectors twice. The next method will show that.

3.5.2 Optimization - Updating

GF(2):

This method is called updating because it doesn't both decrement and increment. Instead, it does both of them in one loop over the coefficients. When the method is simulated, it will be referenced to as *SS-U* and *SR-U*. The updating method do not stand alone, because it isn't smart to use when a new symbol arrives, and the list counting nonzero elements in each row only has to be incremented to. Also when a symbol is removed, it may be faster to use one of the methods described above.

The naming will therefore mean:

- *SS-U*: *Swap Substitute* Without branching and updating
- *SS-BU Swap Substitute* With branching and updating
- *SR-U Swap Restart* Without branching and updating
- *SR-BU Swap Restart* With branching and updating

To understand the method, a truth table has been made where $\hat{\boldsymbol{g}}[i]$ is the i'th coefficient in the coding vector that may change $\hat{\boldsymbol{h}}[i]$ is the i'th coefficient in another coding vector

What the function finds out is then how g changed, and thereby either increment, decrement, or do nothing on the list counting nonzero elements.

$\hat{\boldsymbol{g}}[i]$	$\hat{\boldsymbol{h}}[i]$	$\hat{\boldsymbol{g}}[i] \oplus \hat{\boldsymbol{h}}[i]$	$(\hat{\boldsymbol{g}}[i] \oplus \hat{\boldsymbol{h}}[i]) - \hat{\boldsymbol{g}}[i]$
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	-1

From the truth table it can be seen. Assuming that $\hat{\boldsymbol{g}}[i]$ start equal to zero and the subtraction with $\hat{\boldsymbol{h}}[i]$ results in the same. Then add nothing as the value in $\hat{\boldsymbol{g}}[i]$ stays the same. If the coefficient in $\hat{\boldsymbol{g}}[i]$ was 0 and would become 1 by subtraction (xor), then the number of nonzero elements in the i'th column will be increased by one whenever substitution are performed. Finally, if $\hat{\boldsymbol{g}}[i]$ are 1 and becomes 0, then there will be one less nonzero element in the i'th column after g.

Input: $\hat{g}$, $\hat{h}$, pivotIndex
1 **for** i = *pivotIndex* **to** *symbols* **do**
2 nonzeros[i] = nonzeros[i] + $(\hat{\boldsymbol{g}}[i] \oplus \hat{\boldsymbol{h}}[i]) - \hat{\boldsymbol{g}}[i]$

Algorithm 3.12: Update nonzeros

The performance of this implementation are benchmarked in section 4.2.

For all fields:

In can in principle be done for other fields, but the instead of xor, the arithmetics becomes slower as the field size increases in size

Chapter 4

Simulation

This chapter will present the simulation result performed in this project.

4.1 Encoder simulations

This section will attempt to verify that an encoder's throughput will increase more and more as the sparsity is increased. This is expected because sparse coding vectors combine less symbols.

Figures 4.1 and 4.2 both illustrates throughput versus density of coding vectors with generations of 128 kB and varying density in $GF(2)$ and $GF(2^8)$ respectively. The same decreasing throughput also applies for different densities.

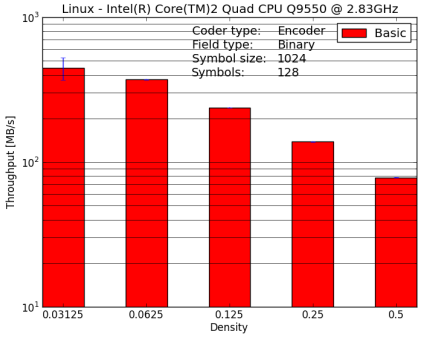

Figure 4.1: Encoder throughput in $GF(2)$

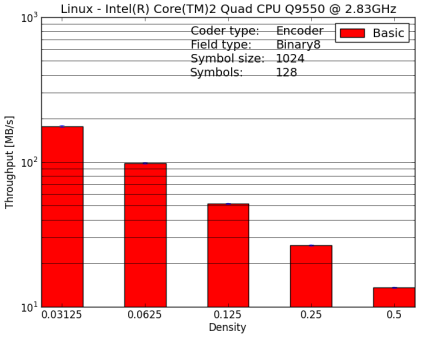

Figure 4.2: Encoder throughput in $GF(2^8)$

4.2 Maintaining nonzero counts

This section will contain throughput measurements of the new decoding algorithms each using different ways to maintain the list counting nonzero elements in each column.

The purpose of these simulations are to decide which implementation, described in 3.5, will perform better in a real implementation.

Originally, simulations were made to exclusively benchmark the three methods that maintains the list counting nonzero elements, but the results were very inconclusive as the compiler optimized differently depending on the context. This led to varying results.
It was therefore concluded that even a detailed assembler examination of the output generated by the compiler wouldn't even be useful as the output in a real implementation would be totally different.

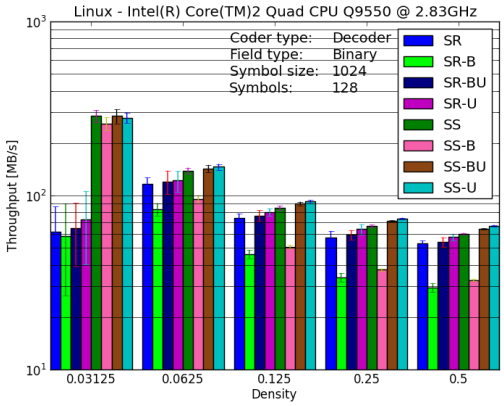

Figure 4.3: Maintaining nonzero counts in $GF(2)$ (128 symbols)

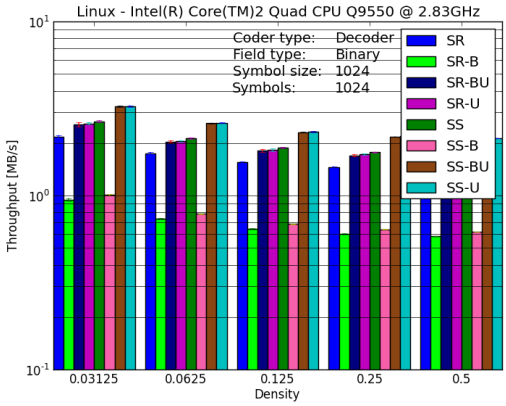

Figure 4.4: Maintaining nonzero counts in $GF(2)$ (1024 symbols)

From the figures above for $GF(2)$, it can be concluded that *SS-B* and *SR-B* are slowest in all cases. It can also be seen that *SS-U* and *SR-U* generally performs better than the other implementations. The methods *SS-BU* and *SR-BU* also performs well, but that is expected to be because the branching method is not used very much and it is therefore the updating implementation that is most significat on the results.

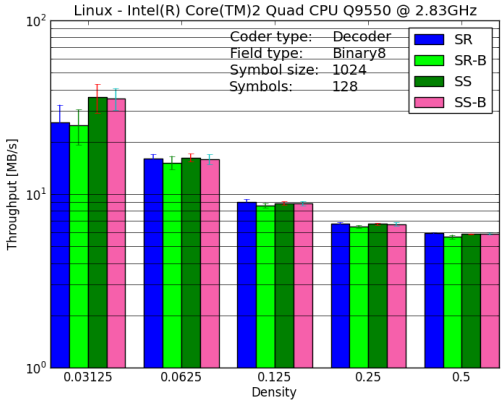

Figure 4.5: Maintaining nonzero counts in $GF(2^8)$ (128 symbols)

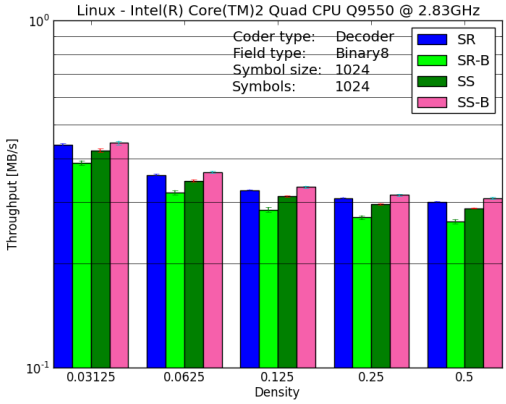

Figure 4.6: Maintaining nonzero counts in $GF(2^8)$ (1024 symbols)

In $GF(2^8)$, it seem that the branching implementation seem to perform better than the other implementations. This is quite unexpected and may be because the compiler do some optimizations steps to either prevent the branching or because it fails to optimize the *SS* for some reason.

Finally what can be concluded from all the graphs in this section is that $SwapSubstitute$ generally performs better than $SwapRestart$

4.3 Sweeping densities

This section will be used to get a general overview of the throughput performance of the algorithms developed and implemented in this project. They will be measured against the existing algorithms that are implemented in Kodo to get a reference of how well they perform.

As described in section 2, the sparse coding vectors can be generated with different densities and different distributions. These densities can be specified depending on the application, so if the application can afford lots of non-innovative packets, it would specify the density to be very low. If non-innovative packets could not be afforded, it would choose the transmit very dense packets.

It is therefore of great interest how well the algorithms perform in all ranges. It has however been decided to only simulate with densities up to 0.5, which means that there is on average 50 percent nonzero values in coding vectors. The range that are intended tested is therefore all densities lower than 50 percent probability of nonzero coefficients. It is however impractical to have densities too close to zero because null vectors would be pointless to transmit.

It has therefore been chosen to measure the throughputs with the following densities:

$$d = \left[\frac{1}{128} \quad \frac{2}{128} \quad \frac{4}{128} \quad \frac{8}{128} \quad \frac{16}{128} \quad \frac{32}{128} \quad \frac{64}{128}\right]$$

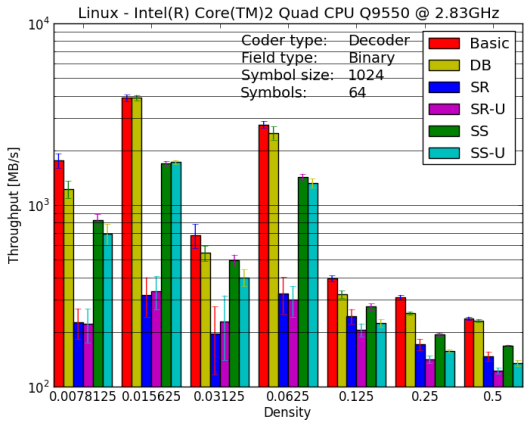

Figure 4.7: Throughput sweep in $GF(2)$ (64 symbols)

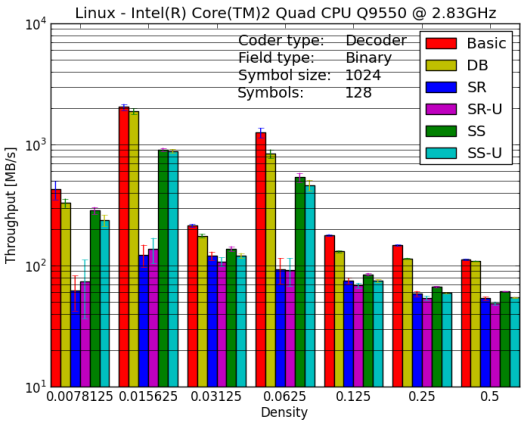

Figure 4.8: Throughput sweep in $GF(2)$ (128 symbols)

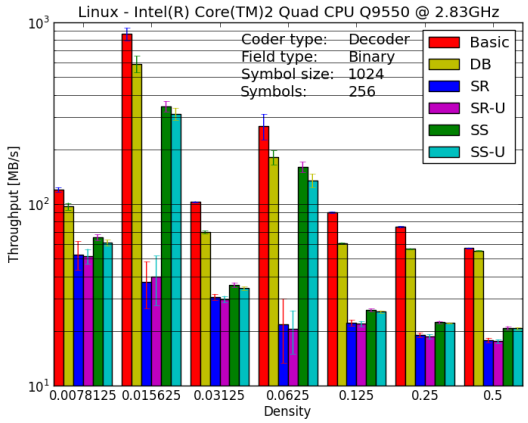

Figure 4.9: Throughput sweep in $GF(2)$ (256 symbols)

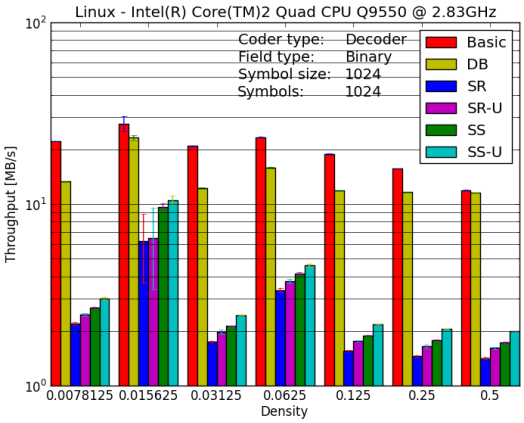

Figure 4.10: Throughput sweep in $GF(2)$ (1024 symbols)

From the graphs above, it can be seen that the Basic decoding algorithm seem to be best in almost all measurement which indicates that the bookkeeping in the *SS* and *SS-U* may require more resources than what is gained by their presence. It can also be seen that *SS* and *SS-U* almost provide same throughput, but *SS-U* does seem to be faster when the number of symbols increase. *SS* seem faster for less symbols. Generally, it is seen that *SS* and *SS-U* follow throughputs of the decoders implemented in Kodo pretty well. They all seem to increase/decrease together. *SR* and *SR-U* do however not seem to always follow the other implementations, which is most likely because they are very sensitive to dependent coding vectors, because they completely restarts and starts all over each time the coefficient matrix, $\hat{\boldsymbol{G}}$, has been filled with symbols and one or more coding vectors are eliminated. It appears to have a very big overhead when it restarts.

The reason throughput measurements seem to fluctuate on the four smallest density measurements may be caused by the following two factors

- Probability of a packet to be innovative
- Density

The two peaks may be caused by the right combinations of being exactly as sparse as possible and still be innovative, but this is uncertain without a detailed analysis of what happens during decoding.

The following graphs illustrates the throughput results for $GF(2^8)$

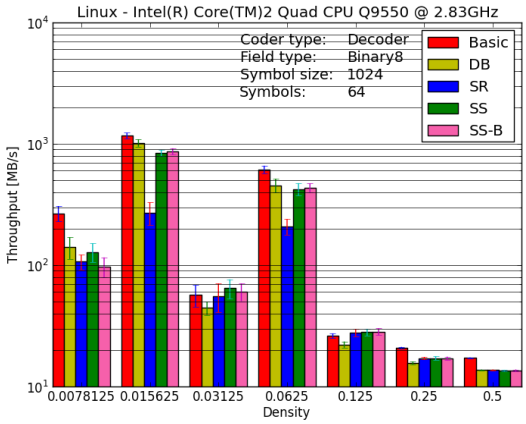

Figure 4.11: Throughput sweep in $GF(2^8)$ (64 symbols)

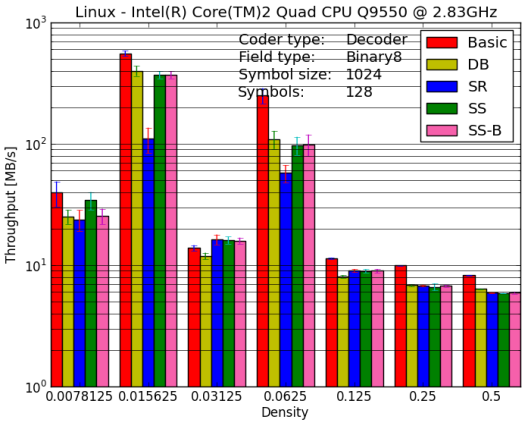

Figure 4.12: Throughput sweep in $GF(2^8)$ (128 symbols)

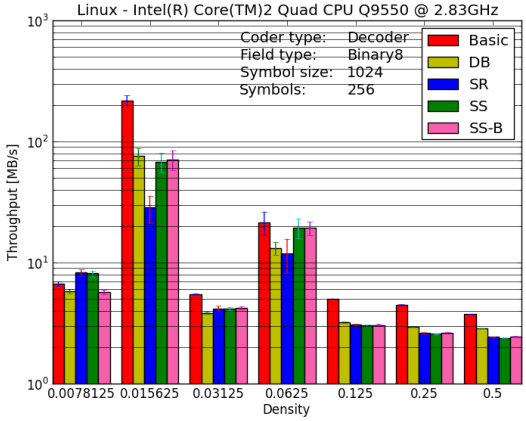

Figure 4.13: Throughput sweep in $GF(2^8)$ (256 symbols)

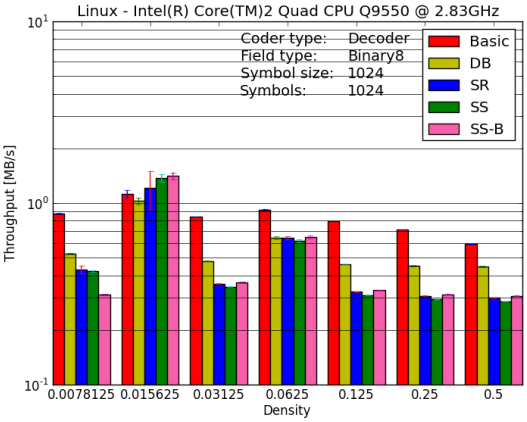

Figure 4.14: Throughput sweep in $GF(2^8)$ (1024 symbols)

As it can be seen, the graphs seem to have the same tendencies as for the field $GF(2)$. The difference is however that SS and $SS-U$ in some cases seem to perform better than the algorithms implemented in Kodo.

There doesn't appear to be a tendency of when *SS* and *SS-B* perform better, but if the densities or written as the average number of nonzero values instead, it may be that there are some tendencies depending on the average number of nonzero values in coding vectors. Table 4.1 shows the densities and the number of nonzero values where *SS* and *SS-B* perform better than the algorithms implemented in Kodo.

Symbols	Densities	avg. nonzero values
64	$\frac{4}{128}$, $\frac{16}{128}$	2,8
128	$\frac{4}{128}$	4
256	$\frac{1}{128}$	2
1024	$\frac{2}{128}$	16

Table 4.1: Average nonzero values in coding vectors

As seen in table 4.1, the average number of nonzero values in coding vectors are actually very low. It would therefore be obvious to make simulations depending on the average number of nonzero values in coding vectors. This is done in section 4.4.

4.4 n nonzeros

In 4.3, it was seen that *SS* and *SS-U* seemed to perform better than the existing implementations at specific densities. These densities may be dependent on the number of nonzero values in the coding vectors generated. This section will therefore present simulations of the throughput versus average number of nonzero values in coding vectors.

The graphs in figure below show simulation data for the field, $GF(2)$, where the density is given as the average number of nonzero values in the coding vectors.

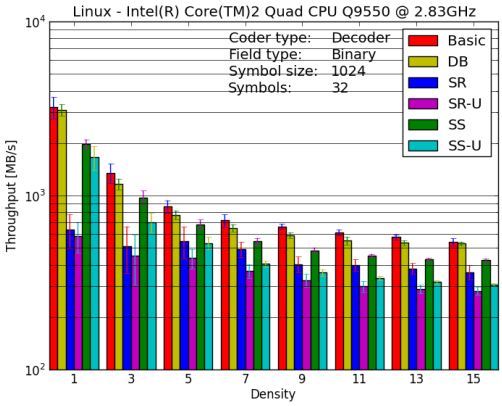

Figure 4.15: Throughput of n nonzero values in $GF(2)$ (32 symbols)

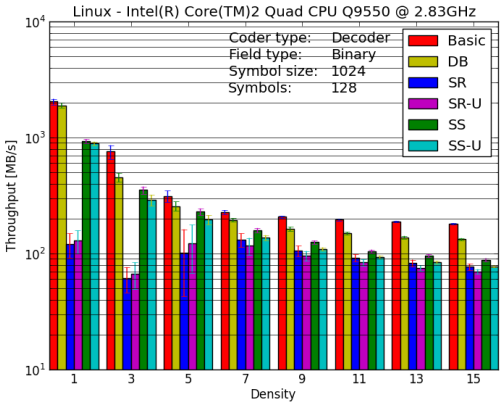

Figure 4.16: Throughput of n nonzero values in $GF(2)$ (128 symbols)

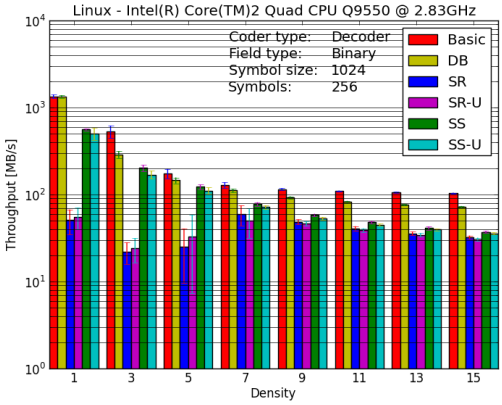

Figure 4.17: Throughput of n nonzero values in $GF(2)$ (256 symbols)

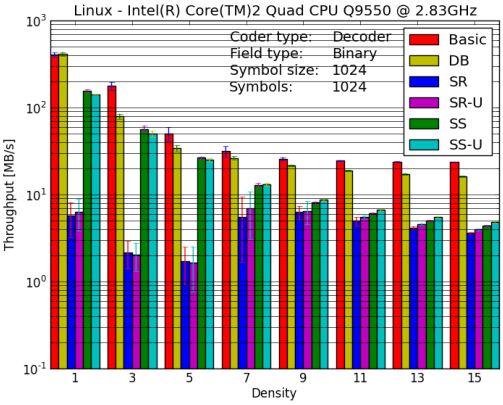

Figure 4.18: Throughput of n nonzero values in $GF(2)$ (1024 symbols)

The simulation has also been performed for the field, $GF(2^8)$ which can be seen below:

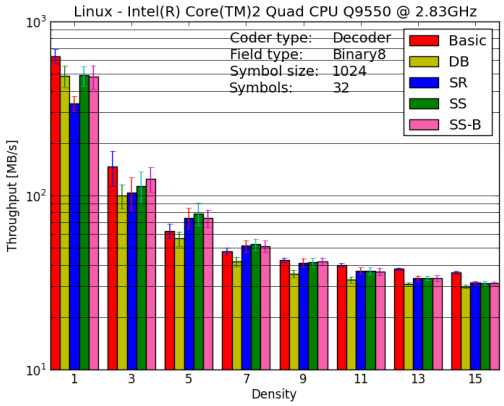

Figure 4.19: Throughput of n nonzero values in $GF(2^8)$ (32 symbols)

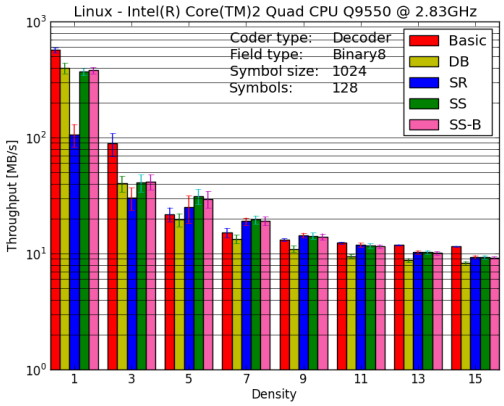

Figure 4.20: Throughput of n nonzero values in $GF(2^8)$ (128 symbols)

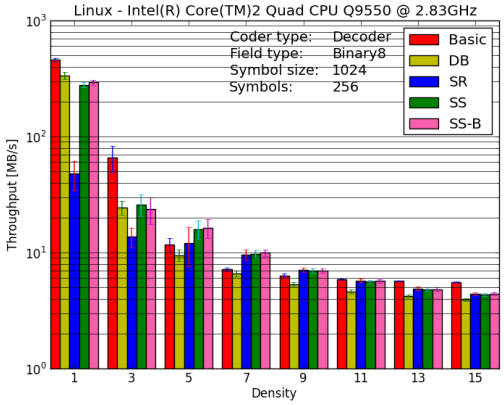

Figure 4.21: Throughput of n nonzero values in $GF(2^8)$ (256 symbols)

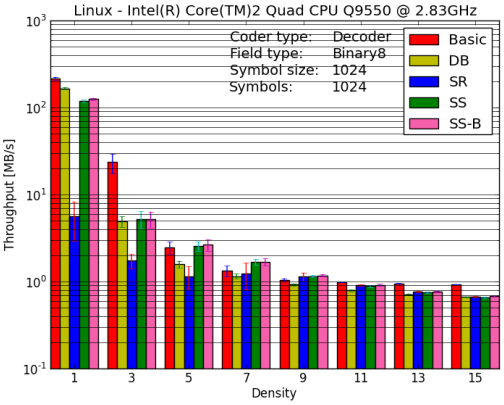

Figure 4.22: Throughput of n nonzero values in $GF(2^8)$ (1024 symbols)

Chapter 5

Conclusion and Assessment

In this project it was investigated whether a sparse decoding algorithm could be developed and implemented to improve decoding throughput for sparse codes. To test it, two sparse decoding algorithms were developed and implemented based on a proposed sparse decoding algorithm and two already existing decoding algorithms.

From the simulations, it could be seen that, it was not possible to beat the basic decoding algorithm in any cases in $GF(2)$. This is most like due to the fact that calculations in $GF(2)$ are extremely fast, so it might actually be that less calculations are performed in the *Swap Substitute* algorithm, but the gain was not enough to compensate for the time spent maintaining a list of nonzero values, swapping columns, and keeping track of the original symbols duing decoding of the generation.

All of the above was however different when using $GF(2^8)$. Here it seemed that *SS* and *SS-B* actually outperform the basic decoding algorithm at specific ranges with a generally minor margin, but with quite significant margins in few cases. This is likely due to the faster memory access when applying $GF(2^8)$, where fields are bytes instead of individual bits. In $GF(2)$, a lot of time may be wasted on bit masking compared to $GF(2^8)$ where bytes are accessed natively. Another cause may be the finite field arithmetics which are significantly faster in $GF(2)$ compared to $GF(2^8)$ which would favor the algorithm with least finite field operations.

In attempts to optimize throughput performance for $GF(2)$ a various of different optimizations were attempted. These optimizations were based on the results obtained by profiling the decoders. The results could then clearify which functions were most timeconsuming, so problems could be narrowed down to particular functionalities that performed poorly. This led to the *increment_nonzeros* and *decrement_nonzeros*, but even after improving them, it would still appear that time might have been better invested in improving the overall algorithm. This is based on the huge difference that can be seen between *Swap Substitute* and *Swap Restart*.

If it was really desired to squeeze more throughput out of the implementations, it would definately be possible.

5.1 Future work

- If there had been more time, some essential graphs could be made. It could for example be a graphs showing the gain of the algorithms compared to the implementations in Kodo. That

way, it could better be concluded how much throughput improvement the algorithms done in this project could provide and at which densities.

- Change the way coding vectors are stored. One could try to only store the indexes and values of each nonzero coefficient. That way, it may be faster to do all the bookkeeping.
- Because algorithms are implemented in kodo, they are portable to a variaty of platforms and devices such as windows, mac, linux, android, and more. If there had been more time, the implementations could also be tested on mobile devices

Bibliography

[1] Arash S. Badr and Chres W. Sørensen. Evaluation of Sparse Network Coding, 2012.

[2] Janus Heide, Morten V. Pedersen, and Frank H. P. Fitzek. Decoding algorithms for random linear network codes. In Vicente Casares-Giner, Pietro Manzoni, and Ana Pont, editors, *NETWORKING 2011 Workshops*, volume 6827 of *Lecture Notes in Computer Science*, pages 129–136. Springer Berlin Heidelberg, 2011.

[3] Soheil Feizi, Daniel E. Lucani, and Muriel Médard. Tunable Sparse Network Coding for Multicast Networks. 2013.

[4] William Stallings. *Cryptography and Network Security*. Prentice Hall, 5 edition, January 2010.

[5] Christina Fragouli, Jean-Yves Le Boudec, and Jörg Widmer. Network Coding: An Instant Primer. *ACM SIGCOMM*, 2006.

Appendices

Appendix A

Content of the CD

- Multiplication and division tables for $GF(2^8)$
- Graphs
- Code