Université Catholique de Louvain
Faculty of applied sciences
Department of computer sciences
Master thesis 2007

Algebraic soft-decoding of Reed-Solomon codes

Arnaud Dagnelies

Promoter:
Prof. P. Delsarte

Thanks to my parents,
my promoter,
all my roommates
and especially
Amine, Ciara,
Camille, Netah,
Yo-Sun and Yo-Yin

Foreword

This text is divided in two parts. The first one, the prerequisites, introduces all the necessary concepts of coding theory. If the reader is familiar with the basics of information and coding theory, he might skip this part and go directly to the second part, which is the core of this thesis. This second part studies an algorithm which improves the decoding of Reed-Solomon codes. For readers familiar with coding theory, the second part is gently introduced also making it directly accessible.

Error-correcting codes are used to add redundancy to data to make it fault tolerant (up to a certain degree). Roughly said, the typical way to do it is to encode sequences of bits to longer sequences of bits by adding structured redundancy in it. That way, even if some bits are corrupted, but not too many, the structured redundancy enables us to retrieve the original data. One of the most efficient and widely used type of error-correcting codes are precisely Reed-Solomon codes. Both because of their good properties and their efficient decoding techniques. The algorithm presented here runs in polynomial time and provides more powerful decoding than conventional decoding algorithms. A more precise and exhaustive description can be found in the introduction of the second part of this text.

We wish you an enjoyable reading.

Contents

Part I

Prerequisites

Chapter 1

Introduction

1.1 Overview

Reliable communication has always been important. Even in our every day life. Imagine your mail has a few kilobytes of data altered on its way, or that an extremely tiny scratch makes your DVD unreadable, or that the message typed on your mobile phone was sent with wrong letters or to a wrong recipient... All these are examples of error detection or correction. In our era of digital communication, reliable transmission of information is a wide ranging concern. It is the aim of this first part to establish a model of communication, to explain *precisely* how it can be modeled and how exactly the information can be made more reliable by means of *error-correcting codes*. To go straight to the subject, a little introductory example is presented thereafter. Then, a communication system model is taken in all its generality in the second chapter. The role of the different components are inspected and refined at each step, by using basic assumptions on the kind of communication models. Most notably, bloc encoders, memoryless channels are reviewed with an emphasis on the distinction between hard and soft-information. Then the basics of coding theory is presented in chapter 3, also in all its generality, along with some bounds on code sizes and decoding paradigms. Before linear codes are reviewed (these are the codes having the greatest use in practice), fundamental mathematics are reviewed in chapter 4. This includes a basic coverage of groups, rings and fields. This enables us to present linear codes in chapter 5 as vector subspaces over finite fields previously presented. This chapter also covers the generator and parity check matrices as well as a brief introduction to syndrome decoding. Then, in chapter 6, finite fields at looked more closely at, explaining basic properties, reviewing polynomials with coefficient in finite fields, how to construct extension fields and lastly the extended Euclidean algorithm which will show its use in the next chapter. It the last chapter 7, Reed-Solomon codes, the subject of the second part, are introduced. It explains what Reed-Solomon codes are, how to decode them using classical algorithms and presents the key equation.

1.2 Introductory example

So, what is it all about?

Let us begin with a simple illustrative example. Imagine a satellite in space which communicates with some station on earth. That is, it receives data or sends data messages across space. Let us take a basic scenario where the station on earth sends a binary message to it, for example 00110. This message is then transformed by a modulator to produce waves that the satellite receives and demodulates to recover the original message. The catch is that interferences in space can provoke misinterpretations by the demodulation which in turns causes erroneous bits in the received message. Still with the example that 00110 was sent, the erroneous message 10110 could be received instead. In the best case scenario, the satellite would ask for a retransmission and in the worst case, it would execute a wrong command with possible potentially disastrous consequences.

Since a retransmission is fairly time consuming, one could wonder if there is no better way to transmit information more reliably. And this is exactly what coding theory is about. By adding redundancy in messages, it is possible to detect or even recover the original message if the distortion is not too important. For example, every bit of the message could be repeated three times so that the message 00110 would be encoded in 000 000 111 111 000. That way, if 010 000 111 111 000 is received the correct message could be easily *decoded* by selecting the bit with greatest occurrence from each triplet. The whole question of coding theory is about how to add redundancy in messages to achieve the best compromises between the quantity of redundancy added and the error-correction ability it provides. Knowing the channel, the demodulator can often provide the information about the probabilities of whether each bit is 0 or 1. This opens the way to more powerful *soft decoding* techniques taking advantage of this information. One of this techniques is the subject of this thesis.

Chapter 2

Basics of communication

"The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point."

Claude E. Shannon - A Mathematical Theory of Communication

2.1 Communication system

Communication is the activity of transmitting information. This transmission can either be made between two places, like a phone call. Or between two points in time, for example the writing of this thesis so that it can be read later on. We shall restrict ourselves to the study of digital communication. That is, the transmission of messages that are sequences of symbols taken from a set called *alphabet*.

Definition 1 *An alphabet $\mathcal{A}$ is a (finite or infinite) set of symbols. By writing $\mathcal{A}_q$, we denote a finite alphabet having exactly q distinct symbols.*

Example
A message on an alphabet $\{0, 1\}$ could be "01001110" as in the introductory example. Or the sentence "HELLO WORLD" with the alphabet of usual English alphabet augmented by the space character.

Digital communication has become predominant in today's world. It ranges from internet, storage disks, satellite communication to digital television and numerous others... Moreover, any analog system can be transformed into digital data by various sampling and signal transformation methods. Typical examples include encoding music in an mp3, numerical cameras, voice recognition and many others.

A digital communication system, in all its generality, can be represented as follows.

$$\begin{tikzcd} \boxed{\text{Source}} \arrow[r] & \boxed{\text{Transmitter}} \arrow[r] & \boxed{\text{Physical Channel}} \arrow[r] & \boxed{\text{Receiver}} \arrow[r] & \boxed{\text{Destination}} \\ & & +\ \text{Noise} \arrow[u] & & \end{tikzcd}$$

- The information source outputs the data to be communicated. It produces messages to be transmitted to the receiving destination. When it is a digital source, these messages are sequences of symbols taken from a finite alphabet.

- The transmitter takes the source data as input and produces an associated signal suited for the channel. The aims of the transmitter can be multiple:

  - that a maximum of information is transmitted per unit of time. This is directly connected to data compression techniques taking advantage of the statistical structure of the data.
  - to ensure a reliable transmission across the noisy channel. In other words, to make it fault tolerant to errors introduced by the channel. This is typically done by adding structured redundancy in the message.
  - to provide message confidentiality. This typically involves encryption which hides or scrambles the message so that unintended listeners cannot discern the real information content from the message.

- The physical channel is the medium used to transmit the signal from the source to the destination. Examples of channels conveying information is conveyed over space like telephone lines, fiber-optic lines, microwave radio channels... Information can also be conveyed between two distinct times like for example by writing data on a computer disk or a DVD and retrieving it later.

As the signal propagates through the channel, or on its storage place, it may be corrupted. For example, the telephone lines suffer from parasitic currents, waves are subject to interference issues, a DVD can be scratched... But these perturbations are regrouped under the term of *noise*. The more noise, the more the signal is altered and the more it is difficult to retrieve the information originally sent. Of course, there are many other reasons for errors like timing jitter, attenuation due to propagation, carrier offset... But all these perturbations lie beyond the scope of this thesis.

- The receiver ordinarily performs the inverse operation done by the transmitter. It reconstructs the original message from the received signal.
- The destination is the system or person for whom the message is intended.

2.2 In more details...

As was said in the previous section, the transmitter can have several roles together. To compress data, to secure data, to make it more reliable and lastly to transmit it as signals suited for the physical channel. Compressing data is also called *source coding*, it consists of mapping sequences of symbols in the original data stream to shorter ones. This is done based on the statistical distribution of the original data: the most frequent sequences are mapped to shorter ones while rare sequences are mapped to longer ones. By doing this, the resulting sequences are on average shorter, i.e. sequences with less symbols. On the opposite, in order to make the sequence of symbols robust to errors, redundancy is added to it. This is called *channel encoding* and consists of mapping shorter sequences to longer ones so that if a few symbols are corrupted the original data can nevertheless be found back.

What if we want the transmitter to do both? This seems contradictory since one reduces the number of sent symbols while the other increases it. However, it is not really. The source coding reduces the redundancy of *unstructured* data which would not provide protection if symbols were corrupted. For example, despite knowing that a file contains on average 99% of zeros, you cannot know which bits were corrupted when sending the file as it is.

On the opposite, channel coding adds *structured* data to improve protection against such errors during the transmission. By taking the compressed file and reapeating three times each bit, you can decode correctly up to one error per three bits introduced.

One could wonder if a technique performing both in a single step could be more efficient than doing it it sequentially. It turns out that performing source and channel coding sequentially tends to be optimal when the treated sequence length tends to infinity. This is known as the *source-channel coding separation theorem* and is one of the results of Shannon's ground breaking work [4]. For finite sequence length, such joint encoding techniques are still a subject of research.

Until now, we spoke only about symbols and about mapping sequences of symbols to other sequences of symbols with better properties. However, the physical channel does not, technically speaking, transmit symbols but signals (waves, voltage, ...). We however assume a one-to-one mapping between symbols and signals which is done by a *modulator* to map a symbol to the corresponding signal and a *demodulator* mapping back a received signal to a received symbol, or informations about the likelihood of each potential symbol. Notice that by separating the source and channel coding, an encryption module can also be conveniently inserted between both. Putting all together, the obtained refined communication model is illustrated below.

$$\begin{tikzcd}
\boxed{\text{Source}} \arrow[r] & \boxed{\text{Source encoder}} \arrow[d] & \\
 & \boxed{\text{Encryption}} \arrow[d] & \\
 & \boxed{\text{Channel encoder}} \arrow[r] & \boxed{\text{Modulator}} \arrow[d] \\
 & & \boxed{\text{Physical Channel}} \arrow[d] \\
 & \boxed{\text{Channel decoder}} \arrow[d] & \boxed{\text{Demodulator}} \arrow[l] \\
 & \boxed{\text{Decryption}} \arrow[d] & \\
\boxed{\text{Destination}} & \boxed{\text{Source decoder}} \arrow[l] &
\end{tikzcd}$$

All three modules: compression, encryption, channel coding are of course optional and not necessarily included in a communication system. The part of interest for us is channel encoding and decoding. As a side note, but important, one consequence of source coding or encryption is that any of them tends to produce equiprobable sequences of symbols. This argument will support an important assumption for the input of the channel encoder later on. Moreover, even if these modules are not present and nothing is known about the source's output, this is still the best assumption that can be made. The main part of interest for us is the channel encoder and decoder and it can now be isolated from the remaining system.

Lastly, the modulator and demodulator constitute the glue between the transmitter, the physical channel and the receiver. Usually, the channel is considered as the modulator, the physical channel and the demodulator together, providing an abstract model having as input and output symbols. However, the received signals, altered by noise, may not match any of the sent ones. So either it is mapped to other symbols in a bigger alphabet or some threshold decision must be used to decide which symbol of the original alphabet it should be. This is seen in more details in the section about channels.

2.3 Channel encoder

The role of the encoder is to add structured redundancy to sequences of symbols. There are different ways of doing this but the biggest class of encoders are by far *block encoders.* They work by cutting the data stream in blocks (of symbols) of fixed size and encoding these blocks one after another. Such a block, a finite sequence of symbols, is also known as a word.

Definition 2 *A word $\underline{\mathbf{w}} = (w_1, ..., w_m)$ of length m over an alphabet $\mathcal{A}$ is an ordered sequence of m symbols taken from $\mathcal{A}$.*

The data block is called the *message word.* By hypothesis, it has a fixed length, say k. The encoder maps each such word to a longer one, called *codeword*, also of fixed size, say n. A reasonable assumption is that both words, the message and the encoded codeword, are defined over the same alphabet $\mathcal{A}_q$. The encoder is thus a mapping from words of length k to words of length n where $n > k$. Words can be seen as points in a space over the alphabet. In this light, the encoder is a mapping of points in $\mathcal{A}_q^k$ to points in $\mathcal{A}_q^n$. More specifically, the encoder is an injective function:

$$Enc : \mathcal{A}_q^k \to \mathcal{C} \subset \mathcal{A}_q^n.$$

If the alphabet has q elements, then q^k message words are mapped onto q^k from q^n possible words. The set of all codewords is thus only a subset of $\mathcal{A}_q^n$ and this set forms the *code* $\mathcal{C} \subset \mathcal{A}_q^n$ which is the image of the encoder function. On a closer look, it turns out that what is of interest is not the mapping, but the code $\mathcal{C}$ itself. This is the subject of the next chapter.

2.4 Channels and demodulators

Channels are the medium used to transmit signals corresponding to symbols, where the transmission of each symbol is assumed to be of equal duration. When noise corrupts a transmitted symbols, it tends to introduce errors having some pattern. One of the most frequent pattern is burst errors: several consecutive symbols are usually corrupted at once. Other patterns include cyclic errors, "echoing" errors and so on. However, we shall restrict ourselves only to channel models introducing errors randomly, without any pattern. These are called *memoryless channels.*

Definition 3 *In a memoryless channel, the nth received symbol y_n depends solely on the nth sent symbol x_n.*

Thus, in such a channel, each symbol is transmitted, and maybe corrupted, independently from the others. The model of any memoryless channel can be characterized by:

- An input alphabet $\mathcal{A}_{in}$: the set of symbols that can be sent on the channel.
- An output alphabet $\mathcal{A}_{out}$: the set of symbols that can be received at the other end of the channel.
- Transition probabilities $p(\mathcal{Y} = y|\mathcal{X} = x)$ for all $x \in \mathcal{A}_{in}, y \in \mathcal{A}_{out}$, denoting the probability of receiving the symbol y when x was sent.

The reason transition probabilities can be expressed this way lies in the fact that the channel is memoryless. For channels with memory, more complex stochastic models would be needed and the transition probabilities would rely on several variables.

Concerning the input and output alphabets, they are not necessarily the same. The output alphabet can even be infinite despite $\mathcal{A}_{in}$ is finite. This is illustrated in the channel presented just now.

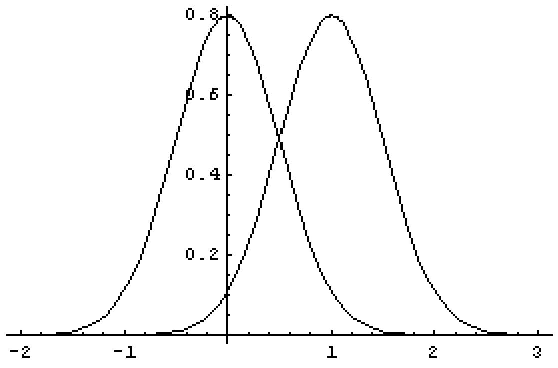
2.4.1 Binary Gaussian channel ...and friends

The Binary Gaussian channel is one of the most used in practice. In this section we present a slight variation thereof. Let its input alphabet be $\{0, 1\}$

and the output be continuous in $\mathbb{R}$, hence an infinite output alphabet. The principle is simple: when the signal is transit in the channel, Gaussian noise $\mathcal{N}(0, \sigma)$ is added to it. Let $\mathcal{X}$ be the random variable standing for the sent bit and $\mathcal{Y}$ the one for the received bit. The probabilities of obtaining a value y when the bit 0 or 1 is sent are distributed according to two normal distributions:

$$p(\mathcal{Y} = y | \mathcal{X} = 0) \sim \mathcal{N}(0, \sigma)$$
$$p(\mathcal{Y} = y | \mathcal{X} = 1) \sim \mathcal{N}(1, \sigma)$$

These are illustrated below for $\sigma = 1/2$.



Knowing the obtained value y, the likelihood of the sent bit can be computed using Bayes' rule:

$$\begin{aligned} p(\mathcal{X} = 0 | \mathcal{Y} = y) &= \frac{p(\mathcal{X} = 0 \cap \mathcal{Y} = y)}{p(\mathcal{Y} = y)} \\ &= \frac{p(\mathcal{Y} = y | \mathcal{X} = 0) p(\mathcal{X} = 0)}{p(\mathcal{Y} = y | \mathcal{X} = 0) p(\mathcal{X} = 0) + p(\mathcal{Y} = y | \mathcal{X} = 1) p(\mathcal{X} = 1)} \end{aligned}$$

Example

Let $\sigma = 1/2$ as in the illustrated normals. If $y = 0.6$ is received, then the likelihood probabilities of the sent bit are:

$$p(\mathcal{X} = 0 | \mathcal{Y} = y) = \frac{0.39 * 0.5}{0.39 * 0.5 + 0.58 * 0.5} = 0.4$$
$$p(\mathcal{X} = 1 | \mathcal{Y} = y) = \frac{0.58 * 0.5}{0.39 * 0.5 + 0.58 * 0.5} = 0.6$$

This means that there are 60% chances the sent bit was a 1 and 40% chances it was a 0.

The knowledge of the likelihood of the sent bit is called *soft-information*, in the example 40% and 60%. This is in opposition to *hard-information* which is the knowledge of only the most likely symbol. For the previous example, just 1. The latter alternative is simpler for the demodulator since a simple threshold rule tells if the output should be a zero or a one. If the input of the channel is assumed to be equiprobable, then the output is 0 if $y < 0.5$ else it is 1. However, by doing this, some information is inherently lost. On the opposite, no information is lost in soft-information consisting of the likelihood of each symbol.

2.4.2 Qary symmetric channel

This channel works with any input alphabet $\mathcal{A}_q$ and the output alphabet is the same. The adjective symmetric refers to the transition probabilities. Given a *crossover probability* p_{err}, the transition probabilities are as follows.

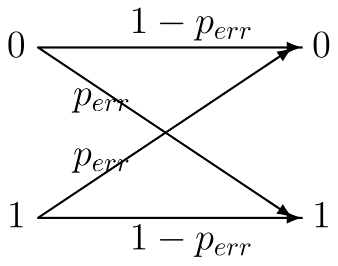
- The probability that the sent symbol α is unchanged upon reception is:

$$\forall \alpha \in \mathcal{A}_q : \quad p(\mathcal{Y} = \alpha | \mathcal{X} = \alpha) = p(\mathcal{X} = \alpha | \mathcal{Y} = \alpha) = 1 - p_{err}$$

- The probability that the sent symbol α became a different symbol β upon reception is:

$$\forall \alpha, \beta \in \mathcal{A}_q, \alpha \neq \beta : \quad p(\mathcal{Y} = \beta | \mathcal{X} = \alpha) = p(\mathcal{X} = \alpha | \mathcal{Y} = \beta) = \frac{p_{err}}{q-1}$$

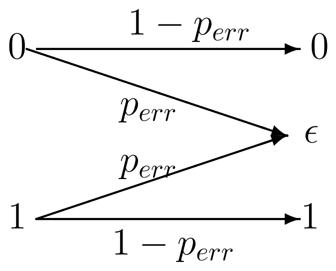
The following example illustrates the binary symmetric channel.



Let us show the relationship between the binary symmetric channel and the Gaussian one we presented before. By including a demodulator which can only provide hard-information to the previous Gaussian channel, the binary symmetric channel is obtained with a crossover probability of $p_{err} = p(0.5 > \mathcal{N}(0, \sigma))$.

2.4.3 Binary erasure channel

Another common variant of binary channels is the use of an additional *erasure* symbol denoted by ϵ. In this channel, either the symbol is correct or it is simply "erased" and there is no way to know what it was. No information at all is available to deduce the sent symbol in the latter case. The graphical representation is as follows.



For example, such channels model appropriately hard drives where there could be bad sectors, or optical disks where scratches make some bytes unreadable and so on.

2.5 Channel decoder

Decoders can basically be divided into two classes. The ones using hard-information, the knowledge of the most likely symbols, are said to perform *hard-decoding*. Their counterpart, taking advantage of soft-information, the likelihood of each symbol, are said to perform *soft-decoding*. These latter decoders are typically both more powerful and more computation intensive. To show that soft-decoding can lead to better error correction, let us illustrate it with the following example.

Example
Let us take as example a binary Gaussian channel where the noise has variance $\sigma^2 = 1$. Suppose either 000 or 111 was sent. These two codewords form the code $\mathcal{C} = \{000, 111\}$.

Let $\underline{\mathbf{y}} = (0.4, 1.2, 0.3)$ be the received values. The hard information would be: $\underline{\hat{\mathbf{y}}} = 010$ and the a hard-decoder based on it would deduce that the most likely sent codeword is $\underline{\hat{\mathbf{x}}} = 000$. On the other hand, by computing the likelihood of each sent symbol, we have:

$$p(\mathcal{X}=0|\mathcal{Y}=0.4)=0.52$$
$$p(\mathcal{X}=1|\mathcal{Y}=0.4)=0.48$$

$$p(\mathcal{X}=0|\mathcal{Y}=1.2)=0.33$$
$$p(\mathcal{X}=1|\mathcal{Y}=1.2)=0.67$$

$$p(\mathcal{X}=0|\mathcal{Y}=0.3)=0.55$$
$$p(\mathcal{X}=1|\mathcal{Y}=0.3)=0.45$$

Since each symbol is independent (because the channel is memoryless), we have:

$$\begin{aligned}
p(\underline{\mathcal{X}}=000|\underline{\mathcal{Y}}=(0.4,1.2,0.3)) &= \gamma p(\mathcal{X}=0|\mathcal{Y}=0.4)p(\mathcal{X}=0|\mathcal{Y}=1.2)p(\mathcal{X}=0|\mathcal{Y}=0.3)\\
&= \gamma 0.09\\
p(\underline{\mathcal{X}}=111|\underline{\mathcal{Y}}=(0.4,1.2,0.3)) &= \gamma p(\mathcal{X}=1|\mathcal{Y}=0.4)p(\mathcal{X}=1|\mathcal{Y}=1.2)p(\mathcal{X}=1|\mathcal{Y}=0.3)\\
&= \gamma 0.14
\end{aligned}$$

Where $\gamma=(0.09+0.14)^{-1}$. Therefore, a decoder able to use this soft-information would conclude that the most likely sent codeword is $\underline{\hat{\mathbf{x}}}=111$! Indeed, 111 is more likely than 000, but both decoders made the best choice based on the information they had when decoding. It is because hard-information inherently looses part of the information.

Chapter 3

Coding theory introduction

In this chapter, we show how the set of codewords forming the code can itself reveal the potential decoding ability of the code. How many errors can be corrected is directly dependent on how the codewords are chosen from $\mathcal{A}^n$, the set of all words of length n. On the same time, coding theory basics are reviewed, explaining main parameters of codes, practical bounds on these and some insights on simple codes.

3.1 What are codes?

As seen in the previous chapter, a code is the set of all the encoded words, the codewords, that an encoder can produce.

Definition 4 *A q-ary code $\mathcal{C}$ of length n is a set of codewords in $\mathcal{A}^n$, where $\mathcal{A}$ is an alphabet of q symbols. The size of the code, noted $|\mathcal{C}|$, is the number of codewords in the code.*

Example
A ternary code of length 5 is the following set:

$AAABB$
$BABAB$
$CCCCC$
$ABCBA$

The ternary alphabet used is $\mathcal{A} = \{A, B, C\}$ and the code size is $|\mathcal{C}| = 4$ (it contains 4 codewords).

Codewords can be seen as vectors in the space $\mathcal{A}^n$ where the ith symbol is the ith coordinate. To compare words, the space $\mathcal{A}^n$ can be equipped with a convenient metric called the *Hamming distance.*

Definition 5 *The Hamming distance between two words $\underline{\mathbf{x}}, \underline{\mathbf{y}} \in \mathcal{A}^n$ is the number of coordinates in which symbols differ.*

$$d_H(\underline{\mathbf{x}}, \underline{\mathbf{y}}) = |\{i | x_i \neq y_i\}|$$

Example
$d_H(01110, 11000) = 3$
$d_H(BLUEBERRY, RASPBERRY) = 4$

The Hamming distance is a metric since it satisfies the triangle inequality.

$$d_H(\underline{\mathbf{x}}, \underline{\mathbf{y}}) \leq d_H(\underline{\mathbf{x}}, \underline{\mathbf{w}}) + d_H(\underline{\mathbf{w}}, \underline{\mathbf{y}})$$

We invite the reader to quickly check it.

3.2 Nearest neighbor decoding

Nearest neighbor decoding is the process of decoding a received word by selecting the codeword at least Hamming distance from it. It is based on the following theorem.

Theorem
Let the channel be a memoryless symmetric one having a crossover probability $p_{err} < \frac{q-1}{q}$ and assume all codewords have equal probabilities to be sent. Under these conditions, the most likely sent codeword is the codeword at least Hamming distance from the received word.

Proof

Let $\underline{\mathbf{y}}$ be the received word and assume that it differs in e places from a codeword $\underline{\mathbf{c}}$. Then the probability that $\underline{\mathbf{c}}$ was sent is:

$$p(\underline{\mathcal{X}} = \underline{\mathbf{c}} | \underline{\mathcal{Y}} = \underline{\mathbf{y}}) = \frac{p(\underline{\mathbf{c}})}{p(\underline{\mathcal{Y}} = \underline{\mathbf{y}})} \prod p(\mathcal{Y}_i = y_i | \mathcal{X}_i = c_i).$$

The probability $p(\underline{\mathbf{y}})$ appears in each whereas $p(\underline{\mathbf{c}})$ is constant due to the equiprobability of the sent codeword hypothesis. Therefore, the above expression is directly proportional to:

$$\prod p(\mathcal{Y}_i = y_i | \mathcal{X}_i = c_i) \leq (1 - p_{err})^{n-e} \left(\frac{p_{err}}{q-1}\right)^e .$$

Since $1 - p_{err} > 1/q$ and $p_{err}/(q-1) < 1/q$ by hypothesis, this function is strictly decreasing with e. Therefore, the probability that $\underline{\mathbf{c}}$ is the sent codeword increases as e decreases so that the word being at least Hamming distance is the most likely one. □

It should be kept in mind that nearest neighbor decoding is maximum likelihood decoding under the implicit assumption that we are working with a ”normal” symmetric channel. To stress how important this assumption is, let us take a small example. Suppose the word $\mathbf{AA}XXXX\mathbf{B}$ is received and either $\mathbf{AA}XXXX$ or $XXXXX\mathbf{B}$ could be decoded. If the probability that an $\mathbf{A}$ was an X before (transmission) is $1/4$ and the probability that a $\mathbf{B}$ was an X before is $1/32$, then it is more likely that the two $\mathbf{A}$'s are erroneous than a single $\mathbf{B}$. In other words, the codeword $XXXXX\mathbf{B}$ would be more likely despite it is the one having the greater distance to $\mathbf{AA}XXXX\mathbf{B}$ compared to the other codeword.

Satisfying the symmetric channel hypothesis, as well as the probability condition concerning it, are crucial for nearest neighbor decoding. But this hypothesis is frequently met in practice. Either because the channel is indeed a symmetric one or because no information about likelihood of symbols is available to the decoder, like in hard-decoding, and thus assuming that any kind of symbol error is equiprobable. This places the hard-decoding problem in a new light. The aim of the hard decoder reduces to find the codeword at nearest distance from the received word. More formally, the most likely sent codeword $\underline{\hat{\mathbf{x}}}$ is the codeword of least distance to $\underline{\mathbf{y}}$.

$$\underline{\hat{\mathbf{x}}} = \text{argmin}_{\underline{\hat{\mathbf{x}}} \in \mathcal{C}} d_H(\underline{\hat{\mathbf{x}}}, \underline{\mathbf{y}})$$

£ Hard decoding thus forgets abouts probabilities and focuses solely on finding the nearest codeword.

Intuitively, one can feel that it is interesting for the code to have a maximum of distance between its codewords.

Definition 6 *The smallest distance between any two distinct codewords of a code $\mathcal{C}$ is called the minimal distance of a code and is noted d.*

$$d = \min_{\underline{\mathbf{x}}, \underline{\mathbf{y}} \in \mathcal{C}, \underline{\mathbf{x}} \neq \underline{\mathbf{y}}} d_H(\underline{\mathbf{x}}, \underline{\mathbf{y}})$$

Indeed, the minimal distance of a code is a fundamental parameter since the error correcting ability of the code is directly related to its minimal distance. For nearest neighbor decoding, in order for the code to provide unambiguous decoding of the received codewords up to e errors, it is necessary and suficient that the minimal distance be at least $d = 2e + 1$. This follows directly from the triangle inequality since no received word $\underline{\mathbf{w}}$ can lie at distance less than or equal to e to two codewords if they are all at a distance $d = 2e + 1$.

Intuition also tells us that soft decoding performances are as well affected by the minimal distance between codewords. However, quantifying it is a difficult task involving channel outcome probability distributions.

3.3 Space and spheres

Back to the codes. We showed that the minimal distance of a code is a crucial parameter, but what is also of interest is that it contains a maximum number of codewords. We will see in this section how the construction of good codes is equivalent to a sphere packing problem. We shall begin by considering the set of words at a given distance from some word and forming a *Hamming sphere.*

Definition 7 *The* Hamming sphere *of radius τ around a word $\underline{\mathbf{w}}$ is the set of all the words which are at a Hamming distance $\leq \tau$ from $\underline{\mathbf{w}}$.*

$$H_\tau(\underline{\mathbf{w}}) = \{\underline{\mathbf{x}} \in \mathcal{A}^n | d_H(\underline{\mathbf{x}}, \underline{\mathbf{w}}) \leq \tau\}$$

Example
Consider the alphabet $\{A, B, C\}$ and the word $AABB$. The Hamming sphere of radius 2 around this word contains :

- at distance 0: $AABB$
- at distance 1: $BABB, CABB, ABBB, ACBB, AAAB, AACB, AABA, AABC$
- at distance 2: $BBBB, BCBB, BAAB, BACB, BABA, BABC, CBBB, CCBB, CAAB, CACB...$

Let $\mathcal{A}$ be an alphabet of q symbols, the number of words in a sphere of radius τ sums up to:

$$\sum_{i=0}^{\tau} \binom{\tau}{i} (q-1)^i$$

Hamming spheres show to be a useful concept, both by assisting understanding and to derive bounds on code parameters. For example, requesting e errors to be unambiguously correctable is equivalent to request that the Hamming spheres of radius e around any two distinct codewords do not intersect. Indeed, if words were in two spheres, it would mean that they are at a distance less than or equal to e from several codewords. This leads to following definitions:

Definition 8 *The packing radius is the greatest possible radius of spheres around each codeword so that they do not intersect.*

The covering radius is the smallest possible radius around codewords so that any word is included in some sphere.

To get good codes, the points in space should be chosen to have the greatest possible packing radius to maximize the minimal distance. And on the same time, have the lowest possible covering radius to not "waste space". Obviously, we always have that *covering radius* $\geq$ *packing radius.* Using these, it is easy to obtain lower and upper bounds on the size of a code depending of its minimal distance:

Theorem
HAMMING BOUND
Given an alphabet of size q and a space of n dimensions, the size of any code with minimal distance $2e+1$ is bounded from above by:

$$|\mathcal{C}| \leq \frac{q^n}{\sum_{i=0}^{e} \binom{n}{i} (q-1)^i}.$$

Proof

This is simply the number of words in the space divided by the number of words in a sphere. □

In a similar manner a lower bound on the number of codewords can be obtained:

Theorem
GILBERT-VARSHAMOV
Given an alphabet of size q and a space of n dimensions, there exists a code with minimal distance d whose size is bounded from below by:

$$|\mathcal{C}| \geq \frac{q^n}{\sum_{i=0}^{d-1} \binom{n}{i} (q-1)^i}.$$

Proof

Let $\mathcal{C}$ be a code of length n and minimal distance d containing a maximum number of codewords. Since it contains a maximum number of codewords, no additional word in space is at a distance d or more from all other codewords. Otherwise, this word could be added to the code. This implies that the spheres of radius $d-1$ around the codewords cover all the space. Thus, the number of codewords times the size of a Hamming sphere of radius $d-1$ must be greater than or equal to the number of words in the whole space. By dividing both sides by the size of a sphere, the theorem's formula is obtained. □

Example
These bounds are however very far apart. Assume we want to find a binary code of length 20 and able to correct unambiguously up to 3 errors. This implies it must have a minimal distance of at leat $d = 2e+1 = 7$. This gives us the following bounds on the size of the code:

$$|\mathcal{C}| \geq \frac{2^{20}}{\sum_{i=0}^{6} \binom{20}{i}} \geq 17$$

$$|\mathcal{C}| \leq \frac{2^{20}}{\sum_{i=0}^{3} \binom{20}{i}} \leq 776$$

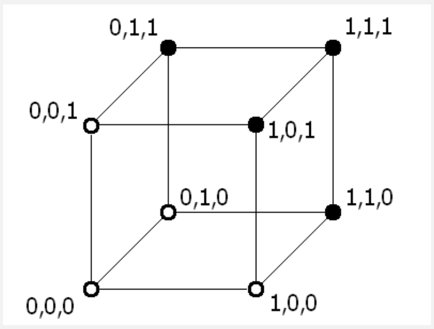
And thus, we know that a code with these parameters and containing at least 17 codewords exists, but it is impossible to find such a code containing more than 776 codewords.

These are not the only bounds and we invite the interested reader to other resources for further informations about them. They can be found in any coding theory book such as [1] or on Wikipedia.

Achieving the Hamming bound happens when the packing radius is equal to the covering radius (why?). Codes achieving this extraordinary property are called *perfect codes* and are very scarce. These are repetition codes, Hamming codes and Golay codes. However, their parameters do not necessarily fit the application. Therefore, it is important to have good codes for a wide range of parameters even if they are not perfect.

Example

This example shows a perfect code and illustrates the concept of spheres. Consider words in the space $\{0,1\}^3$. This space can be illustrated as a cube where each vertex represents a word as illustrated below.



- The Hamming sphere of radius 1 around 000 contains

$$\{000, 001, 010, 100\}$$

- The Hamming sphere of radius 1 around 111 contains

$$\{111, 110, 101, 011\}$$

It is not sufficient to have a code, efficient ways to decode it are necessary. The naive approach to decoding would be to compare the received word to every codeword by measuring it's Hamming distance. Despite the fact that this provides the best possible decoding it has unreasonable time complexity. For example, if there is a codeword associated to any message of 100 bits, then this makes already more than 10^{30} codewords and thus as much comparisons. Hard-decoding paradigms can be subdivided into three categories:

- Bounded distance decoding: this is the most frequent kind of decoding algorithms. Given a code $\mathcal{C}$ and a decoding radius τ less than or equal to the packing radius (usually equal). Bounded distance decoding decodes a received word $\underline{\mathbf{y}} \in \mathcal{A}^n$ to $\underline{\mathbf{x}} \in \mathcal{C}$ if $d_H(\underline{\mathbf{x}}, \underline{\mathbf{y}}) \leq \tau$. I.e. if the received word lies in a sphere of radius τ around some codeword, uniquely determined. In the case the received word is in none of the spheres, a decoding failure is declared. This is why it is also referred as τ-error decoding since it decodes correctly if at most τ errors were introduced.

- List decoding: this problem can be seen as an extension of bounded distance decoding. If we allow the decoding radius τ to be greater than the packing radius, then codewords are no more uniquely determined. List decoding therefore returns a list $\mathcal{L} = \{\underline{\mathbf{x}} \in \mathcal{C} | d_H(\underline{\mathbf{x}}, \underline{\mathbf{y}}) \leq \tau\}$ of all codewords $\underline{\mathbf{x}} \in \mathcal{C}$ in the sphere of radius τ around the received word $\underline{\mathbf{y}}$.

- Nearest codeword decoding: The decoded codeword $\underline{\mathbf{x}} \in \mathcal{C}$ is the one with the minimum distance to the received word $\underline{\mathbf{y}} \in \mathcal{A}^n$. I.e $\underline{\mathbf{x}} = \min_{\underline{\mathbf{x}} \in \mathcal{C}} d_H(\underline{\mathbf{x}}, \underline{\mathbf{y}})$. If several codewords lie at the same distance, there is a number of options possible: the list of these codewords is returned, a codeword is chosen at random or a failure is declared. This enables to correct any received words even if it lies beyond the packing radius. However, this kind of decoding turns out to be computationally difficult in general.

In order to provide efficient encoding and decoding, it is necessary to provide the code with some underlying structure. And for the code to be structured, the alphabet needs some mathematical structure itself. This brings us to the next chapter explaining the basics of fundamental algebraic structures.

Chapter 4

Mathematics fundamentals

This chapter is devoted to covering the basic concepts of groups, rings and fields.

4.1 Groups

The most fundamental algebraic structure is a group. It satisfies the following axioms.

Definition 9 *A group $(G, *)$ is a set G with a binary operation $* : G \times G \rightarrow G$ satisfying the following 3 axioms:*

- *Associativity: For all $a, b, c \in G : (a * b) * c = a * (b * c)$*
- *Identity element: There is an element $e \in G$ such that for all $a \in G$: $e * a = a * e = a$*
- *Inverse element: For each $a \in G$, there is an element $b \in G$ such that $a * b = b * a = e$, where e is the identity element.*

As is stated, a group is simply a set of elements having a neutral and inverses, noted a^{-1} or $-a$ depending on the situation. A group is said to be *commutative*, or *Abelian*, if for $a, b \in G$ we have $ab = ba$.

Example

- The set of even integers, noted $2\mathbb{Z}$ is a commutative group under addition. On the opposite, the set of odd integers is not a group at all since the sum of two odd integers does not lie in the set of odd integers.

- The set of invertible square matrices $\mathbb{R}^{n\times n}$ forms a non-commutative group under multiplication where the neutral element is the identity matrix.
- Let S be an ordered set. Consider G: the set of bijections $\alpha : S \to S$. That is, the elements of G are permutations of the ordered set S. The set G is a group under composition. The inverse is simply the permutation which maps S' to its original state and the neutral is the permutation which does not change anything.

It is straightforward to show that the cancellation laws hold in groups:

$$ab = ac \Rightarrow b = c$$

Since it is sufficient to premultiply both sides by a^{-1} to obtain the equality. Moreover the cancellation laws implies the following theorem.

Theorem
The map $x \to ax : G \to G$ is a bijection.

Proof

To prove that the map is bijective, we will prove that it is both injective and surjective. The cancellation law directly implies that it is injective since $ax_1 = ax_2 \Rightarrow x_1 = x_2$. To show surjectivity, let us show that for every $b \in G$ there exists a value for $x \in G$ so that $ax = b$, this is simply $x = a^{-1}b$. $\square$

By taking a subset of elements satisfying the properties of a group, we obtain a subgroup.

Definition 10 *A subgroup $(S, *)$ of $(G, *)$ satisfies the following axioms:*

- *$S \subset G$ and $S \neq \emptyset$*
- *if $a, b \in S$ then $a * b \in S$*
- *if $a \in S$ then $a^{-1} \in S$*

So that $(S, *)$ is a group by itself with the same neutral element.

Example
The set of even integers $2\mathbb{Z}$ is a subgroup of the integers $\mathbb{Z}$.

By performing the operation (which can be the addition, the multiplication, ...) on a subgroup by an element of a group, we obtain a coset.

Definition 11 *Let $(H, *, e)$ be a subgroup of $(G, *, e)$, both commutative. A coset of H in G is a set of the form*

$$aH = \{ah|h \in H\} = Ha$$

for some fixed $a \in G$.

Example
Let us take as group the integers $\mathbb{Z}$ under addition. Then $5\mathbb{Z}$ is a subgroup of it and $5\mathbb{Z} + 1$ is a coset.

Notice however that a coset is generally not a group. Indeed, it is easy to see that if $a \notin H$, then the coset aH has no neutral element and is therefore not a group. Despite of this, cosets have some nice properties, illustrated in the following theorem.

Theorem
Let H be a subgroup of G.

- If C is a coset of H and $a \in C$ then $C = aH$
- Cosets form a partition on G
- When G is finite, all cosets have the same number of elements.

Proof

To show the first point, let $a \in C = cH$ so that $a = ch$ for some h. By multiplying by h^{-1}, we have $c = ah^{-1}$. On one hand, any element $x \in C$ can be expressed as $x = ch' = ah^{-1}h' \in aH$, thus $C \subset aH$. On the other hand, any element $y \in aH$ can be expressed as $y = ah'' = chh'' \in C$ and thus $aH \subset C$. Combining both, for any $a \in C$ we have $C = aH$.

To show that the cosets form a partition on G, we must show that no element can lie in two distinct cosets. If an element a lies in C and C', then $C = aH = C'$ and thus the sets are either equal or disjoint.

Lastly, that the cosets have the same number of elements as H follows directly from the fact that $x \to ax$ is a bijection on G. $\square$

Example
Let us take $5\mathbb{Z}$, the multiples of 5, as subgroup of the integers $\mathbb{Z}$. The cosets are:

- $\{..., -5, 0, 5, 10, ...\}$
- $\{..., -4, 1, 6, 11, ...\}$
- $\{..., -3, 2, 7, 12, ...\}$
- $\{..., -2, 3, 8, 13, ...\}$
- $\{..., -1, 4, 9, 14, ...\}$

and the properties in the previous theorem are easily checked.

Despite most examples were applied to numbers, it should be kept in mind that these numbers form a particular instance of the problem. In particular, we will see in the next chapter on linear codes that these form a group.

4.2 Rings

Definition 12 *A ring $(R, +, .)$ is a set R with two operations $+$ and $.$ such that:*

- *$(R, +)$ is a commutative group.*
- *$.$ is associative and there exists an element noted 1 such that $a.1 = a = 1.a$ for all $a \in R$*
- *the distributive law holds: for all $a, b, c \in R$:*
  $(a + b).c = a.c + b.c$
  $a.(b + c) = a.b + a.c$

The . is usually omitted so that $a.b$ is abbreviated as ab.

Example
The integers $\mathbb{Z}$ form a ring whereas the set of even numbers $2\mathbb{Z}$ is not a ring. Checking this using the definition is left as a small exercise for the reader.

The most frequent example of rings is modular arithmetic, sometimes also called "clock arithmetic" and works as follows. Two integers a, b are *congruent* modulo n if and only if $a - b$ is a multiple of n.

$$a \equiv b \pmod{n} \quad \Leftrightarrow \quad n|(a-b)$$

The notation on the right means n divides $a - b$. Such a ring is noted $\mathbb{Z}/n\mathbb{Z}$. The sets of congruent integers form n equivalence classes partitioning $\mathbb{Z}$.

Example
If we consider 12 o'clock as the hour 0, then the clock is equivalent to the ring $\mathbb{Z}/12\mathbb{Z}$. Hours "wrap around" after they reach 12. Here are a few examples of computations in this ring:

- $17 + 23 \equiv 5 + 11 \equiv 16 \equiv 4$
- $2 + 10 \equiv 2 + (-2) \equiv 0$
- $10.10 \equiv -2(-2) \equiv 4$

In other words, the result is equivalent to the remainder after dividing by n.
The equivalence classes are:

- $\{..., -12, 0, 12, 24, , ...\}$
- $\{..., -11, 1, 13, 25, ...\}$
- $\{..., -10, 2, 14, 26, ...\}$
- ...and 9 more remaining sets

Let us lastly mention that polynomials with coefficients over a ring R is noted $R[x]$ and they also form a ring. Similarly, if polynomials have several variables, then we note $R[x_1, x_2, ..., x_n]$ for the n variables and they form rings as well. They do indeed form a group under addition and both associativities and the neutral element 1 is satisfied.

However, they do not behave like in usual arithmetics, for example division is not always possible and a polynomial could factor several ways. For example, in $(\mathbb{Z}/6\mathbb{Z})[x]$, we have:

$$(x+2)(x+3) \equiv x^2 + 5x \equiv x(x+5)$$

This motivates the need of a stronger algebraic structure.

4.3 Fields

Fields are mathematical structures behaving with the same rules as usual arithmetic and close to our everyday intuition. A field is a set where operations like addition, subtraction, multiplication or division subject to the laws of commutativity, distributivity, associativity and other "usual" properties.

Definition 13 *A field is a set F or $\mathbb{F}$ with two operations $+$ and $.$ such that:*

- *$(F, +)$ is a commutative group;*
- *$(F^*, .)$, where $F^* = F \setminus \{0\}$, is a commutative group;*
- *the distributive law holds.*

Notice that a field is a ring, by definition, with the additional property that every element has an inverse under multiplication. Some common examples of fields are $\mathbb{R}$, $\mathbb{C}$ and $\mathbb{Q}$. Indeed, every axiom is straightforward to verify. However, the set of integers $\mathbb{Z}$ is not a field. Indeed, for $(\mathbb{Z}^*, .)$ to be a group, any of its element should have an inverse under multiplication. This is clearly not the case since no integers have an inverse in this set except -1 and 1.

Moreover, it should be noted that no flat assumptions are made about operations like subtraction or division. These two can respectively be seen as undoing the operation, i.e. $a + (-b)$ and $a.b^{-1}$. Other familiar properties are not assumed by default but easily proved.

A field is finite when it contains a finite number of elements, referred to as the size of a field and $\mathbb{F}_q$ denotes a field of size q. This notation turns out to be unambiguous because all fields of the same size are isomorphic (identical via a renaming of elements). One of the most common finite fields used in coding theory is the binary field encountered before. The addition and multiplication tables of this field are illustrated below and correspond to XOR and AND binary operations.

+	0	1
0	0	1
1	1	0

×	0	1
0	0	0
1	0	1

What about fields with more elements? Let us take as an example $\mathbb{Z}/5\mathbb{Z}$ which is the ring of the integers modulo 5. The field axioms can easily be verified and we encourage the reader to do it. Thus the ring $\mathbb{Z}/5\mathbb{Z}$ is also a field whose addition and multiplication tables are illustrated below.

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

×	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

This leads us to the question whether all such rings are also fields or not. It can be observed that although the ring of integers modulo 5 is a field, all such rings are not. For example, consider $\mathbb{Z}/6\mathbb{Z}$. It does not form a group under multiplication since the elements 2, 3 and 4 have no multiplicative inverses. For 2 and 4 it is straightforward to see that the product of it with any number results in an even integer and they have therefore no inverse. For 3, it is easy to see as well. By multiplying 3 by any odd integer is equivalent to 3 and by an even integer results in 0.

Theorem
$\mathbb{Z}/n\mathbb{Z}$ is a field if and only if n is prime.

However, they are by no means the only finite fields. They are the simplest and sufficient to illustrate and understand how arithmetic in fields can be done. The key thing to remember is that because of the field axioms, elements can be unambiguously added, subtracted, multiplied and divided, in opposition to previous algebraic structures covered. By taking the alphabet as a field, say $\mathbb{F}$, then $\mathbb{F}^n$ is a vector space (which is itself a group). This vector space structure is practical and will help us defining linear codes in the upcoming section. Finite fields will be investigated further thereafter.

Chapter 5

Linear codes

5.1 Definition

An important family of codes is *linear codes*. They do not only admit simple representations but also have practical reasons by providing efficient techniques of encoding and decoding.

Definition 14 *Let $\mathbb{F}$ be a field. A linear code is a vector subspace of $\mathbb{F}^n$:*

$$\forall a, b \in \mathbb{F},\ \forall \underline{\mathbf{x}}, \underline{\mathbf{y}} \in \mathcal{C} : \quad a\underline{\mathbf{x}} + b\underline{\mathbf{y}} \in \mathcal{C}$$

Example

Over $\mathbb{F}_3$, the following code is linear:
00000
11100
22200
00111
00222
11211
22011
11022
22122

Indeed, any linear combination of codewords is also a codeword.

Since a linear code is a vector subspace, a convenient way to express it is by means of a basis of this subspace. This is the role of the *generator matrix*.

Definition 15 *A generator matrix G of a code $\mathcal{C}$ is a matrix whose rows are vectors forming a basis of the code.*

$$\mathcal{C} = \{m_1\underline{\mathbf{g}}_{1*} + ... + m_k\underline{\mathbf{g}}_{k*} \mid m_1, ..., m_k \in \mathbb{F}\}$$

The dimension k of a linear code is the number of its basis vectors and the size of G is therefore $k \times n$.

Example
A generator matrix for the previous example could be:

$$G = \begin{pmatrix} 11100 \\ 00111 \end{pmatrix}$$

The code has dimension 2 and every codeword from the code can be obtained by a linear combination of these 2 rows.

Notice that G is not unique since there are many possible choices of a basis for $\mathcal{C}$. This generator matrix provides a convenient way to encode a message $\underline{\mathbf{m}} \in \mathbb{F}^k$: by multiplying it by G. The codeword $\underline{\mathbf{x}}$ corresponding to the message $\underline{\mathbf{m}}$ is:

$$\underline{\mathbf{x}} = \underline{\mathbf{m}}G.$$

If the dimension of the code is k and the alphabet has q symbols, then q^k different messages can be encoded. Similarly, in a codeword of n symbols, there are k symbols of information and $n-k$ of redundancy. This is clearly shown when the generator matrix is of the form $G = [I_{k\times k}|A_{k\times n-k}]$. In this case, the first k symbols are the ones of the message $\underline{\mathbf{m}}$ and $n-k$ redundancy symbols are added. This is called *systematic encoding.*

Example
Let the code over $\mathbb{F}_5$ be defined by:

$$G = \begin{pmatrix} 100223344 \\ 010111000 \\ 001000222 \end{pmatrix}$$

The message $\underline{\mathbf{m}} = 123$ would be encoded in

$$\underline{\mathbf{x}} = \underline{\mathbf{m}}G = 123440122$$

The counterpart of the generator matrix is the parity matrix H.

Definition 16 *A parity matrix H of a code $\mathcal{C}$ is a matrix whose rows are vectors forming a basis of the nullspace of the code.*

$$\mathcal{C} = \{\underline{\mathbf{c}} \in \mathbb{F}^n | H\underline{\mathbf{c}}^T = 0\}$$

If the code has dimension k then the dimension of its nullspace is $n-k$ and the size of H is therefore $(n-k) \times n$.

And here again H is not unique. Since the rows of H form a basis of the nullspace of the code whose basis is the rows of G, we have:

$$GH^T = HG^T = 0$$

Notice that the roles of G and H are invertible. This leads to the notion of dual code.

Definition 17 *The dual of a code $\mathcal{C}$ defined by a generator matrix G is the code $\mathcal{C}^\perp$ whose parity matrix is G. Similarly, if H is the parity matrix of $\mathcal{C}$ then H is the generator matrix of $\mathcal{C}^\perp$.*

Both matrices, G and H are easily obtained one from the other. By putting the generator matrix in the form $G = [I_{k\times k}|A_{k\times n-k}]$, a corresponding parity check matrix is obtained by $H = [-A^T_{k\times n-k}|I_{n-k\times n-k}]$ so that multiplying both cancels.

If H has this form, the ith redundancy symbol will be a linear combination of the previous symbols, providing another way to encode messages.

Example

Using the code from the previous example, where the generator matrix has the form $G = [I_{k\times k}|A_{k\times n-k}]$, we have:

$$H = [-A^T_{k\times n-k}|I_{n-k\times n-k}] = \begin{pmatrix} 340100000 \\ 340010000 \\ 240001000 \\ 203000100 \\ 103000010 \\ 103000001 \end{pmatrix}$$

This enables us directly to compute the checks for the message $\underline{\mathbf{m}} = 123$:

- $x_3 = -3x_1 - 4x_2 = 4$
- $x_4 = -3x_1 - 4x_2 = 4$
- $x_5 = -2x_1 - 4x_2 = 0$
- ...

And we obtain the same codeword $\underline{\mathbf{x}} = 123440122$ (of course!) which satisfies $H\underline{\mathbf{x}} = 0$.

As a summary, we can write:

$$\mathcal{C} = \mathrm{Im}(G) = \mathrm{Ker}(H)$$

Lastly, let us mention that the analogy with vector spaces on fields of characteristic zero like the reals $\mathbb{R}$ stops here. For example, the scalar product of a vector by itself can give zero. In other words, the vector is orthogonal to itself. This can of course be extended to a set of vectors and lead to the interesting case of *selfdual* codes. These codes have the interesting property that their generator and parity check matrices are identical.

Example
The following code over the binary field is selfdual:

$$G = \begin{pmatrix} 1100 \\ 0011 \end{pmatrix} = H$$

It is easy to verify that:

$$GH^T = HG^T = 0$$

To summarize it, a code is characterized by four important parameters:

- its alphabet
- its length n
- its dimension k
- its minimal distance d

And a code is sometimes noted with parameters in brackets and has the form $\mathcal{C}[n, k, d]$.

Determining its minimal distance can be done several ways.

Definition 18 *The Hamming weight of a word* $\underline{\mathbf{w}} \in \mathcal{A}^n$ *is the number of non-null symbols the word contains.*

$$w_H(\underline{\mathbf{w}}) = d_H(\underline{\mathbf{0}}, \underline{\mathbf{w}})$$

And using this definition, we can show that the minimal distance of a linear code is equal to the minimum weight of its nonzero codewords.

$$d = \min_{0 \neq \underline{\mathbf{c}} \in \mathcal{C}} w_H(\underline{\mathbf{c}})$$

If there were two codewords, say $\underline{\mathbf{x}}$ and $\underline{\mathbf{y}}$ so that $d_H(\underline{\mathbf{x}}, \underline{\mathbf{y}}) < d$, then $\underline{\mathbf{x}} - \underline{\mathbf{y}}$ would also be in the code and have a Hamming weight less than d which contradicts the hypothesis that we took d as the minimum weight.

For linear codes, one of the most fundamental bounds is the *Singleton Bound* which combines the three main parameters k, n and d as follows.

$$d \leq n - k + 1$$

It can be explained two ways. Given a code $\mathcal{C}[n, k, d]$, if all codewords are projected on $k-1$ coordinates, then (since there are q^k codewords) some codewords must agree on all these $k-1$ coordinates. Thus, these codewords then disagree on at most all other coordinates, i.e. the minimal distance is at most $n - (k-1)$ proving the inequality.

The other way to show the bound makes use of H. For any set of linearly dependent columns of H, there exists a codeword acting on these whose weight is equal to the size of this set. Thus, the minimal weight of codewords ($=d$) is equal to the minimum number of linearly dependent columns. Since any set of $n - k + 1$ columns are necessarily dependent this gives an upper bound on the minimum size of such a set and the above inequality is obtained. Codes achieving this bound are called *maximum distance separable* codes.

5.2 Hard decoding using the syndrome

Let $\underline{\mathbf{x}}$ be the sent codeword and let $\underline{\mathbf{y}}$ be the received word. Since we are working in $\mathbb{F}_q^n$, the error vector can be expressed as $\underline{\mathbf{e}} = \underline{\mathbf{y}} - \underline{\mathbf{x}}$. The error vector $\underline{\mathbf{e}}$ is 0 in the locations where the channel did not introduce errors and nonzero where errors were introduced. By definition, we have:

$$\underline{\mathbf{y}} = \underline{\mathbf{x}} + \underline{\mathbf{e}}.$$

Since a linear code is a vector subspace, it is also a subgroup of $\mathbb{F}^n$. By adding the error vector, $\underline{\mathbf{y}}$ falls in the coset $\mathcal{C} + \underline{\mathbf{e}}$ and all cosets form a partition over $\mathbb{F}^n$. Taking this the other way round, we know that given a received vector $\underline{\mathbf{y}}$, the error vector lies in the coset $\mathcal{C} + \underline{\mathbf{y}}$. The *most likely error vector* is then a word of least weight in this coset and is called *coset leader*. If it is unique, nearest neighbor decoding is unambiguous, else, there are several error vectors of same weight to choose from.

Example
Let the linear code $\mathcal{C}$ over $\mathbb{F}_2$ be defined by:

$$G = \begin{pmatrix} 1111100 \\ 0011111 \end{pmatrix}$$

If the received vector is 1110000 we know that the error is in the coset formed by

$$\{1110000, 0001100, 1101111, 0010011\}$$

so that the most likely error is 0001100 and results in the decoded codeword 1111100. It is nevertheless possible that more errors were introduced so that this choice would be the wrong one and another word of the coset be the correct error word. However, taking the most likely error word, i.e. the coset leader, remains the best choice that can be made.

A nice relationship with the Hamming spheres can be shown. The packing radius is the maximum value so that any coset leader having a weight less than or equal to this radius is unique. The covering radius on the other hand is equal to the maximum weight of any coset leader.

Example
In the previous example, there are $2^5 = 32$ cosets:

- 7 cosets with unique coset leaders of weight 1.
- 19 cosets with unique coset leaders of weight 2.
- 1 coset with 2 coset leaders of weight 2, $\{1100000, 0000011\}$

- 5 cosets with several coset leaders of weight 3.

This means that unambiguous decoding can take place for any single error as well as in 19 of 21 cases of double errors. There 2 double errors where 2 most likely decoded codewords are possible. The packing radius is 1 and the covering radius is 3.

Here comes a general decoding technique. Let $\mathcal{C} + \underline{\mathbf{y}}$ be the coset to which the received word belong. Since the sent codeword $\underline{\mathbf{x}}$ is not known *a priori*, the most likely error $\underline{\hat{\mathbf{e}}}$ is the coset leader of $\mathcal{C} + \underline{\mathbf{y}}$ and we have $\underline{\hat{\mathbf{x}}} = \underline{\mathbf{y}} - \underline{\hat{\mathbf{e}}}$. Thus a way to identify cosets is needed as well as a dictionary mapping the cosets to one of their respective coset leader, having therefore q^{n-k} entries.

A way to identify cosets is the syndrome.

Definition 19 *The syndrome* $\underline{\mathbf{s}}$ *of a received word* $\underline{\mathbf{y}}$ *is:*

$$\underline{\mathbf{s}} = \underline{\mathbf{y}} H^T.$$

Like in medical terminology where a syndrome means a pattern of symptoms helping to identify a disease, the syndrome in coding theory can be used as a unique coset identifier. This fact is illustrated by the following property:

$$\underline{\mathbf{s}} = \underline{\mathbf{y}} H^T = \underline{\mathbf{x}} H^T + \underline{\mathbf{e}} H^T = \underline{\mathbf{e}} H^T.$$

This formula shows that the syndrome is the same for any words of a same coset. The codewords are ”filtered out” so that only the error word affects the syndrome. This provides a straightforward way to make a *syndrome dictionary*: a lookup table mapping syndromes to their corresponding coset leader, i.e. the most likely error word.

Example

Let a linear code $\mathcal{C}$ over $\mathbb{F}_5$ be defined by the following parity check matrix:

$$H = \begin{pmatrix} 011111 \\ 101234 \end{pmatrix}$$

It can correct up to one error since any two columns are independent. The code size would be $5^4 = 625$ which is also the size of the cosets. Therefore, a naive decoder would need 625 comparaisons. On the opposite, there are $5^2 = 25$ cosets that can already be associated with a coset leader:

$\underline{\mathbf{s}}$	$\Rightarrow$	Coset leader (one of)
00	$\Rightarrow$	000000
01	$\Rightarrow$	100000
02	$\Rightarrow$	200000
03	$\Rightarrow$	300000
04	$\Rightarrow$	400000
10	$\Rightarrow$	010000
11	$\Rightarrow$	001000
	$\vdots$	
20	$\Rightarrow$	020000
21	$\Rightarrow$	000020
	$\vdots$	
43	$\Rightarrow$	000400
44	$\Rightarrow$	004000

Now, suppose $\underline{\mathbf{y}} = 123321$ is received. Then

$$\underline{\mathbf{s}} = \underline{\mathbf{y}} H^T = 10$$

And by looking up the table, the most likely codeword is directly obtained:

$$\underline{\hat{\mathbf{x}}} = \underline{\mathbf{y}} - 010000 = 113321$$

Such a syndrome dictionary is especially effective for high rate codes since the syndrome dictionary is then small. For low rate codes, they are however impractical due to their exponential storage needs.

Chapter 6

More on fields

A few chapters before, fields were briefly introduced. It is now time to cover them more in depth. The aims of this section are multiple. First, to explain basic proprieties of fields to become familiar with them, then to review polynomial arithmetic over fields and lastly how to construct extension fields of a given field.

6.1 Field properties

Let us take an element $a \neq 0 \in \mathbb{F}$, where $\mathbb{F}$ is finite. By taking the series $a + a, a + a + a, ...$ there is a point where it will come back to zero and wrap around. If it was not, we would have that two different sums of a's would be equal since the field is finite. Thus the difference of them, also a sum of a's, would be zero leading to the same conclusion. Since the element can be put in evidence, it boils down to knowing how many times 1, the neutral under multiplication, can be added to itself before wrapping to zero since the same applies to any element of $\mathbb{F}$, forming cyclic subgroups of $\mathbb{F}$.

Definition 20 *The characteristic of a finite field $\mathbb{F}_q$ is the smallest integer p such that*

$$\underbrace{1 + 1 + ... + 1}_{p \text{ times}} = 0$$

The characteristic of a finite field is a prime number. Otherwise there would be divisors of zero, i.e. some $a \neq 0, b \neq 0 \in F_q$ so that $ab = 0$. By multiplying by a^{-1} we have that $b = 0$ which contradicts our hypothesis.

Theorem
The size of a finite field is a power of a prime.

Proof

Suppose p is the characteristic of the field $\mathbb{F}_q$ with $q > p$. Take a maximal set of elements $\{\beta_1, \beta_2, ..., \beta_m\}$ in $\mathbb{F}_q$ which are linearly independent over $\mathbb{F}_p$. That is, no two two sums of β's can be equal. Then the field contains, by closure, any linear combination of them.

$$\alpha_1\beta_1 + \alpha_2\beta_2 + ... + \alpha_m\beta_m \in \mathbb{F}_q$$

And no others. F_q is a vector space of dimension m over $\mathbb{F}_p$ and has thus $q = p^m$ elements. $\square$

Of course, multiplication must be defined on this vector space in order to provide the structure of a field. It turns out that all fields of same size are isomorphic. That fields of same size are isomorphic means that one can be obtained from another simply by renaming the elements. This enables us to write $\mathbb{F}_q$ without ambiguity. Before we show how to construct fields whose sizes are powers of a prime, a little introduction about polynomials over fields is necessary. In opposition to polynomials over rings, polynomials over fields have a unique factorization.

Definition 21 *A polynomial $p(x)$ in $\mathbb{F}[x]$ is called irreducible over the field $\mathbb{F}$ if it is non-constant and cannot be represented as the product of two non-constant polynomials from $\mathbb{F}[x]$.*

Example
The polynomial $x^2 + 1$ is irreducible over $\mathbb{F}_3$ but not over $\mathbb{F}_2$ where we have $x^2 + 1 = (x + 1)^2$.

The concept of modular arithmetic can be generalized to polynomials over fields. Then $\mathbb{F}[x]/p(x)$ where $p(x) \in \mathbb{F}[x]$ becomes a set equivalence classes of polynomials.

Theorem
The equivalence classes $\mathbb{F}_q[x]/r(x)$ form a field of size $q^{\deg(r(x))}$ if and only if $r(x) \in \mathbb{F}_q[x]$ is an irreducible polynomial.

The proof is similar to the one of APPENDIX A.1.

Definition 22 *The order of a non-zero element $a \in F_q$ is the smallest integer $m > 0$ so that $a^m = 1$.*

For every finite field $\mathbb{F}_q$ there exists an element of order $q-1$ which is called a *primitive element*. Thus, a finite field is composed of zero and all $q-1$ powers of this primitive element. This in turn means that $\mathbb{F}_q^*$ forms a cyclic group under multiplication whose generator is the primitive element.

Example
Let us take the irreducible polynomial $r(x) = x^3 + x + 1$ over $\mathbb{F}_2$. Said in another way, we have $x^3 = x + 1$. Then we have, along with different representations:

0	$\Leftrightarrow$	000	$\Leftrightarrow$	0
1	$\Leftrightarrow$	100	$\Leftrightarrow$	1
α	$\Leftrightarrow$	010	$\Leftrightarrow$	2
α^2	$\Leftrightarrow$	001	$\Leftrightarrow$	4
$\alpha^3 = \alpha + 1$	$\Leftrightarrow$	110	$\Leftrightarrow$	3
$\alpha^4 = \alpha^2 + \alpha$	$\Leftrightarrow$	011	$\Leftrightarrow$	6
$\alpha^5 = \alpha^2 + \alpha + 1$	$\Leftrightarrow$	111	$\Leftrightarrow$	7
$\alpha^6 = \alpha^2 + 1$	$\Leftrightarrow$	101	$\Leftrightarrow$	5

... $(\alpha^7 = 1)$

This set forms $\mathbb{F}_{2^3}$ and α is a primitive element.

The irreducible polynomial in the example above is special since the primitive element is a root of it. Such a polynomial is called a *primitive polynomial.*

Since $\mathbb{F}_q^*$ is a cyclic group of order $q-1$, we have the interesting property that $\beta^q = \beta$ for all $\beta \in F_q^*$. Therefore, every $\beta \in F_q$ is a root of $x^q - x$, giving rise to the following factorization:

$$x^q - x = \prod_{\beta \in \mathbb{F}_q} (x - \beta)$$

6.2 The extended Euclidean algorithm

We present an important algorithm which will be of later use in decoding specific codes. This algorithm is very general and not limited to finite fields. It computes the smallest common divisor of a pair of integers or a pair of polynomials in one variable. For simplicity, we explain it here by applying it on a pair of integers.

The algorithm works by decomposing every number in smaller and smaller subparts by means of divisions with quotient and remainder. Suppose we want to compute the greatest common divisor of, say, r_0 and r_1, with $r_1 < r_0$. This is denoted by $\gcd(r_0, r_1)$. We will proceed by scrambling these two numbers into smaller and smaller pieces until we obtain a divisor of both. Here is how to proceed:

$$\begin{aligned} r_0 &= q_1 r_1 + r_2 \text{ with } r_2 < r_1 \\ r_1 &= q_2 r_2 + r_3 \text{ with } r_2 < r_1 \\ &\vdots \\ r_{m-1} &= q_m r_m + 0 \text{ with } r_{m+1} = 0 \end{aligned}$$

Or, shortly, we perform:

$$r_{i-1} = q_i r_i + r_{i+1} \text{ with } r_{i+1} < r_i, 1 \leq i \leq m$$

until $r_{m+1} = 0$. That the algorithm is guaranteed to stop is straightforward since the remainders r_i are strictly decreasing at each iteration. To show that the last non-zero remainder r_m is the greatest common divisor of r_0 and r_1, we must proceed by induction. Since r_{i-1} can be expressed by a linear combination of r_i and r_{i+1}, we have, by induction, that r_0 and r_1 can both be expressed by a linear combination of r_m and r_{m+1} also. And, since the latter one is null, both r_0 and r_1 can be expressed as multiples of r_m solely.

Example
Let us compute $\gcd(654, 123)$ over the integers. We have:

$$654 = 5.123 + 39123 = 3.39 + 639 = 6.6 + 36 = 2.3 + 0$$

And thus $\gcd(654, 123) = 3$ where $654 = 214.3$ and $123 = 41.3$. During the process above, notice that the upper divisors and remainders can always be replaced by a sum of "smaller ones". For example, $654 = 5.(3.39+6)+39$, then replace 39 by smaller ones and so on until at some time, everything boils down to a multiple 3.

The extended Euclidean algorithm provides an additional side computation to find the ring elements u and v satisfying Bézout's identity:

$$\gcd(a, b) = ua + vb$$

Over the integers, either u or v is obviously negative. It works hand in hand with the basic Euclidean algorithm. Notice that the remainders, as usual with $r_{i+1} < r_i$, can be expressed as follows:

$$r_{i+1} = r_{i-1} - q_i r_i$$

And thus, r_m can be expressed, by induction, as a sum of multiples of r_0 and r_1. A practical way to keep track of the values u and v is to compute them iteratively at each step of the basic Euclidean algorithm so that:

$$r_i = u_i r_0 + v_i r_1$$

Thus, at each step, u_i and v_i must be updated as follows:

$$u_{i+1} = u_{i-1} - q_1 u_i \quad v_{i+1} = v_{i-1} - q_1 v_i,$$

with the initial values $u_0 = 1, u_1 = 0, v_0 = 0, v_1 = 1$.

Example

Let us compute $\gcd(654, 123)$ over the integers along with the values of u_i and v_i at each step.

i	r_i	q_i	u_i	v_i
0	654	/	1	0
1	123	5	0	1
2	39	3	1	-5
3	6	6	-3	16
4	3	2	19	-101

And the relation $r_i = u_i r_0 + v_i r_1$ can be verfied at each step. Another way to make apparent the computation of the u_i and v_i is the following:

$$\begin{aligned}
39 &= 654 - 5.123 \\
6 &= 123 - 3.39 \\
&= 3(654 - 5.123) \\
&= -3.654 + 16.123 \\
3 &= 39 - 6.6 \\
&= (654 - 5.123) - 6(-3.654 + 16.123) \\
&= 19.654 - 101.123
\end{aligned}$$

Lastly, note that despite the examples were taken with integers, the algorithm is much more general. It can be applied over various algebraic structures and in particular to polynomials over fields. This will be used for a decoding technique in the next chapter.

Chapter 7

Reed-Solomon codes

In 1960, I.S. Reed and G. Solomon introduced a family of error-correcting codes that are doubly blessed. The codes and their generalizations are useful in practice, and the mathematics that lies behind them is interesting.

J. Hall - lecture notes

Reed-Solomon codes, abbreviated RS codes, are designed by oversampling a polynomial constructed from the data. The message to send is mapped to a polynomial and the codeword is defined by evaluating it at several points.

Definition 23 *Generalized Reed-Solomon code*
Let $\underline{\alpha} = (\alpha_1, ..., \alpha_n)$ *be the locations where the Generalized Reed-Solomon code is evaluated, with* $\alpha_i \neq \alpha_j$ *for all* $i \neq j$*. Let* $\underline{\lambda} = (\lambda_1, ..., \lambda_n)$ *be the non-zero normalizing coefficients. Then, the* $GRS_{(n,k,\underline{\alpha},\underline{\lambda})}$ *code is defined as the set of codewords:*

$$\{(\lambda_1 f(\alpha_1), \lambda_2 f(\alpha_2), ..., \lambda_n f(\alpha_n)) | f(x) \in \mathbb{F}_q[x] \textit{ with } \deg(f(x)) < k\}$$

A classic Reed-Solomon code is the same without the normalizing coefficients which constitute the "generalization". The RS code is obtained by evaluating polynomials of degree less than k at n different *locations* α_i. The normalizing coefficients of a GRS code have a less important role since they simply rescale values at each location by a given factor λ_i. Such a GRS code can also be represented by a generator matrix of the type:

$$G = \begin{pmatrix} 1 & 1 & ... & 1 \\ \lambda_1\alpha_1 & \lambda_2\alpha_2 & ... & \lambda_n\alpha_n \\ \lambda_1\alpha_1^2 & \lambda_2\alpha_2^2 & ... & \lambda_n\alpha_n^2 \\ \vdots & \vdots & & \vdots \\ \lambda_1\alpha_1^{k-1} & \lambda_2\alpha_2^{k-1} & ... & \lambda_n\alpha_n^{k-1} \end{pmatrix}$$

However, it is sometimes more convenient to think in terms of polynomials. Let $\underline{\mathbf{m}} = (m_0, ..., m_{k-1}) \in \mathbb{F}_q^k$ be the message to send. Consider now the bijection to $f(x) = m_0 + m_1 x + ... + m_{k-1}x^{k-1}$. This polynomial is called the *encoding polynomial.* The codeword is then constructed by evaluating this polynomial at the predefined positions $\underline{\alpha} = (\alpha_1, ..., \alpha_n)$. The codeword is $\underline{\mathbf{c}} = (\lambda_1 f(\alpha_1), \lambda_2 f(\alpha_2), ..., \lambda_n f(\alpha_n))$.

Example

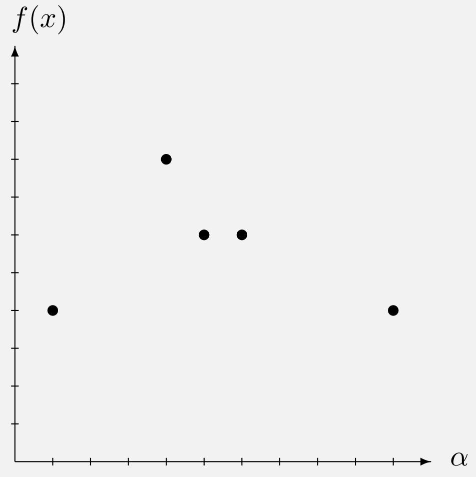
Consider the field $\mathbb{F}_{11}$ and a RS code $\mathcal{C}_{n,k}$ where $k = 3$ and $n = 5$ evaluated over $\underline{\alpha} = (1, 4, 5, 6, 10)$. with $\underline{\lambda} = \underline{\mathbf{1}}$. Let the sent message be $\underline{\mathbf{f}} = (3, 0, 1)$ so that the encoding is $f(x) = 3 + x^2$. Then the resulting codeword is $\underline{\mathbf{x}} = (4, 8, 6, 6, 4)$. Indeed:

$x_1 = f(2) = 3 + 1 = 4$

$x_2 = f(4) = 3 + 16 = 8$

...

The encoding polynomial passes through the n points $(\alpha_i, \lambda_i f(\alpha_i))$ illustrated below:



Let us make some observations about the graph. First, any permutation of the evaluation positions α_i will produce the same graph. This reflects the fact that for given points, the same function encodes it, independent of how the evaluation positions are ordered.

From the algebraic structure of RS codes we can derive many of its properties. For instance, that a Reed-Solomon code meets the Singleton bound so that we have:

$$d = n - k + 1$$

Indeed, since every polynomial is of degree at most $k-1$, it has at most $k-1$ roots. In other words, it is zero in at most $k-1$ positions. This in turn implies that at least $n-(k-1)$ of the positions where it is evaluated are non-zero values. This means that any non-zero codeword has a weight of at least $n-k+1$. Since a *GRS* code is a linear code, the minimum distance is equal to the non-zero codeword of least weight.

Another property we can derive is that the dual of a $GRS_{(n,k,\underline{\alpha},\underline{\lambda})}$ code is a $GRS_{(n,n-k,\underline{\alpha},\underline{\rho})}$ code itself, for some $\underline{\rho}$. To show this, let us consider the product of G and H:

$$GH^T = \begin{pmatrix} 1 & 1 & ... & 1 \\ \lambda_1\alpha_1 & \lambda_2\alpha_2 & ... & \lambda_n\alpha_n \\ \lambda_1\alpha_1^2 & \lambda_2\alpha_2^2 & ... & \lambda_n\alpha_n^2 \\ \vdots & \vdots & & \vdots \\ \lambda_1\alpha_1^{k-1} & \lambda_2\alpha_2^{k-1} & ... & \lambda_n\alpha_n^{k-1} \end{pmatrix} \begin{pmatrix} 1 & 1 & ... & 1 \\ \rho_1\alpha_1 & \rho_2\alpha_2 & ... & \rho_n\alpha_n \\ \rho_1\alpha_1^2 & \rho_2\alpha_2^2 & ... & \rho_n\alpha_n^2 \\ \vdots & \vdots & & \vdots \\ \rho_1\alpha_1^{n-k-1} & \rho_2\alpha_2^{n-k-1} & ... & \rho_n\alpha_n^{n-k-1} \end{pmatrix}^T = 0$$

This forms a system of $n-1$ equations of the form:

$$\sum_i \lambda_i\rho_i\alpha_i^j = 0 \quad \text{with } 0 \leq j \leq n-2$$

where the ρ_i are the unknowns. Since they are n of them, there exists a non zero solution, proving that the dual of a *GRS* code is another *GRS* code (evaluated at the same positions).

7.1 Not so classical decoding

The bounded distance decoding algorithm presented here is not the classical decoding explained in most introductory books. It is an alternative, first presented by Gao [3] offering an elegant, yet simple, solution to the decoding problem.

Theorem

GAO DECODING ALGORITHM

Input: the received word $\underline{\mathbf{y}} = (y_1, ..., y_n) \in \mathbb{F}_q^n$

Output: the encoding polynomial $f(z)$ or "Decoding failure"

- Step 1: Set $p_0(z) = \prod_{i=1}^{n}(z - \alpha_i)$ and find the (unique) polynomial $p_1(z)$ of degree $n-1$ such that $p_1(\alpha_i) = y_i$ for $i = 1, ..., n$.
- Step 2: Apply the extended euclidean algorithm on p_0 and p_1 and stop when the remainder $p_m(z)$ has degree less than $\frac{n+k}{2}$.
- Step 3: Perform the division $p_m(z) = f(z)v_m(z) + r(z)$. If $r(z) = 0$ and $\deg(f(z)) < k$ then output $f(z)$ as the encoding polynomial, else declare a "Decoding failure".

Proof

In the following proof, we assume that e errors were introduced.

$$e = |\{i | f(\alpha_i) \neq y_i\}|$$

Let

$$w(z) = \prod_i (z - \alpha_i) \ , i | f(\alpha_i) = y_i$$

$$\bar{w}(z) = \prod_i (z - \alpha_i) \ , i | f(\alpha_i) \neq y_i$$

Let us moreover define the polynomial $\Delta(z)$ of degree at most $e-1$ such that:

$$\Delta(\alpha_i) = \frac{y_i - f(\alpha_i)}{\bar{w}(\alpha_i)} \ , i | f(\alpha_i) \neq y_i$$

Since there are e constraints and $\deg(\Delta(z)) \leq e - 1$, this polynomial exists and is unique. In order to make the proof lighter and the reading easier, the variable will be hidden by default so that f is directly written instead of $f(z)$ and so on. Using these polynomials, both p_0 and p_1 can be expressed as follows:

$$p_0 = \prod_{i=1}^{n}(z - \alpha_i) = \bar{w}w$$

$$p_1 = \bar{w}\Delta + f$$

Indeed, we have $p_1(\alpha_i) = y_i$ for every i and is of degree less than n as required. As a first step, we shall prove that applying the extended Euclidean algorithm to (w, Δ) gives the same sequence of quotients, for m iterations, as if it was applied to (p_0, p_1). Let $r_0 = w$ and $r_1 = \Delta$, then by applying the algorithm we obtain the successive remainders and quotients satisfying:

$$r_{i-1} = r_i q_i + r_{i+1} \text{ , with } 1 \leq i \leq m, \deg(r_{i+1}) < \deg(r_i)$$

where $r_{m+1} = 0$. Let us now show that the same m successive quotients are obtained by applying the algorithm to p_0 and p_1. To do this, let us directly define p_i as:

$$\begin{aligned} p_i &= u_i p_0 + v_i p_1 \\ &= u_i \bar{w} w + v_i \bar{w}\Delta + f \\ &= \bar{w} r_i + v_i f \end{aligned}$$

The relationship to verify to show that the sequence of quotients is the same for both is:

$$\begin{aligned} p_{i-1} &= q_i p_i + p_{i+1} \\ \bar{w} r_{i-1} + v_{i-1} f &= \bar{w} q_i r_i + q_i v_i f + \bar{w} r_{i+1} + v_{i+1} f \end{aligned}$$

In the latter relationship, we see that the quotients indeed satisfy the equality. However, it must also be shown that $\deg(p_{i+1}) < \deg(p_i)$ in order to be correct. For $0 \leq i \leq m$, the degree of p_i is:

$$\deg(p_i) = \deg(\bar{w} r_i + v_i f).$$

The degree of each v_i is less than or equal to the degree of r_0, itself equal to e by definition. Thus, we have $\deg(v_i f) \leq e + (k-1)$ whereas the degree of $\bar{w}$ is $n - e$. Under the assumption that $e < \frac{n-k+1}{2}$ we obtain:

$$\deg(\bar{w} r_i) \geq n - e > e + k - 1 \geq \deg(v_i f)$$

And thus the degree of p_i is equal to the degree of $\bar{w} r_i$ which is strictly decreasing because the degree of the remainders r_i are strictly decreasing themselves. This ends the first part of the proof showing that the same sequence of consecutive quotients is obtained and therefore also the same u_i and v_i during the first m steps. Looking at p_{m+1} we have:

$$p_{m+1} = \bar{w} r_{m+1} + v_{m+1} f$$

And since $r_{m+1} = 0$ by definition, this expression becomes:

$$f = \frac{p_{m+1}}{v_{m+1}}$$

Giving the desired result. $\square$

Example

Consider the code $GRS[5,3,3]$ over $\mathbb{F}_5$ evaluated at $\underline{\alpha} = (0,1,2,3,4)$ with $\underline{\lambda} = \underline{\mathbf{1}}$. Assume the received word is $\underline{\mathbf{y}} = (1,3,0,2,0)$, then by applying the algorithm, we have:

$$p_0 = z^5 - z$$

And, by Lagrangian interpolation, we can find p_1 passing through all the points:

$$\begin{aligned} p_1 &= \sum_i y_i \frac{\prod_{j\neq i}(z-\alpha_j)}{\prod_{j\neq i}(\alpha_i-\alpha_j)} \\ &= 1\frac{z^4-1}{(-1)(-2)(-3)(-4)} + 3\frac{z(z-2)(z-3)(z-4)}{1(-1)(-2)(-3)} + 2\frac{z(z-1)(z-2)(z-4)}{3.2.1(-1)} \\ &= 4z^4 + z^3 + 4z^2 + 3z + 1 \end{aligned}$$

By applying the extended Euclidean algorithm, the following results are obtained:

i	p_i	q_i	u_i	v_i
0	$z^5 - z$		1	0
1	$4z^4 + z^3 + 4z^2 + 3z + 1$	$4x + 4$	0	1
2	$2x^2 + 3x + 1$	$STOP$	1	$x + 1$

and there is no need to continue further by computing the quotient q_2 since the stopping condition has already been met, namely $\deg(p_2) = 2 < \frac{n+k}{2} = 4$. By dividing the remainder p_2 by v_2, we obtain:

$$2x^2 + 3x + 1 = (x+1)(2x+1) + 0$$

And therefore our decoding is successful and the encoding polynomial is $f(z) = 2x + 1$. By re-encoding, we obtain:

$$\underline{\hat{\mathbf{x}}} = (1,3,0,2,4)$$

which differs only in the last symbol from the received word $\underline{\mathbf{y}}$.

The source code for a GRS encoder and a "Euclidean decoder" can be found in the appendix.

7.2 The key equation

This subsidiary section contains complementary material about the key equation. It constitutes the central part of several well known decoding algorithms widely used in practice.

Theorem
The dual of a $GRS_{(n,k,\underline{\alpha},\underline{\lambda})}$ code is a $GRS_{(n,n-k,\underline{\alpha},\underline{\rho})}$ code itself with:

$$\rho_i^{-1} = \lambda_i \prod_{j\neq i}(\alpha_i - \alpha_j)$$

Proof

The fact that the dual is a *GRS* code itself has been proved in the beginning of this chapter. Let us now prove that the coefficients are the correct ones. Let $f(x)$ be an encoding polynomial of degree at most $k-1$ of the *GRS* code and let $g(x)$ be an encoding polynomial of degree at most $n-k-1$ of the dual *GRS* code. Let us show that:

$$\sum \lambda_i f(\alpha_i)\rho_i g(\alpha_i) = 0$$

The function $f(x)g(x)$ can be expressed using the Lagrange interpolation as:

$$f(x)g(x) = \sum_i f(\alpha_i)g(\alpha_i)\frac{\prod_{i\neq j}(x-\alpha_j)}{\prod_{i\neq j}(\alpha_i-\alpha_j)}$$

When instantiating x to α_i, all terms vanish except the one which is equal to $f(\alpha_i)g(\alpha_i)$ as it should be.

Since the product of $f(x)$ by $g(x)$ is a polynomial of degree at most $n-2$, equating the coefficients of x^{n-1} on both sides gives:

$$\begin{aligned}
0 &= \sum_i f(\alpha_i)g(\alpha_i)\frac{1}{\prod_{i\neq j}(\alpha_i-\alpha_j)}\\
&= \sum_i \lambda_i f(\alpha_i)\frac{1}{\lambda_i\prod_{i\neq j}(\alpha_i-\alpha_j)}g(\alpha_i)\\
&= \sum_i (\lambda_i f(\alpha_i))(\rho_i g(\alpha_i))
\end{aligned}$$

Which completes the proof. □

Example
Let us define the $GRS[n = 7, k = 3, d = 5]$ over $\mathbb{F}_{11}$ evaluated at $\underline{\alpha} = (1, 2, 3, 4, 5, 6, 7)$. It can correct up to 2 errors. The multiplication table of $\mathbb{F}_{11}$ can be found in APPENDIX X.X.

$$G = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 4 & 9 & 5 & 3 & 3 & 5 \end{pmatrix}$$

The normalizing coefficients of the dual GRS code are:

$$\underline{\rho} = (9, 1, 3, 7, 3, 1, 9).$$

This leads to the following parity matrix:

$$H = \begin{pmatrix} 9 & 1 & 3 & 7 & 3 & 1 & 9 \\ 9 & 2 & 9 & 6 & 4 & 6 & 8 \\ 9 & 4 & 5 & 2 & 9 & 3 & 1 \\ 9 & 8 & 4 & 8 & 1 & 7 & 7 \end{pmatrix}$$

Knowing the expression of the paritiy check matrix, the syndrome can be expressed as follows:

$$\begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_{n-k} \end{pmatrix} = \begin{pmatrix} 1 & 1 & ... & 1 \\ \rho_1\alpha_1 & \rho_2\alpha_2 & ... & \rho_n\alpha_n \\ \rho_1\alpha_1^2 & \rho_2\alpha_2^2 & ... & \rho_n\alpha_n^2 \\ \vdots & \vdots & & \vdots \\ \rho_1\alpha_1^{n-k-1} & \rho_2\alpha_2^{n-k-1} & ... & \rho_n\alpha_n^{n-k-1} \end{pmatrix} \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_{n-k} \end{pmatrix}$$

And, by isolating each component of the syndrome:

$$s_j = \sum_i e_i \rho_i \alpha_i^{j-1}$$

Let us now take every syndrome component into account:

$$\begin{aligned} s(z) &= \sum_{j=1}^{n-k} s_j z^{j-1} \\ &= \sum_{j=1}^{n-k} \sum_i e_i \rho_i \alpha_i^{j-1} z^{j-1} \\ &= \sum_i e_i \rho_i \sum_{j=1}^{n-k} \alpha_i^{j-1} z^{j-1} \\ &= \sum_i e_i \rho_i \left(\frac{1-(\alpha_i z)^{n-k}}{1-\alpha_i z} \right) \end{aligned}$$

Since terms with $e_i = 0$ do not contribute to the latter sum, it is equivalent to consider this sum over the set of error locations $\mathcal{I} = \{i | e_i \neq 0\}$. With a few manipulations, the above equation becomes:

$$\prod_{i \in \mathcal{I}} (1-\alpha_i z) s(z) = \sum_{i \in \mathcal{I}} e_i \rho_i \prod_{j \in \mathcal{I} \setminus \{i\}} (1-\alpha_j z) \pmod{z^{n-k}}$$

This expression can be expressed more elegantly as the *key equation*:

$$\sigma(z) s(z) = \omega(z) \pmod{z^{n-k}}$$

with

$$\sigma(z) = \prod_{i \in \mathcal{I}} (1-\alpha_i z)$$

and

$$\omega(z) = \sum_{i \in \mathcal{I}} e_i \rho_i \prod_{j \in \mathcal{I} \setminus \{i\}} (1-\alpha_j z) \pmod{z^{n-k}}$$

The polynomial $\omega(z)$ is called the *error locator polynomial* because it tells us at which coordinates the errors are:

$$\mathcal{I} = \{i | \sigma(\alpha_i^{-1}) = 0\}$$

Once these error locations are known, the *error evaluator polynomial* $\omega(z)$ gives us the exact error at a given position:

$$\forall i \in \mathcal{I} : e_i = \frac{\omega(\alpha_i^{-1})}{\rho_i \prod_{j \in \mathcal{I} \setminus \{i\}} (1 - \alpha_j \alpha_i^{-1})}$$

Therefore, the decoding of a codeword necessitate to solve the key equation by finding $\sigma(z)$ and $\omega(z)$. There exist several efficient techniques to do this and we invite the interested reader to the extensive literature about this subject, for example [1].

7.3 Conclusion

Reed-Solomon codes have many advantages. They provide excellent error-correcting abilities since they reach the Singleton bound. Moreover, k can be chosen freely and therefore *GRS* codes can be used for nearly any rate. Lastly, they have efficient bounded distance decoding techniques decoding up to $\frac{n-k}{2}$ errors and polynomial running time in $\mathcal{O}(n^2)$.

One drawback they have is that they are based on large alphabets. If they are mapped to smaller alphabets, their length and dimension increase proportionally but their distance does not change. These mapped codes however remain excellent concerning error bursts since a block of successive errors affects only a few symbols in the bigger alphabet. Because error bursts is a common pattern, these codes are widely used in practice. Let us stress again the fact that mapping a *GRS* code to a smaller alphabet is not well suited if the errors are completely random.

Part II

Soft decoding

Chapter 8

Introduction

8.1 What do we achieve here?

The algebraic algorithm we present here is a breakthrough in the field. Classical algorithms either perform hard-decoding up to $\frac{n-k}{2}$ errors, or are running in exponential time and were thus of very limited practical use due to their complexity exploding behavior. This algorithm outperforms both by providing soft-decoding in polynomial time. Moreover, it can handle hard-decoding also as a special case. For both, the gains are substantial. For hard-decoding, it provides list-decoding beyond the classical $\frac{n-k}{2}$ bound for low rates easily. For high rates this is also the case but the cost grows drastically in this latter case. However, its real strength lies in soft-decoding, taking advantage of the full information available. Using soft-decoding, the performances improvement of the error correction ability is substantial while running in polynomial time. Moreover, it has some other advantages. For example, the trades-off between decoding complexity and decoding performance can easily be fine tuned. Another side effect is that erasures are naturally handled. Moreover, and not unimportant, the algorithm can be extended to BCH and algebraic-geometry codes. Lastly, as bonus if I may say, the algorithm provides beautiful underlying mathematics.

8.2 History and results

In 1996, Sudan presented a new kind of algorithm for hard-decoding Reed-Solomon codes [5]. He placed the problem in a new light and transformed the problem of decoding to a problem of polynomial fitting. This led to a new algorithm providing list-decoding beyond the error-correction radius of

$\frac{n-k}{2}$ for low rate RS codes. This triggered a renewed and intense interest in the research community. A few years later, Sudan and Guruswami extended the algorithm and provided a better error-correction ability at any rate [6]. However, increased error-correction ability comes hand-in-hand with increased computational cost. For high rate codes, these are significant. Then Koetter and Vardy generalized it one step further to take advantage of the soft-information from the channel [9]. This leads to the algebraic soft-decision Reed-Solomon decoding algorithm we present here. The algorithm has two main parts consisting in interpolation and factorization of bivariate polynomials (over finite fields). For these two tasks, efficient solutions were presented by [12] and [13]. Research is still active in this "hot" area, various modifications and improvements were already published and further advancements are to be expected.

8.3 General outline

The first chapter covers Sudan's initial algorithm in full length. It offers an understanding of the global process, of the algorithm as a whole and will be valid for the other algorithms too. Then, the extension of Guruswami-Sudan is implicitly presented in chapter two leading directly to a presentation of the core theorem of the algorithm: a condition on successful decoding. This is presented in a fashion naturally made to be extended to the soft-decoding adaptation from Koetter and Vardy in chapter three. This one consists of a mapping of the soft-information from the channel to the multiplicity matrix introduced later and a few comments on performances. While chapter one gave the overview and chapter two and three give reasons on why it works and what conditions should be satisfied, the two next chapter are about practical tasks. As for Sudan's initial algorithm, two main steps remain: the interpolation and the factorization of a bivariate polynomial. Efficient ways of performing these are seen in chapter four and five, respectively. Lastly, some recent improvements are briefly outlined, and a conclusion is given.

Chapter 9

The Sudan algorithm

In this chapter, we cover completely Sudan's algorithm which pioneered the field. This algorithm was the first in its genre and the source of all the subsequent algorithms like the algebraic soft decoding which is in fact just a layer upon it. Therefore, understanding the easier Sudan algorithm first helps greatly to understand the big picture and to grasp some of the underlying concepts also valid for the other algorithms.

9.1 Introduction

In 1996, Madhu Sudan presented in [5] a new and innovative way of decoding Reed-Solomon and algebraic geometry codes. Before, every decoding scheme was based on the analysis of the syndrome. The new approach of Sudan is based on a geometrical interpretation of the problem. This did not only opened new ways of considering the decoding problem of Reed-Solomon codes, but also provided a more powerful *list-decoding.* Before, all the classical decoding algorithms decoded at a distance up to $\frac{n-k}{2}$. We will see that Sudan's algorithm performs *list-decoding* beyond this bound for low rate codes.

9.2 Decoding problem reformulation

As seen in chapter 7, codewords from a RS code are the evaluation of an encoding polynomial at given locations. Let some $f(z)$ be the encoding polynomial and let $\underline{\alpha} = (\alpha_1, ..., \alpha_n)$ be the locations where it is evaluated. The codeword can be represented as a set of n points $(\alpha_i, f(\alpha_i))$ as we saw before.

Let $\underline{\mathbf{x}} = (x_1, ..., x_n)$ be the sent codeword so that $x_i = f(\alpha_i)$, and $\underline{\mathbf{y}} = (y_1, ..., y_n)$ be the received word. Moreover, let e be the numbers of errors. In other words, $x_i \neq y_i$ at e different positions. This also means that by comparing (α_i, x_i) with (α_i, y_i), we see that e points differ, those corrupted by noise, while $n - e$ are identical. A polynomial $f(x)$ is said to *pass through* a point (α, β) if $f(\alpha) = \beta$.

So, by paraphrasing what is said above, the encoding polynomial passes through $n - e$ of the received points. This leads to the objective to find all polynomials passing through at least $n - e$ of the received points.

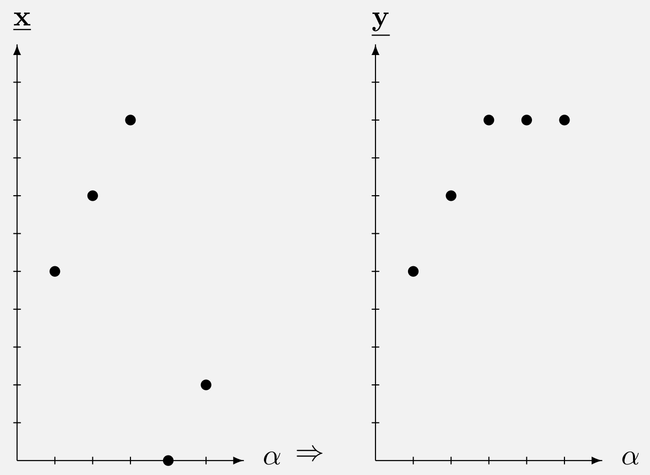
Example

The encoding polynomial $2z+3$ over $\mathbb{F}_{11}$ evaluated at $\underline{\alpha} = (1,2,3,4,5)$ gives the following set of points:

$$\{(\alpha_1, x_1), ..., (\alpha_5, x_5)\} = \{(1,5), (2,7), (3,9), (4,0), (5,2)\}$$

Now, suppose that the received codeword has been corrupted in the second and third position ($|e| = 2$) so that the following set of points is obtained:

$$\{(\alpha_1, y_1), ..., (\alpha_5, y_5)\} = \{(1,5), (2,7), (3,9), (4,9), (5,9)\}$$



Here, the polynomials of degree less than 2, i.e. lines, passing through at least 3 points from (α_i, y_i) are:

- $2z + 3$ which passes through (1,5), (2,7) and (3,9)

- 9 which passes through (3,9), (4,9) and (5,9)

In this case, both codewords would be at a distance 2 from $\underline{\mathbf{y}}$ and are two equiprobable candidates for decoding.

What we get is a list of all polynomials corresponding to codewords at a distance at most e from the received codeword $\underline{\mathbf{y}}$. To our delight, the list is usually rather short. Before going further, it should be noted that we do not know the value of e beforehand. Therefore, we will instead use a constant τ denoting the decoding radius, or in other words, the number of points that can differ.

As can be seen, the decoding problem has shifted from finding the nearest codeword to a polynomial fitting problem. This problem translation has two advantages. First, the polynomial fitting problem is more general since it can be generalized to other fields like the reals or complex while it can also be used in a much larger panel of applications. Secondly, this abstraction enables us to put codes by side. It is now a polynomial fitting problem and can be handled independently, without requesting any error-correcting codes knowledge. However, the connection will be kept all the way since it is the aim of this work. To resume the whole, let us restate the problem to refresh our mind:

Input:

- n points: (α_i, y_i) for $1 \leq i \leq n$
- the decoding radius τ
- the maximum degree $k-1$ of the fitting polynomial

Output:

- a list $\mathcal{L}$ containing all polynomials $f(z)$ of degree less than k passing through at least $n-\tau$ points.

As a reminder, this is equivalent to the decoding problem. Indeed, the polynomials we will get will be the ones that encode codewords being at a distance less than or equal to τ from the received codeword.

The main question this triggers is: how to (efficiently) find those polynomials? Subsidiary questions are: for which values of τ does it work? And lastly, how likely is it to obtain more than one fitting polynomial? All these questions will be answered in this order.

It is now time to introduce Sudan's algorithm. Before we attack the core of the algorithm and the key aspect, we will need to define some vocabulary.

9.3 Weighted degree

Definition 24 *The (a, b) weighted degree of a monomial $x^i y^j$ is defined as:*

$$wdeg_{(a,b)} x^i y^j = ai + bj$$

Any polynomial is by definition just a finite weighted sum of monomials: $Q(x, y) = \sum q_{ij} x^i y^j$. It is therefore natural to extend the definition of weighted degree to polynomials in the following manner:

Definition 25 *The (a,b) weighted degree of a polynomial $Q(x, y)$ is the largest weighted degree of its monomials.*

Example
$\text{wdeg}_{(1,7)} x^2 y^3 = 2 + 3 * 7 = 23$
$\text{wdeg}_{(1,4)}(x^6 + x^3 y + xy^2) = \max(6, 7, 9) = 9$

The notion of weighted degree can be extended to any number of variables but two will suffice for our needs. The use of this can be illustrated in the following situation. Consider a bivariate polynomial $Q(z, F)$ and a univariate polynomial $f(z)$. If we replace F by $f(z)$, we get:

$$Q(z, f(z)) = \sum q_{ij} z^i (f(z))^j$$

Therefore, if the maximum degree of $f(z)$ is at most $k-1$ then the maximum degree of $Q(z, f(z))$ is upper bounded by the $(1, k-1)$ weighted degree of $Q(z, F)$.

9.4 Overview

The algorithm is based on a bivariate polynomial called $Q(z, F)$ [1] whose $(1, k-1)$ weighted degree is Ω. When F is replaced by a polynomial $f(z)$

[1] It is often noted as $Q(x, y)$ in the literature but we have preferred the $Q(z, F)$ notation in this text. The reason of this choice is to improve clarity and to avoid possible confusion between the variables x, y and the codewords $\underline{\mathbf{x}}, \underline{\mathbf{y}}$.

of degree less than k in $Q(z, F)$, we get a univariate polynomial $Q(z, f(z))$ of degree less than or equal to Ω as seen previously.

Now, assume that we choose $Q(z, F)$ so that $Q(\alpha_i, y_i) = 0$ for every i. We say that $Q(z, F)$ is zero or vanishes at this point. An interesting viewpoint is the algebraic curve defined by $Q(z, F) = 0$, this curve passes through every point (α_i, y_i).

If $f(\alpha_i) = y_i$ for some α_i then $Q(z, f(z))$ is zero as well for this value α_i since $Q(\alpha_i, f(\alpha_i)) = Q(\alpha_i, y_i) = 0$. Thus, every α_i such that $f(\alpha_i) = y_i$ is also a root of the polynomial $Q(z, f(z))$. Therefore $Q(z, f(z))$ is either the zero polynomial or has as many roots as the number of points that $f(z)$ passes through.

In the case $f(z)$ passes through more than Ω points, the polynomial $Q(z, f(z))$ should also have as many roots. Since a non-zero polynomial cannot have more roots than its degree, there is no other possibility than being the zero polynomial. To summarize it, if $f(z)$ passes through more than Ω points, then $Q(z, f(z)) = 0$. Which in turn implies that $f(z)$ is an F-root of $Q(z, F)$!

This condensed explanation will now be seen in more detail, step by step and illustrated by a few examples for a better understanding.

9.5 Constraints for Q

The polynomial $Q(z, F)$ should have two properties, it should vanish at all points and have a weighted degree as small as possible. More formally,

Let $Q(z, F) = q_{ij} z^i F^j$, with $q_{ij} \in \mathbb{F}_q$. We want to ensure that:

- $Q(\alpha_i, y_i) = 0$ for every i.
- $\Omega = wdeg_{(1,k-1)} Q(z, F)]$ is minimal.

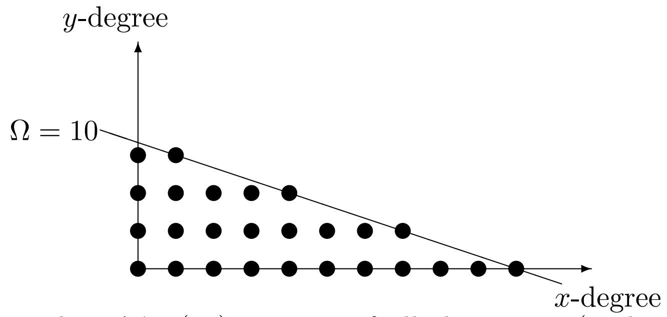
The first condition regroups n constraints, one for each point, that we must satisfy. To find an appropriate $Q(z, F)$ it is therefore sufficient to solve a system of n equation, one for each point, where the unknowns are the coefficients q_{ij} of the polynomial and the $z^i F^j$ terms are evaluated at each point. A non-trivial solution exists provided that the number of unknowns is greater than n. Or in other words, we need more than n distinct monomials at hand.

9.6 Monomial enumeration

The second condition requests that the weighted degree Ω of $Q(z, F)$ should be minimal, while the number of monomials is greater than n. It is obvious that the number of monomials is directly dependent on the maximum weighted degree Ω they can have.

Definition 26 *The set $\mathcal{M}_{a,b}(\Omega)$ denotes the set of monomials having an (a, b) weighted degree less than or equal to Ω.*

The monomials can be conveniently represented on an infinite array. On the picture below are shown all monomials of (1,3) weighted degree less than or equal to 10.



Observe that $\mathcal{M}_{1,3}(10)$ consists of all the points (with integer coordinate) under the line going from $y = 10/3$ to $x = 10$. Thus, the number of monomials of (a, b) weighted degree less than Ω is strictly greater than the area of the triangle $\frac{\Omega^2}{2ab}$.

Let us now compute the exact number of monomials having a $(1, k-1)$ weighted degree at most Ω. Let $L = \lfloor \Omega/(k-1) \rfloor$, there are:

- $\Omega + 1$ monomials of the form x^i
- $\Omega + 1 - (k-1)$ monomials of the form $x^i y$
- $\Omega + 1 - 2(k-1)$ monomials of the form $x^i y^2$
- ...

Therefore, we obtain:

$$\begin{aligned} |\mathcal{M}_{1,k-1}(\Omega)| &= \sum_{i=0}^{L}(\Omega + 1 - i(k-1)) \\ &= (L+1)(\Omega + 1 - L(k-1)/2) \\ &> \frac{\Omega^2}{2(k-1)} \end{aligned}$$

Example
What is the amount of distinct monomials having a (1,3) weighted degree at most 4? The answer is $\left|\mathcal{M}_{(1,3)}(4)\right| = 7$ and the corresponding list of monomials is: $\mathcal{M}_{(1,3)}(4) = 1, x, x^2, x^3, x^4, y, xy$

9.7 Roots vs degree of Q

Let us now go to the core theorem which reveals the use of this.

Theorem
Given n points (α_i, y_i) where all α_i have distinct values and let $\mathrm{wdeg}_{1,k-1}(Q(z,F)) = \Omega$.

If

- $Q(\alpha_i, y_i) = 0$ for $i \in [1...n]$
- $f(\alpha_j) = y_j$ for $j \in J$ where J is a subset of $[1...n]$
- $\deg(f(z)) < k$ and $|J| > \Omega$

Then

$$(F - f(z))|Q(z,F)$$

Proof

If $f(\alpha_j) = y_j$ and $Q(\alpha_j, y_j) = 0$ then

$$Q(\alpha_j, f(\alpha_j)) = 0$$

$$\Updownarrow$$

$$(z-\alpha_j)|Q(z,f(z))$$

And $Q(z,f(z))$ is either the all-zero polynomial or it can be expressed as

$$Q(z,f(z)) = \prod_{j\in J}(z-\alpha_j)\tilde{Q}(z)$$

For some $\tilde{Q}(z)$. On one side the degree of $Q(z,f(z))$ is at most Ω and on the other side, $Q(z,f(z))$ has $|J|$ roots. Since $|J| > \Omega$ by hypothesis and the fact that a polynomial connot have more distinct roots than its degree, $Q(z,f(z))$ must therefore be the all-zero polynomial so that $f(z)$ is an F-root of $Q(z,F)$. $\square$

This leads to Sudan's algorithm which can be summarized as follows:

- Find the polynomial $Q(z,F)$ of least weighted degree passing through all the points (α_i, z_i).
- Factorize the polynomial and consider the factors of the form $(F - f(z))$.
- Output the list of polynomials $f(z)$ of degree less than k passing through the most points.

9.8 Example

We will summarize this section with a small but complete example illustrating the algorithm. Let us work in the field $\mathbb{F}_{11}$ with the following parameters:
$n = 5$
$k = 2$
$d = 5 - 2 + 1 = 4$
$k - 1 = 1$ (lines)

The encoding polynomial shall be $x+1$ evaluated at $(1,2,3,4,5)$ so that we have:

$$\underline{\mathbf{x}} = (2,3,4,5,6)$$

Assume the received codeword is:

$$\underline{\mathbf{y}} = (2,3,4,0,0)$$

Can we find (a list containing) the correct codeword back? We see that when $\Omega = 2$ we have $|\mathcal{M}_{1,k-1}(\Omega)| = 6$ which is sufficient to find a $Q(x,y)$ satisfying the 5 constraints. Moreover, if $\mathrm{wdeg}_{1,1}Q(z,F) \leq 2$, we can find any polynomial passing though at least 3 points, which means that we can indeed correct 2 errors! (Which could not be possible with classical decoding!)

Let $Q(x,y) = \sum_{i+j\leq 3} q_{ij}x^i y^j$. The coefficients must satisfy following constraints:

$$\begin{aligned}
q_{00}\ 1 + q_{10}\ 1 + q_{01}\ 2 + q_{20}\ 1 + q_{11}\ 2 + q_{02}\ 4 &= 0\\
q_{00}\ 1 + q_{10}\ 2 + q_{01}\ 3 + q_{20}\ 4 + q_{11}\ 6 + q_{02}\ 9 &= 0\\
q_{00}\ 1 + q_{10}\ 3 + q_{01}\ 4 + q_{20}\ 9 + q_{11}\ 1 + q_{02}\ 5 &= 0\\
q_{00}\ 1 + q_{10}\ 4 + q_{01}\ 5 + q_{20}\ 5 + q_{11}\ 3 + q_{02}\ 3 &= 0\\
q_{00}\ 1 + q_{10}\ 5 + q_{01}\ 6 + q_{20}\ 3 + q_{11}\ 8 + q_{02}\ 3 &= 0
\end{aligned}$$

By resolving this system of equations, we get:

$$Q(z,F) = -F - zF + F^2$$

And by factorizing, we obtain:

$$Q(z,F) = F(F - (z+1))$$

which contains one erroneous polynomial (the zero polynomial) and the encoding polynomial $z+1$ giving back the correct codeword $(2,3,4,5,6)$.

Concerning the previous step, it should be clear that $Q(z,F)$ has $(z+1)$ as F-root. Since $Q(z,(z+1))$ is zero at $z = 1,2$ and 3 while being of degree at most 2, $Q(z,(z+1))$ can only be zero itself. Indeed, we can verify:

$$Q(x,(x+1)) = -(x+1)-x(x+1)+(x+1)^2 = -x-1-x^2-x+x^2+2x+1 = 0$$

We hope this illustrative example enlightens the reader about the concepts used.

9.9 Performances

To finish this section, let us look at how many errors in polynomial fitting we can handle. In order to find $Q(z,F)$ passing through all the points, we must satisfy:

$$|\mathcal{M}_{1,k-1}(\Omega)| > n$$

where Ω is the $(1, k-1)$ weighted degree of $Q(z, f(z))$. This condition is fulfilled if

$$\frac{\Omega^2}{2(k-1)} > n$$

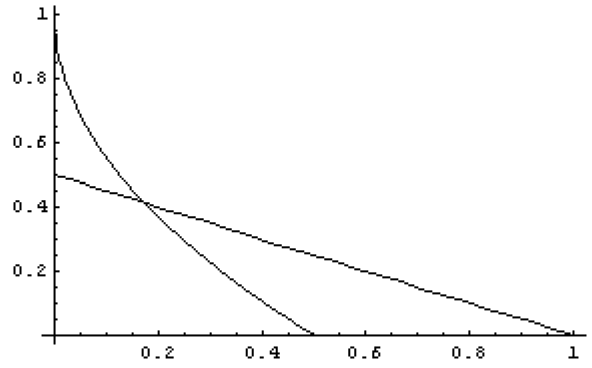
. The decoding is successful when $f(z)$ passes through more than Ω points. In other words, there should be less than $n - \Omega$ errors. By putting both formulas together, the algorithm is succesful if:

$$e < n - \Omega = n - \sqrt{2n(k-1)}$$

This bound can be fine-tuned by looking at the inequality more closer but it is sufficient to illustrate the better performances at low code rates. Let us remember that, in comparison, classical decoding algorithms correct up to $e = \frac{n-k}{2}$ errors. Let us define $\epsilon = e/n$ as the error correcting ability. When n tends to infinity, we have:

- For classical decoding: $\epsilon = \frac{1-r}{2}$
- For Sudan's algorithm: $\epsilon = 1 - \sqrt{2r}$

And both are illustrated on the graph below.



The error correction ability is high with fitting functions of low degree but drops rapidly as the degree increases. This algorithm is very interesting for low rate codes, however it becomes quickly inefficient as the code rate grows. But as we will see, there are ways to improve a lot this algorithm to get good results also for higher degree fitting functions, and thus for higher code rates.

Chapter 10

Zeros, roots and constraints

This chapter is devoted to understanding the relationship between higher order zeros, the roots of $Q(z, f(z))$ and the constraints on the polynomial $Q(z, F)$ to satisfy these zeros. Using the algebraic curve comparison, it means that the curve passes through given points several times. In Sudan's algorithm, $Q(z, F)$ is determined so that it is zero at each received point. The idea remains the same but it is generalized to each point with zeros of any multiplicity. The consequences of this will be the core theorem of the algorithm seen in the next chapter.

10.1 Zeros of higher multiplicity

For a univariate polynomial $f(x) = f_0 + f_1x + f_2x^2 + f_3x^3 + f_4x^4 + ...$, we say that $f(x)$ has a zero of multiplicity (or order) m at the origin if $f_0 = f_1 = ... = f_{m-1} = 0$ so that it can be expressed as $f(x) = \sum_{i \geq m} f_i x^i$. Or equivalently we can say that $x^m | f(x)$. All these conditions are strictly equivalent. As a side note, a null multiplicity is no zero at all.

To know the multiplicity of a zero at a given point α, we just have to translate the polynomial so that its new origin is at α. This corresponds to the translated polynomial $f(x + \alpha)$. Therefore, the zero multiplicity of $f(x)$ at α is simply the zero multiplicity of $f(x + \alpha)$ at the origin.

We can define the multiplicity of zeros for $Q(z, F)$ exactly the same way.

Definition 27 *A polynomial $Q(z, F)$ has a zero of multiplicity or order m at (α, β) if $Q(z + \alpha, F + \beta) = \sum_{i+j \geq m} q_{ij} z^i F^j$.*

$Q(z + \alpha, F + \beta)$ is simply the polynomial $Q(z, F)$ where the origin has been translated to the point (α, β). Having a multiplicity of m just means that this translated polynomial has no monomials of degree less than m.

Recall that in the Sudan algorithm, we proved that if $f(\alpha) = \beta$ and $Q(\alpha, \beta) = 0$ then $(z - \alpha)|Q(z, f(z))$. This result will now be extended to zeros of higher multiplicities in the following theorem.

Theorem
DIVISORS THEOREM
If $Q(z, F)$ has a zero of order m at (α, β) and $f(\alpha) = \beta$ then

$$(z - \alpha)^m | Q(z, f(z))$$

Proof

By translation of z to $z + \alpha$, proving that $(z - \alpha)^m|Q(z, f(z))$ is equivalent to proving that $z^m|Q(z + \alpha, f(z + \alpha))$.

Let us now consider the polynomial $f(z + \alpha) - \beta$. Since it is zero at the origin (for $z = 0$), we can rewrite it to become $f(z + \alpha) - \beta = z\tilde{f}(z)$ for some $\tilde{f}(z)$.

By replacing $f(z + \alpha)$ in the translated equation, we get $Q(z + \alpha, z\tilde{f}(x) + \beta)$ and since $Q(z, F)$ has a zero of order m at (α, β) we get the final expression:

$$Q(z + \alpha, z\tilde{f}(z) + \beta) = \sum_{i+j \geq m} q_{ij} z^i z^j \tilde{f}(z)^k$$

which is divisible by z^m as we wanted to show. □

Example
Let $f(z) = z^2 - 4$ and $Q(z, F) = (F - z - 2)^2$.

When $(\alpha, \beta) = (3, 5)$, it can be shown that $f(\alpha) = \beta$ and $Q(z, F)$ has a zero of multiplicity 2 at this point. Let us verify this. For $f(\alpha)$ we have:

$$f(3) = 9 - 4 = 5.$$

And for $Q(z + \alpha, F + \beta)$ we have:

$$Q(z + 3, F + 5) = ((F + 5) - (z + 3) - 2)^2 = (F - z)^2$$

which has indeed a zero of multiplicity two.

Lastly, the theorem tells us that $(z-3)^2|Q(z,f(z))$ in this case and we can verify:

$$Q(z,f(z)) = (z^2-z-6)^2 = (z-3)^2(z+2)^2.$$

And the desired result is obtained.

This can now be extended to any number of points. If $f(z)$ passes through n such points (α_i,β_i) and $Q(z,F)$ has zeros of multiplicity m_i at these, then we have:

$$\prod_i (z-\alpha_i)^{m_i}|Q(z,f(z))$$

This is exactly the same reasoning as for Sudan's algorithm.

10.2 Expressing the constraints

Let us now rename the parameters of $Q(z,F)$ to become $Q(x,y)$. This is just a renaming for convenience, x has nothing to do with a codeword and just stands for z. And as well about y which just stands for F. The subject of this section is to determine the constraints that $Q(x,y)$ must satisfy in order to have zeros of given multiplicities at given points. Or better said, if we express $Q(x,y)$ as:

$$Q(x,y) = \sum q_{ij}x^iy^j$$

The question becomes: how can the coefficients q_{ij} be chosen so that $Q(x,y)$ has indeed zeros of given multiplicities at the given points? More precisely, for some points (α,β), the polynomial $Q(x,y)$ must have a zero of multiplicity m. This means that

$$Q(x+\alpha,y+\beta) = \sum_{i+j\geq m} \tilde{q}_{(\alpha,\beta),ij}x^iy^j$$

where $\tilde{q}_{(\alpha,\beta),*}$ are the coefficients of the polynomial translated to the point (α,β).

Since the translation is a linear transformation, the *translated coefficients* $\tilde{q}_{(\alpha,\beta),ij}$ can be expressed as a linear combination of the original ones. To see this, let us decompose the translated polynomial:

$$\begin{aligned} Q(x+\alpha, y+\beta) &= \sum_{i,j} q_{ij}(x+\alpha)^i (y+\beta)^j \\ &= \sum_{i,j} q_{ij} \left[\sum_{u=0}^{i} \begin{pmatrix} i \\ u \end{pmatrix} x^u \alpha^{i-u}\right] \left[\sum_{v=0}^{j} \begin{pmatrix} j \\ v \end{pmatrix} y^v \beta^{j-v}\right] \end{aligned}$$

Now we can establish a relationship between the coefficients q_* of the initial polynomial and the coefficients $\tilde{q}_{(\alpha,\beta),*}$ of the translated one. By isolating terms of given degree $x^u y^v$ from $Q(x+\alpha, y+\beta)$, we obtain:

$$\tilde{q}_{(\alpha,\beta),uv} = \text{coef}(x^u y^v) = \sum_{i \geq u, j \geq v} q_{ij} \begin{pmatrix} i \\ u \end{pmatrix} \begin{pmatrix} j \\ v \end{pmatrix} \alpha^{i-u} \beta^{j-v}$$

In a field of characteristic 0, this awkward expression is equivalent to a very convenient one, namely the *Hasse derivative.*

Definition 28 *The* (u, v)*th* Hasse (mixed partial) derivative *of* $Q(x, y)$ *evaluated at* (α, β) *is:*

$$DH_{u,v,(\alpha,\beta)}\ Q(x,y) = \left(\frac{1}{u!v!} \frac{d^u}{dx^u} \frac{d^v}{dy^v} Q(x,y)\right)\Bigg|_{(\alpha,\beta)}$$

Indeed, this definition is equivalent to the previous expression (...provided the field is of characteristic 0). The sharp-eyed reader may have noticed that in other fields, the Hasse derivative would in some cases reduce to $\frac{0}{0}$ if we treat it in a rigorous way. For example, the p_{th} derivative of x^n in $\mathbb{F}_q$, where q is a power of p, would reduce to $\frac{0}{0}$ since in this field $p \equiv 0$. Therefore, the two expressions are quite similar but are not strictly equivalent because of the field characteristic. From now on, we shall use the notation $D_{u,v,(\alpha,\beta)}$ to express the coefficients $\tilde{q}_{(\alpha,\beta),uv}$ to remind the extreme similarity with the Hasse derivative.

$$\tilde{q}_{(\alpha,\beta),uv} = \text{coef}(x^u y^v) = D_{u,v,(\alpha,\beta)} Q(x,y)$$

Example

Let $Q(x, y) = x^3 + y^2 + xy \in \mathbb{F}_{11}[x, y]$ and the point $(2, 5)$. What are the coefficients $\tilde{q}_{(2,5),ij}$ of $Q(x + 2, y + 5)$?

We have:

$$\begin{array}{lcl} D_{0,0,(2,5)}\ Q(x,y) = & (x^3+y^2+xy)\big|_{(2,5)} & = 43 = -1(mod 11) \\ D_{1,0,(2,5)}\ Q(x,y) = & (3x^2+y)\big|_{(2,5)} & = 17 = 6(mod 11) \\ D_{2,0,(2,5)}\ Q(x,y) = & (3x)|_{(2,5)} & = 6(mod 11) \\ D_{3,0,(2,5)}\ Q(x,y) = & 1 & \\ D_{0,1,(2,5)}\ Q(x,y) = & (2y+x)|_{(2,5)} & = 12 = 1(mod 11) \\ D_{1,1,(2,5)}\ Q(x,y) = & 1 & \\ D_{0,2,(2,5)}\ Q(x,y) = & 1 & \end{array}$$

Thus, $Q(x+2, y+5)$ can be expressed as:

$$Q(x+2, y+5) = y^2 + xy + y + x^3 + 6x^2 + 6x - 1$$

In order to have a zero of multiplicity m at (α, β), we must have all the coefficients of the translated polynomial $\tilde{q}_{i,uv} = 0$ whenever $u+v \leq m$. This is equivalent to satisfying the set of constraints:

$$\mathcal{D} = \{D_{u,v,(\alpha,\beta)}\ Q(x,y) = 0 | u + v \leq m\}$$

And this set contains $\frac{m(m+1)}{2}$ linear constraints since this is the number of couples u, v satisfying $u + v \leq m$.

This can be generalized to any number of points with zeros of any multiplicities easily. Moreover, each constraint can be expressed as a linear combination of the coefficients from the initial polynomial $Q(x, y)$. A system of linear equations is obtained where each equation expresses a constraint.

Example
Let $Q(x,y) = q_{00} + q_{10}x + q_{20}x^2 + q_{01}y + q_{11}xy$. Is it possible to find values for the different q_{ij} so that $Q(x,y)$ has a zero of order 2 at $p = (2,3)$?

The answer is yes! Indeed $Q(x,y)$ has a zero of order 2 at p if it satisfies this set of constraints:

$$\mathcal{D} = \{D_{0,0,p}, D_{1,0,p}, D_{0,1,p}\}$$

So that the following system of equations is obtained:

$$\begin{array}{rcl} D_{0,0,p}Q & = & q_{00} + 2q_{10} + 4q_{20} + 3q_{01} + 6q_{11} = 0 \\ D_{1,0,p}Q & = & q_{10} + 4q_{20} + 3q_{11} = 0 \\ D_{0,1,p}Q & = & q_{01} + 2q_{11} = 0 \end{array}$$

By solving it, we obtain that $Q(x,y)$ can be any linear combination of:

$$4 - 4x + x^2 \text{ and } 6 - 3x - 2y + xy$$

When we take a closer look at both of these components, we see that they are equivalent to:

$$(x-2)^2 \text{ and } (x-2)(y-3)$$

And it is easily seen that both of these components have second order zeros at $(2,3)$.

To conclude, let us say that $Q(x,y)$ can be found in polynomial time since it is sufficient to solve a system of linear equations expressing the required constraints. Since a zero of multiplicity m_i at a point p_i can be expressed as $\frac{m_i(m_i+1)}{2}$ linear constraints on the coefficients of $Q(x,y)$, we obtain a system of linear equations with $\sum_i \frac{m_i(m_i+1)}{2}$ constraints.

Finding a nonzero polynomial satisfying these constraints can be done by solving a system of linear equations provided that there are more coefficients than constraints. Therefore, the weighted degree of $Q(x,y)$ must be just high enough to have more coefficients than this value.

Chapter 11

The core theorem

11.1 Multiplicity Matrix

In order to keep track of the zeros of $Q(z,F)$, more precisely of where they are located and what multiplicities they have, let us define the multiplicity matrix. From now on, we will assume that $\underline{\alpha} = (\alpha_1, ..., \alpha_n)$ are the locations where the RS code is evaluated and $\underline{\beta} = (\beta_1, ..., \beta_q)$ be the vector containing all the elements of $\mathbb{F}_q$.

Definition 29 *The multiplicity matrix $M \in \mathbb{N}^{n\times q}$ is defined so that for all i and j, the polynomial $Q(z,F)$ has a zero of multiplicity m_{ij} at the point (α_i, β_j).*

Example

Let the field we are working in be $\mathbb{F}_5$ so that $\underline{\beta} = (0,1,2,3,4)$ and let $\underline{\alpha} = (0,2,4)$. Then the following multiplicity matrix:

$$M = \begin{pmatrix} 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 8 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 \end{pmatrix}$$

means that the polynomial $Q(z,F)$ has:

- A zero of multiplicity 7 at (0,3)
- A zero of multiplicity 8 at (2,2)
- A zero of multiplicity 9 at (4,1)

Before stating the core theorem of this section, a little vocabulary and tools are needed. Two important notions associated with M are its *cost* and the *score* of a word for M that we will define shortly. But first, let us define the inner product since it will help us to define the two others.

Definition 30 *The inner product $\bullet$ of two matrices A and B is defined as: $A \bullet B = \sum_{ij} a_{ij} b_{ij}$*

Now that we know what the inner product is, the cost and the score can be defined with more ease.

Definition 31 *The cost of the matrix M is defined as:*

$$Cost(M) = \frac{1}{2} M \bullet (M + J) = \sum_{ij} \frac{1}{2} m_{ij}(m_{ij} + 1)$$

where J is the all one matrix.

The cost of a matrix M is the number of linear constraints on the coefficients of $Q(z, F)$ that must be satisfied.

Definition 32 *The score of the word $\underline{\mathbf{w}}$ for the matrix M is*

$$Score_M(\underline{\mathbf{w}}) = M \bullet \langle \underline{\mathbf{w}} \rangle = \sum_{i,j | x_i = \beta_j} m_{ij}$$

Where $\langle \underline{\mathbf{w}} \rangle$ is the $n \times q$ matrix with entries $\langle \underline{\mathbf{w}} \rangle_{ij} = 1$ when $x_i = \beta_j$ and $\langle \underline{\mathbf{w}} \rangle_{ij} = 0$ else. In other words, for each row i, there is a 1 at the column j when $w_i = \beta_j$ and 0 elsewhere.

The following brief example illustrates these definitions and will certainly reveal their simplicity.

Example
Consider the matrix M of the previous example.

$$M = \begin{pmatrix} 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 8 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 \end{pmatrix}$$

Its cost of M is:

$$Cost(M) = 28 + 36 + 45 = 109,$$

and the score of a word $\underline{\mathbf{w}} = (3, 0, 1)$ is:

$$Score_M(\underline{\mathbf{w}}) = \begin{pmatrix} 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 8 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 \end{pmatrix} \bullet \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} = 7 + 0 + 9 = 16.$$

The score is the sum of the multiplicities associated with the points (α_i, w_i).

Using these two notions, the following theorem can be stated elegantly. It defines a condition for which $f(z)$ is an F-root of $Q(z, F)$ and is the cornerstone of the whole algorithm.

Theorem
DECODING THEOREM
Let $Q(z, F)$ be the polynomial satisfying the multiplicities defined by M with $\mathrm{wdeg}_{1,k-1}(Q(z, F)) = \Omega$, and let $f(z)$ be an encoding polynomial evaluated in $\underline{\alpha}$ resulting in the codeword $\underline{\mathbf{c}}$ (we have $f(\alpha_i) = c_i$).
If

$$|\mathcal{M}_{1,k-1}(\Omega)| > Cost(M) \quad \text{and} \quad Score_M(\underline{\mathbf{c}}) > \Omega(M)$$

then

$$(F - f(z)) | Q(z, F).$$

Proof

Recall that $Cost(M)$ is the number of constraints that $Q(z, F)$ must satisfy. Therefore, the condition

$$|\mathcal{M}_{1,k-1}(\Omega)| > Cost(M)$$

ensures that there are more monomials at hand than linear equations and thus a non-zero $Q(z, F)$ exists.

Since $Q(z, F)$ has a zero of multiplicity m_{ij_i} at (α_i, c_i) for all i and with j_i so that $c_i = \beta_{j_i}$, we obtain by the *divisor theorem* that:

$$\sum_i (z - \alpha_i)^{m_{ij_i}} | Q(z, f(z)).$$

Thus, the polynomial $Q(z, f(z))$ is either the all-zero polynomial or has (at least) $\sum_i m_{ij_i} = Score_M(\underline{\mathbf{c}})$ roots.

Recall that the degree of $Q(z, f(z))$ is bounded from above by the weighted degree of $Q(z, F)$, equal to Ω. Thus, when

$$Score_M(\underline{\mathbf{c}}) > \Omega(M)$$

then the polynomial $Q(z, f(z))$ would have more roots than its degree and it must therefore be the all-zero polynomial. In other words, we have:

$$(F - f(z))|Q(z, F)$$

which concludes the proof. □

Let us stress that understanding the previous theorem and its proof is crucial. That $f(z)$ is an F-root of $Q(z, F)$ means that we can "extract it" by factorizing $Q(z, F)$. This reduces to a very simple but strong statement: if

$$Score_M(\underline{\mathbf{x}}) > \Omega(M)$$

then the decoding is successful. The algorithm naturally flows from the theorem and is presented in the upcoming section. For the moment, we can already see that increasing the multiplicities in M increases the potential score as well as the cost of the matrix. Thus, it is important to choose the values in a suitable manner. We will see that with higher values we can obtain finer decoding but since the cost of M is the number of constraints to satisfy, it is directly linked to the algorithm complexity. This provides a trade-off between correction ability and computational complexity.

11.2 Algorithm and example

The algorithm steps can be illustrated as follows:

$$\begin{tikzcd}
\text{channel information} \arrow[d, "\text{Choosing Multiplicities}"'] \\
M \arrow[d, "\text{Interpolation}"'] \\
Q(z,F) \arrow[d, "\text{Factorization}"'] \\
\prod (F - f(z))Q' \arrow[d, "\text{Selection}"'] \\
f(z)
\end{tikzcd}$$

Let us now look at the steps more closely and at the same time give an overview of what is awaiting us in the next few chapters.

- The first step of the algorithm consists of determining the multiplicity matrix M given the soft or hard-information of the channel to optimize the probability of successful decoding. This is the subject of the next chapter.
- Given M, the next step is to find a polynomial $Q(z, F)$ of least weighted degree satisfying the multiplicities defined by M. With $c = Cost(M)$, we saw that such a polynomial can be found by solving a system of c linear equations, having a little more than c unknowns. Solving this system of equation with classical means would result in a complexity of $\mathcal{O}(c^3)$. In chapter 13, we present a more effective way to "interpolate" such a polynomial with a complexity of $\mathcal{O}(c^3/k)$.
- Once $Q(z, F)$ is obtained, the factors of type $F - f(z)$ where $\deg(f(z)) \leq k - 1$ must be filtered out in order to output the list $\mathcal{L}$ of all of such functions. An efficient way of doing this in polynomial time is seen in chapter 14.
- Lastly, once the list of functions is obtained, they are reencoded to test their likelihood using the channel information and the most likely one is returned as the final output. This last operation is simple and how to do this is mentioned in the end of next chapter.

An important part is however missing for our understanding right now: what is the meaning of M? how are these multiplicities chosen? Although this is explained in the next chapter, let us say that an entry m_{ij} corresponds to the level of confidence we have that $f(\alpha_i) = \beta_j$. It turns out that one way to construct M is to choose multiplicities directly proportional to the likelihood that the ith symbol is β_j. Thus, high multiplicities at given points reflect a high probability that the encoding function passes through these points whereas low multiplicities mean that they have low probabilities to pass through these.

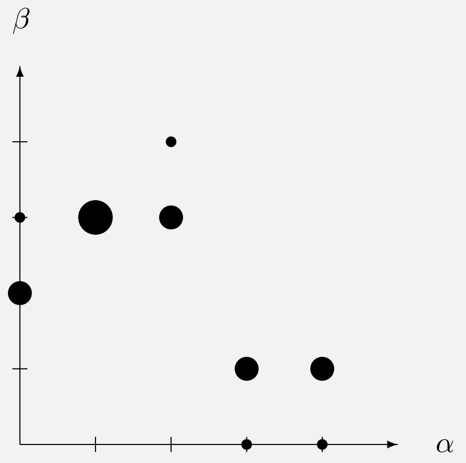
Example
Assume a RS code over $\mathbb{F}_5$ having parameters $n = 5$ and $k = 3$ with evaluator locations $\underline{\alpha} = (0, 1, 2, 3, 4)$. Assume that, based on the channel's information, the following multiplicity matrix is computed (we will discover how in the next chapter):

$$M = \begin{pmatrix} 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 2 & 1 \\ 1 & 2 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 \end{pmatrix}$$

The cost of M is:

$$Cost(M) = (3 + 1) + (6) + (3 + 1) + (1 + 3) + (1 + 3) = 22.$$

Since $k = 3$, the weighted degree of $Q(z, F)$ must be equal to $\Omega = 8$ so that $|\mathcal{M}_{1,k-1}(\Omega)| = 25 > 22$. The following figure shows a graphical representation of M where the dots have a radius proportional to the multiplicity at this point.



A polynomial $Q(z, F)$ satisfying these multiplicities is:

$$\begin{aligned} Q(z, F) \quad = \quad & 4 + 4z + 2z^2 + 3z^3 + z^5 + z^6 + 4z^7 + 3F + 2zF + 2z^2F + z^4F \\ & +2z^5F + z^6F + 4F^2 + 3zF^2 + 2z^2F^2 + 4z^3F^2 + 3F^3 \end{aligned}$$

By factorizing it, we obtain:

$$Q(z,F) = [F-(2+z)][F-(2+2z)][F-(3+z+z^2)]$$

Thus, three potential encoding functions are considered:

- $z+2$
- $2z+2$
- $3+z+z^2$

When computing the points they pass through and summing the multiplicities at these points, we conclude that a single function has a high enough score, namely:

$$f(z) = z+2.$$

The function passes through the points

$$(0,2),(1,3),(2,4),(3,0),(4,1)$$

And its score $\underline{\hat{\mathbf{x}}}$ is:

$$Score_M(\underline{\hat{\mathbf{x}}}) = 2+3+1+1+2 = 9 > \Omega = 8$$

Computing the score can be done quickly by looking at the graph above and looking at the multiplicities of the points directly. In the end, the algorithm outputs $f(z) = z+2$ which is the best candidate for decoding.

Chapter 12

Multiplicity assignment

In the previous chapter, we saw that the output list of the algorithm contains the sent codeword $\underline{\mathbf{x}} \in \mathcal{C}$ under the following condition:

$$Score_M(\underline{\mathbf{x}}) > \Omega(M),$$

where $\Omega(M)$ is the smallest integer such that $|\mathcal{M}_{1,k-1}(\Omega)| > Cost(M)$.

Knowing this, the remaining question is: how to choose the matrix M? This chapter shows how to find "good" matrices M in the sense that they maximize the probability of correct decoding, or better said, an approximation thereof.

12.1 The reliability matrix

Let us start where everything starts: the channel. In our case, the input consists of symbols taken from a finite set, namely the elements of the field $\mathbb{F}_q$. We will denote the input as the discrete variable $\mathcal{X}$ which takes values in $\mathbb{F}_q$.

Under the assumption that the channel is memoryless, it can be characterized by transition functions $p(\mathcal{Y} = y|\mathcal{X} = \beta_j)$ for every $\beta_j \in \mathbb{F}_q$. They define the probability of receiving y when having sent β_j.

Given these probabilities, it is easy to compute the reverse conditional probabilities using Bayes' rule:

$$p(\mathcal{X} = \beta_j|\mathcal{Y} = y) = \frac{p(y|\mathcal{X} = \beta_j)p(\mathcal{X} = \beta_j)}{\sum_l p(y|\mathcal{X} = \beta_l)p(\mathcal{X} = \beta_l)}$$

Moreover, under the assumption that $\mathcal{X}$ is uniformly distributed over $\mathbb{F}_q$, the above formula can be simplified into:

$$p(\mathcal{X} = \beta_j | \mathcal{Y} = y) = \frac{p(y|\mathcal{X} = \beta_j)}{\sum_l p(y|\mathcal{X} = \beta_l)}$$

If we perform this computation for every received symbol y_i, we obtain the reliability matrix.

Definition 33 *The reliability matrix* Π *is an* $n \times q$ *matrix with entries defined as:* $\pi_{i,j} = p(\mathcal{X}_i = \beta_j | \mathcal{Y}_i = y_i)$.

To summarize it, each entry of Π is the probability that the ith symbol from the sent codeword is β_j. We saw in the previous chapter that a codeword from an RS code can be represented as a set of n points $(\alpha_i, x_i = f(\alpha_i))$. Placed in this light, the entries correspond to the confidence we have that $f(\alpha_i) = \beta_j$ (based on the y_i value received).

The sent and received codewords can be represented, respectively, by a vector of random variables $\underline{\mathcal{X}} = (\mathcal{X}_1, ..., \mathcal{X}_n)$ and $\underline{\mathcal{Y}} = (\mathcal{Y}_1, ..., \mathcal{Y}_n)$. Note however that the a posteriori probability $p(\underline{\mathcal{X}} = \underline{\mathbf{x}} | \underline{\mathcal{Y}} = \underline{\mathbf{y}})$ are not equal to $\prod_i p(\mathcal{X} = x_i | \mathcal{Y} = y_i)$ since the $\mathcal{X}_i$ are not independent. The equality would apply if the vectors $\underline{\mathbf{x}}$ would a priori be taken from $\mathbb{F}_q^n$. However, this is not the case since they are taken from $\mathcal{C}_{(n,k)} \subset \mathbb{F}_q^n$. Moreover, it should be pointed out that finding $p(\underline{\mathcal{X}} = \underline{\mathbf{x}} | \underline{\mathcal{Y}} = \underline{\mathbf{y}})$ would be equivalent to maximum likelihood decoding which is NP-complete.

12.2 Setting the problem

The sent codeword $\underline{\mathbf{x}}$ is unknown to the decoder. The only information the decoder has at hand is some stochastic information about the likelihood of each sent symbol. Therefore, from the decoder's perspective, the sent codeword can be seen as a random variable $\underline{\mathcal{X}} = (\mathcal{X}_1, ..., \mathcal{X}_n)$. The probability that a codeword $\underline{\mathbf{w}} \in \mathbb{F}_q^n$ was sent can be stated as follows:

$$p(\underline{\mathcal{X}} = \underline{\mathbf{w}} | \underline{\mathbf{y}}) = \begin{cases} 0 & \text{if } \underline{\mathbf{w}} \notin \mathcal{C} \\ \gamma^{-1} \prod p(\mathcal{X}_i = w_i | y_i) & \text{if } \underline{\mathbf{w}} \in \mathcal{C} \end{cases}$$

where $\gamma = \sum_{\underline{\mathbf{c}} \in \mathcal{C}} \prod_i p(\mathcal{X}_i = c_i | y_i)$.

Given this distribution, the optimal multiplicity matrix M_{opt} is the one which maximizes the probability of decoding:

$$M_{opt} = \text{argmax}_{M_{opt}} p(Score_M(\underline{\mathcal{X}}) > \Omega(M))$$

Unfortunately, this optimization problem is inherently difficult because of two reasons:

- the matrix M affecting both sides of the inequality,
- the conditional probabilistic model for $p(\underline{\mathcal{X}} = \underline{\mathbf{x}})$.

Moreover, Koetter and Vardy showed that this optimization problem is not tractable and that finding the optimal M_{opt} reduces to an NP-hard problem [9]. Therefore, a simplified model is used, tackling both of the above difficulties while remaining a good approximation.

Let $\mathcal{M}_c$ be the (finite) set of matrices whose cost is c. To deal with the first issue, the solution is to find $M \in \mathcal{M}_c$ maximizing the expected score among matrices of equal cost c, that is

$$M = \text{argmax}_{M \in \mathcal{M}_c} E[Score_M(\underline{\mathcal{X}})].$$

The justification is twofold. First, it sounds intuitively reasonable and makes sense. Secondly, Koetter and Vardy computed bounds on the error probability based on this approximation showing it is asymptotically "good" [9].

Based on this reformulated problem, we are now concerned with how to compute the expected score $E[Score_M(\underline{\mathcal{X}})]$ of a matrix M. To do this, the following approximation is needed, concerning the probability that $\underline{\mathbf{x}}$ was sent:

$$\hat{p}(\underline{\mathcal{X}} = \underline{\mathbf{x}} | \underline{\mathbf{y}}) = \prod p(\mathcal{X}_i = x_i | y_i).$$

This would be the a posteriori distribution of $\underline{\mathcal{X}}$ given $\underline{\mathbf{y}}$ if the codewords were a priori uniformly taken in the entire space $\mathbb{F}_q^n$. Therefore, the decoder does not use all the available information and this results in a sub-optimal problem. However, it enables one to solve the (approximate) optimization model because the expectation of the score can now be computed:

$$\begin{aligned} \hat{E}[Score_M(\underline{\mathcal{X}})] &=^{def} \sum_{\underline{\mathbf{x}} \in F_q^n} M \bullet \underline{\mathbf{x}} \hat{p}(\underline{\mathcal{X}} = \underline{\mathbf{x}} | \underline{\mathbf{y}}) \\ &= M \bullet \sum_{\underline{\mathbf{x}} \in F_q^n} \underline{\mathbf{x}} \hat{p}(\underline{\mathcal{X}} = \underline{\mathbf{x}} | \underline{\mathbf{y}}) \\ &= M \bullet \Pi. \end{aligned}$$

Indeed, Π is the component-wise addition of $\langle \underline{\mathbf{x}} \rangle \, p(\underline{\mathbf{x}})$. To conclude this section, we thus look for M such that

$$M = \text{argmax}_{M \in \mathcal{M}_c} M \bullet \Pi$$

The two upcoming sections present two *multiplicity assignment algorithms*, shortly *MAA*s, solving exactly this approximated problem.

12.3 Greedy MAA

One way to find $M \in \mathcal{M}_c$ which maximizes $M \bullet \Pi$ is to construct it iteratively. Starting with M_0 as the all-zero matrix, the algorithm greedily increments one of the entries of M at each iteration. Incrementing an entry by one has two consequences:

- the cost is increased by $m_{ij} + 1$, this is the number of additional constraints to satisfy;
- the expected score is increased by π_{ij}.

Therefore, the best choice is to increment the entry having the best trade off between gain and cost, that is, the greatest ratio $\frac{\pi_{ij}}{m_{ij}+1}$.

Theorem
Input:

- the reliability matrix Π
- an integer s, indicating the number of interpolation points

Output:
the multiplicity matrix $M = \text{argmax}_M M \bullet \Pi$ among matrices of same cost.

Proof

Let us prove that the output of the algorithm is indeed the matrix $M = \text{argmax}_M M \bullet \Pi$ among the matrices of same cost. To facilitate the understanding, a geometric interpretation is used. Let us associate to each position in M an infinite sequence of rectangles $R_{ij,1}, R_{ij,2}, ...$ having their length and height defined as follows:

- $length(R_{ij,l}) = l$
- $height(R_{ij,l}) = \pi_{ij}/l$

Notice that the area of a rectangle $R_{ij,l}$ is π_{ij}. Now, let the set of rectangles associated to a multiplicity matrix M be

$$\mathcal{R} = \{R_{ij,l} | 1 \leq i \leq n, 1 \leq j \leq q, 1 \leq l \leq m_{ij}\}$$

This set contains s rectangles, the number of interpolation points counted with their multiplicities. Increasing the entry m_{ij} is by analogy adding the rectangle $R_{ij,m_{ij}+1}$, where its length is the number of constraints added and its area is the increase of the expected score. By putting all these rectangles side by side, the cost of M can conveniently be expressed as the total length of the rectangles:

$$Cost(M) = \frac{1}{2}\sum_{ij} m_{ij}(m_{ij}+1) = \sum_{ij}\sum_{l=1}^{m_{ij}} l = \sum_{R\in\mathcal{R}} length(R).$$

And the expected score of M is the total surface of all rectangles:

$$M \bullet \Pi = \sum_{ij} m_{ij}\pi_{ij} = \sum_{ij}\sum_{l=1}^{m_{ij}} \pi_{ij} = \sum_{R\in\mathcal{R}} area(R).$$

In this context, maximizing M for a given cost is equivalent to maximizing the total area given its total length. It is obvious to see that this is done by taking the set of the tallest rectangles and this is exactly what the algorithm does. At each iteration, it picks the tallest rectangle. □

Notice that no comment was made about the number of iterations $s = \sum m_{ij}$ which is also the number of interpolation points counted with their multiplicities. Typically, s is chosen dynamically and the algorithm is stopped just before $\Omega(M)$ (which directly depends on the cost) exceeds some fixed threshold.

12.4 Proportional MAA

In the greedy MAA, the tallest rectangle available is taken at each iteration so that all the remaining ones are smaller than some height threshold. Therefore, for any value $0 < h < \pi_{\max}$, the algorithm includes, at some iteration, all the rectangles taller than h whereas the remaining ones are smaller. For all i, j we have:

$$\frac{\pi_{ij}}{m_{ij}} \geq h > \frac{\pi_{ij}}{m_{ij}+1}$$

When looking closer at it, the non-strict and the strict inequalities can be swapped. Moreover, under the condition that h would be different than the height of any rectangles, two strict inequalities could be used. However, the formula as it is above is sufficient for us. Let us rewrite it:

$$\frac{\pi_{ij}}{h} - 1 < m_{ij} \leq \frac{\pi_{ij}}{h}$$

For convenience, let us take $\lambda = h^{-1}$. The expression of m_{ij} becomes:

$$m_{ij} = \lambda \pi_{ij} - \epsilon \quad \text{,with } 0 \leq \epsilon < 1$$

This, in turn, is just the definition of the floor of $\lambda \pi_{ij}$. Moreover, if $\lambda < \frac{1}{\pi_{\max}}$ is taken, then the above formula gives 0 for every m_{ij} and this corresponds to the initial all-zero matrix for M. Therefore, for any fixed value of λ, the greedy MAA produces, after an unknown number of iteration, a matrix M which is:

$$M = \lfloor \lambda M \rfloor$$

Thus, the multiplicity matrix proportional to Π is the optimal one among multiplicity matrices of same cost. This provides a very efficient and practical way to obtain the multiplicity matrix quickly and easily: M is simply chosen proportionally to Π. We call this the *proportional MAA*.

It is a shortcut of the greedy MAA obtaining instantly but exactly what would be produced after many iterations by the greedy MAA. Nevertheless, the latter one is still useful and the best method to choose M is to combine both as follows. First, choose a reasonable λ and multiply the reliability matrix multiplier by it to obtain a "rough" multiplicity matrix M. Then, to tweak it, apply the greedy MAA on it and stop just before an offset of $\Omega(M)$. That way, the score is optimized in the sense that any additional increment on M would cause $\Omega(M)$ to increase, which is unfavorable for the main decoding condition $Score_M(\underline{\mathbf{x}}) > \Omega(M)$.

12.5 Asymptotic hard-decoding performances

Notice that hard information is like having a reliability matrix with ones for the most likely symbols and zeros for the others. In this light, Sudan's algorithm is clearly a special case of this general algorithm with n interpolation points of multiplicity 1. The next historical step was to assign higher multiplicities to each such point and was developed by Guruswami-Sudan [6]. Every point (α_i, y_i) is assigned a multiplicity of $m = \lambda$ for some fixed λ. Let us compute the asymptotical error correcting ability as λ tends to infinity. To construct $Q(z, F)$, we must ensure that:

$$\frac{\Omega^2}{2(k-1)} > \mathcal{M}_{1,k-1}(\Omega) > \frac{1}{2}n\lambda(\lambda+1).$$

By isolating Ω we obtain:

$$\Omega > \sqrt{n(k-1)\lambda(\lambda+1)}$$

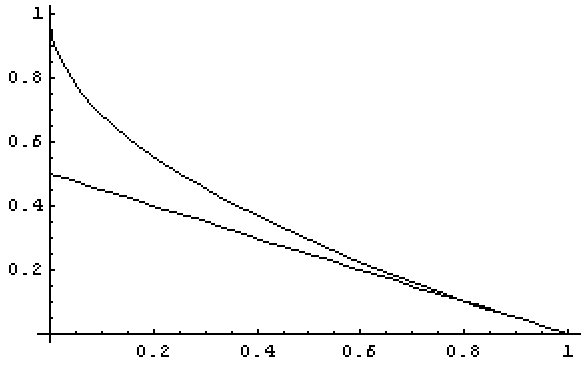
Since the score is $(n-e)\lambda$ the decoding is successful when:

$$(n-e)\lambda > \sqrt{n(k-1)\lambda(\lambda+1)}$$

When λ and n tend to infinity, the condition becomes:

$$\epsilon < 1 - \sqrt{r}$$

This bound is illustrated on the graph hereunder, together with the error correction ability of conventional decoders, namely $\epsilon = \frac{n-k}{2}$.



We see that the error correction ability of this kind of decoder is always (at any rate) superior to conventional decoders. However, it should be kept in mind that this holds when λ tends to infinity and therefore also the complexity and runtime of the algorithm. For fixed values of λ, the bound varies between the bound above and the bound presented in Sudan's algorithm.

12.6 Asymptotic soft-decoding performances

These are harder to characterize due to their nature. It makes no sense to talk in terms of error-correction ability since it does not depend on the number of erroneous symbols but on the probabilities themselves.

Before going on, let us come back the last example of the preceding chapter. In this example, the soft-decoding algorithm was successful, giving as output $f(z) = z + 2$. When looking at the multiplicity matrix, one can figure out that the hard-information corresponding to the matrix would have been $(2, 3, 3, 1, 1)$. When evaluating the decoded function, it results in the codeword $(2, 3, 4, 0, 1)$. This corresponds to two errors in terms of hard-information. Since the minimal distance of the code is $d = 3$, conventional decoders could only have corrected a single error and would therefore have failed decoding the example.

Lastly, to select the codeword, it is sufficient to:

- First, reencode every obtained function so that a list of codewords $\mathcal{L} = \{\underline{\mathbf{c}}_1, ..., \underline{\mathbf{c}}_l\}$ is obtained.
- Then, select the codeword $\underline{\mathbf{c}}_i \in \mathcal{L}$ maximizing $\Pi \bullet \langle \underline{\mathbf{c}}_i \rangle$, proportional to the probability of being the sent codeword.

Performances rely heavily on the code rate as well as the type of channel. Moreover, improvements offering a better optimization model and better MAAs than the initial work of Koetter and Vardy appeared several times since then. Performances of the MAA presented here can be found in the original paper of Koetter and Vardy [9]. Another paper of interest is the one from Jiang and Narayanan which studies the performances of this algorithm using this MAA analytically for the binary symmetric channel and binary erasure channel [14]. The latest and best result however has been reported by El-Khamy and McEliece in [15]. Based on an improved optimization model, they obtain an MAA resulting in much better decoding performances than with previously known MAAs.

Chapter 13

Kötter's interpolation

We saw that finding a polynomial having zeros of given multiplicities at given points reduces to satisfying a set of constraints $\mathcal{D}$. In this chapter, we review how to efficiently find the polynomial $Q(x, y)$ of minimal weighted degree and satisfying this set of constraints. As usual, some notions will be needed as prerequisites for the understanding of the algorithm and its proof, so here we go.

13.1 Monomial ordering

Let $\mathbb{M}[x, y]$ denote the set of monomials in x, y. The subject of this part is to define a total order for $\mathbb{M}[x, y]$. That is, for every pair $\phi_A = x^{i_A}y^{j_A}$ and $\phi_B = x^{i_B}y^{j_B}$, it must be possible to say that ϕ_A is greater than or smaller than ϕ_B according to the defined order. Not surprisingly, we shall order the monomials according to their weighted degree. That is, the relation $\prec$ is defined so that $\phi_A \prec \phi_B$ whenever $\text{wdeg}_{1,k-1}(\phi_A) < \text{wdeg}_{1,k-1}(\phi_B)$.

However, this is not yet a complete order. Indeed, there can be several monomials of the same weighted degree. For example, when $k - 1 = 3$, the three monomials x^6, x^3y, y^2 have all a weighted degree of 6. Therefore, a way to distinguish them is needed. Arbitrarily, we say that the one with higher y-degree is greater.

Definition 34 *The complete order $\prec$ is defined as $\phi_A \prec \psi_B$ if and only if*

$$\begin{cases} wdeg_{1,k-1}(\phi_A) < wdeg_{1,k-1}(\phi_B) \\ \text{or} \\ (wdeg_{1,k-1}(\phi_A) = wdeg_{1,k-1}(\phi_B) \text{ and } \deg_y(\phi_A) < \deg_y(\phi_B) \end{cases}$$

Example
Consider the case where $k-1=5$ and two monomials are x^6y^2 and x^4y^3. In this context, $x^6y^2 \prec x^4y^3$ since $\mathrm{wdeg}_{1,k-1}(x^6y^2) = 16 < \mathrm{wdeg}_{1,k-1}(x^4y^3) = 19$

Example
As a second example, when $k-1=3$ and the two monomials are x^8 and x^2y^2. Then $x^8 \prec x^2y^2$ since $\mathrm{wdeg}_{1,k-1}(x^8) = 8 = \mathrm{wdeg}_{1,k-1}(x^2y^2)$ but $\deg_y(x^8) = 0 < \deg_y(x^2y^2) = 2$

This complete order also provides us a way to enumerate the monomials unambiguously. Monomials $\phi_i \in \mathbb{M}[x,y]$ can now be linearly ordered and form a sequence $1 = \phi_0 \prec \phi_1 \prec \phi_2 \prec \phi_3 \prec ...$

Example
For $k-1=3$, the resulting ordering is: $1 \prec x \prec x^2 \prec x^3 \prec y \prec x^4 \prec xy \prec x^5 \prec x^2y \prec x^6 \prec x^3y \prec y^2 \prec ...$

Any polynomial $Q(x,y)$ can be expressed as a sum of these ordered monomials, i.e. $Q(x,y) = \sum_i q_i\phi_i$. Expressing it this way leads to two new concepts, the *leading monomial* and the *rank*.

Definition 35 *The leading monomial of a polynomial $Q(x,y)$ is the monomial of highest order. More formally: $LM(Q) = \phi_I$ where $Q(x,y) = \sum_{i=0}^{I} q_i\phi_i$ with $q_i \neq 0$.*

Definition 36 *The rank of a polynomial is the order of its leading monomial.*

Example
Let $k-1=3$ and $Q(x,y) = 6x^4 + 5xy$. We obtain $\mathrm{LM}(Q) = xy$ and $\mathrm{Rank}(Q) = \mathrm{Rank}(\mathrm{LM}(Q)) = 5$

As a side note, the total order can also be used to represent $Q(x,y)$ as a tuple. This is simply done by defining that the ith entry in the tuple is the coefficient of the ith monomial.

13.2 Kernels of constraints

As was seen previously, every constraint in our problem is of the form:

$$D_{u,v}Q(x,y)|_{(\alpha,\beta)} = 0 \ : \ D_iQ.$$

The expression on the left is the complete expression and on the right an abbreviated form of it. Because the left notation is pretty heavy, the abbreviated one D_iQ will be adopted. This is done purely for convenience, it is just a syntactic change where D_iQ denotes the ith constraint. Notice that we also dropped the variables of Q, again just for convenience.

The operator D_i is what is called a $linear functional$. This means simply that it maps vectors to values in a linear way.

Definition 37 *A linear functional on a vector space S over a field $\mathbb{F}$ is a function $f : V \rightarrow \mathbb{F}$ which satisfies the following two properties:*

- $\forall u, v \in S : \quad f(u+v) = f(u) + f(v)$
- $\forall \alpha \in \mathbb{F}, u \in S : \ f(\alpha u) = \alpha f(u)$

It should be clear to the reader that D_i is a linear functional. Indeed Q can be seen as a vector and the derivative is then evaluated at some point resulting in a value. The two other properties are also satisfied for derivatives, thus for D_i.

Using a strandard terminology, the set of all polynomials Q so that $D_iQ = 0$ is called the kernel of D_i.

Definition 38 $\ker_{D_i} = \{Q | D_iQ = 0\}$.

If $Q_A, Q_B \in \ker_{D_i}$ then $\alpha Q_A + \beta Q_B \in \ker_{D_i}$ also since D_i is a linear functional. Indeed, we have:

$$D_i(\alpha Q_A + \beta Q_B) = \alpha D_i Q_A + \beta D_i Q_B = \alpha 0 + \beta 0 = 0$$

In other words, the kernel is a vector subspace of the polynomial space.

13.3 The algorithm

Now that we know about ranks and kernels, "finding Q" can be reformulated as follows. Find a Q of least rank so that $Q \in \ker_{D_1} \cap ... \cap \ker_{D_{Cost}}$

The idea behind Kötter's interpolation algorithm is to satisfy the constraints one after another. Therefore, let us define the cumulative kernels.

Definition 39 *The cumulative kernel $\mathcal{K}_i$ is defined as: $\mathcal{K}_i = \mathcal{K}_0 \cap \ker(D_1) \cap \ldots \cap \ker(D_i)$ where $\mathcal{K}_0$ is the set of all polynomials of weighted degree at most Ω.*

The cumulative kernel is simply the set of polynomials that satisfy the set of the first i constraints.

INITIALIZATION

Let L be the maximum y-degree of $Q(x, y)$, i.e. $L = \lfloor \Omega/(k-1) \rfloor$. The algorithm initializes $L + 1$ polynomials as follows:

$$\begin{aligned} Q_0(x, y) &= 1 \\ Q_1(x, y) &= y \\ Q_2(x, y) &= y^2 \\ &\ldots \\ Q_L(x, y) &= y^L \end{aligned}$$

In this initialization, each of the $L + 1$ polynomials Q_j belongs to a specific set. Namely the set $\mathcal{Q}_j$ of polynomials whose leading monomial have a y-degree of j.

Definition 40 *Let $\mathcal{Q}_j$ be the set of polynomials whose leading monomial has a y-degree of j. More formally: $\mathcal{Q}_j = Q \in \mathbb{F}_q[x, y] | LM(Q) = x^? y^j$.*

Example

Consider $k - 1 = 4$. In this case the polynomial $Q(x, y) = 3x^7y^2 + 5x^2y^3$ has as leading monmial x^7y^2 and thus $Q \in \mathcal{Q}_2$.

An important fact is that $Q_j \in \mathcal{Q}_j$ will remain true at each step of the algorithm.

ITERATION

At each iteration, one additional constraint is satisfied. At the same, the y-degree of the leading monomial of every Q_j keeps constant. That is, it is ensured that $LM(Q_j) = x^? y^i$ after each iteration.

At the ith iteration, the constraint D_i has to be satisfied. It has the form: $D_u, vQ(x, y)|_{(\alpha,\beta)} = 0$.

The order in which the constraints are satisfied is crucial but we are not yet ready for it. A condition on the ordering of constraint satisfaction will be explained in the proof. Let us ignore it for the moment. To ensure that every polynomial $Q_j(x, y)$ satisfies this constraint, the following is done:

1. For each j, compute compute the discrepancy λ_j, i.e. the result of the derivative on Q_j:
$$\lambda_j = D_i Q_j$$
If the discrepancy is zero. We are happy. The constraint is already satisfied and there is no need to change Q_j. Else...

2. Among all Q_j having a non zero discrepancy, select the one of least order. The index of this polynomial will be noted j^*.

3. For each $j \neq j^*$ where $\lambda_j \neq 0$, the updated polynomial Q'_j is: $Q'_j = \lambda_{j^*} Q_j - \lambda_j Q_{j^*}$.

4. The updated polynomial Q'_{j^*} is: $Q'_{j^*} = (x - \alpha) Q_{j^*}$.

This way, after each iteration, one more constraint is satisfied and Q'_j is the polynomial of least rank in its set.

Theorem
ITERATIVE INTERPOLATION THEOREM

At each iteration of the algorithm, assume that every $D_{u-1,v}$ constraint is

satisfied before $D_{u,v}$.
If $\forall j : Q_j \in \ker_{i-1} \cap \mathcal{Q}_j$

then Q'_j is the polynomial of least rank in $\mathcal{Q}_j \cap \mathcal{K}_i$.

Proof

In the first case where the discrepancy is zero, the constraint is already satisfied. Moreover, if the polynomial was of lowest rank in its set, it will remain so since

$$\mathcal{K}_i = \mathcal{K}_{i-1} \cap \ker_{D_i} \subseteq \mathcal{K}_{i-1}.$$

In the other case of a non-zero discrepancy, we update the polynomial according to: $Q'_j = \lambda_{j^*} Q_j - \lambda_j Q_{j^*}$.

Since D_i is a linear functional and since by hypothesis both Q_j and Q_{j^*} belong to $\mathcal{K}_{i-1}$, it is straightforward that $(\lambda_{j^*}Q_j - \lambda_j Q_{j^*}) \in \mathcal{K}_{i-1}$. Moreover:

$$D_i(\lambda_{j^*}Q_j - \lambda_j Q_{j^*}) = \lambda_{j^*}D_iQ_j - \lambda_j D_i Q_{j^*} = \lambda_{j^*}\lambda_j - \lambda_j\lambda_{j^*} = 0.$$

Thus $Q'_j \in \mathcal{K}_{i-1} \cap \ker_{D_i} = \mathcal{K}_i$. Additionally, the rank of Q'_j has not changed since a polynomial of lower rank has been subtracted. Thus if Q_j was of lowest rank in its set, it will remain so since $\mathcal{K}_i \subseteq \mathcal{K}_{i-1}$.

The last point is a little more complicated. The lowest ranked polynomial Q_{j^*} is updated by multiplying it by $(x-\alpha)$. Again, it is obvious that $Q'_{j^*} \in \mathcal{K}_{i-1}$. But is it also in $\ker_{D_i}$? Here comes in play the condition on constraints ordering that we omitted previously.

$$D_iQ = \frac{d}{dx}D_{u-1,v}(x-\lambda_{j^*})Q_{j^*} = D_{u-1,v}Q_{j^*} + (x-\alpha)D_{u,v}Q_{j^*}$$

When evaluating the latter expression at (α,β), the equality becomes:

$$D_{u,v}(x-\lambda_{j^*})Q_{j^*}|_{(\alpha,\beta)} = D_{u-1,v}Q_{j^*}|_{ab}$$

Thus, if the constraint constraint $D_{u-1,v}Q(x,y)|_{(\alpha,\beta)}$ is already satisfied, before $D_iQ = D_{u,v}Q(x,y)|_{(\alpha,\beta)} = 0$ and is therefore in $\ker_{D_i}$, else not necessarily. □

END

To summarize it, after the ith iteration, every polynomial Q_j is the lowest rank polynomial in $\mathcal{K}_i \cap \mathcal{Q}_j$. Thus, at the end of the algorithm, after a number of iterations equal to the cost of M, the Q_j obtained satisfy all the constraints. Notice that the union of all $\mathcal{Q}_j$ sets is the set of all polynomials with y-degree less than or equal to L. The lowest ranked among the Q_j polynomials is therefore also the lowest ranked possible from the whole set of polynomials with y-degree less than or equal to L.

Example

Let us work over $\mathbb{F}_5$ with $\underline{\alpha} = (1,2)$ and $k=4$. Let the multiplicity matrix be:

$$M = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 2 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

The aim is to find the polynomial $Q(z, F)$ having zeros of multiplicity m_{ij} at (α_i, β_j). More precisely:

- A zero of multiplicity 1 at (1,4)
- A zero of multiplicity 2 at (2,0)

This is equivalent to satisfying the following constraints:

- $D_{0,0}Q(z, F)|_{(1,4)} = 0$
- $D_{0,0}Q(z, F)|_{(2,0)} = D_{0,1}Q(z, F)|_{(2,0)} = D_{1,0}Q(z, F)|_{(2,0)}$

The weighted degree of $Q(z, F)$ must be at least $\Omega = 4$.
INITIALISATION
$Q_0 = 1$
$Q_1 = y$

FIRST ITERATION
Let the constraint to satisfy for the first iteration be $D_{0,0}Q(z, F)|_{(1,4)}$. We have:
$\lambda_0 = D_{0,0}Q_0|_{(1,4)} = 1$
$\lambda_1 = D_{0,0}Q_1|_{(1,4)} = 4$
$j^* = 0$
Thus:
$Q'_0 = (x - 1)Q_0 = x - 1$
$Q'_1 = 1Q_1 - 4Q_0 = y - 4$

SECOND ITERATION: $D_{0,0}Q(z, F)|_{(2,0)}$
$\lambda_0 = D_{0,0}Q_0|_{(2,0)} = -2$
$\lambda_1 = D_{0,0}Q_1|_{(2,0)} = -4$
$j^* = 0$
$Q'_0 = (x - 2)(x - 1) = x^2 + 2x + 2$
$Q'_1 = -2(y - 4) + 4(x - 1) = 3y + 4x + 4$

THIRD ITERATION: $D_{0,1}Q(z, F)|_{(2,0)}$
Remark: the ordering condition imposes us to process $D_{0,1}$ before $D_{1,0}$.
$\lambda_0 = D_{0,1}Q_0|_{(2,0)} = 0$
$\lambda_1 = D_{0,1}Q_1|_{(2,0)} = 3$
$Q'_0 = Q_0$

$Q'_1 = (x-2)(3y+4x+4) = 3xy+4y+4x^2+x+2$

FOURTH ITERATION: $D_{1,0}Q(z,F)|_{(2,0)}$
$\lambda_0 = D_{1,0}Q_0|_{(2,0)} = 2x+2|_{(2,0)} = 1$
$\lambda_1 = D_{1,0}Q_1|_{(2,0)} = 3y+3x+1|_{(2,0)} = 2$
$j^* = 0$
$Q'_0 = (x-2)(x^2+2x+2) = x^3+3x+1$
$Q'_1 = 1(3xy+4y+4x^2+x+2) - 2(x^2+2x+2) = 3xy+4y+2x^2+2x+3$

And thus we obtain two polynomials satisfying the zeros of given multiplicities. By factorizing both polynomials, the zeros become visible:
$Q_0 = x^3+3x+1 = (x-2)^2(x-1)$
$Q_1 = 3xy+4y+2x^2+2x+3 = 3(y-x+2)(x-2)$

The interpolation ends by selecting the polynomial of least order, Q_0 in this case.

13.4 Performances

The question is now: how does it compare to classical matrix solving? We do $|\mathcal{D}|$ iterations. At each iteration, $L+1$ polynomials are updated where L is in $\mathcal{O}(|\mathcal{D}|/k)$. Evaluating a constraint costs $\mathcal{O}(|\mathcal{D}|)$ as well. Thus the algorithm runs in $\mathcal{O}(|\mathcal{D}|^3/k)$.

13.5 Pseudo-code

```
input: M, k
output: Q

L = floor(ComputeOmega(M) / (k-1))
for l = 0 to L
Q[l] = y^l

for i = 1 to n
for j = 1 to q
if (M[i,j] != 0)
  m = M[i,j]
  a = alphas[i]
  b = Fq[j]
  for u = 0 to m - 1
  for v = 0 to m - u - 1
    lowestL = -1
    for l = 0 to L
      lambda[l] = ComputeD(u,v,Q,a,b)
      if (lambda[l] != 0)
        if (lowestL == -1 or rank(Q[l]) < rank(Q[lowestL]))
          lowestL = l

    if (lowestL != -1)
      for l = 0 to L
        if (lambda[l] != 0 and l != lowestL)
          Q[l] = lambda[lowestL] * Q[l] - lambda[l] * Q[lowestL]

      Q[lowestL] = (x - a) * Q[lowestL]

lowestL = -1
for l = 0 to L
if (lowestL == -1 or rank(Q[l]) < rank(Q[lowestL]))
  lowestL = l

return Q[lowestL]
```

Chapter 14

Factorization

In this chapter, we investigate how to find the factors of the type $(y - f(x))$ in $Q(x, y)$. More specifically, only the factors with $f(x)$ of degree less than k. The brute force method would be to test whether $f(x)$ is a y-root of $Q(x, y)$ or not for each polynomial $f(x)$. But of course, this would be outrageously expensive in terms of computation time. For the same price, one could directly test the best candidate among all possible codewords. This is of course unreasonable and we therefore need an efficient algorithm running in polynomial time.

The one we shall present here is known as the *Roth-Ruckenstein factorization algorithm*, named after its inventors [13].

14.1 Roth-Ruckenstein factorization

Like in the interpolation algorithm, the idea here is not to get all at once but the opposite. It is to find out the coefficients of the polynomials one by one. Let us express a polynomial $f(x)$ as:

$$f(x) = f_0 + f_1 x + f_2 x^2 + ...$$

Assume $f(x)$ is a y-root of $Q(x, y)$ so that $Q(x, f(x)) = 0$. By instantiating x to 0, we have $f(0) = f_0$ and also that $Q(0, f_0) = 0$. In other words, it means that f_0 is a y-root of $Q(0, y)$.

Definition 41 *Let* $\langle\langle Q(x, y)\rangle\rangle$ *be the* normalized polynomial *defined as follows:*

$$\langle\langle Q(x, y)\rangle\rangle = Q(x, y)/x^m$$

where m *is the greatest integer such that* $x^m | Q(x, y)$*.*

The first coefficient f_0 is of course also an y-root of $\langle\langle Q(0,y)\rangle\rangle$:

$$(y - f_0)|Q(0,y) \Leftrightarrow (y - f_0)|x^m \langle\langle Q(0,y)\rangle\rangle \Leftrightarrow (y - f_0)| \langle\langle Q(0,y)\rangle\rangle$$

The reason to normalize is to avoid $Q(0,y)$ to be the all-zero polynomial. Using $\langle\langle Q(0,y)\rangle\rangle \neq 0$, testing all the y-roots gives the first coefficient f_0 for all the polynomials in the list. Now that we have f_0 at hand, the question is how to find f_1. Since we have:

$$(y - (f_0 + f_1 x + f_2 x^2 ...))|Q(x,y)$$

The key idea is to perform a smart change of variable. Namely, replacing y by $xy + f_0$ so that we have:

$$\begin{array}{rcl} (xy + f_0 - (f_0 + f_1 x + f_2 x^2 ...)) & | & Q(x, xy + f_0) \\ & \Updownarrow & \\ x(y - (f_1 + f_2 x + f_3 x^2 ...)) & | & Q(x, xy + f_0) \\ & \Updownarrow & \\ (y - (f_1 + f_2 x + f_3 x^2 ...)) & | & \langle\langle Q(x, xy + f_0)\rangle\rangle = Q'(x,y) \end{array}$$

Informally, we "killed" the term f_0 and it is now exactly the same situation as before but headed by f_1. Thus, exactly the same procedure can be repeated, enabling us to "pick" that second coefficient f_1 the same way by taking the roots of $Q'(0,y)$. By repeating the process iteratively, all coefficients can be picked one after another until all coefficients of $f(x)$ are obtained. Let us now prove the key iteration more formally.

Theorem
Let us define

$$\tilde{f}_i(x) = f_i + f_{i+1}x + f_{i+2}x^2 + ...$$

and

$$Q_{i+1}(x,y) = \langle\langle Q_i(x, xy + f_i)\rangle\rangle .$$

If

$$(y - \tilde{f}_i(x))|Q_i(x,y)$$

so that f_i is an y-root of $Q_i(0,y)$. Then, by performing the change of variable using f_i we obtain:

$$(y - \tilde{f}_{i+1}(x))|Q_{i+1}(x,y)$$

so that f_{i+1} is a y-root of $Q_{i+1}(0,y)$.

Proof

By hypothesis, we have:

$$\begin{array}{rcl} (y - \tilde{f}_i(x)) & \mid & Q_i(x,y) \\ & \Updownarrow & \\ (y - (f_i + f_{i+1}x + f_{i+2}x^2 + ...)) & \mid & Q_i(x,y) \end{array}$$

By performing the change of variable $y \to xy + f_i$ we obtain:

$$\begin{array}{rcl} (xy + f_i - (f_i + f_{i+1}x + f_{i+2}x^2 + ...)) & \mid & Q_i(x, xy + f_i) \\ & \Updownarrow & \\ x(y - (f_{i+1} + f_{i+2}x + f_{i+3}x^2 + ...)) & \mid & Q_i(x, xy + f_i) \\ & \Updownarrow & \\ (y - (f_{i+1} + f_{i+2}x + f_{i+3}x^2 + ...)) & \mid & \langle\langle Q_i(x, xy + f_i)\rangle\rangle \\ & \Updownarrow & \\ (y - \tilde{f}_{i+1}(x)) & \mid & Q_{i+1}(x,y) \end{array}$$

as desired. □

Notice that this process splits up like a tree, for any coefficient f_i found, there may be multiple roots to $Q_{i+1}(0,y)$. This gives rise to several possible polynomials sharing the same first i coefficients but with different remaining ones. The process of picking each coefficient is repeated until the maximum degree of $f(x)$ is reached.

Example

Let us work in $\mathbb{F}_5$ and look for the factors $(y - f(x))$ with $\deg(f(x)) \leq 2$ in:

$$Q(x,y) = 2x + x^2 + x^3 + 3x^4 + y + 2xy + 4x^3y + 2y^2 + xy^2$$

Since this polynomial is already normalized, no need to divide it further by a power of x. Thus:

$$Q_0(x,y) = 2x + x^2 + x^3 + 3x^4 + y + 2xy + 4x^3y + 2y^2 + xy^2$$

The polynomial $Q_0(0,y) = y + 2y^2$ has two roots: 0 and 2. Let:

- Q_1 be the polynomial corresponding to the change of variable $y \to xy$

- Q'_1 be the polynomial corresponding to the change of variable $y \to xy + 2$

Let us begin with Q_1 and go back to Q'_1 later.

$$Q_1(x,y) = 2 + x + x^2 + 3x^3 + y + 2xy + 4x^3y + 2xy^2 + x^2y^2$$

And $Q_1(0,y)$ has a single root: 3.

$$Q_2(x,y) = 1y^1 + 4x^1y^1 + 1x^2y^1 + 4x^3y^1 + 2x^2y^2 + 1x^3y^2$$

And $Q_2(0,y)$ has a single root: 0. This results in the first polynomial

$$f(x) = 3x$$

Let us now go back to the other alternative Q'_1.

$$Q'_1(x,y) = 1x^1 + 4x^2 + 3x^3 + 4y^1 + 1x^1y^1 + 4x^3y^1 + 2x^1y^2 + 1x^2y^2$$

And $Q'_1(0,y)$ has a single root: 0.

$$Q'_2(x,y) = 1 + 4x^1 + 3x^2 + 4y^1 + 1x^1y^1 + 4x^3y^1 + 2x^2y^2 + 1x^3y^2$$

And $Q'_2(0,y)$ has a single root: 1. This results in the second polynomial

$$f(x) = x^2 + 2$$

In the end, the factorization outputs the two factors $3x$ and $x^2 + 2$.

14.2 Pseudo-code

```
ListOfPolynoms = {}
void FactorizeRR(int i, BivarPoly Q_i, Poly f)
  if(Q_i(x,0) == 0)
    ListOfPolynoms += f
  for each gamma in F_q
    if(Q_i(0,gamma) == 0)
      clone := f
      clone[i] = gamma
      if(i == k-1)
        ListOfPolynomials += clone
```

```
else
  Q_i_next += Normalize(Q_i(x, xy + gamma))
  FactorizeRR(i+1 , Q_i_next, clone)
```

Chapter 15

Program notes

In this chapter, we present the source code provided along with this text. It tells what the API provides, how to use it and a few implementation notes. This includes examples of code excerpts to demonstrate how to use the API. The program is primarily written for educational purposes, to illustrate how the algorithm works, to experiment with the algebraic soft-decoding and to see it in action. Besides of this, it provides also a limited API that can be used for performing computation with polynomials in finite fields, implementing encoders, decoders and other communication system components. Lastly, it can also be used to provide benchmarks. Despite the program favors clarity in spite of performances, these ones should be acceptable and have the same running time asymptotic properties as in the theory. The language C# was chosen because of several reasons: it is well known, it is relatively efficient, it is multi-platform (in theory at least...) and lastly it offers operator overloading which increases a lot readability as well as comfort when writing and using the source code.

As a side note, a free IDE called visual C# express is provided by Microsoft. This IDE can be downloaded freely on the net, is extremely easy to set up and getting the source code running is just a matter of minutes.

The program can be divided in four main parts:

- Algebra tools: classes for finite fields and polynomials over such fields
- Communication system: interfaces for encoders, channels and decoders
- Channel models: the different kind of channel models implemented
- Reed-Solomon: encoders and decoders for Reed-Solomon codes

15.1 Algebra

15.1.1 Finite fields

Two kinds of finite fields are currently implemented: prime fields and fields whose size are a power of two. Their source code is in `PrimeField.cs` and `BinaryExtensionField.cs` respectively. A finite field element is represented externally by the `struct FFE`, the choice of being a `struct` and not an `object` is for performance reasons. Internally, the finite field element is represented by an integer for performance reasons as well. For extension fields $\mathbb{F}_{p^m}$, the following bijection to integers $\mathbb{F}_{p^m} \leftrightarrow \mathbb{N}$ is used:

$$a_0 + a_1 x + ... + a_m x^m \Leftrightarrow \sum_{i=0}^{m} a_i p^i$$

Both classes are written in order to take advantage of the structure of the field for these sizes. For example, for fields whose size is a power of 2, bitwise addition is performed directly whereas modular arithmetic is used in prime fields. All operations are in $\mathcal{O}(1)$. In fields whose size is a power of 2, primitive polynomials are hard-coded and enable to make fields of size up to $2^1 6$.

And now a few example. To construct a field:

```
FiniteField Fq = new PrimeField(101); // q = p = 101
FiniteField Fq = new BinaryExtensionField(8); // q = 2^8 = 1024
```

To get elements of a field:

```
FFE a = Fq[3];
FFE b = Fq[47];
```

To perform computations in fields:

```
FFE result = a * Fq[21] + b / Fq[33] + (Fq[2]^3402);
Console.WriteLine(result);
```

The reason the exponent has been put in parenthesis is that this operator in C# initially stands for bitwise XOR and has no precedence over other operators.

Remark: since `FFE` is a struct, it is created by default upon declaration with all fields uninitialized. Doing so results in broken field elements. `myFiniteField[i]` should always be used to get elements.

A prime extenstion field class can be implemented and seamlessly integrated in the API provided it implements the `FiniteField` interface.

15.2 Polynomials

To be more precise, bivariate polynomials. Polynomials of different degrees in x,y can be added, subtracted, multiplied and powers of them can be taken. Except for polynomial powers, all other operations are efficiently implemented. The next few code lines illustrates how polynomials can be used.

```
// Let Fq = FFE.DEFAULT_FIELD
Polynomial p = new Polynomial(new int[,]{{0,3,0},{0,0,2}}); // result: p(x,y) = Fq[3]
Polynomial q = new Polynomial(4,3); // a polynomial whose max x-degree is 4 and max y
q[3,3] = Fq[5]; // result: q(x,y) = Fq[5]x^3y^3

Console.WriteLine((p*q - Fq[21]*p)^2);

Polynomial poly = Fq[1]; // implicit conversion
Polynomial x = Polynomial.GetX();

for(int i=0; i<Fq.q; i++)
poly = poly * (x - Fq[i]);

Console.WriteLine(poly); // prints x^q - x
```

Internally, polynomials store an array of `FFE` being the coefficients of the powers of x and y.

15.3 Communication System

15.3.1 Interfaces

The communication system is represented by three key interfaces which are utterly simple.

- `Encoder`: any encoder must implement this interface which consists of the single method `FFE[] Encode(FFE[] message)`.
- `Channel`: any channel must implement this interface which consists of two methods:
  - `void Send(FFE[] word)`
  - `double[,] Receive()`: returns the reliability matrix for the last sent word

- `FFE[] Decode(double[,] RM)`: given the reliability matrix as input, perform the decoding algorithm. For hard-decoding, the auxiliary method `FFE[] GetQuantizedReceivedWord(double[,] RM)` can be used.

By means of these interfaces, various components can be put together to form the system. Moreover, it is easy to extend the current program by adding different kind of channels, other decoders and so on. That way, running simulation in a standardized way becomes easy:

```
FiniteField Fq = new MyFiniteField(...);

Encoder encoder = new MyEncoder(...);
Channel channel = new MyChannel(...);
Decoder decoder = new MyDecoder(...);

for (int i = 0; i < howManyTimes; i++)
{
FFE[] m = RandomMessage();
FFE[] x = encoder.Encode(m);
channel.Send(x);
double[,] RM = channel.Receive();
FFE[] hatx = decoder.Decode(RM);
}
```

15.3.2 Channels

Three channels are implemented:

- a noiseless channel outputting the same as is sent.
- a Qary symmetric channel with custom transition probabilities.
- a ”binary extension channel” explained next.

The last channel works only for the transmission of symbols from a finite field whose size is 2^m for some m. Each symbol is mapped onto bits, transmitted an underlying binary symmetric channel of crossover probability p_{err}, then the received sequence of bits is mapped back to the symbol it stands for. The probability that β was sent if γ is received depends on the number of locations in which bits differ, noted b_e:

$$p(\mathcal{X} = \beta | \mathcal{Y} = \gamma) = p_{err}^{b_e}(1 - p_{err})^{m-b_e}$$

Example
Let $m = 4$ and $p_err = 0.1$ and assume that 1100 is received. Then:

- 1100 has a probability of 0.9^4 of being the sent symbol.
- $1000, 0100, 111, 1101$ have each a probability of $0.1 * 0.9^3$ of being the sent symbol.
- ...
- 0011 has a probability of 0.1^4 of being the sent symbol.

15.3.3 Encoders and decoders

A Reed-Solomon encoder is implemented. It works for any finite field and any valid parameters. The locations where the encoding polynomial is evaluated can be specified or chosen by default. Two decoders were implemented:

- A bounded-distance hard-decoder performing the Gao algorithm presented in chapter 7 and decoding up to $\frac{n-k}{2}$ errors.
- The algebraic soft-decoder for Reed-Solomon codes described in this thesis. It uses the greedy MAA with the number of total multiplicities is given, the Kötter interpolation and the Roth-Ruckenstein factorization.

In the first line of the source code of `RSSoftDecoder.cs` are preprocessor directives included to indicate if the algorithm should execute in normal, verbose or mute mode.

- In the verbose mode, every step is printed on the screen with every temporary polynomial obtained during the interpolation and factorization steps.
- In the normal mode, a dot is printed each time a constraint is satisfied during the interpolation. Once interpolation has finished, the resulting polynomial is printed. For factorization, a dot is printed for each root found and upon termination the list of potential encoding polynomials found are printed.
- In the mute mode, nothing at all is printed.

Code to run tests directly can be found in `Benchmark.cs`.

Chapter 16

Conclusion

In practice, the advantage of this algebraic soft decoding algorithm depends on:

- the rate of the code
- the kind of channel
- the available computational power

Since the rate of a code is directly chosen depending on the magnitude of the noise in the channel (for low noise, high code rates are used and vice-versa), these two arguments in fact boil down to the same one. Both for low rate codes and for very noisy channels, this algorithm offers tremendous benefits. Unfortunately, neither of these situations are frequent in practice. For example, optical fiber has crossover probabilities in the range of 10^{-12}, CD-Roms use RS codes with parameters $k = 239$ and $n = 255$, and so on. For these higher rate codes and low noise channels, the benefits are lower and more expensive (computationally) but nevertheless present. If the system can afford these computational resources, this alternative should be taken into account.

Another advantage for this algorithm is that it can easily be fined tuned between performances in decoding ability and rapidity of execution. Indeed, both are directly dependent on the number of interpolation points.

Despite it does not look very attractive for the moment at high rates, some very promising improvements for this algorithm emerged during the last few years. Here are some of them:

- There is the fact that it can be extended for soft-decoding *algebraic geometry codes* which are superior to *RS* codes.
- In [10][11], four authors show the equivalence between the initial interpolation problem and a transformed interpolation problem of lesser complexity by means of a genuine change of variable. The transformed interpolation problem has the property that a significant amount of constraints can be pre-solved. This reduces the number of constraints in the shifted interpolation problem by at least $\frac{n}{n-k}$.
- El-Khamy and McEliece showed a better optimization problem for the assignment of the multiplicity matrix providing an MAA leading to greatly improved error correcting ability [15].
- In [17], Guruswami and Rudra showed that by carefully selection the locations at which the Reed-Solomon is evaluated, cyclic errors up to the capacity can be corrected. He calls this *folded Reed-Solomon codes* and takes advantage of relation between polynomials evaluated at specified locations.
- Iterative algorithms to perform the interpolation while giving the intermediate results to be factored are being developed.
- Lastly, some papers go toward an implementation of this algorithm on an electronic level with VLSI.

Combining these recent advances together, it would not be surprising to have significantly increased decoding performances while having a significant reduction in computational complexity. This would make this decoding algorithm also attractive at high rates. It is surely only a matter of time until they are applied in practice.

Bibliography

[1] The Theory of Error-Correcting Codes
- F.J. MacWilliams and N.J.A. Sloane
North-Holland, Amsterdam, 1977, 762 pp.

[2] Lecture notes
- J. Hall
http://www.mth.msu.edu/~jhall/classes/codenotes/coding-notes.html

[3] A new algorithm for decoding Reed-Solomon codes
- S. Gao
Communications, Information and Network Security (V. Bhargava, H. V. Poor, V. Tarokh and S. Yoon, Eds.), Kluwer Academic Publishers, 2003, p. pp. 55–68
http://www.math.clemson.edu/faculty/Gao/papers/RS.pdf

[4] A Mathematical Theory of Communication
- C. Shannon
http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf
The Bell System Technical Journal, Vol. 27, pp. 379-423, 623-656, July, October, 1948

[5] Decoding of Reed-Solomon codes beyond the error-correction bound
- M. Sudan
Journal of Complexity, 13(1): 180–193, March 1997
http://people.csail.mit.edu/madhu/papers/reeds-journ.ps

[6] Improved decoding of Reed-Solomon and algebraic-geometry codes
- V. Guruswami, M. Sudan
IEEE Transactions on Information Theory, 45(6): 1757–1767, September 1999.
http://people.csail.mit.edu/madhu/papers/venkat-journ.ps

[7] Reflections on "Improved decoding of Reed-Solomon and algebraic-geometry codes"
- V. Guruswami, M. Sudan
IEEE Information Theory Society Newsletter, Volume 52, Number 1, ISSN 1059-2362, pages 6-12, March 2002.
http://people.csail.mit.edu/madhu/papers/reflections.ps

[8] The Guruswami-Sudan Decoding Algorithm for Reed-Solomon Codes.
- R. J. McEliece
The Interplanetary Network Progress Report, IPN PR 42-153, January-March 2003, pp. 1-60
http://tmo.jpl.nasa.gov/progress_report/42-153/153F.pdf

[9] Algebraic soft-decision decoding of Reed-Solomon codes
- R. Koetter, A. Vardy
IEEE Transactions on Information Theory, vol. 49, no. 11, November 2003
http://www.eecs.berkeley.edu/~dolecek/coding/KoetterVardy03.pdf

[10] Efficient interpolation and factorization in algebraic soft-decision decoding of Reed-Solomon codes
- R. Koetter, J. Ma, A. Vardy, and A. Ahmed

[11] A Complexity Reducing Transformation in Algebraic List Decoding of Reed-Solomon Codes
- Ralf Koetter, Alexander Vardy

[12] Fast Generalized Minimum Distance Decoding of Algebraic-Geometry and Reed-Solomon Codes
- R. Kötter
IEEE Transaction in Information Theory, vol. 42, no. 3, pp. 721-737, May 1996

[13] Efficient Decoding of Reed-Solomon Codes beyond Half the Minimum Distance
- R. Roth and G. Ruckenstein
Trans. Info. Theory, vol. 46, no. 1, pp. 246256, Jan. 2000

[14] Algebraic Soft-Decision Decoding of Reed-Solomon Codes Using Bit-level Soft Information
- Jing Jiang and Krishna R. Narayanan

in Proc. Allerton Conference on Communications, Control and Computing 2006 and submitted to IEEE Trans. on Information Theory.
http://www.ece.tamu.edu/~jjiang/bit_gmd.pdf

[15] Interpolation Multiplicity Assignment Algorithms for Algebraic Soft-Decision Decoding of Reed Solomon Codes
M. El-Khamy and R. J. McEliece
AMS-DIMACS volume on Algebraic Coding Theory and Information Theory, vol. 68, 2005.
http://www.its.caltech.edu/~mostafa/pubs/DimacsElk8c.pdf

[16] Codes and Curves
- Judy L. Walker
expository monograph, published by the AMS in the IAS/Park City Mathematical Subseries of the Student Mathematical Series.
http://www.math.unl.edu/~jwalker7/papers/rev.pdf

[17] Achieving List Decoding Capacity Using Folded Reed-Solomon Codes
- V. Guruswami and A. Rudra
Invited paper at Allerton 2006
http://www.cs.washington.edu/homes/venkat/pubs/papers/FRS-allerton.pdf

[18] Iterative Algebraic Soft-Decision List Decoding of Reed-Solomon Codes
- M. El-Khamy. and R. J. McEliece
http://arxiv.org/PS_cache/cs/pdf/0509/0509097v1.pdf

[19] aximum-Likelihood Decoding of Reed-Solomon Codes is NP-hard
V. Guruswami and A. Vardy
SODA 2005; accepted to IEEE Trans. Info. Theory
http://www.cs.washington.edu/homes/venkat/pubs/papers/mldrs.pdf

Part III

Appendix

Appendix A

Source code

A.1 Algebra

```
using System;
using System.Collections.Generic;
using System.Text;

namespace SoftDecoding
{
public abstract class FiniteField
{
public readonly int p;
public readonly int n;
public readonly int q;

public readonly FFE[] elements;

protected FiniteField(int p, int n)
{
    this.p = p;
    this.n = n;
    this.q = (int) Math.Pow(p, n);
    this.elements = new FFE[q];
    for (int i = 0; i < q; i++)
        elements[i] = new FFE(i);
    FFE.DEFAULT_FIELD = this;
}
```

```
public abstract FFE Add(FFE a, FFE b);
public abstract FFE Substract(FFE a, FFE b);
public abstract FFE Multiply(FFE a, FFE b);
public abstract FFE Divide(FFE a, FFE b);
public abstract FFE Opposite(FFE a);
public abstract FFE Power(FFE a, int exp);


public FFE this[int value]
{
    get
    {
        return this.elements[value];
    }
}

public FFE this[int[] coefs]
{
    get
    {
        return this[AsValue(coefs)];
    }
}




public int[] AsCoefs(FFE a)
{
    return AsCoefs(a.value);
}

public int[] AsCoefs(int value)
{
    int[] coefs = new int[n];
    int ppow = 1;
    for (int i = 0; i < n; i++)
    {
        coefs[i] = (value % (p*ppow))/ppow;
        ppow *= p;
    }
    return coefs;
}

public int AsValue(int[] coefs)
{
    if (coefs.Length != n)
        throw new ArgumentException("Wrong number of coefficients. Should be " + n +

    int value = 0;
    int ppow = 1;
    for (int i = 0; i < n; i++)
    {
        value += coefs[i] * ppow;
        ppow *= p;
    }
    return ((value % q) + q) % q;
}


}
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace SoftDecoding
{
public struct FFE
{
internal static FiniteField DEFAULT_FIELD;

internal readonly int value;


internal FFE(int value) {
    this.value = value;
}



public override String ToString() {
    if (DEFAULT_FIELD.n == 1)
        return this.value.ToString();
    else
    {
        String ans = "";
        int[] coefs = DEFAULT_FIELD.AsCoefs(value);
        for (int i = 0; i < coefs.Length; i++)
            ans += coefs[i];
        return ans;
         /*
        if (this.value == 0)
            return "X";

        for (int i = 0; i < DEFAULT_FIELD.q; i++)
        {
            int power = (DEFAULT_FIELD[2] ^ i).value;
            if (this.value == power)
                return i.ToString();
        }

        throw new Exception("Field element is not the power of the generator element.
        */
    }
}


public static FFE operator +(FFE a, FFE b) {
    return DEFAULT_FIELD.Add(a, b);
}

public static FFE operator -(FFE a) {
    return DEFAULT_FIELD.Opposite(a);
}
```

```
public static FFE operator -(FFE a, FFE b) {
    return DEFAULT_FIELD.Substract(a, b);
}

public static FFE operator *(FFE a, FFE b) {
    return DEFAULT_FIELD.Multiply(a, b);
}

public static FFE operator /(FFE a, FFE b) {
    return DEFAULT_FIELD.Divide(a, b);
}

public static FFE operator ^(FFE a, int exp) {
    return DEFAULT_FIELD.Power(a, exp);
}

public static bool operator ==(FFE a, FFE b) {
    return a.value == b.value;
}

public static bool operator !=(FFE a, FFE b){
    return a.value != b.value;
}

public static String VectorToString(FFE[] vector)
{
    String ans = "";
    for (int i = 0; i < vector.Length; i++)
        ans += vector[i].ToString() + " ";
    return ans;
}
}

}
```

```
using System;
using System.Collections.Generic;
using System.Text;
```

```
namespace SoftDecoding
{
class PrimeField : FiniteField
{
protected int[,] powersTable;
protected int[] inverses;

public PrimeField(int p) : base(p, 1)
{
    powersTable = new int[p, p];
    inverses = new int[p];

    for (int i = 0; i < p; i++)
        for (int j = 0; j < p; j++)
            if (j == 0)
                powersTable[i, j] = 1;
            else
                powersTable[i, j] = powersTable[i, j - 1] * i % p;

    for (int i = 1; i < p; i++)
        for (int j = 1; j < p; j++)
            if ((i * j) % p == 1)
            {
                inverses[i] = j;
                break;
            }
}

public override FFE Add(FFE a, FFE b)
{
    return new FFE((a.value + b.value) % p);
}

public override FFE Substract(FFE a, FFE b)
{
    return new FFE((a.value - b.value + p) % p);
}

public override FFE Multiply(FFE a, FFE b)
{
    return new FFE((a.value * b.value) % p);
}

public override FFE Divide(FFE a, FFE b)
{
    if (b.value == 0)
        throw new DivideByZeroException();

    return new FFE((a.value * inverses[b.value]) % p);
}

public override FFE Opposite(FFE a)
{
    return new FFE(p - a.value);
}

public override FFE Power(FFE a, int exp)
{
    if (a.value == 0)
        if (exp == 0)
            return FFE.DEFAULT_FIELD[1];
        else
            return a;
    else
        return new FFE(powersTable[a.value, exp % (q-1)]);
}
}
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace SoftDecoding
{
class BinaryExtensionField : FiniteField
{
```

```
protected readonly int[] IRREDUCIBLE_POLYNOMIALS = new int[]{
    0, //0
    1, // = 2^1
    3, // = 2^2
    3, // = 2^3
    3, // = 2^4
    5, // = 2^5
    3, // = 2^6
    3, // = 2^7
    29, // = 2^8
    17, // = 2^9
    9, // = 2^10
    5, // = 2^11
    83, // = 2^12
    27, // = 2^13
    1091,  // = 2^14
    3,  // = 2^15
    989  // = 2^16
};

//protected readonly int[,] productsTable;
//protected readonly int[] inverses;
//protected readonly int[,] powersTable;
protected readonly int[] logs;
protected readonly int[] powers;

public BinaryExtensionField(int n) : base(2,n)
{
    /*
    productsTable = new int[q,q];
    inverses = new int[q];
    powersTable = new int[q, q - 1];
    */
    powers = new int[q-1]; //val = alpha^i => val = powers[i]
    logs = new int[q]; //val = alpha^i => i = logs[val]
    logs[0] = -1; // log(0) = -infinity

    int val = 1;
    for (int i = 0; i < q-1; i++)
    {
        while (val >= q)
            val = (val % q) ^ (IRREDUCIBLE_POLYNOMIALS[n] * (val >> n));
        powers[i] = val;
        logs[val] = i;

        val = 2 * val;
    }
}

public override FFE Add(FFE a, FFE b)
{
    return this.elements[a.value ^ b.value];
}

public override FFE Substract(FFE a, FFE b)
{
    return this.elements[a.value ^ b.value];
}

public override FFE Multiply(FFE a, FFE b)
{
    if (a.value == 0 | b.value == 0)
        return this.elements[0];
    else
        return this.elements[powers[(logs[a.value] + logs[b.value])%(q-1)]];
}

public override FFE Divide(FFE a, FFE b)
{
    if (b.value == 0)
        throw new DivideByZeroException();
    else if (a.value == 0)
        return this.elements[0];
    else
        return this.elements[ powers[
            (q - 1 + logs[a.value] - logs[b.value]) % (q - 1)
            ]];
}

public override FFE Opposite(FFE a)
```

```
{
    return a;
}

public override FFE Power(FFE a, int exp)
{
    if (a.value == 0)
        if (exp == 0)
            return this.elements[1];
        else
            return this.elements[0];
    else
        return this.elements[powers[(logs[a.value]* exp) % (q - 1)]];
}
}
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace SoftDecoding
{
struct Polynomial
{
private FFE[,] values;

public FFE this[int i, int j]
{
    get
    {
        if (i < values.GetLength(0) & j < values.GetLength(1))
            return values[i, j];
        else
            return FFE.DEFAULT_FIELD[0];
    }
    set
    {
        if (i < values.GetLength(0) & j < values.GetLength(1))
            values[i, j] = value;
        else
        {
            FFE[,] old = values;
            values = new FFE[Math.Max(i, old.GetLength(0)), Math.Max(j, old.GetLength
            for (int ii = 0; ii < old.GetLength(0); ii++)
                for (int jj = 0; jj < old.GetLength(1); jj++)
                    values[ii, jj] = old[ii, jj];
            values[i, j] = value;
        }
    }
}

public Polynomial(int[,] values)
{
    this.values = new FFE[values.GetLength(0), values.GetLength(1)];
    for (int i = 0; i < values.GetLength(0); i++)
        for (int j = 0; j < values.GetLength(1); j++)
            this.values[i, j] = FFE.DEFAULT_FIELD[values[i,j]];
}

public Polynomial(FFE[,] values)
{
    this.values = values;
}

public Polynomial(FFE[] values)
{
    this.values = new FFE[values.Length, 1];
    for (int i = 0; i < values.Length; i++)
        this.values[i,0] = values[i];
}


public static Polynomial GetX()
{
    return new Polynomial(new FFE[,] { { FFE.DEFAULT_FIELD[0] }, { FFE.DEFAULT_FIELD[
}
```

```
public static Polynomial GetY()
{
    return new Polynomial(new FFE[,] { { FFE.DEFAULT_FIELD[0], FFE.DEFAULT_FIELD[1] }
}

public int GetMaxDegX() {
    return values.GetLength(0)-1;
}

public int GetDegreeX()
{
    for (int i = this.GetMaxDegX(); i >= 0; i--)
        for (int j = 0; j <= this.GetMaxDegY(); j++)
            if (values[i, j] != FFE.DEFAULT_FIELD[0])
                return i;
    return 0;
}

public int GetMaxDegY(){
    return values.GetLength(1)-1;
}

public Polynomial(int maxDegX, int maxDegY)
{
    this.values = new FFE[maxDegX+1, maxDegY+1];
}




public static Polynomial operator +(Polynomial P, Polynomial Q)
{
    Polynomial res = new Polynomial(
        Math.Max(P.GetMaxDegX(), Q.GetMaxDegX()),
        Math.Max(P.GetMaxDegY(), Q.GetMaxDegY()));

    for (int i = 0; i <= res.GetMaxDegX(); i++)
        for (int j = 0; j <= res.GetMaxDegY(); j++)
            res[i, j] = P[i, j] + Q[i, j];
    return res;
}


public static Polynomial operator -(Polynomial P, Polynomial Q)
{
    Polynomial res = new Polynomial(
        Math.Max(P.GetMaxDegX(), Q.GetMaxDegX()),
        Math.Max(P.GetMaxDegY(), Q.GetMaxDegY()));

    for (int i = 0; i <= res.GetMaxDegX(); i++)
        for (int j = 0; j <= res.GetMaxDegY(); j++)
            res[i, j] = P[i, j] - Q[i, j];

    return res;
}


public static Polynomial operator *(Polynomial P, Polynomial Q)
{
    Polynomial res = new Polynomial(P.GetDegreeX() + Q.GetDegreeX(), P.GetMaxDegY() +

    for (int pi = 0; pi <= P.GetDegreeX(); pi++)
        for (int pj = 0; pj <= P.GetMaxDegY(); pj++)
            if(P[pi,pj] != FFE.DEFAULT_FIELD[0])
                for (int qi = 0; qi <= Q.GetDegreeX(); qi++)
                    for (int qj = 0; qj <= Q.GetMaxDegY(); qj++)
                        res[pi+qi, pj+qj] += P[pi, pj] * Q[qi, qj];

    return res;
}


public static Polynomial operator ^(Polynomial P, int exp)
{
    if (P == GetX())
    {
        Polynomial res = new Polynomial(exp, 0);
        res[exp,0] = FFE.DEFAULT_FIELD[1];
        return res;
    }
    else if (P == GetY())
    {
        Polynomial res = new Polynomial(0, exp);
        res[0, exp] = FFE.DEFAULT_FIELD[1];
        return res;
    }
    else
    {
        Polynomial ans = FFE.DEFAULT_FIELD[1];
        for (int i = 0; i < exp; i++)
            ans *= P;
        return ans;
    }
}




// implicit conversion from an FFE to a polynomial
public static implicit operator Polynomial(FFE value) {
    return new Polynomial(new FFE[,] { { value } });
}


public FFE Evaluate(FFE xVal, FFE yVal)
{
    FFE res = FFE.DEFAULT_FIELD[0];
    for (int i = 0; i <= GetMaxDegX(); i++)
        for (int j = 0; j <= GetMaxDegY(); j++)
            res += this.values[i,j]*(xVal^i)*(yVal^j);

    return res;
}



public override string ToString()
```

```
{
    String ans = "";
    FFE zero = FFE.DEFAULT_FIELD[0];
    for (int j = 0; j <= GetMaxDegY(); j++)
        for (int i = 0; i <= GetMaxDegX(); i++)
            if (this[i, j] != zero)
                ans += this[i, j] + (i == 0 ? "" : "x^" + i) + (j == 0 ? "" : "y^" +

    if (ans.Length > 0)
        return ans.Substring(0, ans.Length - 2);
    else
        return "0";
}


public static bool operator ==(Polynomial P, Polynomial Q)
{
    return !(P != Q);
}

public static bool operator !=(Polynomial P, Polynomial Q)
{
    int maxDegX = (int) Math.Max(P.GetMaxDegX(), Q.GetMaxDegX());
    int maxDegY = (int) Math.Max(P.GetMaxDegY(), Q.GetMaxDegY());

    for (int i = 0; i <= maxDegX; i++)
        for (int j = 0; j <= maxDegY; j++)
            if (P[i, j] != Q[i, j])
                return true;

    return false;
}
}

}
```

A.2 Communication system interfaces

```
using System;
using System.Collections.Generic;
using System.Text;

namespace SoftDecoding
{
interface Encoder
{
    FFE[] Encode(FFE[] message);
}
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace SoftDecoding
{
interface Channel
{
    void Send(FFE[] word);
    double[,] Receive(); // returns the reliability matrix
}
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace SoftDecoding
{
interface Decoder
{
    FFE[] Decode(double[,] RM);
}
}
```

A.3 Reed-Solomon encoders/decoders

```
using System;
using System.Collections.Generic;
using System.Text;

namespace SoftDecoding
{
class RSEncoder : Encoder
{
public readonly int n;
public readonly int k;
public readonly FFE[] alphas;



internal RSEncoder(int k, FFE[] alphas)
{
    this.n = alphas.Length;
    this.k = k;
    this.alphas = alphas;
}

public static FFE[] GetDefautLocation(FiniteField Fq, int n)
{
    if (n > Fq.q)
        throw new ArgumentException("More locations than field elements is not allowe

    FFE[] alphas = new FFE[n];
    if (n < Fq.q)
        for (int i = 0; i < n; i++)
            alphas[i] = Fq[i + 1];
    else // n == q
        for (int i = 0; i < n; i++)
            alphas[i] = Fq[i];

    return alphas;
}

public FFE[] Encode(FFE[] message)
{
    Polynomial f = new Polynomial(message);
    FFE[] x = new FFE[n];
    for (int i = 0; i < n; i++)
        x[i] = f.Evaluate(alphas[i], FFE.DEFAULT_FIELD[0]);
    return x;
}
}
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace SoftDecoding
{
class GaoDecoder : Decoder
{
protected FiniteField Fq;
protected int k;
protected int n;
protected FFE[] alphas;

public GaoDecoder(FiniteField Fq, FFE[] alphas, int k)
{
    this.Fq = Fq;
    this.alphas = alphas;
    this.n = alphas.Length;
    this.k = k;
}

FFE[] Decoder.Decode(double[,] RM)
{
    FFE[] y = new FFE[n];

    for (int i = 0; i < n; i++)
    {
        int maxJ = 0;
```

```
            double max = 0;
            for (int j = 0; j < Fq.q; j++)
            {
                if (RM[i, j] > max)
                {
                    max = RM[i, j];
                    maxJ = j;
                }
            }
            y[i] = Fq[maxJ];
        }
        return this.Decode(y);
    }

    private FFE[] Decode(FFE[] y)
    {
        Polynomial[] p_i = new Polynomial[3];
        Polynomial[] v_i = new Polynomial[3] { Fq[0], Fq[0], Fq[1] };

        Polynomial x = Polynomial.GetX();

        // p_1 = prod_j (x - alpha_j)
        p_i[1] = Fq[1];
        for (int i = 0; i < n; i++)
            p_i[1] *= (x - alphas[i]);

        // p_2 = sum y_i prod (x - alpha_j)/(alpha_i - alpha_j)
        p_i[2] = Fq[0];
        for (int i = 0; i < n; i++)
        {
            Polynomial fact = y[i];
            for (int j = 0; j < n; j++)
                if (j != i)
                    fact *= Fq[1]/(alphas[i] - alphas[j])*(x - alphas[j]);
            p_i[2] += fact;
        }

        if (p_i[2].GetDegreeX() < k)
        {
            FFE[] hatx = new FFE[n];
            for (int i = 0; i < n; i++)
                hatx[i] = p_i[2].Evaluate(alphas[i], Fq[0]);
            return hatx;
        }



        // the Euclidean division begins...
        Polynomial[] QR;
        do
        {
            p_i[0] = p_i[1];
            p_i[1] = p_i[2];

            v_i[0] = v_i[1];
            v_i[1] = v_i[2];

            QR = GetQuotientRemainder(p_i[0], p_i[1]);

            p_i[2] = QR[1];
            v_i[2] = -Fq[1] * QR[0] * v_i[1] + v_i[0];
        }
        while(2 * p_i[2].GetDegreeX() >= n+k);

        //p_i[2] = f * v[2] + r
        QR = GetQuotientRemainder(p_i[2], v_i[2]);
        if (QR[1] == Fq[0])
        {
            // Success
            FFE[] hatx = new FFE[n];
            for (int i = 0; i < n; i++)
                hatx[i] = QR[0].Evaluate(alphas[i], Fq[0]);
            return hatx;
        }
        else
        {
            // Failure
            return new FFE[n];
        }
    }
```

```
private Polynomial[] GetQuotientRemainder(Polynomial dividend, Polynomial divisor)
{
    if (divisor == Fq[0])
        throw new DivideByZeroException();

    if(dividend.GetDegreeX() < divisor.GetDegreeX())
        throw new ArgumentException("The degree of the dividend cannot be less than t

    Polynomial remainder = Fq[1] * dividend; // to make a "by value copy" of the divi
    Polynomial quotient = new Polynomial(0,0);
    Polynomial x = Polynomial.GetX();

    int degDiv = divisor.GetDegreeX();
    FFE leadingCoeficient = divisor[degDiv,0];

    for (int deg = dividend.GetDegreeX(); deg >= degDiv; deg--)
    {
        FFE coef = remainder[deg, 0] / leadingCoeficient;
        quotient += coef * (x ^ (deg - degDiv));
        remainder -= coef * divisor * (x ^ (deg - degDiv));
    }

    return new Polynomial[] { quotient, remainder };
}
}
}
```

```
//#define VERBOSE
#define MUTE

using System;
using System.Collections.Generic;
using System.Text;

namespace SoftDecoding
{
class RSSoftDecoder : Decoder
```

```
{
int k;
int n;
FFE[] alphas;
FiniteField Fq = FFE.DEFAULT_FIELD;

private int nbMultiplicities;




public RSSoftDecoder(int k, FFE[] alphas, int nbMultiplicities)
{
    if (alphas == null)
        throw new ArgumentNullException("'alphas' cannot be null.");

    if (!(0 < k & k < alphas.Length & alphas.Length <= Fq.q))
        throw new ArgumentException("Arguments 'k' and 'n' do not satisfy: 0 < k < n

    for (int i = 0; i < n; i++)
        for (int j = i+1; j < n; j++)
            if(alphas[i] == alphas[j])
                throw new ArgumentException("Evaluating locations must all be distinc

    this.k = k;
    this.n = alphas.Length;
    this.alphas = alphas;
    this.nbMultiplicities = nbMultiplicities;
}



public FFE[] Decode(double[,] RM)
{
    if (RM == null)
        throw new ArgumentNullException("RM cannot be null");

    if (RM.GetLength(0) != n | RM.GetLength(1) != Fq.q)
        throw new ArgumentException("RM matrix should have size n x q (" + n + " x "
    int[,] M = GreedyMAA(RM);

    int omega = ComputeOmega(Cost(M));
    List<Polynomial> polyList = ListDecode(M, omega);

    if (polyList.Count == 0)
    {
        //throw new Exception("Decoding failed");
        return new FFE[n];
    }

    double maxProba = 0;
    Polynomial best = polyList[0];

    foreach (Polynomial poly in polyList)
    {
        double proba = 1;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < Fq.q; j++)
                if (poly.Evaluate(alphas[i], Fq[0]) == Fq[j])
                    proba *= RM[i, j];

        if (proba > maxProba)
        {
            maxProba = proba;
            best = poly;
        }
    }
#if !MUTE
    Console.WriteLine("\n--== RESULT ==--\n" + best);
#endif
    FFE[] x = new FFE[n];
    for (int i = 0; i < n; i++)
        x[i] = best.Evaluate(alphas[i], Fq[0]);

    return x;
}

public int[,] GreedyMAA(double[,] RM)
{
    int[,] M = new int[n, Fq.q];

    for (int m = 0; m < nbMultiplicities; m++)
    {
        double heighest = 0;
        int hi = 0;
        int hj = 0;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < Fq.q; j++)
                if (RM[i, j]/(M[i,j] + 1) > heighest)
                {
                    heighest = RM[i, j]/(M[i,j] + 1);
                    hi = i;
                    hj = j;
                }
        M[hi, hj]++;
    }
    return M;
}

/*
public FFE[] Decode(FFE[] y)
{
    if (y == null)
        throw new ArgumentNullException("'y' cannot be null");

    int[,] M = new int[n, Fq.q];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < Fq.q; j++)
            if(y[i] == Fq[j])
                M[i, j] = nbMultiplicities / n; //nbMultiplicities must be a multiple

    int omega = ComputeOmega(n * nbMultiplicities * (nbMultiplicities + 1) / 2);
    List<Polynomial> polyList = ListDecode(M, omega);


    Polynomial best = polyList[0];
    int maxAgree = 0;

    foreach (Polynomial poly in polyList)
    {
        int agree = 0;
        for (int i = 0; i < n; i++)
            if (poly.Evaluate(alphas[i], Fq[0]) == y[i])
                agree++;

        if (agree > maxAgree)
        {
            maxAgree = agree;
            best = poly;
        }
    }
#if !MUTE
    Console.WriteLine("\n--== RESULT ==--\n" + best);
#endif
    FFE[] x = new FFE[n];
    for (int i = 0; i < n; i++)
        x[i] = best.Evaluate(alphas[i], Fq[0]);

    return x;
}
*/

public List<Polynomial> ListDecode(int[,] M, int omega)
{
    Polynomial Q = FindQ(M, omega);
#if !MUTE
    Console.WriteLine("\n--== Interpolation finished ==--\n" + Q+"\n");
#endif
    List<Polynomial> polyList = new List<Polynomial>();
    FactorizeRR(Q, 0, new Polynomial(0,0), polyList);
#if !MUTE
    Console.WriteLine("\n--== Factorization finished ==--");
    foreach (Polynomial poly in polyList)
        Console.WriteLine(poly);
#endif
    return polyList;
}
```

```
public static long C(int j, int n)
{
    long ans = 1;

    if (j > n / 2)
        j = n-j;

    for (int i = 0; i < j; i++)
        ans *= (n - i);

    ans /= Fact(j);

    return ans;
}


static int Fact(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * Fact(n - 1);
}

// divides by the greatest power of X possible and shrink the array as much as possib
Polynomial NormalizeX(Polynomial Q)
{
    int xPowerDivisor = -1;

    for (int i = 0; i <= Q.GetMaxDegX() & xPowerDivisor == -1; i++)
        for (int j = 0; j <= Q.GetMaxDegY(); j++)
            if (Q[i, j] != Fq[0])
            {
                xPowerDivisor = i;
                break;
            }

    if (xPowerDivisor == -1) {
        return new Polynomial(0,0);
    }
    int xDegreesInExcess = -1;
    for (int i = Q.GetMaxDegX(); i > 0 & xDegreesInExcess == -1; i--)
        for (int j = 0; j <= Q.GetMaxDegY(); j++)
            if (Q[i, j] != Fq[0])
            {
                xDegreesInExcess = Q.GetMaxDegX() - i;
                break;
            }

    if (xPowerDivisor > 0 | xDegreesInExcess > 0)
    {
        Polynomial divided = new Polynomial(Q.GetMaxDegX() - xPowerDivisor - xDegrees
        for (int i = 0; i <= divided.GetMaxDegX(); i++)
            for (int j = 0; j <= divided.GetMaxDegY(); j++)
                divided[i, j] = Q[i + xPowerDivisor, j];

        return divided;
    }
    else
        return Q;
}


public Polynomial Y_To_XY_Plus_Gamma(Polynomial Q, FFE gamma)
{
    Polynomial result = new Polynomial(Q.GetMaxDegX() + Q.GetMaxDegY(), Q.GetMaxDegY(

    for (int i = 0; i <= Q.GetMaxDegX(); i++)
        for (int j = 0; j <= Q.GetMaxDegY(); j++)
            for (int d = 0; d <= j; d++)
                result[i + d, d] += Fq[(int) C(d, j) % Fq.p] * Q[i, j] * (gamma ^ (j

    return result;
}



Polynomial HasseDerivative(Polynomial Q, int dx, int dy)
{
    if (dx == 0 & dy == 0)
        return Q;

    Polynomial DQ = new Polynomial(Math.Max(Q.GetMaxDegX() - dx,0), Math.Max(Q.GetMax
    for (int i = 0; i <= Q.GetMaxDegX() - dx; i++)
        for (int j = 0; j <= Q.GetMaxDegY() - dy; j++)
        {
            int ii = i + dx;
            int jj = j + dy;
            long ci = C(dx, ii);
            long cj = C(dy, jj);
            DQ[i, j] = Fq[(int) (C(dx,ii) % Fq.p)] * Fq[(int) (C(dy,jj) % Fq.p)] * Q[
        }
    return DQ;
}




public int Cost(int[,] M)
{
    int cost = 0;

    for (int i = 0; i < M.GetLength(0); i++)
        for (int j = 0; j < M.GetLength(1); j++)
            cost += M[i, j] * (M[i, j] + 1) / 2;

    return cost;
}


public int ComputeOmega(int cost)
{
    // this is a lower bound on Omega(cost)
    int omega = (int) Math.Sqrt(2 * (k - 1) * cost) - 1;
    int L = omega / (k - 1);

    // now, we find the least Omega such that NbMonoms(Omega) > cost
    int nbMonoms = (L + 1) * (omega+1 - L * (k - 1) / 2);
    while (nbMonoms <= cost)
    {
        omega++;
        L = omega / (k - 1);
        nbMonoms = (L + 1) * (omega+1 - L * (k - 1) / 2);
    }
    return omega;
}


public Polynomial FindQ(int[,] M, int omega)
{
    int L = omega / (k - 1);

    Polynomial[] Q = new Polynomial[L + 1];
    int[] wdeg = new int[L + 1];
    for (int l = 0; l <= L; l++)
    {
        Q[l] = new Polynomial(omega, L);
        Q[l][0, l] = Fq[1];
        wdeg[l] = l * (k - 1);
    }


    FFE[] lambdas = new FFE[L + 1];
    int lowestL;


    for (int i = 0; i < alphas.Length; i++)
    for (int j = 0; j < Fq.q; j++)
    if (M[i, j] != 0)
    {
#if VERBOSE
        Console.WriteLine("i,j: " + alphas[i] + "," + Fq[j]);
#endif

        for (int u = 0; u < M[i, j]; u++)
        for (int v = 0; v < M[i, j] - u; v++)
        {
#if !MUTE
#if !VERBOSE
            Console.Write(".");
#endif
#endif
            lowestL = -1;
            for (int l = 0; l <= L; l++)
            {
                Polynomial Duv = HasseDerivative(Q[l], u, v);
                lambdas[l] = Duv.Evaluate(alphas[i], Fq[j]);
#if VERBOSE
                Console.WriteLine("Q["+l+"](x,y) = " +Q[l] + "\n    => D_" + u + ","
#endif
                if (lambdas[l] != Fq[0])
                    if (lowestL == -1 || wdeg[l] < wdeg[lowestL])
                        lowestL = l;
            }
#if VERBOSE
            Console.WriteLine("\n" + "Lowest l is: " + lowestL + "\n");
#endif

            if (lowestL != -1)
            {
                for (int l = 0; l <= L; l++)
                if (lambdas[l] != Fq[0] & l != lowestL)
                    Q[l] =  Q[l] - (lambdas[l]/lambdas[lowestL]) * Q[lowestL];


                Q[lowestL] = (Polynomial.GetX() - alphas[i]) * Q[lowestL];
                wdeg[lowestL] += 1;
            }
        }
    }


    return Q[ArgMin(wdeg)];
}

static int ArgMin(int[] array)
{
    int argMin = -1;
    for (int i = 0; i < array.Length; i++)
        if (argMin == -1 || array[i] < array[argMin])
            argMin = i;

    return argMin;
}




public void FactorizeRR(Polynomial Q, int i, Polynomial f, List<Polynomial> fList)
{
    Q = NormalizeX(Q);

#if VERBOSE
    Console.WriteLine("i: " + i + " - k: " + k);
#elif !MUTE
    Console.Write(".");
#endif

    for (int j = 0; j < Fq.q; j++)
    {
        FFE eval = Q.Evaluate(Fq[0], Fq[j]);
       if(eval == Fq[0])
      {
#if VERBOSE
            Console.WriteLine(Q);
            Console.WriteLine("Q(0," + Fq[j] + ") = " + Q.Evaluate(Fq[0], Fq[j]));
#endif
          Polynomial otherF = f + Fq[j] * (Polynomial.GetX()^i);
            if (i == k - 1)
                fList.Add(otherF);
            else
            {
                Polynomial nextQ = Y_To_XY_Plus_Gamma(Q, Fq[j]);
                FactorizeRR(nextQ, i + 1, otherF, fList);
            }
        }
    }
}
```

```
}
}
```

A.4 Channels

```
using System;
using System.Collections.Generic;
using System.Text;

namespace SoftDecoding
{
public class NoiselessChannel : Channel
{
protected FiniteField Fq;

public NoiselessChannel(FiniteField Fq)
{
    this.Fq = Fq;
}

FFE[] received;

public void Send(FFE[] word)
{
    received = word;
}


public double[,] Receive()
{
    int n = received.Length;
    double[,] RM = new double[n, Fq.q];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < Fq.q; j++)
            if (received[i] == Fq[j])
                RM[i, j] = 1.0;
    return RM;
}
public void SetVariance(double variance)
{
    //nothing to do
}
}
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace SoftDecoding
{
class QarySymetricChannel : Channel
{
public readonly FiniteField Fq;
public readonly double p_err;

private Random random = new Random();

private FFE[] received;
private double[,] RM;

public QarySymetricChannel(FiniteField Fq, double p_err)
{
    this.Fq = Fq;
    this.p_err = p_err;
}


public void Send(FFE[] word)
{
    int n = word.Length;
    received = new FFE[n];
    for (int i = 0; i < word.Length; i++)
        if (random.NextDouble() > p_err)
            received[i] = word[i];
        else
```

```
            do
                received[i] = Fq[random.Next(Fq.q)];
            while (received[i] == word[i]);

    RM = new double[word.Length, Fq.q];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < Fq.q; j++)
            if (received[i] == Fq[j])
                RM[i, j] = 1.0 - p_err;
            else
                RM[i, j] = p_err / (Fq.q - 1);
}

public double[,] Receive()
{
    return RM;
}


}
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace SoftDecoding
{
class ExtendedBinaryChannel : Channel
{
FiniteField Fq;
double p_err;

double[,] RM;
FFE[] received;

private Random random = new Random();
private int m;
```

```
public ExtendedBinaryChannel(FiniteField Fq, double p_err)
{
    if(Fq.p != 2)
        throw new Exception("This channel is only valid for symbols belonging to bina

    this.Fq = Fq;
    this.p_err = p_err;
    this.m = Fq.n;
}


private int Dist(FFE a, FFE b)
{
    int ans = 0;
    int[] coefsA = Fq.AsCoefs(a);
    int[] coefsB = Fq.AsCoefs(b);
    for (int i = 0; i < m; i++)
       if(coefsA[i] != coefsB[i])
            ans++;
    return ans;
}

public void Send(FFE[] word)
{
    int n = word.Length;
    int[] noise = new int[m];

    received = new FFE[n];
    for (int i = 0; i < word.Length; i++)
    {
        for (int j = 0; j < m; j++)
            if (random.NextDouble() > p_err)
                noise[j] = 0;
            else
                noise[j] = 1;

        received[i] = word[i] + Fq[noise];
    }
    RM = new double[n, Fq.q];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < Fq.q; j++)
        {
            int dist = Dist(received[i], Fq[j]);
            RM[i, j] = Math.Pow(p_err, dist) * Math.Pow(1.0 - p_err, m-dist);
        }
}


public double[,] Receive()
{
    return RM;
}

}
}
```