Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

DNA Data Storage using Hamming and Reed-Solomon Codes

Thesis submitted to the
Delft Institute of Applied Mathematics
in partial fulfillment of the requirements

for the degree of

BACHELOR OF SCIENCE
in
APPLIED MATHEMATICS

BY

Eva Slingerland

Delft, Netherlands
June 2019

BSc thesis Applied Mathematics

"DNA Data Storage using Hamming and Reed-Solomon Codes"

(Dutch title: "DNA Data Opslag met Hamming en Reed-Solomon Codes")

E. Slingerland

4572580

Delft University of Technology

Supervisor

Dr.ir. J. H. Weber

Thesis committee

Dr. B. van den Dries

Dr. D. C. Gijswijt

June, 2019

Delft

Preface

This thesis is the final step for the Bachelor Applied Mathematics at the Delft University of Technology. Like most high schools, my school required students to do a small research in one of their profile courses. This is where my interest for the topic of this thesis started, since I decided to research cryptographic methods. More specifically, I looked into the story of Alan Turing and his contributions to the Second World War. This interest made me attend a symposium called 'The Key to your Privacy: the Science behind Cyber Security' in the first year of my bachelor. My interest grew more and more and I chose to follow the elective course called Applied Algebra: Codes and Cryptosystems, which taught me more about the mathematics behind those cryptographic methods and about coding theory. Therefore, when I heard that I could do my thesis on the topic of coding theory, I immediately said yes. Furthermore, the research was combined with another subject that I have always found really interesting: DNA, which I had only come across during my biology classes, though it was one of my favourite topics there.

Before I started, I had never heard of the opportunity to save data on DNA. However, when I began reading more about the topic, it made sense. Within our society, a lot of data is produced. Therefore, it seems logical that problems will occur if people try to save all of it because servers and computers have a limited storage capacity. DNA has many advantages, of which the most important one is that it can store a lot of information , thus it is an obvious choice. Moreover, DNA based data storage is a topic that is currently being researched a lot. For example, the paper from Takahashi et al. in [1], that I mostly focused on, was published in the beginning of this year. So even after finishing this thesis, I will definitely keep an eye out for new developments in this field.

Lastly, I would like to thank my supervisor J. Weber for his great guidance. He gave me a lot of feedback during the writing process and helped me out whenever I did not understand something. Finally, I would like to express my gratitude to the thesis committee for reading and reviewing this thesis.

Eva Slingerland
Delft, June 2019

Abstract

Nowadays, enormous amounts of data are produced on a daily basis. Whether it is a cute family picture, a funny cat video, or a scientific paper, all of the data is stored. The challenge in data storage nowadays is about finding a way to store a lot of data, in such a way that it will stay preserved for many years without too much maintenance and such that if errors occur, the data can still be retrieved. DNA is a great choice for this, since the DNA from extinct species that lived 10,000 years ago can still be retrieved, it does not need maintenance and it is estimated that it can store 5 PB per gram [2]. However, in reading and writing DNA, substitution, insertion and deletion errors occur, so the data needs to be protected against these errors. Therefore, several coding methods have already been invented and researched. Takahasi et al. [1] designed a full automated system for writing, storing and reading data, which consisted only of the word *hello*, using DNA.

This thesis focuses on the coding method used by [1], namely a Hamming code, and compares it to the implementation of a DNA based Reed-Solomon code, applied to the same data. An analysis is made based on the net information density, GC-weight, homopolymer runs and the error detection and correction properties. As expected, there is a trade-off between the net information density and the error detection and correction properties. Although the net information density of the Reed-Solomon code is lower, it can correct more errors and it has the potential of also being applied to a bigger data set.

Contents

1 Introduction

The beginning of this thesis will discuss why the topic of DNA based data storage should be investigated and what part of it this thesis researches. Also, an overview is given of what each chapter is about.

1.1 Motivation

As explained by Bornholt et al. in [3], more and more data is being produced on a daily basis. A big fraction of all the data is *archival data*, which is also called *cold data*. This means that the data is meant for being stored away and that it does not need to be accessible at every moment. Archival data is typically stored on magnetic and optical media, but there are some downsides to this approach. Despite the fact that the storage density is about 100 GB/mm^3, it would be impossible to store zettabytes, so billions of gigabytes, of data without needing millions of units and a lot of space. On top of that, this type of media is not durable and needs a lot of maintenance, since they only last up to 30 years right now. Therefore, there is a desperate need of an alternative to store the enormous amounts of data.

In [3], the researchers also explain why DNA seems to be a perfect solution: it is eight orders of magnitude denser than tape, it has a storage capacity of 5 PB, so 5 million GB per gram DNA, it is self-replicating and it stays intact for at least several thousands of years without high maintenance costs as long as it is preserved in a dry, cold an dark environment [2]. But, DNA also comes with downsides, since quite a lot of errors occur in the reading and writing processes of DNA [4]. This is were coding theory comes in, to secure the data before it is written on DNA. A lot of research has been done, with different codes like the Huffman code [5], Reed-Solomon codes [6][2][7] and Fountain codes [8]. Several have already been successful in completely retrieving their data, or in implementing random access, but a lot of chemical and biological processes in this are involved as well. Therefore, in order to be able to actually implement DNA based storage, more research needs to be done into a system that automatically takes care of those processes and secures the data at the same time.

1.2 Thesis Statement

One recent research paper attracted our attention: "Demonstration of End-to-End Automation of DNA Data Storage" by Takahashi et al. [1]. In that paper, a fully automated system is described for encoding data, writing it on DNA, reading the DNA and decoding the data again. This seems like the first real step into practical use of DNA based data storage. However, the data only consisted of the 5 byte string *hello* and a Hamming code was used, which can only detect 2 errors and correct 1 error, and which had not been used by researchers before in this context. Therefore, we decided to look into this code and to improve it by using a Reed-Solomon code that was altered for DNA use.

In this thesis, methods that have been developed so far will be studied first. Next, the Hamming code from the previously stated paper will be explained an analysed, before it is investigated whether it is possible to use the Reed-Solomon code instead. So, this thesis investigates whether it would be better to use the Reed-Solomon rather than the Hamming code for DNA based storage.

1.3 Organisation of thesis

This section describes how the remainder of this report is organised and what each chapter is about.

Chapter 2: Prerequisites. In this chapter, an overview of prerequisites containing necessary concepts that are used throughout this thesis is given. It will provide an introduction to DNA, data storage and coding theory. All this information is needed in order to understand the problem and the researched methods that are discussed in Chapter 3.

Chapter 3: Previously researched methods. This chapter will begin by discussing some constraints and properties to build a code and to measure the quality of it. Then three methods that were the first to describe large scale DNA storage will be explained and a short overview is given on what has been done since then.

Chapter 4: Hamming code. Firstly, this chapter will give a general introduction to the Hamming code. Then it will explain what parameters are chosen to apply it to DNA. Next, it will give a step-by-step description on how encoding and decoding works, before examples are worked out to show how errors are handled. More examples can be found in Appendix A.1. The next step is to implement an error simulation to test the code, and finally a detailed analysis of this code is given based on the net information density, GC-weight, homopolymer runs and the error detecting and correcting properties.

Chapter 5: Reed-Solomon code. This chapter has the same structure as Chapter 4. So, firstly a general introduction to the Reed-Solomon code is provided. Next, the parameters are chosen and justified, before the encoding and decoding mechanisms are explained. Again, examples are worked out, with additional ones in Appendix A.2. Then the error simulation is implemented and lastly the whole process will be carefully analysed.

Chapter 6: Comparison. In this chapter the results from the previous chapters are summarised and compared. The trade-off between the net information density and the error detection and correction of both methods is analysed. Moreover, the comprehensibility of the implementations and the applicability of both methods to a bigger data set is evaluated.

Chapter 7: Conclusion and discussion. This final chapter will conclude the results that have been found in this thesis. It will also discuss these results and give recommendations for further research.

2 Prerequisites

This chapter will explain the basics of DNA, data storage and coding theory. In Section 2.1, the structure of DNA together with the process of writing and reading DNA is explained. Next, Section 2.2 gives an introduction to data storage, after which Section 2.3 describes how to use DNA to save data. Lastly, Section 2.4 describes basic concepts within coding theory.

2.1 DNA

In order to be able to explain the process of storing data on DNA, a short introduction on the basic properties of DNA will be given. The information below is retrieved from [3] and [9].

DNA, which is short for *deoxyribonucleic acid*, can be found in the cells of all organisms. It is a source for storing information. For example, it encodes the genetic information that instructs a cell how to make proteins. DNA is made up of two strands of paired *nucleotides*(nt) that are wrapped around each other in a so called *double helix form*. These nucleotides are sometimes also referred to as the building blocks of DNA, since the order in which they appear contains the most important information of the DNA. A nucleotide consists of a five-sided sugar, a phosphate group and a *base*. There are four different types of nucleotides, each defined by a specific base: *A (Adenine), C (Cytosine), G (Guanine)* and *T (Thymine)*. Throughout this thesis the abbreviations *A, C, G* and *T* will be used. Two single strands of DNA bond together in a specific way based on the nucleotides to get the double helix form: A and T are always paired together and C and G are always paired together, in the sense that they are always each others opposites in the two strands. This is called *complementary base pairing*.

DNA replication is the process in nature where DNA molecules are copied or created. This is done by all living cells and occurs typically as part of cell division. During this process, the DNA strands are separated under the influence of the enzyme helicase, where each strand serves as a template for the production of the counterpart. *DNA polymerase* is the enzyme that reads the existing DNA strand and produces two new strands that match the existing ones. After the copy is made, the copied strands are put together to form a new strand of DNA that can again be copied. To start this whole process, *primers* are needed, which are short pieces of DNA where the DNA polymerase enzyme can bind to. They indicate the beginning and end of the region that needs to be copied.

This whole process can luckily also be done artificially in a laboratory, by a so called *polymerase chain reaction* (PCR). Then, also DNA polymerase and primers are used. It is frequently applied to make a huge amount of copies of a certain DNA strand that needs to be analysed. The process of creating DNA molecules, either naturally or artificially, is called *DNA synthesis*. Thus, this technique can be used to 'write' a certain strand of DNA. When talking about artificial DNA, the term *oligonuclotide* comes up quite often. This is a short piece of single stranded DNA, and is often used for genetic testing, research and forensics.

Another interesting process is called *DNA sequencing*, where the order of the nucleotides in a strand of DNA is determined. This can also be seen as a way to 'read' the DNA. Again, PCR is used and primers are needed, since having a lot of copies makes it easier to read the DNA. *Illumina* is the platform that is most used right now and overall has the best performance of modern sequencing technologies in terms of yield and accuracy. However, there are also other options like the Pacific Biosciences single-molecule real time technology and the Oxford Nanopore sequencing systems. Unfortunately, the costs of the first one are fairly high and the error rates are significantly higher than those of Illumina sequences. For the second one, a hand-held device called MinION has been developed, which is relatively cheap. However, it again comes with high error rates, so it is not a good alternative [4]. This thesis is mostly be

concerned with encoding and decoding DNA and therefore the details of these processes will not be discussed in further details.

Nonetheless, the errors that appear in these processes are interesting for this research. There are 3 types: deletion errors, insertion errors and substitution errors. One of the aspects on which the coding methods will be evaluated is if and how these errors are handled. Since the topic of DNA data storage is already quite challenging and because of the limited time for this thesis, it was decided to focus on substitution errors, since they are easier to detect and correct in comparison to deletion errors and insertion errors. Also, according to Yazdi et al. [4] substitution errors are most frequently encountered. By putting certain constraints on the coding schemes, the number of errors that are made in the synthesis and sequencing processes can be decreased. This will be explained more thoroughly in Section 3.1.

2.2 Data storage

When data is saved, whether it is a movie, a picture or just a text file, the data is converted to bits. This thesis only considers text files. In [10], it is explained that to save text, a letter or symbol is first converted to an ASCII number. ASCII stands for American Standard Code Information Interchange and is a numerical representation of letters, characters and certain actions, with values between 0 and 255. Then this number is written in bit form, with a total of 8 bits. Therefore, 1 letter equals 1 byte, so 8 bits. In abbreviations, a small b corresponds to a bit, and a capital B corresponds to a byte. This distinction is important, since for example the difference between MB and Mb is confused a lot. One MB is equal to 10^6 bytes, so 8×10^6 bits. However, to prevent some of the confusion, Mb is often written as Mbit. Other important units from big to small are given below.

$$1 \text{ ZB} = 10^3 \text{ EB} = 10^6 \text{ PB} = 10^9 \text{ TB} = 10^{12} \text{ GB} = 10^{15} \text{ MB} = 10^{18} \text{ kB} = 10^{21} \text{ B}$$

2.3 From data to DNA

According to [2], it is estimated that one gram of DNA can hold 5 PB of data. This, together with the fact that DNA does not need maintenance and that DNA from extinct species from 10,000 years ago can still be recovered, makes it an excellent alternative for the normal data storage that is used nowadays [4]. But how exactly should data be represented in DNA?

There are three basic steps for DNA data storage. First, binary data is encoded to DNA strands, so strings with a quaternary alphabet consisting of the letters A, C, G and T. Next, the DNA strands are synthesised and the data is stored. Finally, the DNA strands can be read using DNA sequencing to retrieve the data.

This thesis is mostly concerned with the first step of transferring data to DNA. The problem is that data is represented binary, so in bits that are either zero or one, and that DNA consists of four nucleotides: A, C, G and T. So the question is: what is a safe way to transfer from a binary to a quaternary alphabet, taking into account the big amount of errors that are made during DNA synthesis? For this, the existing coding techniques need to be altered in such a way that they can be applied to a quaternary alphabet.

2.4 Coding theory

Using information from [10], basic concepts from coding theory will now be explained. There is not a unique definition of the word *coding*, but in this thesis it is defined as a way of securing data such that, if a limited number of errors occur, the data can still be retrieved. Suppose that certain information $\mathbf{s}$ needs to be stored. Then this is converted to bits to get a *message word*

m, which contains all of the information. Next, extra bits are added to secure this information, which is called *redundancy*. There is a consideration to make in this: storing data costs money, so the more data there is, the more expensive it will be. On the other hand, if the data is not secured in a proper way, so if not enough redundancy is added, then the data might not be retrievable due to errors. Thus, a balance needs to be found in the amount of redundancy that is added. Adding redundancy to **m** yields the *codeword* **c**. *Encoding* is defined in this sense as converting a message word **m** to a codeword **c**, and *decoding* is the operation the other way around. The length of a message word is denoted by k and the length of a codeword is denoted by n. Therefore, the amount of redundancy is equal to $n - k$.

The way in which the codewords are constructed, so how the redundancy is added, is determined by which *coding method* is used. This thesis discusses the *Hamming* and *Reed-Solomon* codes in DNA context. To clarify which parameters are chosen, this is denoted by the $\text{Ham}(n, k)$ and $\text{RS}(n, k)$ codes.

To make sure that errors can be detected and corrected, it is necessary to ensure that not all words are codewords. In that way, if a word is retrieved that is not a codeword, it is certain that an error has occurred. For example, one can add a parity bit to a codeword **m** that is equal to 0 if the number of ones in **m** is even, and the parity bit is 1 if the number of ones in **m** is odd. In this way, all the codewords will have an even number of ones. If a word is retrieved with an odd amount of ones, there surely must be an error. In this case, the *distance* between codewords is at least two, since two codewords have at least at two places in which they differ. One error can be detected, but it cannot be corrected since it is not known where the error is. When two errors occur, it may not be possible to detect them, since two errors could change one codeword into another codeword. The *minimum distance* d of a code is the smallest distance between any two codewords. Then the code can detect $d - 1$ errors and correct $\lfloor \frac{d-1}{2} \rfloor$ errors.

In Sections 4.1 and 5.1, more details are given on how the Hamming and Reed-Solomon codes add redundancy to message words to obtain codewords. Note that in the explanation and example above, codes were applied to bits and this does not necessarily have to be the case. Since the goal is to use DNA to save data, in this thesis codes are also applied to quaternary digits, so $\{0, 1, 2, 3\}$. For convenience, these symbols are called *quads* from now on.

3 Previously researched methods

In this chapter, various methods that have been used by researchers to encode data on DNA are discussed. Firstly, several constraints that are used are explained and motivated. Next, the basic principles of the methods of Church et al. [11], Goldman et al. [5] and Grass et al. [6], are described since they are often seen as the founding fathers of archival DNA-based storage [4] and it shows how the used methods differ a lot from each other. Finally, a comparison of the aforementioned methods and recent developments will close off this chapter.

3.1 Constraints

This section explains and motivates certain constraints that are used in the design of DNA based codes. These constraints reduce the amount of synthesis and sequencing errors [4] and can be used to analyse the quality of a code.

Net information density

One easy approach for representing data in DNA would be to map every two bits to one base. For example, one could propose to let 00 correspond to A, 01 to C, 10 to G and 11 to T. This gives a DNA string of $n/2$ bases from a string of n binary bits. Unfortunately, this easy solution does not work properly, since many substitution, insertion and deletion errors are made during the synthesis and sequencing steps and these errors can not be detected or corrected with this method [3]. However, Erlich et al. [8] explain how this actually provides an upper bound on the amount of information that can be stored in each nucleotide. Therefore, they concluded that the *information capacity* of a nucleotide can reach 2 bits. It is thus interesting to calculate the *net information density* for different methods, which represents the amount of information stored on one nucleotide, since this is a way to compare different methods. This is why this property is also included in Table 2. The closer the net information density is to the information capacity, the better the method is in terms of storage efficiency. It can be calculated by the formula below. Note that primers and file identification information are not included in the calculation.

$$\text{net information density} = \frac{\text{number of input information bits}}{\text{number of bases in resulting DNA string}}$$

GC-weight

Another requirement for DNA strands is that they should have a certain *GC-weight*, or *GC-content*. This is the percentage of G and C nucleotides in a DNA strand. Insertion and deletion errors in sequencing are caused by a GC-weight of more than 60% [12]. Furthermore, the DNA strands will be more stable for GC-weight close to 50%, therefore making the computation less error prone [4]. This is why the GC-weight of a DNA strand should be close to 50%.

Homopolymer runs

Repetitions of the same nucleotide is called a *homopolymer run*. For example, in the sequence $TGAAAACAGT$ there is a homopolymer run of Adenine with length 4. According to Blawat et al. [2], a sequence of more than 3 identical nucleotides could lead to difficulties during sequencing. Schouhamer Immink et al. [13] argue that there is a significant higher chance of sequencing errors when the same nucleotide is repeated. This is because every DNA nucleotide is read as a signal in DNA sequencing. If nucleotides are repeated, a machine may read them as a single signal, therefore dropping the nucleotides while sequencing [12]. This explains why a long homopolymer run that consists of more than 3 nucleotides can result in an increase of insertion and deletion errors. Therefore, long runs should not be allowed in order to avoid sequencing errors. This kind of constraint is also called the *runlength constraint*.

3.2 The Church et al. Method

Church et al. [11] were the first to describe a large scale method for DNA storage. Their method works by a one-bit-per-base encoding, in contrary to previous works who all encoded two bits per base. Each bit 0 was converted to either A or C and each bit 1 was converted to either G or T. The choice between the nucleotides was made based on the runlength constraint, which was set to 3. So, no homopolymer runs of length greater than 3 were allowed. In this way, there was also control over the GC-weight. Next, they cut the long DNA strand into information blocks of 96 nucleotides. This was done because long synthetic DNA strands can be hard to assemble. A unique block identification code of 19 nucleotides was added to each information block. Lastly, primers of 22 nucleotides were added at the beginning and end of each block, where the primer at the end was the reverse complement of the primer at the beginning. This gave data blocks consisting of 159 nucleotides.

This method was tested with a data set of approximately 5.27 Mb, so 5.27×10^6 bits, consisting of an html-coded draft of a book that included 53,426 words, 11 JPG images and one JavaScript program. So in total the data consisted of 658,776 bytes. This was a huge increase in data storage, since no research before this had encoded more than 10,000 bits, so more than 1250 bytes [11]. In the end, there were 54,898 data blocks of 159 nucleotides. All of them were recovered with a total of 10 bit errors out of 5.27 million bits. Most of those errors were found at the end of the oligonucleotides where there was only single coverage during sequencing. However, in the DNA 22 errors were made, but only 10 of them led to bit changes. For example, when an A was changed to a C, then the resulting bit stayed a 0 even though there was an error.

Their paper made a huge step towards DNA based data storage, but it had some flaws. For example, there was no system to detect and correct the errors that were made. Therefore, if errors were made, it was impossible to retrieve the original data. Luckily, soon Goldman et al. [5] came up with an improved system.

3.3 The Goldman et al. Method

It was only a year after the publication of Church et al. when Goldman et al. [5] published a paper about practical, high-capacity, low-maintenance information storage in synthesised DNA. Their encoding method was more complex, but seemed a safer way to store data. Firstly, they compressed the data by converting every byte through a Huffman code into 5 or 6 base 3 digits, or *ternary digits*, which are also called *trits*. Next, Goldman et al. solved the problem of homopolymer runs by using a special table as shown in Table 1. This ensured that every nucleotide was different from the one before it.

	next trit		
previous nt	0	1	2
A	C	G	T
C	G	T	A
G	T	A	C
T	A	C	G

Table 1: Goldman's table for encoding the trits to the nucleotides [5].

The next step was to divide the DNA strand into segments of length 100 such that every segment overlapped in 75 bases with the adjacent segments. In this way, every base was covered 4 times in different DNA segments. Also, the reverse complement was taken of alternate segments. Again indexing information was added, which consisted of 2 trits of file identification, 12 trits

indicating which information block it was, 1 parity check trit for the previous information and 1 base at the beginning and end indicating whether the segment was reverse complemented or not. Finally, primers of 33 nucleotides were added at the beginning. In that way, every DNA strand had a total of 150 nucleotides.

Their method was a big improvement compared to the one described by Church et al. For example, the maximum length of a homopolymer run was 1 and every base was covered four times. Therefore, the chances of sequencing errors were lower and if an error occurred, the data could still be retrieved. On top of that, they reverse complemented alternate segments for data security and added a parity check in the indexing information.

They tested their method on data of 757,051 bytes, which is more than what Church et al. stored. It consisted of 5 files: all 154 of Shakespeare's sonnets, a scientific paper in PDF format, an JPEG image, an MP3 file of Martin Luther King's 1963 'I have a dream' speech and a Huffman code. In the end, there were 153,335 data blocks of 117 nucleotides.

Unfortunately, not all DNA sequences could be decoded without intervention, so this method was still not perfect. Although this way of encoding data might look save, there are some downsides to it. For example, only a single parity-check and four fold coverage were used as error correction methods. This is not efficient in terms of redundancy, since every information base is copied 4 times in total, so this will result in high sequencing costs. Plus, research by Yazdi et al. claims that using the Huffman code can lead to huge errors due to sequencing noise [14]. Therefore, it is fair to say that also this method does not fulfil all requirements for DNA data storage.

3.4 The Grass et al. method

To improve the error correction and detection problem of previous methods, Grass et al. [6] used a coding method that is well known for correcting a lot of errors: the Reed-Solomon code. This code will be explained more thoroughly in Chapter 5. Thus, this section only gives a short explanation of their method.

Firstly, every two bytes of a digital file were mapped to three elements of the Galois Field of size 47, so $\mathbb{F}(47)$, which is a base conversion from 256^2 to 47^3. The resulting B information arrays had dimension $m \times k$ and were denoted by $M_b,\ b = 1, ..., B$. The next step was called *outer encoding*. Here, a RS(n, k) code over $\mathbb{F}(47)$ was applied to each block M_b, resulting in codeword blocks C_b. Next, a unique index of length l was added to each column to produce vectors $m_{b,1}, ..., m_{b,n}$ of length $K = l + m$. The *inner encoding* step now used a RS(N, K) code to obtain codewords $c_{b,i}$. Finally, each element in these codewords was written as a string of three nucleotides of which the nucleotides in the second and third position were not equivalent. For this, a codon wheel was designed to map every element of $\mathbb{F}(47)$ to a unique DNA strand with the above mentioned property. In that way, it was ensured that no more than 3 nucleotides were repeated in the total DNA string. So, the resulting total DNA string had length $3N$, before primers were added to the beginning and end.

This method was again tested on data, this time of 82,895 bytes, so approximately 83 KB. It consisted of the Swiss Federal Charter from 1291 and the English translation of the Method of Archimedes. Encoding it resulted in 4991 sequences of 117 nucleotides. Adding the primers gave a total length of 158 nucleotides. An average of 0.7 nucleotide errors per sequence had to be corrected by the inner code. Of the total sequences, the outer code had to account for a loss of 0.3 % and correct 0.4%. In the end, all the data was recovered error-free.

Again, there are some downsides to this method. The biggest one is that only 83 kB of data was stored in the experiment, so the question is whether it is also applicable to larger amounts of data.

3.5 Comparison of techniques so far

After describing the first successful DNA based data storage techniques, not all details from new developments will be discussed, since there has been a lot of research and not all of it is relevant for this thesis. Instead, in Tables 2 and 3 the information from the research in [11], [5], [6], [14], [3], [2], [15], [8], [7] and [1] is summarised.

Authors	Year	Size of data (MB)	Net information density (bits/nt)
Church et al. [11]	2012	0.65	0.83
Goldman et al.[5]	2013	0.75	0.33
Grass et al. [6]	2015	0.083	1.14
Yazdi et al. [14]	2015	0.017	1.58
Bornholt et al. [3]	2016	0.15	0.88
Blawat et al. [2]	2016	22	0.92
Yazdi et al. [15]	2017	0.011	1.72
Erlich et al. [8]	2017	2.11	1.57
Organick et al. [7]	2018	200	1.10
Takahasi et al. [1]	2019	5.0×10^{-6}	1.25

Table 2: Properties of researched DNA based storage methods.

Authors	Year	R.A.	Full recov.	Method for robustness	Error det./cor.
Church et al. [11]	2012	No	No	None	None
Goldman et al. [5]	2013	No	No	Repetition	Detection
Grass et al. [6]	2015	No	Yes	RS	Correction
Yazdi et al. [14]	2015	Yes	Yes	BRDS	Correction
Bornholt et al. [3]	2016	Yes	No	Repetition	None
Blawat et al. [2]	2016	No	Yes	RS	Correction
Yazdi et al. [15]	2017	Yes	Yes	Homopol. check	Correction
Erlich et al. [8]	2017	No	Yes	Fountain	Correction
Organick et al. [7]	2018	Yes	Yes	XOR rand. and RS	Correction
Takahasi et al. [1]	2019	No	Yes	Hamming	Correction

Table 3: Continuation of properties of researched DNA based storage methods.

The tables above contain properties of the different coding methods that are explained in the cited papers. The *net information density* has already been discussed and it can be calculated that the average net information density is 1.13. R.A. denotes whether *random access* was available. If this is the case, then part of the data can be accessed without having to read all the DNA. Another way of looking at it is like this: it is the question of when looking for a sentence in a book, the whole book has to be read or not. This is sometimes also considered as a constraint on codes for DNA. Most of the time, random access is achieved by exploiting the primers, as explained in [3]. *Full recovery* is achieved when the information was fully retrieved without any error or intervention. From the column that states the methods that are used for robustness, it can be concluded that many different coding methods are used. These include repetitions, Reed-Solomon codes, Bounded Running Digital Sum codes, Fountain codes and Hamming codes. The last column classifies if errors could be detected and corrected. For more information and details we refer to the cited works.

4 Hamming code

This chapter will explain how the Hamming code can be used for DNA data storage. Section 4.1 gives a general introduction to the Hamming code. The next three sections describe what parameters are chosen in [1] and how to encode and decode the Hamming code applied to DNA. Next, some examples will be worked out in Section 4.5 and in Section 4.6 the error simulation is described. Finally, in Section 4.7 an analysis is made of this method.

4.1 General properties of Hamming code

Before the Hamming code is applied to DNA, this section will give a general introduction to the Hamming code, using information from [10]. The Hamming code is a linear code, which means that it is a subspace of a vector space, so all its elements can be described in terms of a basis. A *generator matrix* is a matrix G whose rows form a basis for this code. For a Ham(n, k) code, G is a $k \times n$ matrix. So, multiplying a message word $\mathbf{m}$ of length k with generator matrix G will give a codeword $\mathbf{c}$ of length n. When G is of the form $[\ I_k \mid A\ ]$, then G is in *standard form*. In that case, encoding comes down to adding extra symbols behind the message word. A *parity-check matrix* is an $n \times (n-k)$ matrix H where all columns are linearly independent and where it holds that $\mathbf{c}$ is a codeword if and only if $\mathbf{c}H = \mathbf{0}$. It can be used for both error detection and error correction, because if the outcome of $\mathbf{c}H$ does not equal $\mathbf{0}$, then it indicates where an error was made. For a Hamming code, all the rows in H are different and no row is the multiple of another row. If H is of the form $\begin{bmatrix} A \\ I_{n-k} \end{bmatrix}$, then H is in standard form. For a linear code it holds that G is in standard form if and only if H is in standard form. Also, when a code has a minimum distance d, then every $d-1$ rows of H are linearly independent. For a Hamming code, the minimum distance is always equal to 3 and every word has a unique closest codeword. This provides information about how many errors can be detected and corrected. The minimum distance between any 2 codewords is equal to 3, so if one or two errors occur in a codeword, then the resulting word will never be a codeword. This is because then at least three errors should happen, so the Hamming code can always detect two errors. If one error occurs, then the unique closest codeword is equal to the original codeword. However, if two errors occur, then the unique closest codeword is not the original codeword, so if the error would be corrected, it would be incorrectly decoded. Therefore, the Hamming code can correct only 1 error.

4.2 Hamming code applied to DNA

In [1], the Ham(31,26) code is used to encode the data. The data was quite simple and consisted only of the string $\mathbf{s} = \textit{hello}$. The main aim of that research was to build a fully automated DNA data storage system that could encode the data into a DNA strand, then synthesise the DNA onto a oligonucleotide, after which it was read and decoded again. Using the matrix A that is shown below, they constructed the 26×31 generator matrix G by taking $G = [\ I_{26} \mid -A\ ]$ and the 31×5 parity-check matrix H by taking $H = \begin{bmatrix} A \\ I_5 \end{bmatrix}$.

$$A = \begin{bmatrix} 1&1&0&1&1&0&1&0&1&0&1&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1 \\ 1&0&1&1&0&1&1&0&0&1&1&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1 \\ 0&1&1&1&0&0&0&1&1&1&1&0&0&0&1&1&1&1&0&0&0&0&1&1&1&1 \\ 0&0&0&0&1&1&1&1&1&1&1&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1 \\ 0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1 \end{bmatrix}^T$$

Note that the code is constructed over $\mathbb{Z}_4 = \{0, 1, 2, 3\}$, which is not a field. In the supplementary information of [1], the Python code that was used for encoding and decoding was given. In order to understand the encoding and decoding procedure, this code was analysed and it will now be discussed more thoroughly.

4.3 Encoding data to DNA using Ham(31,26) code

This section will explain step-by-step how data is encoded using the Ham(31,26) code in order to save it on DNA.

Step 1. Firstly, the string $\mathbf{s} = hello$ is converted to data. For this, the letters are first converted from $\mathbf{s}$ to ASCII symbols, where the UTF-8 encoding that is standard in Python is used. Next, those ASCII symbols are converted to base 4 numbers and concatenated. The resulting message is denoted by $\mathbf{m}$. As explained in Section 2.4, a number that is either 0, 1, 2 or 3 is called a quad, just like a number that is either 0 or 1 is a bit. So, the elements in $\mathbf{m}$ are quads.

letter	$\rightarrow$	*ASCII*	$\rightarrow$	*base* 4
h	$\rightarrow$	104	$\rightarrow$	1220
e	$\rightarrow$	101	$\rightarrow$	1211
l	$\rightarrow$	108	$\rightarrow$	1230
l	$\rightarrow$	108	$\rightarrow$	1230
o	$\rightarrow$	111	$\rightarrow$	1233

So, this results in the message $\mathbf{m} = 12201211123012301233$ of 20 quads.

Step 2. The next step is to add a *cyclic redundancy check (CRC)* that consists of SHA-256, which is a Secure Hash Algorithm. This is a cryptographic hash function that uses the input data to compute a unique string of symbols. This is put into base 4 form, and the 6 right-most quads are taken. The output of this is denoted by $\mathbf{h}$, which is in this case equal to $\mathbf{h} = 110213$. Concatenating $\mathbf{m}$ to $\mathbf{h}$ gives $\mathbf{a}$, which is given by $\mathbf{a} = 12201211123012301233110213$. This has a total length of 26 quads.

Step 3. This is where the matrix G that was previously defined comes in, since in this step G is used to encrypt the message $\mathbf{a}$. To do this, $\mathbf{a}$ is multiplied with G and the result is denoted as $\mathbf{b}$. Since G is in standard form, this comes down to adding redundancy quads to $\mathbf{a}$. Using $\mathbf{a}$ that was calculated in Step 2, this yields $\mathbf{a}G = \mathbf{b} = 1220121112301230123311021321131$, with 31 quads in total.

Step 4. A parity check quad $\mathbf{p}$ is added at the beginning. For this, the sum of $\mathbf{b}$ is taken, modulo 4. Therefore, when the parity is checked after an error has occurred, the error can be detected. In this case, the parity quad is 3, so this is added to the beginning of $\mathbf{b}$. This gives the final codeword $\mathbf{c} = 3122012111230123012331102132113 1$, consisting of 32 quads.

Step 5. The final step is to convert $\mathbf{c}$ to DNA. This is done by simply mapping {0,1,2,3} to $\{A, C, G, T\}$, so a 0 becomes an A, a 1 becomes a C, etc. This mapping yields the DNA string $\mathbf{d} = TCGGACGCCCGTACGTACGTTCCAGCTGCCTC$.

4.4 Decoding DNA to data using Ham(31,26) code

The previous section explains how to encode data such that it can be saved on DNA. The question that remains is how to decode a DNA sequence and what errors can be corrected in that process. So, suppose a DNA strand $\tilde{\mathbf{d}}$ is retrieved after sequencing. In the steps below, the tilde is constantly used to emphasise that errors might have occurred.

Step 1. The first step is to convert $\tilde{\mathbf{d}}$ back to quads, i.e. back to a codeword $\tilde{\mathbf{c}}$.

Step 2. Next, $\tilde{\mathbf{c}}$ is split up into $\tilde{\mathbf{b}}$ and the parity quad $\tilde{\mathbf{p}}$, and $\tilde{\mathbf{b}}$ is decoded. For this, the parity-check matrix H is used to calculate an error vector $\mathbf{e} = \tilde{\mathbf{b}}H$. Based on $\mathbf{e}$, the error position and error value can be calculated. The error value indicates the difference between the received quad and the original quad. There are 3 situations to distinguish:

a) If $\mathbf{e} = \mathbf{0}$, then no errors have been detected. However, it is not completely sure that in this case there has not been any error, both because there could be an error in the parity quad and because this code is only able to detect single and double errors. So, it could be that $\mathbf{e} = \mathbf{0}$ but that there are 3 errors or more. In any case, 0 is returned for both the error position and the error value. See Example 4.1, and Examples A.1 and A.4 in Appendix A.1.

b) If the non zero entries in $\mathbf{e}$ are all the same, then $\mathbf{e}$ can be used to obtain the value and position of the error. For example, if $\mathbf{e} = [33000]$, then it is certain that there is an error. Again, it could be that there are actually two errors, for example if one error occurs in the parity quad, or if two errors occurred in such a way that the non zero entries in $\mathbf{e}$ are all the same. See Example 4.2, and Examples A.2 and A.3 in Appendix A.1.

c) If the non zero entries in $\mathbf{e}$ are different numbers, so for example if $\mathbf{e} = [12300]$, then there are multiple errors. In that case, $None$ is returned for both the error position and value. See Example 4.3.

Step 3. In situation b) from above, the data can be fixed and decoded into $\tilde{\mathbf{b}}_{dec}$. This is done by subtracting (modulo 4) the error value from the quad in the error position to obtain the original quad. For example, if the error value is 3 and the received quad in the error position is 0, then the original quad is equal to $0 - 3 = -3 \mod 4 \equiv 1$. Apparently, in that case a C was replaced by an A in the DNA string. So, $\tilde{\mathbf{b}}_{dec}$ is returned as decoded data. In situation a) and c) from above, $\tilde{\mathbf{b}}_{dec} = \tilde{\mathbf{b}}$ and $\tilde{\mathbf{b}}_{dec} = None$, are returned respectively.

Step 4. For this step, the parity quad $\tilde{\mathbf{p}}$ is checked with our potentially fixed data $\tilde{\mathbf{b}}_{dec}$. Based on this, and the information before, the type of error is determined and returned.

$$error\_type = \begin{cases} -1, & \text{if too many errors are detected} \\ 0, & \text{if no errors are detected} \\ 1, & \text{if the parity quad mismatches} \\ 2, & \text{if there is a correctable error} \\ 3, & \text{if there is a correctable error and the parity quad mismatches} \end{cases}$$

Step 5. Taking the first 26 quads of $\tilde{\mathbf{b}}_{dec}$ yields $\tilde{\mathbf{a}}$. The cyclic redundancy check is now used to see if there are any errors left in $\tilde{\mathbf{a}}$. For this, $\tilde{\mathbf{a}}$ is split into the first 20 quads, denoted by $\tilde{\mathbf{m}}$, and

the last 6 quads, denoted by $\tilde{\mathbf{h}}$. The CRC for $\tilde{\mathbf{m}}$ is computed and compared to $\tilde{\mathbf{h}}$. This check will return $True$ or $False$.

Step 6. Finally, $\tilde{\mathbf{m}}$ is converted to $\tilde{\mathbf{s}}$ in the reverse order of Step 1 of encoding $\mathbf{s}$. For this, the first four quads of $\tilde{\mathbf{m}}$ are read as a base 4 number. Then this number is converted to an ASCII number, after which it is converted to a string symbol. Then, the next four quads of $\tilde{\mathbf{m}}$ are read as a base 4 numbers and using ASCII converted to a string symbol, which is joined with the first string symbol. This is done until there are no more quads to convert. The resulting string is denoted by $\tilde{\mathbf{s}}$ and is returned together with the result of the CRC check.

4.5 Examples of errors

This section illustrates how different types of errors are handled by this code. Additional examples can be found in Appendix A.1.

Example 4.1. No errors
$\tilde{\mathbf{d}} = TCGGACGCCCGTACGTACGTTCCAGCTGCCTC$
$\tilde{\mathbf{c}}$ = 3 1220 1211 1230 1230 1233 110213 21131
$\tilde{\mathbf{b}}$ = 1220 1211 1230 1230 1233 110213 21131
$err\_type = 0$
$\mathbf{e}$ = [00000]
$(err\_pos, err\_val) = (0, 0)$
$\tilde{\mathbf{a}}$ = 1220 1211 1230 1230 1233 110213
$\tilde{\mathbf{m}}$ = 1220 1211 1230 1230 1233
CRC pass = $True$
$\tilde{\mathbf{s}}$ = h e l l o

Example 4.2. One error: non-parity base
$\tilde{\mathbf{d}} = T\underline{A}GGACGCCCGTACGTACGTTCCAGCTGCCTC$
$\tilde{\mathbf{c}}$ = 3 0220 1211 1230 1230 1233 110213 21131
$\tilde{\mathbf{b}}$ = 0220 1211 1230 1230 1233 110213 21131
$err\_type = 2$
$\mathbf{e}$ = [33000]
$(err\_pos, err\_val) = (0, 3)$
$\tilde{\mathbf{a}}$ = 1220 1211 1230 1230 1233 110213
$\tilde{\mathbf{m}}$ = 1220 1211 1230 1230 1233
CRC pass = $True$
$\tilde{\mathbf{s}}$ = h e l l o

Example 4.3. Two errors: 2 in non-parity bases
$\tilde{\mathbf{d}} = T\underline{AA}GACGCCCGTACGTACGTTCCAGCTGCCTC$
$\tilde{\mathbf{c}}$ = 3 0020 1211 1230 1230 1233 110213 21131
$\tilde{\mathbf{b}}$ = 0020 1211 1230 1230 1233 110213 21131
$err\_type = -1$
$\mathbf{e}$ = [13200]
$(err\_pos, err\_val) = (None, None)$
$\tilde{\mathbf{a}}$ = *Error: too many errors made*
$\tilde{\mathbf{m}} = -$
CRC pass = -
$\tilde{\mathbf{s}}$ = -

4.6 Error simulation

In [1], groups of 4000 reads were collected for decoding. In order to test the error detection and correction properties of this code, a simulation was made where random errors were inserted.

The DNA strand $\mathbf{d} = TCGGACGCCCGTACGTACGTTCCAGCTGCCTC$ is presented as a string in Python. Unfortunately, strings are immutable in this programming language. So to make the simulation easier, and because the mapping from the bases to the quads is the same, quads were randomly changed in $\mathbf{c} = 31220121112301230123311021321131$ to simulate substitution errors. It is possible to change the values in $\mathbf{c}$ in Python, since it is not a string but an array.

Firstly, $\mathbf{c}$ was copied 4000 times. These copies are also referred to as *samples*. Next, the desired total number of substitution errors is specified. For now, a certain number is taken, though it is also possible to choose a random number between some boundaries for this. Next, a random number between 0 and 3999 is chosen to indicate in which sample an error will be inserted, and a random number between 0 and 31 is chosen to indicate in which position in the sample, so in which quad, the error will take place. If there has already been an error in that sample at that spot, so if the combination of the two random numbers has already been generated before, two random new numbers are generated, since it is unfavourable to make an error twice in the same place in the same sample. Finally, a random quad is chosen that is not equal to the quad that is in the place where the error will be inserted. In this way, it is guaranteed that an error will actually take place.

Therefore, by replacing the quad in the sample that was randomly chosen, in the position that was randomly chosen, by a quad that was randomly chosen, errors are inserted with as much randomness as possible. Note that the results can differ for different runs of the implementation. Therefore, a seed value was set, which was chosen to be equal to zero, to be able to compare the results from the Hamming and Reed-Solomon codes.

In order to make an analysis about how the Hamming code performs on the samples after errors have been inserted, the total number of error needs to be determined. In [2], a mean error rate of approximately 3.3×10^{-3} is found. In [7], a substitution rate of 4.5×10^{-3} was found in the payload region, so the region without the primers. In total, there are $4000 \times 32 = 128,000$ bases in the samples. So, if 500 substitution errors are made, then the error rate is equal to $500/128,000 = 3.9 \times 10^{-3}$, which is in accordance with the results from [2] and [7]. This is why the decision was made to insert a total of 500 substitution errors in the samples.

Since the decoding scheme of the Hamming code specifies what type of error is made, an analysis can be made of the amount of times a certain type of errors was made.

Type of error	Amount
-1	17
0	3528
1	10
2	435
3	10

Table 4: Amount of errors of a certain type when making 500 errors.

From Table 4, it can be seen that in 3528 samples, no errors were made at all. Furthermore, in 435 samples there was a correctable error, but in 17 samples, too many errors were made to decode the message.

Unfortunately, the conclusion that the program draws about the type of error, is not always correct. For example, when two errors occur it could be that $\mathbf{e}$ contains the same non-zero

values, so the code thinks it is a correctable error, though it should be type -1 error. Also, when three errors occur, it could be that $\mathbf{e} = \mathbf{0}$, in which case it is marked as a type 1 error, in stead of a type -1 error. Examples of this have been worked out in Appendix A.1.

The question that is actually interesting is in how many cases the code incorrectly decodes the message and in how many cases it cannot decode the message. From the previously described decoding procedure, it is clear that the sample was incorrectly decoded when the CRC pass returns $False$. This is because the CRC determines if there are any errors left after decoding. These errors cannot be corrected anymore, so the decoded string will not be equal to $\mathbf{s}$. Therefore, this is classified as an *incorrect decoding*. If after decoding $None$ is returned, then it was impossible to decode since too many errors were detected. Thus, this is called an *impossible decoding*. Making this distinction when decoding will give the results that are shown in Table 5.

Type of decoding	Amount
Successful decoding	3978
Incorrect decoding	5
Impossible decoding	17

Table 5: Amount of decodings of a certain type when making 500 errors.

In this sense, it can be concluded that, when 500 random errors are made, 22 of the 4000 samples could not be decoded successfully, either because it was impossible to decode or because the decoding was incorrect.

4.7 Analysis

In this section, several qualities of the code are examined, including the net information density, the GC-weight, the maximum homopolymer run and the error detecting and correcting properties.

Net information density

As explained in Section 3.5, the net density it equal to the amount of input information bits divided by the total number of bases in our DNA string. Our input was the string $\mathbf{s} = hello$, consisting of 5 bytes, so 40 bits. Our DNA string $\mathbf{d}$ has 32 bases. Therefore, this yields the following result.

$$\text{net information density} = \frac{\text{number of input information bits}}{\text{number of bases in resulting DNA string}} = \frac{40}{32} = 1.25$$

Looking at Table 2, it can be seen that this is a bit above the average of 1.13, so using the Hamming code to encode the data is quite good in terms of net information density.

GC-weight

The resulting DNA string was equal to $\mathbf{d} = TCGGACGCCCGTACGTACGTTCCAGCTGCCTC$. To calculate the GC-weight, the total number of G and C bases in $\mathbf{d}$ needs to be counted.

Base	Amount
A	4
C	8
G	13
T	7

Using the table above, the GC-weight of this method can be calculated.

$$\text{GC-weight} = \frac{\text{number of } G \text{ and } C \text{ bases}}{\text{total number of bases}} = \frac{21}{32} \approx 65.63\%$$

This is quite high, since the GC-weight needs to be below 60% to avoid deletion and insertion errors and to improve the stability [4] [12].

Homopolymer runs

Again, the DNA string $\mathbf{d}$ is examined. Determining the maximum homopolymer run is easy, since the only things that needs to be done is to look for repetitions of the same base.

$$\mathbf{d} = TCGGACG\underline{CCC}GTACGTACGTTCCAGCTGCCTC$$

So, it can be concluded that the maximum homopolymer run is equal to 3. As explained in Section 3.1, a sequence of more than 3 identical nucleotides could lead to difficulties during sequencing and a significant higher chance of sequencing errors [2] [13]. Therefore, this result is good enough. However, no method was used to limit this number, it was pure coincidence that it did not exceed 3.

Error detecting and correcting properties

This encoding and decoding scheme has three ways of securing the data: first there is the parity-check matrix H, then there is the parity quad that was added, and there is the CRC pass. So suppose one error occurred during sequencing and $\tilde{\mathbf{d}}$ is retrieved. Then the parity-check matrix H can detect and correct this error. However, if the error was made in the parity quad $\tilde{\mathbf{p}}$, then H will not detect it, but the error in $\tilde{\mathbf{p}}$ does not influence the decoding so $\mathbf{s}$ can still be retrieved. Now suppose two errors occurred in $\tilde{\mathbf{d}}$, that are both not in $\tilde{\mathbf{p}}$. In that case, H will detect that there was an error. If $\mathbf{e}$ contains different non-zero values, then multiple errors have occurred, so no attempt is made to decode, since the code is not able to correct more than one error. However, it could happen that two errors occur but $\mathbf{e}$ contains the same non-zero values. Then the assumption is made that only one error has occurred and therefore an attempt is made to correct it. H can only detect and correct one error, so it will incorrectly decode $\tilde{\mathbf{d}}$. This is where the parity quad $\tilde{\mathbf{p}}$ comes in, since it is now checked whether the received $\tilde{\mathbf{p}}$ matches the parity that is calculated of the incorrectly decoded $\tilde{\mathbf{d}}$. These will probably not match, so it can be concluded that additional errors have taken place and therefore the CRC pass will then also return $False$. A worked out example of this can be found in Appendix A.1. So, if multiple errors occur, then the CRC can sometimes still detect them. In that way, this code is a SECDED code, so a Single Error Correcting, Double Error Detecting code. This means that only one error can be corrected and if two errors are detected, they cannot be corrected. This can be improved by implementing the Reed-Solomon code that can correct multiple errors.

5 Reed-Solomon code

In [2], [6] and [7], Reed-Solomon codes are used for encoding the data. One of the main advantages of this encoding technique is that it can correct multiple errors. Therefore, an attempt was made to try to improve the results from the previous chapter by using a Reed-Solomon code. This chapter explains how the Reed-Solomon code can be used for DNA based data storage. The same structure as last chapter is used. So, firstly a general introduction of the Reed-Solomon code is given. Next, the choice of parameters is motivated and the encoding and encoding procedures are explained. Again, an error simulation is implemented and finally an analysis will be given of the Reed-Solomon code applied to DNA.

5.1 Introduction to Reed-Solomon codes

Using information from [10], this section gives an introduction to Reed-Solomon codes. Reed-Solomon codes are defined over a finite field $\mathbb{F}(p^r)$, with p prime and r a positive integer. For example, when $\mathbb{F}(2^8)$ is used as field, a symbol consists of 8 bits, one of which is 10101010 for instance. Then, in total there are 2^8, so 256, possible symbols. This field is actually used a lot, since the ASCII table consists of 256 symbols to represent letters and characters. Normally, for a Reed-Solomon code over $\mathbb{F}(p^r)$ with minimum distance d, $n = p^r - 1$ and $k = 2^r - d - 1$ are taken as parameters. With these parameters, a total of $\lfloor \frac{n-k}{2} \rfloor$ errors can be corrected. For this code, not a generator matrix, but a generator polynomial g is used as a basis. This is of the form $g(x) = (x + \alpha)(x + \alpha^2) \dots (x + \alpha^{d-1})$, where α is the primitive element of $\mathbb{F}(p^r)$. The parity-check polynomial h is of the form $h(x) = (x^n - 1)/g(x)$. From polynomials g and h, generator matrix G and parity-check matrix H can be obtained.

5.2 Reed-Solomon applied to DNA

In order to apply the Reed-Solomon code to DNA data storage, the field $\mathbb{F}(256)$ is used. This choice is logical, since the code needs to be applied a string, namely $\mathbf{s} = hello$, so ASCII symbols can be used to represent the letters. In order to improve the Hamming code from previous chapter, the goal is to be able to correct at least 2 errors. Therefore, it is necessary to choose n and k such that $\lfloor \frac{n-k}{2} \rfloor = 2$. Since $\mathbb{F}(256)$ is used, it might seem logical to choose $n = 255$ and $k = 251$. However, the string $\mathbf{s}$ consists of only 5 symbols, so it is more convenient to have $k = 5$. This is why a *shortened Reed-Solomon code* is used. Shortening a RS(n, k) code with minimum distance d by a symbols will yield a RS$(n - a, k - a)$ code with minimum distance d. Therefore, the RS(255,251) code is shortened into a RS(9,5) code. This means that, when each symbols in the codeword is converted to a base 4 number, the total number of quads will be equal to 36, so the DNA string will consist of 36 nucleotides. This is longer than the DNA string that was used for the Hamming code. However, this was necessary, since choosing $n = 8$, so shortening from RS(255,252) to RS(8,5) would yield 32 nucleotides, but in that case only one error can be corrected. Because the encoding and decoding schemes of the Reed-Solomon code are fairly complicated, it was decided to not implement this ourselves. Therefore, the unireedsolomon 1.0 package that was available on PyPI under an MIT license was applied. This package can encode and decode a possibly shortened Reed-Solomon code over $\mathbb{F}(256)$ for a given n and k.

5.3 Encoding data to DNA using the RS(9,5) code

This section will explain step-by-step how data is encoded using the RS(9,5) code.

Step 1. Firstly, the strings$\mathbf{s} = hello$ is converted to ASCII symbols. This is needed, since it will return values between 0 and 255 and the field that is used is $\mathbb{F}(256)$. This yields the message

$\mathbf{m}$ = 104 101 108 108 111. Here, it is important to notice that the length of $\mathbf{m}$ is five, since each ASCII symbol is counted as an element with length 1, so 104 is one symbol in $\mathbb{F}(256)$.

Step 2. In this step, the message $\mathbf{m}$ is encoded with the RS(9,5) code. Here, $k = 5$ means that the message $\mathbf{m}$ consists of 5 symbols, and the encoded message $\mathbf{b}$ will have 9 symbols. Encoding is done by the previously mentioned package using a generator polynomial g, which comes down to adding 4 symbols from $\mathbb{F}(256)$. This results in
$\mathbf{b}$ = 104 101 108 108 111 127 24 174 193. If wanted, this can be converted back to a string, which will be equal to $\mathbf{s}$' = h e l l o ↑ ®Á, but for the next step the ASCII symbols of $\mathbf{b}$ are needed. Note that symbol 127 is converted to a white space.

Step 3. Translating the ASCII symbols to base 4 numbers and concatenating them, yields
$\mathbf{c}$ = 122012111230123012331333012022323001, which is quite similar to $\mathbf{c}$ that was found in Section 4.3. However, in this case $\mathbf{c}$ has length 36, so it contains 36 quads instead of 32.

Step 4. The final step is again to convert quads to bases. The same mapping as before is used, so from {0,1,2,3} to $\{A, C, G, T\}$. This results in the DNA string
$\mathbf{d} = CGGACGCCCGTACGTACGTTCTTTACGAGGTGTAAC$.

Note that the main difference between the encoding schemes of the Hamming and Reed-Solomon codes is the order in which the data is encoded and is converted to quads. For the Hamming code, the string $\mathbf{s}$ was first converted to quads, before the CRC and the matrix G were used to encode the message. However, for the Reed-Solomon code, the string was first encoded and it was converted to quads afterwards. This also changes the order of the decodings steps, which are described in the next section.

5.4 Decoding DNA to data using the RS(9,5) code

This section describes how a DNA strand that was encoded using the Reed-Solomon code from above can be decoded. For this, suppose that the DNA strand $\tilde{\mathbf{d}}$ is retrieved after sequencing.

Step 1. In the first step, the bases in $\tilde{\mathbf{d}}$ are converted back to quads to gain $\tilde{\mathbf{c}}$.

Step 2. Next, $\tilde{\mathbf{c}}$ is divided into parts of 4 quads. Each of those parts is then read as a base 4 number, and converted to an ASCII symbol, so to an element of $\mathbb{F}(256)$. Putting these together yields $\tilde{\mathbf{b}}$ that has length 9.

Step 3. It is now time to start decoding and checking $\tilde{\mathbf{b}}$. For this, the Reed-Solomon decoding function from the previously mentioned package is used. This will return $\tilde{\mathbf{b}}_{dec}$. Also, a *RS check* is performed to check if $\tilde{\mathbf{b}}_{dec}$ is a valid codeword. All codewords are multiples of the generator polynomial g, so $\tilde{\mathbf{b}}_{dec}$ is a codeword if g divides $\tilde{\mathbf{b}}_{dec}$. If this check returns $True$, then $\tilde{\mathbf{b}}_{dec}$ is a valid codeword. However, if the check returns $False$, then $\tilde{\mathbf{b}}_{dec}$ is not a codeword, so the decoding was incorrect.

Step 4. To find $\tilde{\mathbf{m}}$, the first 5 symbols of $\tilde{\mathbf{b}}$ are taken. The final step in decoding the DNA string that was encoded with the RS(9,5) code is to translate these ASCII symbols back to characters to form string $\tilde{\mathbf{s}}$.

5.5 Examples of errors

This section illustrates how different types of types of errors are handled by the RS(9,5) code. Additional examples can be found in Appendix A.2.

Example 5.1. No errors
$\tilde{\mathbf{d}} = CGGACGCCCGTACGTACGTTCTTTACGAGGTGTAAC$
$\tilde{\mathbf{c}}$ = 1220 1211 1230 1230 1233 1333 0120 2232 3001
$\tilde{\mathbf{b}}$ = 104 101 108 108 111 127 24 174 193
$\tilde{\mathbf{b}}_{dec}$ = 104 101 108 108 111 127 24 174 193
RS check = $True$
$\tilde{\mathbf{m}}$ = 104 101 108 108 111
$\tilde{\mathbf{s}}$ = h e l l o

Example 5.2. Two errors: in different parts
$\tilde{\mathbf{d}} = \underline{A}GGA\underline{A}GCCCGTACGTACGTTCTTTACGAGGTGTAAC$
$\tilde{\mathbf{c}}$ = 0220 0211 1230 1230 1233 1333 0120 2232 3001
$\tilde{\mathbf{b}}$ = 40 37 108 108 111 127 24 174 193
$\tilde{\mathbf{b}}_{dec}$ = 104 101 108 108 111 127 24 174 193
RS check = $True$
$\tilde{\mathbf{m}}$ = 104 101 108 108 111
$\tilde{\mathbf{s}}$ = h e l l o

Example 5.3. Three errors: in different parts
$\tilde{\mathbf{d}} = \underline{A}GGA\underline{A}GCC\underline{A}GTACGTACGTTCTTTACGAGGTGTAAC$
$\tilde{\mathbf{c}}$ = 0220 0211 0230 1230 1233 1333 0120 2232 3001
$\tilde{\mathbf{b}}$ = 40 37 44 108 111 127 24 174 193
$\tilde{\mathbf{b}}_{dec}$ = 40 37 44 108 111 127 24 174 193
RS check = $False$
$\tilde{\mathbf{m}}$ = 40 37 44 108 111
$\tilde{\mathbf{s}}$ = (% , l o

5.6 Error simulation

To evaluate the error detection and correction properties of this code, again an error simulation was made, in the same way as in Section 4.6. So again, 4000 copies were made of
$\mathbf{c}$ = 122012111230123012331333012022323001 and errors were inserted randomly.

In this case, the code does not return what type of error has been made, but again a distinction can be made based on the type of decoding. When the RS check returns $False$, the decoding was unsuccessful, since there are still errors left. Therefore, this type of decoding is called an *incorrect decoding*. If too many errors are made, the code will raise an error and not decode anything. In that case, it is classified as an *impossible decoding*. Inserting 500 errors randomly and making this distinction yields the results that are shown in Table 6.

Type of decoding	Amount
Successful decoding	3999
Incorrect decoding	0
Impossible decoding	1

Table 6: Amount of decodings of a certain type when making 500 errors.

Remember from Section 4.6 that after 500 errors 22 of the 4000 samples could not be decoded successfully. However, in this case, only 1 out of 4000 samples cannot be decoded successfully, so this is an improvement.

5.7 Analysis

Just like Section 4.7, this section analyses the different properties of the RS(9,5) code, based on the net information density, the GC-weight, the maximum homopolymer run and the error detecting and correcting properties.

Net information density
The input is the same string $\mathbf{s} = hello$ as before, consisting of 5 bytes, so 40 bits. However, the amount of bases in the DNA string $\mathbf{d}$ is higher when using the RS(9,5), because instead of 32 bases, it contains 36 bases. Therefore, this results in the following net information density:

$$\text{net information density} = \frac{\text{number of input information bits}}{\text{number of bases in resulting DNA string}} = \frac{40}{36} \approx 1.11$$

This is worse than the net information density that was calculated for the Hamming code. It could be improved by taking the RS(8,5) code, but as explained, then only one error can be corrected. Therefore, by improving the error correction, the net information density decreased. Though, the result is not even that bad since it is just below the average of the net information densities of Table 2.

GC-weight
For the GC-weight, the DNA string $\mathbf{d}$ is evaluated, which was equal to $\mathbf{d} = CGGACGCCCGTACGTACGTTCTTTACGAGGTGTAAC$. The table below shows an overview of the occurrence of each base, which is then used to calculate the GC-weight.

Base	Amount
A	7
C	10
G	10
T	9

$$\text{GC-weight} = \frac{\text{number of } G \text{ and } C \text{ bases}}{\text{total number of bases}} = \frac{20}{36} \approx 55.56\%$$

This is smaller than 60%, so within the wanted boundary. Therefore, the DNA will be more stable during synthesis.

Homopolymer runs
Once more, the DNA string $\mathbf{d}$ is analysed to determine the maximum homopolymer run.

$$\mathbf{d} = CGGACG\underline{CCC}GTACGTACGTTC\underline{TTT}ACGAGGTGTAAC$$

Again, the maximum homopolymer run is equal to 3. Like before, this is still within limits. Unfortunately, this result was again due to coincidence, so it would be wiser to construct a method to control this.

Error detecting and correcting properties

The Reed-Solomon code was chosen because it could improve the error correction. As explained earlier, by choosing the parameters n and k specifically, one can choose for themselves how many errors the code can correct in the field that it is applied to. In this case, it was necessary for the code to correct at least 2 base errors. However, by the way it was applied to DNA data storage, it can actually correct up to 8 base errors. This is because the Reed-Solomon code that was used works in $\mathbb{F}(256)$ and can thus correct 2 errors in the ASCII symbols. In the DNA, every ASCII symbol is represented by four bases. So, if multiple errors occur in the bases that represent one ASCII symbol, it corresponds to only one ASCII symbol changing, which means that the Reed-Solomon code can still correct it. A worked out example can be found in A.2. Therefore, this is a big improvement in terms of error detection and correction compared to the Hamming code.

6 Comparison

In this chapter the Ham(31,26) and RS(9,5) codes are compared. For this, the information from previous chapters is used. In Section 6.1, the net information density and error detection and correction of both methods is compared and the trade-off between the two qualities is discussed. Section 6.2 evaluates the implementations of both methods. Finally, Section 6.3 and Section 6.4 explain if and how both methods could be applied to a bigger set of data.

6.1 Net information density and error detection and correction

In Section 4.7, the net information density for the Hamming code was calculated, which was equal to 1.25. This was because for the Hamming code only 32 nucleotides were needed. For the Reed-Solomon code, in Section 5.7 the net information density was calculated, which was equal to 1.11 because 36 nucleotides are used. So, in terms of the net information density the Hamming code is better.

However, when looking at error detection and correction, the Reed-Solomon code clearly outperforms the Hamming code, since it can correct 2 errors in stead of 1. On top of that, as explained in Section 5.7, in a lot of cases it can correct even more than 2 errors when multiple errors take place in the same part of the DNA that corresponds to one symbol of $\mathbb{F}(256)$. The error simulations in Section 4.6 and Section 5.6 also proved this, since it was observed that when 500 errors occur, the Reed-Solomon could not decode 1 sample successfully, opposed to 22 samples of the Hamming code.

The error simulation was also executed with a RS(8,5) code, which has the same net information density as the Ham(31,26) code, and in that case the performances were similar in terms of failed decodings. It is clear that improving the error correction decreases the net information density. Therefore, there seems to be a trade-off between the net information density and the error detecting and correcting properties. The net information density gives an indication of how much data can be stored on a nucleotide. So the higher this number, the less nucleotides are needed and thus the lower the costs for synthesising and sequencing. For many years, the costs for DNA synthesising and sequencing techniques were pretty high, so it was important to have a high net information density. However, according to Organick et al. [7], the biotechnology industry has made substantial progress to lower the costs. This means that in our opinion, the small difference in net information density between the Hamming and Reed-Solomon code will not cause a big difference in the final costs nowadays. Hence, the advantage of being able to correct more errors is a more important factor in this trade-off.

The error simulation was expanded to compare the results of both methods. Here, the number of successful, incorrect and impossible decodings for any number of errors between 0 and 4000 was calculated. In this, it was observed that when 1000 errors occur in total, almost 100 of the 4000 samples could not be successfully decoded using the Hamming code, in contrast to only 5 unsuccessful decodings with the Reed-Solomon code. Again, this shows the big difference in the error correction properties of the methods. As expected, the Reed-Solomon code indeed performed better than the Hamming code for any number of errors.

6.2 Implementations

In the beginning of this thesis, the Python code from the research of Takahashi et al. in [1] was studied. Their code had a quite simple structure: firstly two functions were determined: one for converting a number to a base 4 number and one for creating the generator matrix G and parity-check matrix H, based on k. Then, a class was build with several functions: one for

converting the string to data, one for encoding the data, one for converting the encoded data to DNA, one for converting DNA back to data, several ones for decoding and lastly one for converting the decoded data to a string. This implementation was pretty straightforward and good to understand. In the implementation of the Reed-Solomon code that could be applied to DNA a similar structure was used.

The Hamming encoding and decoding steps were easy to understand and follow. However, for the Reed-Solomon code, this was different since the unireedsolomon 1.0 package was used for encoding and decoding, which consisted of multiple files that all contained a great deal of lines. Despite the attempt to understand it, it was not possible to completely figure out how the used encoding and decoding functions work. So although the implementation works fine, it is harder to get a full picture of how it exactly works. It was also observed that the Reed-Solomon error simulation was slightly faster than the Hamming error simulation. Therefore, it can be concluded that the Hamming code was easier to understand and implement than the Reed-Solomon code even though the Reed-Solomon code performs better in terms of error correction and it is a bit faster.

6.3 Applicability of Hamming code to bigger data

So far, the Hamming code was only applied to the simple string $\mathbf{s} = hello$. It is questionable whether it is realistic to apply this to a bigger data set. Encoding was done trough the generator matrix G that was given in Section 4.2. This matrix was created for the Ham(31,26) code. Although of course a new generator matrix can be made for a higher value of n and k to be capable of applying the Hamming code to larger data, this does not seem useful. Not only because in that case quite some changes need to be made to the Python code, but also because the distance d of a Hamming code is always equal to 3. Therefore, if a different value is chosen for n and k, then the error correction will not improve, so it will not be possible to correct more errors. However, if n is larger, so if the oligonucleotide is longer, then there will actually be more errors since it consists of more nucleotides. Therefore, it is expected that the larger the data, the worse the performance of the Hamming code. Consequently, the Hamming code is not suitable for bigger data.

6.4 Applicability of Reed-Solomon code to bigger data

For now, the RS(9,5) code was used, since the string $\mathbf{s} = hello$ of length 5 needed to be encoded. However, this could of course be expanded to a bigger code like the RS(255,223) code, that can correct 16 errors. This can already be implemented with the current Python code, as it is suitable for any input parameters n and k. In that case, the net density would increase greatly to $\frac{223\times 8}{255\times 4} = 1.75$. The GC-weight depends on the input string, so it unknown if that will still be lower that the set boundary of 60%. For this thesis, the main goal was to improve the code that was used in [1]. Therefore, expanding the code to make is applicable to bigger data was not researched any further. We do believe that Reed-Solomon codes should be used for DNA data storage, seeing their many advantages and the fact that the research in for example [7] looks promising.

7 Conclusion and discussion

In this chapter, a short summary from the previous chapters is given and the results are concluded. These results are then discussed and recommendations for further research on this topic are given.

7.1 Conclusion

This thesis was aimed at improving the encoding and decoding method that was described in the paper "Demonstration of End-to-End Automation of DNA Data Storage" by Takahasi et al. [1] for DNA based data storage. In order to do this, methods that have been researched before that paper was published were analysed and certain properties of a code to measure its quality were investigated: net information density, GC-weight, homopolymer runs and error detecting and correcting properties. The method of Takahashi et al. was based on a Ham(31,26) code, which was compared to a RS(9,5) code in this thesis. An introduction to both methods and a description of the encoding and decoding steps were given. By working out examples, both codes were investigated to see how they responded to errors. Then, error simulations were made where randomly 500 errors were inserted in 4000 copies of the data, after which a careful analysis of both methods was made, before they were compared.

In Section 6.1, it was observed that the net information density for the Hamming code is better than the net information density for the Reed-Solomon code. However, when the effect of the net information density on the synthesising and sequencing costs was inspected more closely, it was concluded that this difference was not too important.

Another property where the Hamming code seemed to be better than the Reed-Solomon code was the implementation, since the Hamming code was much easier to understand. Again, the claim was made that this difference does not necessarily make the Hamming code better. This is because nowadays, quite a lot of methods, like for example neural networks, are implemented and used even though they are far too complex for a human being to understand. However, since computers can do a lot of sophisticated operations, their output is used even though it is not exactly certain how that was generated from the input. Therefore, the conclusion is that the fact that the Reed-Solomon code excels in other important qualities like error correction weighs heavier than the fact that the implementation is not as simple as the implementation of the Hamming code.

Unfortunately, the Hamming code had a GC-weight of more than 60%, which could lead to more deletion and insertion errors, as well as more unstable DNA in general. On top of that, the Hamming code could only correct one error, which caused quite a lot of failed decodings in our error simulation. These two things are both improved by the Reed-Solomon code, with a GC-weight of 56% and the ability to correct at least 2 errors. Although it is not possible to measure the influence of the improved GC-weight, the error simulation showed that in only one sample the Reed-Solomon was not successful in decoding.

In Section 6.3 and 6.4, it was concluded that the Reed-Solomon code also has more potential to work properly on a bigger data set, since for a larger k and n, the number of errors that the code is able to correct can increase. This is not the case for the Hamming code.

Counting up all these arguments, it is fair to conclude that the Reed-Solomon code is more suitable for a DNA based storage system than the Hamming code.

7.2 Discussion

The focus of this thesis was to create a DNA based Reed-Solomon code and compare it to the Hamming code that was used in [1]. However, multiple remarks can be made about this research.

The biggest one is probably the fact that only one word, namely *hello*, was encoded. The choice for this was motivated by the fact that the goal was to mainly look into the research that was described in [1]. In order to implement DNA based storage systems in reality, more research needs to be done into bigger sets of data. Also, it was only researched how to encode and decode the data itself, and nothing was investigated about the primers and file identification that should be added. For bigger data sets, this needs to be researched as well, because according to Blawat et al. [2] modern synthesizers can produce olignucleotides of 250 nucleotides with acceptable errors. For longer oligonucleotides the error rates increase drastically, so for a big file the data needs to be cut into pieces, and every oligonucleotide needs to contain extra information about which part of the file it is representing. In that case, also random access and rewritability of the data could be investigated.

Furthermore, only substitution errors were implemented in the simulation and deletion and insertion errors were not. However, according to Yazdi et al. [4], these are less common than substitution errors. Moreover, according to that research there is an indication that error rates are higher in regions where there is a homopolymer length longer than 15. So far, only homopolymer runs of length 3 occurred. Unfortunately, this was pure coincidence since nothing was done to prevent homopolymers runs of a certain length. The same holds for the GC-weight, which also was not controlled in any way. Every time, the same mapping was used to map an A to a 0, a C to a 1, a G to a 2 and a T to a 3 and the other way around. In our opinion, more research could be done into this mapping. For example, one could also first encode the data, and then count the resulting zeros, ones, twos and threes. Based on that, one could choose the mapping in such a way that a good GC-weight is achieved. The information about which mapping is chosen then needs to be added as additional information to the oligonucleotide. This was looked into and it was observed that this would improve the GC-weight, but the effects could not been researched since the DNA synthesising and sequencing were not actually performed. Thus, the effect of an improved GC-weight is unknown and therefore this was not included in this thesis. This immediately introduces a new discussion point, since in this thesis the errors were simulated and the data was not actually stored on DNA. So, another recommendation is to encode and decode data with the described Reed-Solomon code using DNA.

Therefore, it is recommended that more research is done in finding a method that is applicable to a big set of data, includes primers and additional file information, prevents long hompolymer runs and has a good GC-weight.

References

[1] C. N. Takahashi, B. H. Nguyen, K. Strauss, and L. Ceze, "Demonstration of End-to-End Automation of DNA Data Storage," *Scientific Reports*, vol. 9, p. 4998, 12 2019.

[2] M. Blawat, K. Gaedke, I. Hütter, X.-M. Chen, B. Turczyk, S. Inverso, B. W. Pruitt, and G. M. Church, "Forward Error Correction for DNA Data Storage," *Procedia Computer Science*, vol. 80, pp. 1011–1022, 1 2016.

[3] J. Bornholt, R. Lopez, D. M. Carmean, L. Ceze, G. Seelig, and K. Strauss, "A DNA-Based Archival Storage System," *IEEE Micro*, vol. 37, no. 3, pp. 98–104, 2016.

[4] S. M. H. T. Yazdi, H. M. Kiah, E. R. Garcia, J. Ma, H. Zhao, and O. Milenkovic, "DNA-Based Storage: Trends and Methods," *IEEE Transactions on Molecular, Biological, and Multi-Scale Communications*, vol. 1, pp. 230–248, 7 2015.

[5] N. Goldman, P. Bertone, S. Chen, C. Dessimoz, E. M. LeProust, B. Sipos, and E. Birney, "Towards practical, high-capacity, low-maintenance information storage in synthesized DNA," *Nature*, vol. 494, pp. 77–80, 2 2013.

[6] R. N. Grass, R. Heckel, M. Puddu, D. Paunescu, and W. J. Stark, "Robust chemical preservation of digital information on DNA in silica with error-correcting codes," *Angewandte Chemie - International Edition*, vol. 54, pp. 2552–2555, 2 2015.

[7] L. Organick, S. D. Ang, Y.-J. Chen, R. Lopez, S. Yekhanin, K. Makarychev, M. Z. Racz, G. Kamath, P. Gopalan, B. Nguyen, C. N. Takahashi, S. Newman, H.-Y. Parker, C. Rashtchian, K. Stewart, G. Gupta, R. Carlson, J. Mulligan, D. Carmean, G. Seelig, L. Ceze, and K. Strauss, "Random access in large-scale DNA data storage," *Nature Biotechnology*, vol. 36, pp. 242–248, 2 2018.

[8] Y. Erlich and D. Zielinski, "DNA Fountain enables a robust and efficient storage architecture.," *Science (New York, N.Y.)*, vol. 355, pp. 950–954, 3 2017.

[9] I. Miko and L. LeJeune, "Essentials of Genetics," Cambridge: MA: NPG Education, 2009.

[10] S. Ling and C. Xing, *Coding Theory: A First Course*. Cambridge: Cambridge University Press, 2004.

[11] G. M. Church, Y. Gao, S. Kosuri, and C. T. Clelland, "Next-generation digital information storage in DNA.," *Science (New York, N.Y.)*, vol. 337, p. 1628, 9 2012.

[12] D. Limbachiya, M. K. Gupta, and V. Aggarwal, "Family of Constrained Codes for Archival DNA Data Storage," *IEEE Communications Letters*, vol. 22, pp. 1972–1975, 10 2018.

[13] K. A. Schouhamer Immink and K. Cai, "Design of Capacity-Approaching Constrained Codes for DNA-Based Storage Systems," *IEEE Communications Letters*, vol. 22, pp. 224–227, 2 2018.

[14] S. M. H. T. Yazdi, Y. Yuan, J. Ma, H. Zhao, and O. Milenkovic, "A Rewritable, Random-Access DNA-Based Storage System," *Scientific Reports*, vol. 5, p. 14138, 11 2015.

[15] S. M. H. T. Yazdi, R. Gabrys, and O. Milenkovic, "Portable and Error-Free DNA-Based Data Storage," *Scientific Reports*, vol. 7, p. 5011, 12 2017.

A Extra examples

A.1 Extra examples for Section 4.5

Example A.1. One error: parity base
$\tilde{\mathbf{d}} = \underline{A}CGGACGCCCGTACGTACGTTCCAGCTGCCTC$
$\tilde{\mathbf{c}}$ = 0 1220 1211 1230 1230 1233 110213 21131
$\tilde{\mathbf{b}}$ = 1220 1211 1230 1230 1233 110213 21131
$err\_type = 1$
$\mathbf{e}$ = [00000]
(err_pos,err_val) = (0,0)
$\tilde{\mathbf{a}}$ = 1220 1211 1230 1230 1233 110213
$\tilde{\mathbf{m}}$ = 1220 1211 1230 1230 1233
CRC pass = $True$
$\tilde{\mathbf{s}}$ = h e l l o

Example A.2. Two errors: 1 in parity base, 1 in non-parity base
$\tilde{\mathbf{d}} = \underline{AA}GGACGCCCGTACGTACGTTCCAGCTGCCTC$
$\tilde{\mathbf{c}}$ = 0 0220 1211 1230 1230 1233 110213 21131
$\tilde{\mathbf{b}}$ = 0220 1211 1230 1230 1233 110213 21131
$err\_type = 3$
$\mathbf{e}$ = [33000]
(err_pos,err_val) = (0,3)
$\tilde{\mathbf{a}}$ = 1220 1211 1230 1230 1233 110213
$\tilde{\mathbf{m}}$ = 1220 1211 1230 1230 1233
CRC pass = $True$
$\tilde{\mathbf{s}}$ = h e l l o

Example A.3. Two errors: e contains the same non-zero values
$\tilde{\mathbf{d}} = TCGGACGCCCGTACGTACGTTCCAGCT\underline{TG}CTC$
$\tilde{\mathbf{c}}$ = 3 1220 1211 1230 1230 1233 110213 32131
$\tilde{\mathbf{b}}$ = 0220 1211 1230 1230 1233 110213 32131
$err\_type = 3$ (wrong conclusion)
$\mathbf{e}$ = [11000]
$(err\_pos, err\_val) = (0, 1)$
$\tilde{\mathbf{a}}$ = 0220 1211 1230 1230 1233 110213
$\tilde{\mathbf{m}}$ = 0220 1211 1230 1230 1233
CRC pass = $False$
$\tilde{\mathbf{s}}$ = (e l l o

Example A.4. Three errors: such that e = 0.
$\tilde{\mathbf{d}} = T\underline{A}GGACGCCCGTACGTACGTTCCAGCT\underline{TG}CTC$
$\tilde{\mathbf{c}}$ = 3 0220 1211 1230 1230 1233 110213 10131
$\tilde{\mathbf{b}}$ = 0220 1211 1230 1230 1233 110213 32131
$err\_type = 1$ (wrong conclusion)
$\mathbf{e}$ = [00000]
$(err\_pos, err\_val) = (0, 0)$
$\tilde{\mathbf{a}}$ = 0220 1211 1230 1230 1233 110213

$\tilde{\mathbf{m}}$ = 0220 1211 1230 1230 1233
CRC pass = $False$
$\tilde{\mathbf{s}}$ = (e l l o

A.2 Extra examples for Section 5.5

Example A.5. One error
$\tilde{\mathbf{d}} = \underline{A}GGACGCCCGTACGTACGTTCTTTACGAGGTGTAAC$
$\tilde{\mathbf{c}}$ = 0220 1211 1230 1230 1233 1333 0120 2232 3001
$\tilde{\mathbf{b}}$ = 40 101 108 108 111 127 24 174 193
$\tilde{\mathbf{b}}_{dec}$ = 104 101 108 108 111 127 24 174 193
RS check = $True$
$\tilde{\mathbf{m}}$ = 104 101 108 108 111
$\tilde{\mathbf{s}}$ = h e l l o

Example A.6. Two errors: in the same part
$\tilde{\mathbf{d}} = \underline{AA}GACGCCCGTACGTACGTTCTTTACGAGGTGTAAC$
$\tilde{\mathbf{c}}$ = 0020 1211 1230 1230 1233 1333 0120 2232 3001
$\tilde{\mathbf{b}}$ = 8 101 108 108 111 127 24 174 193
$\tilde{\mathbf{b}}_{dec}$ = 104 101 108 108 111 127 24 174 193
RS check = $True$
$\tilde{\mathbf{m}}$ = 104 101 108 108 111
$\tilde{\mathbf{s}}$ = h e l l o

Example A.7. Eight errors: in two parts
$\tilde{\mathbf{d}} = \underline{AAACAAAA}CGTACGTACGTTCTTTACGAGGTGTAAC$
$\tilde{\mathbf{c}}$ = 0001 0000 1230 1230 1233 1333 0120 2232 3001
$\tilde{\mathbf{b}}$ = 1 0 108 108 111 127 24 174 193
$\tilde{\mathbf{b}}_{dec}$ = 104 101 108 108 111 127 24 174 193
RS check = $True$
$\tilde{\mathbf{m}}$ = 104 101 108 108 111
$\tilde{\mathbf{s}}$ = h e l l o

Example A.8. Six errors: in different parts
$\tilde{\mathbf{d}} = \underline{A}GGA\underline{A}GCC\underline{A}GTA\underline{A}GTA\underline{A}GTT\underline{A}TTTACGAGGTGTAAC$
$\tilde{\mathbf{c}}$ = 0220 0211 0230 0230 0233 0333 0120 2232 3001
$\tilde{\mathbf{b}}$ = 40 37 44 44 47 63 24 174 193
$\tilde{\mathbf{b}}_{dec}$ = *Error: too many errors made*
RS check = -
$\tilde{\mathbf{m}}$ = -
$\tilde{\mathbf{s}}$ = -