Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

DEPARTMENT OF INFORMATION ENGINEERING
UNIVERSITY OF PADOVA

Corso di Laurea Triennale in Ingegneria dell'Informazione

Reed-Solomon Codes

(Codici di Reed-Solomon)

Laureando

Tommaso Martini

Relatore

Prof. Tomaso Erseghe

25 luglio 2013

Anno Accademico 2012/2013

Abstract

This bachelor degree thesis has the purpose of presenting a general overview on *Reed-Solomon codes* as a subclass of *cyclic codes* and *BCH codes*, using a mathematical approach to describe and prove their many practical aspects. After briefly exposing *GaloisFields* theory, fundamental for the creation of error correcting codes, we will introduce a systematic encoding strategy through generator polynomial and a decoder based on *Berlekamp-Massey* and *Forney algorithms*. Every explanation will be followed by examples referring to a $RS(255, 223)$ code. Moreover a *Matlab* implementation af a system *encoder - channel - decoder* has been realised and the whole code can be found in this work.

Questa tesi di laurea triennale ha lo scopo di fornire una panoramica generale sui codici di Reed-Solomon come sottoclasse dei codici ciclici e dei codici BCH, utilizzando un approccio matematico per descrivere e dimostrare i loro numerosi aspetti vantaggiosi. Dopo aver brevemente esposto la teoria dei Campi di Galois, fondamentali per la costruzione di codici correttori d'errore, presenteremo una strategia di codifica sistematica tramite polinomio generatore e un decodificatore basato sugli algoritmi di Berlekamp-Massey e di Forney. Ogni spiegazione sarà seguita da esempi che fanno riferimento ad un codice $RS(255, 223)$. *È stata, inoltre, realizzata un'implementazione in codice Matlab di un sistema codificatore - canale - decodificatore e l'intero codice può essere trovato all'interno di questo elaborato.*

Contents

Chapter 1

Introduction

1.1 Channel coding

Imagine you have to write a letter to a friend: first of all you have to compose your message by lining up the words you need. Every word is made by one or more symbols taken from a finite set: in this case the alphabet (we will assume not to use punctuation).

Once the letter has been written, you have to send the paper on which the message is reported. The paper will probably travel through several stages before arriving to the receiver and, most times, your letter will not have an easy path to the destination: it could get wet, ripped or partially erased and a part of it could even be lost. In the worst case some letters can not be read by your friend or they could be misunderstood (e.g. a "i" could be damaged to transform into an "l"). All of these eventualities undermine the possibility for the receiver to properly understand your message or a part of that.

In telecommunication the question about transmitting a digital message has to face the same problems: a message made by sequences of symbols taken by a finite alphabet has to be sent through a noisy channel, which could damage or lose some of them.

A practical solution to the loss of a part of the message, and the easiest one, is trivially sending it twice or more times. Once the receiver owns several copies of the letter, even if all of those are partially unintellegible, he can interpolate them to piece together the original message. Of course this is a very expensive and wasteful solution, because we have to send twice all the letter even if only a word will be corrupted.

The smartest and most adopted strategy is adding to every word some redundancy symbols: if a word is made by k symbols (we will assume for

sake of simplicity that all the words have the same length), m symbols that are totally insignificant for the message itself can be added in order to help the receiver to correctly decode what has been written. Redundancy allows a decoder, under certain conditions, to detect corrupted symbols and even to correct them. This technique of a controlled addition of redundancy is called *channel coding*.

It is interesting to notice that spoken languages already provide a basic form of channel coding; as a matter of fact a language vocabulary does not contains any combination of alphabet characters. This assure a certain redundancy of the language, which is fundamental to help a listener to clearly understand a speech even though some letters or some sounds are not correctly received or heard at all.

For example: *osberve tht evon f some lettrs are missong or put i thy wkong place anp sme wods are not corpectpy writoen, hte sontence is stull cowprehensibme, unlss thepe are toi manc erros.*

As noticed by Pierce in [19, p. 143], intrinsic channel coding of the English language allows us to decode a sequence of sounds, or letters in the above example, as a known word that has a large part of tokens in common with the received one.

There are several ways to encode information words before transmitting them through a noisy channel. This document has been written to give a general overview on *Reed-Solomon codes*.

1.2 Hystorical overview

The birth of *channel coding* can be traced back to 1946, when a mathematician named Richard Wesley Hamming of the Bell Laboratories created a first trivial example of code using 7-bit words with three reduncancy bits: the *Hamming Code (7, 4)*, able to correct one bit error (a related article was published only in 1950: [9]). In 1948 Claude Elwood Shannon, also employed at the Bell Laboratories, published one of the milestones of information theory: article "*A Mathematical Theory of Communication*" [23], where he proved that the error probability on the message symbols of a transmission can be arbitrarily low if a suitable coding is applied. This result, based on Nyquist and Hartley works of 1928 [17][10] and known with the name of *Channel Capacity Theorem*, is fundamental because it tells us that, thanks to channel coding, we are able to reach the highest reliability in digital communication, considering, of course, some tradeoffs between electrical power, circuitry and computational complexity, information rate and

so on. After *Hamming Codes*, *Golay Codes* (1946) and *Reed-Muller Codes* (1954) represented the development of the first block codes, offering a higher robustness, transmission power being equal.

In 1957 Eugene Prange, code thorist at the Air Force Cambridge Research Laboratory of Massachusetts, first introduced the idea of *cyclic codes* and studied their many properties [20]. A subclass of *cyclic codes* was the subject of the research of French mathematician Alexis Hocquenghem and of Indian American mathematician Ray Chandra Bose and his student Dijen K. Ray-Chaudhuri. The former published a paper about these new and more powerful codes in 1959 [11], the latter in 1960 [5]. Because of the simultaneity and independence of the discoveries, the codes are now referred to by using the three scientists initials: *BCH codes*.

Irving Reed and Gustave Solomon were two mathematicians working at the Lincoln Laboratory of the Massachusetts Institute of Technology (MIT) of Boston. In 1960 they presented their particular class of non-binary *BCH codes* in the paper "*Polynomial Codes over Certain Finite Fields*" [21]. Even though their codes presented significant benefits with respect to previous codes, they have been used only since almost 1970, because of their onerous decoding computation, as also explained by Berlekamp in [2, p. vii]. In 1960 computer scientist William Wesley Peterson had already suggested a decoding algorithm with complexity $\mathcal{O}(d_{min}^3)$ [18], as also said in [4, p. 176], but the turning point occurred in 1968, when American mathematician Elwyn Berlekamp proposed his version of a $\mathcal{O}(d_{min}^2)$ decoder for *BCH codes* [2], as explained in [4, p. 180]. A few months later information theorist James Lee Massey realised that Berlekamp algorithm was applicable to linear feedback shift registers (*LFSR*) and thus easily implementable by electrical circuits [15]. The *Berlekamp-Massey algorithm* allowed the wide spred of *Reed-Solomon codes* in telecommunication systems.

Reed-Solomon codes are nowadays one of the best encoding strategies to correct and control errors and they are used in several areas: data storage in digital devices (CD, DVD, Blue-Ray), satellite transmissions, high speed modems (ADSL), deep space communications, digital television, secret sharing techniques, QR-codes, wireless and mobile communications and so on.

1.3 Original approach to *Reed-Solomon codes*

The original logic behind the coding technique, as Reed and Solomon described in their paper [21], is very simple. Suppose you have a word of k symbols $[m_0\ m_1\ m_2\ \ldots\ m_{k-1}]$ to transmit. A polynomial of degree $k-1$ can

be made up using the k symbols as coefficients:

$$p(x) = m_0 + m_1 x + m_2 x^2 + \ldots + m_{k-2} x^{k-2} + m_{k-1} x^{k-1} \tag{1.1}$$

The polynomial $p(x)$ is then evaluated in n settled points $[q_0\ q_1\ q_2\ \ldots\ q_{n-1}]$ in order to obtain a vector of n symbols (it must be $n > k$), which will represent the code word:

$$\begin{aligned} c_0 &= p(q_0) \\ c_1 &= p(q_1) \\ c_2 &= p(q_2) \\ &\vdots \\ c_{n-1} &= p(q_{n-1}) \end{aligned}$$

The word actually transmitted is $[c_0\ c_1\ c_2\ \ldots\ c_{n-1}]$. The receiver, who must know the n values in which the polynomial has been evaluated, has to reconstruct original coefficients by solving a system of k variables $[m_0\ m_1\ m_2\ \ldots\ m_{k-1}]$ and n equations:

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} p(q_0) \\ p(q_1) \\ p(q_2) \\ \vdots \\ p(q_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & q_0 & q_0^2 & \cdots & q_0^{k-1} \\ 1 & q_1 & q_1^2 & \cdots & q_1^{k-1} \\ 1 & q_2 & q_2^2 & \cdots & q_2^{k-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & q_{n-1} & q_{n-1}^2 & \cdots & q_{n-1}^{k-1} \end{bmatrix} \cdot \begin{bmatrix} m_0 \\ m_1 \\ m_2 \\ \vdots \\ m_{k-1} \end{bmatrix} \tag{1.2}$$

Of course some of the transmitted symbols $[c_0\ c_1\ c_2\ \ldots\ c_{n-1}]$ will be corrupted by the noisy channel (scratches on the CD surface, imperfections of the hard disk, radio frequency interference ...) and the received n-ple $[\hat{c}_0\ \hat{c}_1\ \hat{c}_2\ \ldots\ \hat{c}_{n-1}]$ will be different from the original one. Luckily algebra tells us that k points are sufficient to uniquely identify a $k-1$ degree polynomial [26, p. 3-4] and, unless too much errors have been introduced or a too small number of points q has been used, the reconstructed polynomial will properly approximate the original one, $p(x)$.

1.4 Chapters contents

Coding as introduced is a valid way to use *Reed-Solomon codes*, but this is not the only approach, and not the most used as well. In the following chapters another definition, with its related encoding technique, will be exposed. Before reaching it, however, it is worth recalling and presenting the

background algebraic tools that will be used. **Chapter 2: Galois Fields** gives a brief revision of algebraic theory, focusing on the theory of *finite fields* and their properties, essential for the construction of efficient error correcting codes. **Chapter 3: Reed-Solomon Codes** can be seen as a sort of matryoshka: using a quite chronological slant, we will start from the most generic class of block codes, linear codes, and we will go deeper and deeper to *Reed-Solomon codes*, the most specific class. For every subclass of codes (*linear*, *cyclic*, *BCH*) we will focus only on the characteristics which are fundamental to understand and build *RS codes*. In the end of the chapter we will investigate one of the most employed codes: the *RS(255, 223)*, which will be used as example for the whole document. **Chapter 4: Reed-Solomon Encoder** explains the basic strategy to encode in systematic form an input word through a *Reed-Solomon code*, while **Chapter 5: Reed-Solomon Decoder** illustrates the Berlekamp-Massey algorithm for a quick decoding. Chapter 4 makes use of the *RS(255, 223)* to integrate the explanation, while, for the sake of clearness, exmaples in Chapter 5 concern a shorter code *RS(7, 4)*. Eventually, **Chapter 6: Matlab Implementation** exposes the practical side of this work, giving a brief view of what and how has been made by the *Matlab* programming language.

Chapters **2**, **3**, **4** and **5** have as main reference Monti's work: *Teoria dei Codici: Codici a Blocco* [16]. Quite all the here exposed definitions and theorems are investigated in more detail in this book and, unless different specified, the reader is addressed to that for further information.

Chapter 2

Galois Fields

Reed-Solomon codes are realised by the use of some as powerful as simple algebraic structures, which are the basic background for coding theory. In this section a short summary about these fundamental mathematical instruments will be exposed, introducing the concepts of *group*, *ring*, *field* and, eventually, of *Galois Field*. A more detailed view on these topics can be found in [4, pp. 16-41, 65-90] and [14, pp. 1-13] and, for a more complete mathematical explanation, in [3] and [25].

2.1 Groups

A *group* $\mathfrak{G}$ with an operation "$*$" is referred to with notation: $(\mathfrak{G}, *)$ and can be defined as it follows:

- $(\mathfrak{G}, *)$ is non-empty: $\exists a \in \mathfrak{G}$.
- Operation "$*$" is associative: $a * (b * c) = (a * b) * c$ with $a, b, c \in \mathfrak{G}$.
- Operation "$*$" has the identity element, indicated with the symbol e, that is: $\forall a \in \mathfrak{G}\ a * e = e * a = a$.
- Every element a in $\mathfrak{G}$ has an inverse with respect to operation "$*$", indicated with symbol a', that is: $\forall a \in \mathfrak{G}\ \exists\ a' \in \mathfrak{G} \mid a * a' = a' * a = e$.

A *group* is said to be *abelian*, or *commutative*, if operation "$*$" is commutative, that is: $\forall a, b \in \mathfrak{G}\ a * b = b * a$.

2.1.1 Finite Groups

When a *group* $\mathcal{G}$ has a finite number of elements, then we can talk about a *finite group* and its numerosity q is called its *group order*. Let's consider

any element a of the *finite group* $\mathcal{G}$. Operating on a through "$*$" we will obtain:

$$\begin{aligned} a^2 &= a * a \\ a^3 &= a * a * a \\ a^4 &= a * a * a * a \\ &\vdots \end{aligned}$$

Since the group has a finite number of elements, sooner or later we will find two powers of a equal to each other: $a^i = a^j$ with $i < j$.

Then, it is possible to execute these following steps:

$$\begin{aligned} a^i &= a^j \\ (a^i)' * a^i &= (a^i)' * a^j \\ e &= a^{j-i} \end{aligned}$$

Notice that a^{j-i} makes sense beacuse $j > i$, but i and j are not the only values that verify the equality. If we define m as the smallest integer such that $a^m = e$, that is the lowest difference $j - i$, then m is said to be the *order* of a with respect to operation "$*$".

If an element a has got order m, then the first m powers of a are certainly distinct, otherwise the order would be lower. As a matter of fact, assuming that $a^h = a^k$ with $0 < h < k < m$, then we should have $e = a^{k-h}$, with $k - h < m$, which is in contradiction with the hypotheses of order m. It follows that, since in $\mathcal{G}$ there are only q different elements, the order of an element is always lower or equal to q.

Theorem 1. *Let m be the maximum among element orders of an* abelian finite group*, then m is a multiple of the order of any other element of the group.*

Proof. Let a be the element of the *abelian finite group* $\mathcal{G}$ of maximum order m. Let's choose another element of $\mathcal{G}$ with order $n < m$: both m and n can be divided in a certain number of factors. Let's write m and n using all the divisors $p_1, p_2, \ldots, p_\nu$ of both m and n, with 0 as exponent in case any integer is not present in the decomposition of one of the two values:

$$\begin{aligned} m &= p_1^{m_1} \cdot p_2^{m_2} \cdot \ldots \cdot p_\nu^{m_\nu} \\ n &= p_1^{n_1} \cdot p_2^{n_2} \cdot \ldots \cdot p_\nu^{n_\nu} \end{aligned}$$

Let's impose an absurd condition: m is not a multiple of n. This means that there is at least one factor p_i which has an exponent in n greater than its exponent in m: $n_i > m_i$ with $1 \leq i \leq \nu$.

For any positive integer $1 \leq j \leq \nu$ it is possible to find an element a_j of $\mathcal{G}$ which has got order $p_j^{m_j}$, as a matter of fact we can take:

$$a_j = a^{\frac{m}{p_j^{m_j}}} = a^{\frac{p_1^{m_1} \cdot p_2^{m_2} \cdot \ldots \cdot p_j^{m_j} \cdot \ldots \cdot p_\nu^{m_\nu}}{p_j^{m_j}}} = a^{p_1^{m_1} \cdot p_2^{m_2} \cdot \ldots \cdot \tilde{p}_j^{m_j} \cdot \ldots \cdot p_\nu^{m_\nu}} \tag{2.1}$$

where notation $\tilde{p}_j$ means that element p_j is not present in the product. Then $a_j^{p_j^{m_j}} = a^{p_1^{m_1} \cdot p_2^{m_2} \cdot \ldots \cdot p_j^{m_j} \cdot \ldots \cdot p_\nu^{m_\nu}} = a^m = e$.

In the same way let's take an element b_i of $\mathcal{G}$ with order $p_i^{n_i}$ and let's consider the element

$$c = a_1 * a_2 * \ldots * a_{i-1} * b_i * a_{i+1} * \ldots * a_\nu$$

where a_l, $l = 1, 2, \ldots, \nu$, is a generic element made up as exposed in (2.1). Since all the p_l are prime factors, c has order:

$$p_1^{m_1} \cdot p_2^{m_2} \cdot \ldots \cdot p_{i-1}^{m_{i-1}} \cdot p_i^{n_1} \cdot p_{i+1}^{m_{i+1}} \cdot \ldots \cdot p_\nu^{m_\nu} = m \frac{p_i^{n_i}}{p_i^{m_i}}$$

We can notice that $p_i^{n_i} > p_i^{m_i}$, because $n_i > m_i$. This implies that $\frac{p_i^{n_i}}{p_i^{m_i}} > 1$ and $m \frac{p_i^{n_i}}{p_i^{m_i}} > m$, which is equivalent to say that c has greater order than a, which is, for hypotheses, the maximum order element. In this fact we find the contradiction, therefore $\forall i \; 1 \leq i \leq \nu$ we must have $m_i \leq n_i$, that is m is a multiple of n.

This demonstrates that the maximum order of an *abelian finite group* is multiple of the order of any of its elements. □

A *cyclic group* is a *finite group* in which there exists at least one element a whose powers cover all the elements of the group; in other words, its order is $m = q$:

$$a, \quad a^2, \quad \ldots, \quad a^{q-1}, \quad a^q = e, \quad a^{q+1} = a, \quad a^{q+2} = a^2, \quad \ldots$$

Since we cannot have higher order than the cardinality of the set, we can deduce that, for *cyclic groups*, the order q of the set is also the maximum order of the elements and, from the previous theorem, also a multiple value for every other order.

2.2 Rings

A *ring* $\mathfrak{R}$ is a set of elements in which two algebraic operations, "$+$" (called *addition*) and "$\cdot$" (called *multiplication*), are defined. Intuitively it

is a set where addition, subtraction and multiplication, but not division, are allowed. All of these operations between two elements of the *ring* give as result another element belonging to the *ring*.

A *ring* is a set $(\mathfrak{R}, +, \cdot)$ with the following characteristics:

- $(\mathfrak{R}, +)$ is an *abelian group*:
  1. Operation "+" is associative: $a+(b+c) = (a+b)+c$ with $a, b, c \in \mathfrak{R}$
  2. Operation "+" has got the identity element, indicated with the symbol 0, that is: $\forall a \in \mathfrak{R}\ a + 0 = 0 + a = a$
  3. Every element a in $\mathfrak{R}$ has an inverse with respect to the operation "+", indicated with the symbol $-a$, that is: $\forall a \in \mathfrak{R}\ \exists -a \in \mathfrak{R} \mid a + (-a) = 0$
  4. Operation "+" is commutative, that is $\forall a, b \in \mathfrak{R}\ a + b = b + a$.
- Operation "$\cdot$" is associative: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ with $a, b, c \in \mathfrak{R}$.
- Multiplication "$\cdot$" is distributive with respect to addition "+", that is: $\forall a, b, c \in \mathfrak{R}\ a \cdot (b + c) = a \cdot b + a \cdot c$.

2.2.1 Operations modulo-p

It is useful, before moving on to finite sets, to introduce operations *modulo-p*, which will be widely used in the following. In order to describe this kind of operations it is worth defining the operator *reminder*, $\mathcal{R}_p[\cdot]$:

$$a = q \cdot p + r \quad \text{with } r < q \text{ and } r < p \quad \Rightarrow \quad \mathcal{R}_p[a] = r$$

Addition modulo-p

Addition modulo-p can be indicated with notation $\mathcal{R}_p[a + b]$ and it is defined as:

$$\mathcal{R}_p[a + b] = r \ \Leftrightarrow\ (a + b) = q \cdot p + r \quad \text{with } r < q \text{ and } r < p$$

Multiplication modulo-p

Multiplication modulo-p can be indicated with notation $\mathcal{R}_p[a \cdot b]$ and it is defined as:

$$\mathcal{R}_p[a \cdot b] = r \ \Leftrightarrow\ (a \cdot b) = q \cdot p + r \quad \text{with } r < q \text{ and } r < p$$

2.2.2 Examples of important Rings

In the following we will make a wide use of two particular *rings*, thus it is worth briefly introducing them right now. Moreover, also the finite version of these two *rings* are now introduced.

Ring of integer numbers

Let's start by considering the *ring* of integer numbers: $\mathbb{Z}$. It is easy to see that this set satisfies all the properties of the *rings*, as a matter of fact $\mathbb{Z}$ is provided with the operation of addition "$+$" and multiplication "$\cdot$" with the respective identity elements 0 and 1 and it has an inverse element with respect to addition, but not to multiplication. We can now build a ring $\mathcal{R}$, also called $\mathbb{Z}/q$, with a finite number q of elements, as it follows:

1. consider the set of the first q positive integers, and 0, in $\mathbb{Z}$: $\{0, 1, 2, \ldots, q-1\}$
2. define the operation *addition* as the addition modulo q: $\forall a, b \in \mathcal{R}\ a + b \doteq \mathcal{R}_q[a + b]$ where the symbol "$+$" in the left memeber denotes the operation of addition in the set $\mathbb{Z}$
3. define the operation *multiplication* as the multiplication modulo q: $\forall a, b \in \mathcal{R}\ a \cdot b \doteq \mathcal{R}_q[a \cdot b]$ where the symbol "$\cdot$" in the left memeber denotes the operation of multiplication in the set $\mathbb{Z}$

It is easy to show that the set got in this way is still an algebraic ring, but we will omit here this trivial proof.

Ring of polynomials

The set of all the polynomials with coefficients in $\mathbb{Z}$ forms a *ring*. It is always possible to sum and multiply two polynomials, but not always the result of a division between two polynomials is still included in the polynomial ring because of the presence of a possible reminder (e.g. $x^3 \div (x^2 + x) = x - 1$ with x as reminder). It is possible to create a finite ring of polynomials, but, as we would need to use *finite fields* in order to define them, we will postpone the analysis of this set.

2.3 Fields

A *field* $\mathfrak{F}$ is a set of elements in which two algebraic operations, "$+$" (called *addition*) and "$\cdot$" (called *multiplication*), are defined. Intuitively it

is a set where addition, subtraction, multiplication and division are allowed. All of these operations between two elements of the *field* give as result another element belonging to the *field*.

Formally a *field* is a set $(\mathfrak{F}, +, \cdot)$ with the following characteristics:

- $(\mathfrak{F}, +)$ is an *abelian group*:

  1. Operation "$+$" is associative: $a+(b+c) = (a+b)+c$ with $a, b, c \in \mathfrak{F}$
  2. Operation "$+$" has got the identity element, indicated with the symbol 0, that is: $\forall a \in \mathfrak{F}\ a + 0 = 0 + a = a$
  3. Every element a in $\mathfrak{F}$ has an inverse with respect to the operation "$+$", indicated with the symbol $-a$, that is: $\forall a \in \mathfrak{F}\ \exists -a \in \mathfrak{F} \mid a + (-a) = 0$
  4. Operation "$+$" is commutative, that is $\forall a, b \in \mathfrak{F}\ a + b = b + a$.

- $(\tilde{\mathfrak{F}_0}, \cdot)$, that is the set $\mathfrak{F}$ deprived of the identity element of the addition, 0, is an *abelian group*:

  1. Operation "$\cdot$" is associative: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ with $a, b, c \in \tilde{\mathfrak{F}_0}$
  2. Operation "$\cdot$" has the identity element, indicated with the symbol 1, that is: $\forall a \in \tilde{\mathfrak{F}_0}\ a \cdot 1 = 1 \cdot a = a$
  3. Every element a in $\tilde{\mathfrak{F}_0}$ has an inverse with respect to operation "$\cdot$", indicated with the symbol a^{-1}, that is: $\forall a \in \tilde{\mathfrak{F}_0}\ \exists\ a^{-1} \in \tilde{\mathfrak{F}_0} \mid a \cdot a^{-1} = 1$
  4. Operation "$\cdot$" is commutative, that is $\forall a, b \in \tilde{\mathfrak{F}_0}\ a \cdot b = b \cdot a$.

- Multiplication is distributive with respect to addition, that is: $\forall a, b, c \in \mathfrak{F}\ a \cdot (b + c) = a \cdot b + a \cdot c$.

2.4 Galois Fields

A *finite field*, or *Galois Field*, is a field with a finite number of elements and it is usually referred to with the symbol $GF(q)$ where q is the number of elements in it. The special properties of this kind of sets make them suitable for the composition of error correcting codes particularly simple to encode and decode.

2.4.1 Examples of important Galois Fields

In the following we will make use, above all, of two kinds of *Galois Field*, which we will prove to be equivalent: *finite field of integer numbers* and *finite field of polynomials*.

Finite field of integer numbers

It is possible to prove that a *ring* of integer numbers $\mathbb{Z}/q$, as defined in (2.2.2), is a *finite field* if and only if q is prime.

Theorem 2. *$\mathbb{Z}/q$ is a* finite field $GF(q) \Leftrightarrow$ *q is prime.*

Proof. $\Rightarrow$) Let's proceed by contradiciton: q is not prime. Then there exists $a, b \in [1, q-1]$ such that $q = a \cdot b$. Since $\mathbb{Z}/q$ is a *field* for hypotheses, every element has got an inverse: $\exists\ a^{-1} \mid a \cdot a^{-1} = 1$; then $(a \cdot a^{-1}) \cdot b = 1 \cdot b = b$.

Hence:

$$a^{-1} \cdot [a \cdot b] = \mathcal{R}_q[a^{-1} \cdot \mathcal{R}_q[a \cdot b]] = \mathcal{R}_q[a^{-1} \cdot \mathcal{R}_q[q]] = \mathcal{R}_q[a^{-1} \cdot 0] = \mathcal{R}_q[0] = 0$$

Then we would have: $[a \cdot a^{-1}] \cdot b = b = 0$, but this is impossible, because $q = a \cdot b = a \cdot 0$ cannot be a zero element. Therefore, q must be prime.

$\Leftarrow$) The purpose is now to prove that, if q is prime, then every element has got an inverse with respect to multiplication. Given two positive integers s and t, it is always possible to find a couple of integers a and b such that $MCD[s,t] = as + bt$ [16, p. 83]. Then, since $MCD[s,q] = 1$ for any $s \in [1, q-1]$,

$$\begin{aligned} 1 &= \mathcal{R}_q[1] \\ &= \mathcal{R}_q[as + bq] = \mathcal{R}_q[as] + \mathcal{R}_q[bq] \\ &= \mathcal{R}_q[\mathcal{R}_q[a]\mathcal{R}_q[s]] + \mathcal{R}_q[\mathcal{R}_q[b]\mathcal{R}_q[q]] \\ &= \mathcal{R}_q[\mathcal{R}_q[a]s] + \mathcal{R}_q[\mathcal{R}_q[b] \cdot 0] \\ &= \mathcal{R}_q[\mathcal{R}_q[a]s] + 0 \\ &= \mathcal{R}_q[\mathcal{R}_q[a]s] \end{aligned}$$

Since $\mathcal{R}_q[1] = \mathcal{R}_q[\mathcal{R}_q[a] \cdot s]$, then $\mathcal{R}_q[a] \cdot s = 1$ and thus $s^{-1} = \mathcal{R}_q[a]$. In the same way we can find an inverse element for every element of the *finite field*. □

Finite field of polynomials

Before defining the *finite field* of polynomials, it is worth completing the definition of a *ring* of polynomials beginned in section (2.2.2): it is possible to create a *ring* of this kind by using the following proceeding:

1. choose a *finite field* of integers $\mathcal{C}$ and a polynomial $p(x)$ with coefficients in $\mathcal{C}$ whose degree is an arbitrary integer $n > 0$ (for $n = 0$ we would still get $\mathcal{C}$)

2. consider the set of polynomials with coefficients in the *finite field* $\mathcal{C}$ with degree less than n

3. define the operation *addition* as the usual addition between polynomials (remember that coefficients now belongs to a finite field, so their sum gives as result the sum modulo q)

4. define the operation *multiplication* as the reminder modulo $p(x)$ of the polynomial product, that is: $\forall a(x), b(x) \in \mathcal{C}[x]$, set of the polynomials with coefficients in $\mathcal{C}$, $a(x) \cdot b(x) = \mathcal{R}_{p(x)}[a(x) \cdot b(x)]$, where $\mathcal{R}_{p(x)}[f(x)]$ denotes the reminder polynomial of the division of $f(x)$ by $p(x)$.

It can be shown that the set thus obtained is a finite ring, but we will omit here the proof. It is also interesting to note that the numbers of element in this ring is q^n, because for each of the n coefficients corresponding to a power of the variable x (from $x^0 = 1$ to x^{n-1}) we can choose one of the q elements in the finite field $\mathcal{C}$.

Theorem 3. *$GF(q)[x]/f(x)$ is a* finite field $\Leftrightarrow$ *$p(x)$ is an irreducible polynomial in $\mathcal{C} = GF(q)$.*

Proof. $\Rightarrow$) Let's proceed by contradiciton: $p(x)$ is not irreducible. Then there exists $a(x), b(x) \in GF(q)[x]$ such that $p(x) = a(x)b(x)$, with $deg[a(x)] < deg[p(x)]$ and $deg[b(x)] < deg[p(x)]$. Since $GF(q)[x]$ is a *field* for hypotheses, every element has got an inverse: $\exists\ a^{-1}(x) \mid a(x)a^{-1}(x) = 1$, then $(a(x)a^{-1}(x))b(x) = 1 \cdot b(x) = b(x)$. Therefore:

$$\begin{aligned} a^{-1}(x)(a(x)b(x)) &= \mathcal{R}_{p(x)}[a^{-1}(x)\mathcal{R}_{p(x)}[a(x)b(x)]] \\ &= \mathcal{R}_{p(x)}[a^{-1}(x)\mathcal{R}_{p(x)}[p(x)]] \\ &= \mathcal{R}_{p(x)}[a^{-1}(x) \cdot 0] = \mathcal{R}_{p(x)}[0] = 0 \end{aligned}$$

Then we would have: $(a(x)a^{-1}(x))b(x) = b(x) = 0$, but this is impossible, because $p(x) = a(x)b(x) = a(x) \cdot 0$ cannot be a zero element, otherwise the chosen polynomial $p(x)$ would be an integer, breaking the rule saying that

its degree must be $n > 0$. Therefore, $p(x)$ must be irreducible.

$\Leftarrow$) The purpose is now to prove that, if $p(x)$ is irreducible, then every element has got an inverse. Given two polynomials $s(x)$ and $t(x)$, it is always possible to find a couple of polynomials $a(x)$ and $b(x)$ such that $MCD[s(x), t(x)] = a(x)s(x)+b(x)t(x)$ [16, p. 91]. Then, since $MCD[s(x), q(x)] = 1$ for any $s(x) \in GF(q)[x]$:

$$\begin{aligned}
1 &= \mathcal{R}_{p(x)}[1] \\
&= \mathcal{R}_{p(x)}[a(x)s(x) + b(x)q(x)] = \mathcal{R}_{p(x)}[a(x)s(x)] + \mathcal{R}_{p(x)}[b(x)p(x)] \\
&= \mathcal{R}_{p(x)}[\mathcal{R}_{p(x)}[a(x)]\mathcal{R}_{p(x)}[s(x)]] + \mathcal{R}_{p(x)}[\mathcal{R}_{p(x)}[b(x)]\mathcal{R}_{p(x)}[p(x)]] \\
&= \mathcal{R}_{p(x)}[\mathcal{R}_{p(x)}[a(x)]s(x)] + \mathcal{R}_{p(x)}[\mathcal{R}_{p(x)}[b(x)] \cdot 0] \\
&= \mathcal{R}_{p(x)}[\mathcal{R}_{p(x)}[a(x)]s(x)] + 0 \\
&= \mathcal{R}_{p(x)}[\mathcal{R}_{p(x)}[a(x)]s(x)]
\end{aligned}$$

Since $\mathcal{R}_{p(x)}[1] = \mathcal{R}_{p(x)}[\mathcal{R}_{p(x)}[a(x)]s(x)]$, then $\mathcal{R}_{p(x)}[a(x)]s(x) = 1$ and thus $s^{-1}(x) = \mathcal{R}_{p(x)}[a(x)]$. In the same way we can find an inverse element for every element of the *finite field*. □

It is now possible to build any *Galois Field* of cardinality $q = p^n$ just by creating a *finite field* of polynomials in $\mathcal{C}$ with degree less than n and considering each of them as an element of the new *finite field*. For example, since 4 is not prime, we can not build the *finite field* of cardinality 4 just using the modulo 4 operations (2 would not have an inverse element with respect to the operation "$\cdot$"). Instead, we can build a field $GF(4) = GF(2^2)$ using $x^2 + x + 1$ as prime polynomial in $GF(2) = \{0, 1\}$.

2.4.2 Properties of Galois Fields

Thanks to their special properties, *Galois Fields* are a very powerful mathematical tool and thay are the foundation of *Reed-Solomon codes* theory. In this section we introduce an overview of all their most useful characteristics.

Fermat Theorem

Theorem 4 (Fermat Theorem)**.** *Let $\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_{q-1}$ be non-zero elements of a* finite field $\mathcal{C}$ *with order q* $\Rightarrow$

$$x^{q-1} = \prod_{i=1}^{q-1}(x - \alpha_i)$$

Proof. From the definition of *finite field*, non-zero elements form a finite multiplicative group of cardinality $q-1$. For Theorem 1 every element of the multiplicative group has order which divides the maximum order, $q-1$. Let's consider a generic element $\alpha_l \in \mathcal{C}_{\tilde{0}}$ with order h:

$$\alpha_l^{q-1} = (\alpha_l^h)^{\frac{q-1}{h}} = 1^{\frac{q-1}{h}} = 1$$

Since $\alpha_l^{q-1} = 1$ we deduce that every $\alpha_l \in \mathcal{C}_{\tilde{0}}$ is a root of the polynomial $x^{q-1} - 1$.

On the other hand, since $x^{q-1} - 1$ has got $q-1$ factors $(x - \alpha_l)$ and there are exactly $q-1$ different non-zero element α_l of $\mathcal{C}$, these are all and the only roots of $x^{q-1} - 1$:

$$x^{q-1} = \prod_{i=1}^{q-1}(x - \alpha_l) \quad \text{with } \alpha_l \in \mathcal{C}_{\tilde{0}}$$

□

Primitive Element and Primitive Polynomial

A *primitive element* α of a *finite field* $\mathcal{C}$ with order q is an element with multiplicative order $q-1$, that is its multiplicative powers cover all the non-zero elements of the *Galois Field*. Powers of α provide a very comfortble way to point an element of the $finitefield$ and in the following we will often refer to an element in $GF(q)$ with the relative power of the *primitive element*.

Theorem 5. *Every* finite field *has got at least one* primitive element.

Proof. For Theorem 1 the order h of any of the non-zero elements of $\mathcal{C}$, α_l, divides the maximum order, let's say m. Since $m = h \cdot p$ for any integer p:

$$\alpha_l^m = \alpha_l^{h \cdot p} = (\alpha_l^h)^p = (1)^p = 1$$

Therefore: $\alpha_l^m = 1$, that is any α_l is a root of $x^m - 1$. Since there are $q-1$ non-zero elements of $\mathcal{C}$, $x^m - 1$ must have at least $q-1$ roots, that is $m \geq q-1$. On the other hand, since any of the element order divides the group order, $m \leq q-1$. These two implications bring to $m = q-1$. Then, every *finite field* has got an element with order $q-1$, that is a *primitive element*. □

From this theorem it follows that every *Galois Field* deprived by the zero constitutes a *cyclic multiplicative group*.

A *primitive polynomial* $f(x)$ is a prime polynomial (that is irreducible and monic) such that $GF(q)[x]/f(x)$, that is extension modulo $f(x)$ of any

Galois Field $GF(q)$, has got polynomial x as primitive element. If we build a *Galois Field* by using a *primitive polynomial*, we can refer to every element of the field $GF(q^n)$, with n degree of $f(x)$, as a power of x. Since we do not treat here how to find *primitive polynomials*, we will take them from pre-compiled tables.

Theorem 6. *$\forall p$ prime integer and $\forall n \in \mathbb{N}\ \exists\ f(x)$ primitive polynomial with degree n on $GF(p)$.*

Minimal Polynomial

Let $\mathcal{C}_0$ be a *Galois field*, $GF(p)$, let $\mathcal{C}$ be an extension of $GF(p)$, $GF(p^n)$, and let $\alpha \neq 0$ be any element in $\mathcal{C}$. The monic polynomial whose coefficients are taken from $\mathcal{C}_0$, which have the least degree and such that it vanishes when evaluated in α, is called *minimal polynomial* of α on $\mathcal{C}_0$ and it is expressed by notation: $f_\alpha(x)$.

Theorem 7. *Let $\mathcal{C}$ be a* finite field *extension of the* finite field *$\mathcal{C}_0$ and let be $\alpha \in \mathcal{C}$, $\alpha \neq 0 \Rightarrow \exists!\ f_\alpha(x)$, minimal polynomial of α on $\mathcal{C}_0$.*

Proof. First of all we can prove that for every element of a *finite field* with cardinality q there exists at least one polynomial with coefficients in $\mathcal{C}_0$ which vanishes if evaluated in it. Let's consider any element $\alpha \in \mathcal{C}$. For *Fermat Theorem* 4, α is a root of $x^{q-1} - 1$, which has got coefficients in $\mathcal{C}_0$. Thus, we are sure that a polynomial of this kind exists, because element 1 and -1 must belong to every field.

Let's prove that $f_\alpha(x)$ is irreducible by contradiction: if we suppose $f_\alpha(x)$ is not irreducible, then $f_\alpha(x) = s(x)t(x)$ with $deg[s(x)] \geq 1$ and $deg[t(x)] \geq 1$. Since $f_\alpha(\alpha) = 0$, then necesserily $s(\alpha) = 0$ or $t(\alpha) = 0$. In this case $f_\alpha(x)$ would not be the polynomial with coefficients in $\mathcal{C}_0$ which vanishes in α with the least degree, that is in contradiction with the hypotheses. Therefore, $f_\alpha(x)$ must be irreducible.

Eventually, we have to demonstrate that $f_\alpha(x)$ is the only *minimal polynomial*. Suppose there is a polynomial $g(x)$ such that $g(x) = q(x)f_\alpha(x)+r(x)$ with $deg[r(x)] < deg[f_\alpha(x)]$. If $g(\alpha) = 0$, that is α is a root of $g(x)$, then it must be $g(\alpha) = q(\alpha)f_\alpha(\alpha) + r(\alpha) = q(\alpha) \cdot 0 + r(\alpha) = r(\alpha) = 0$. Since a polynomial with smaller degree than $f_\alpha(x)$ which vanishes in α cannot exist, $r(x)$ must be the null polynomial: $r(x) = 0$. Therefore: $g(x) = q(x)f_\alpha(x)$ and $g(x)$ is a multiple of $f_\alpha(x)$.

If we suppose $g(x)$ is monic and with the same degree than $f_\alpha(x)$, that is if we assume there are two *minimal polynomials*, then $q(x)$ must be $q(x) = 1$.

This implies that $g(x)$ and $f_\alpha(x)$ are the same polynomial. In other words $f_\alpha(x)$ is the only *minimal polynomial*. □

Numerosity and existance of Galois Fields

Theorem 8. *The number of elements of any* Galois field *can be expressed in the form* $q = p^n$ *with* p *prime integer and* $n > 0$, $n \in \mathbb{N}$.

Proof. Let $f_\alpha(x) = x^n + f_{n-1}x^{n-1} + \ldots + f_1 x + f_0$ be the *minimal polynomial* for α with coefficients in $GF(p)$, where α belongs to the extension $\mathcal{C}$ of $\mathcal{C}_0 = GF(p)$. This means that

$$f_\alpha(\alpha) = \alpha^n + f_{n-1}\alpha^{n-1} + \ldots + f_1\alpha + f_0 = 0$$

and then

$$\begin{aligned}\alpha^n &= -(f_{n-1}\alpha^{n-1} + \ldots + f_1\alpha + f_0) \\ &= -\sum_{i=0}^{n-1} f_i\alpha^i \quad \text{with } f_i \in \mathcal{C}_0\end{aligned}$$

We have, thus, found a representation of the element α^n through elements that belong to $\mathcal{C}_0$. This can be done for every power of α as well, as a matter of fact:

$$\begin{aligned}\alpha^{n+1} &= \left(-\sum_{i=0}^{n-1} f_i\alpha^i\right)\cdot\alpha \\ &= -\sum_{i=0}^{n-1} f_i\alpha^{i+1} \\ &= -f_{n-1}\alpha^n - \sum_{i=0}^{n-2} f_i\alpha^{i+1} \\ &= -f_{n-1}\left(-\sum_{i=0}^{n-1} f_i\alpha^i\right) - \sum_{i=0}^{n-2} f_i\alpha^{i+1}\end{aligned}$$

If we choose α a *primitive element*, which always exists in *finite fields* (see Theorem 6), then we can write every element of $\mathcal{C}$ through coefficients in $\mathcal{C}_0$ just by iterating the shown proceeding.

This notation with elements in $\mathcal{C}_0$ is unique for every element of $\mathcal{C}$; if there were two representations of α:

$$s_{n-1}\alpha^{n-1} + s_{n-2}\alpha^{n-2} + \ldots + s_1\alpha + s_0 = t_{n-1}\alpha^{n-1} + t_{n-2}\alpha^{n-2} + \ldots + t_1\alpha + t_0$$

then we could subtract them, getting:

$$(s_{n-1} - t_{n-1})\alpha^{n-1} + (s_{n-2} - t_{n-2})\alpha^{n-2} + \ldots + (s_1 - t_1)\alpha + (s_0 - t_0) = 0$$

which would be a polynomial with smaller degree than n vanishing in α; but this is impossible because the *minimal polynomial* has degree n for hypotheses.

We can represent p^n elements with this notation, because we can choose the n coefficients of the polynomial with maximum degree $n-1$ in p different ways among p elements of $\mathcal{C}_0 = GF(p)$. Thanks to the linearity of *fields*, every linear combination of elements in $\mathcal{C}$ must give as result still an element in $\mathcal{C}$. This guarantees that the p^n elements written through the exposed notation must all belong to $\mathcal{C}$. Therefore, any *Galois Field* has p^n elements, with p a certain prime integer an n a positive value. □

Theorem 8 has got a great relevance for *finite fields* theory, because it tells us that we can build every *Galois Field* just by choosing a prime integer p and any prime polynomial with degree n and coefficients in $GF(p)$ and evaluate $GF(q) = GF(p^n) = GF(p)[x]/f(x)$. It is possible to prove that, given a *finite field* $GF(p)$ and a positive integer n, it is always possible to find a prime polynomial of degree n with coefficients in $GF(p)$ [16, p. 109-110].

2.5 Operations in the Galois Fields

Once we know what is and how to build a *finite field*, we would like to use it for executing some operations between its elements. The possible operations on a *finite field* are the fundamental four operations: *addition*, *subtraction*, *multiplication* and *division*. In the next following we will examine each of them, observing how thay can be computed in the easiest way.

Suppose we have a *Galois Field* $GF(q) = GF(p^n)$ where p is prime and n is any positive integer. For what has been said in the previous section by Theorem 8, every *finite field* can be expressed in this way. Moreover, if $n = 1$ we simply have the set of the first n integers and the operations are defined as operations modulo p, otherwise, if $n > 1$, we can associate to every element of $GF(p^n)$ a polynomial in $GF(p)[x]/f(x)$, where $f(x)$ is a primitive polynomial of degree n. It is much more convenient to use the polynomial notation when $n > 1$.

2.5.1 Addition

If $n = 1$ the addition is easily defined as addition modulo p (see section (2.2.1) for its formal description).

If $n > 1$, we can replace the two elements to sum, let's say c_1 and c_2, with their relative polynomial representations with coefficients in $GF(p)$:

$$c_1 \to a_0 + a_1 x + ... + a_{n-1} x^{n-1}$$
$$c_2 \to b_0 + b_1 x + ... + b_{n-1} x^{n-1}$$

and sum the two polynomials. This operation is now very easy to execute, because we can simply sum the corresponding couples of coefficients relative to each power and, since p is prime and the coefficients belong to $GF(p)$, the addition is made modulo p. Once we have the resulting polynomial, we have to convert it to its corresponding element of $GF(p^n)$. To speed up this operation we can make use of a precompiled table which associates to every element its polynomial form.

2.5.2 Subtraction

Subtraction is very similar to addition and we can proceed in the same way: operation modulo p if $n = 1$ and polynomial subtraction if $n > 1$.

2.5.3 Multiplication

If $n = 1$ the multiplication is defined as multiplication modulo p (see section (2.2.1) for its formal description).

If $n > 1$, it is comfortable to take advantage of the properties of primitive elements: if we consider every non-zero element of the *finite field* as a power of the primitive element α, multiplication between two elements reduces to the sum of their exponents:

$$\alpha^i \cdot \alpha^j = \alpha^{i+j}$$

If $i + j$ is greater than $p^n - 1$, we can reduce it to a lower power remembering that $\alpha^{p^n - 1} = 1$ and therefore

$$\alpha^{i+j} = \alpha^{k+(p^n-1)} = \alpha^k \cdot \alpha^{p^n-1} = \alpha^k \cdot 1 = \alpha^k$$

Also in this case, as for addition and subtraction, it is useful to have a table associating every element of $GF(p^n)$ to its logarithmic representation to quickly compute multiplication between two elements.

2.5.4 Division

Division is very similar to multiplication and we can proceed in the same way: operation modulo p if $n = 1$ and difference between the esponents of the logarithmic form if $n > 1$.

Chapter 3

Reed-Solomon Codes

Reed-Solomon codes can be seen as a special subclass of *BCH codes*. In order to define what a *Reed-Solomon code* is, following a different approach from the polynomial one exposed in section (1.3), it is first necessary to introduce more general types of codes for channel coding, each a particular subclass of another:

block codes
↑
linear block codes
↑
cyclic codes
↑
BCH codes
↑
Reed-Solomon codes

3.1 Block codes

Given a finite alphabet $\mathcal{A}$ of q symbols, a *block code* is a function that assigns to a k-symbol word an n-symbol word. A stream of symbols belonging to the alphabet $\mathcal{A}$ arrives to the encoder, which "cuts" it in blocks of k symbols each that are uniquely mapped in a set of words of n symbols. It is easy to see that there are q^k possible words which are uniquely mapped in one of the q^n possible code words.

3.2 Linear block codes

A *block code* is *linear* if its words are a vectorial subspace of the vectorial space made by all the n-ples whose elements are in the *finite field* $GF(q)$: $\mathcal{V}_n^{(q)} \doteq (GF(q))^n$ [16, p. 31]; addition between two code words always gives as result a code word as well. This property is very useful in the decoding process, especially decoding by *syndrome*. Thanks to linearity, a *linear block code* is uniquely identified by a *generator matrix* $\mathcal{G}$.

A code is said to be *systematic* if the n-symbol code word $\mathbf{c}$ is made up by adding $n-k$ redundancy symbols to the k-symbol input word $\mathbf{u}$:

$$\mathbf{u} = [u_0\ u_1\ \ldots\ u_{k-1}] \quad \rightarrow \quad \mathbf{c} = [c_0\ c_1\ \ldots\ c_{n-k-1}\ u_0\ u_1\ \ldots\ u_{k-1}]$$

Systematic codes are comfortable because they allow to know the original input word just by observing the resulting code word. As one can suppose, however, channel noise corruption does not permit such an easy decoding.

For *linear block codes* we can define some useful and important parameters we will often use in the following (we will refer to a generic word with the vector $\mathbf{v}$).

- **Hamming weigth** $w(\mathbf{v})$: number of non-zero components in the word $\mathbf{v}$.

  For binary codes *Hamming weigth* also represents the number of positions in the code word filled by "1".

- **Hamming distance** $d(\mathbf{v}, \mathbf{u})$: number of positions at which the two words contain different symbols:

  $$d(\mathbf{v}, \mathbf{u}) = \sum_{i=0}^{l} x_i \quad \text{where } x_i = \begin{cases} 1 & \text{if } v_i \neq u_i \\ 0 & \text{if } v_i = u_i \end{cases}$$

  and l is the vector length of $\mathbf{v}$ and $\mathbf{u}$.

- **minimum Hamming weigth of a code** $\mathcal{C}$: $w_{min} \doteq min\{w(\mathbf{v})\}$ with $\mathbf{v} \in \mathcal{C}$, $\mathbf{v} \neq 0$.

- **minimum Hamming distance of a code** $\mathcal{C}$: $d_{min} \doteq min\{d(\mathbf{v}, \mathbf{u})\}$ with $\mathbf{v}, \mathbf{u} \in \mathcal{C}$, $\mathbf{v} \neq \mathbf{u}$.

Theorem 9. *$\mathcal{C}$ is a binary* linear block code $\Rightarrow d_{min} = w_{min}$.

Proof. Since $\mathcal{C}$ is binary $d(\mathbf{v}, \mathbf{u}) = w(\mathbf{v} - \mathbf{u})$ because the difference between two binary vectors, equal to their addition, is made by a XOR comparison. This means that $\mathbf{v} - \mathbf{u}$ has a "0" in the positions where the two vectors have equal symbols and "1" in the positions where the two vectors have different symbols. Therefore the vector $\mathbf{v} - \mathbf{u}$ has got as many "1" as the number of positions at which the two original vectors differ each others. □

Theorem 10 (Singleton bound)**.** $\mathcal{C}$ *is a* linear block code $\Rightarrow d_{min} \leq n-k+1$

Proof. Since $d_{min} = w_{min}$, we can demonstrate the sentence for the lowest weigth code word.

It can be proved that every code has an equivalent *systematic* code [16, p. 34][?, p. 50], that is every code can be transformed into a systematic code thorugh elementary algebraic operations on the *generator matrix* without changing any of its carachteristics (we will not investigate this property in more detail). Let's consider the systematic code equivalent to the code $\mathcal{C}$. A codeword of this code is made by an input word preceded (or followed) by $n - k$ redundancy symbols. Among the input words there is certainly a vector in the form: [0 0 ... 0 1]. As a matter of fact the input dictionary includes all the combinations of $q = 2$ symbols in k positions. Therefore the minimum weight word is formed in this way:

$$[x\ x\ \ldots\ x\ 0\ 0\ \ldots\ 0\ 1]$$

where x can be either "0" or "1". In the worst case (highest weigth) each of the $n - k$ leftmost symbols (x) are all "1". The minimum number of "1" in a code word is thus over limited by $1 + (n - k)$. This means that $d_{min} = w_{min} < n - k + 1$. □

3.2.1 Error detection

A code $\mathcal{C}$ *detects* an error when it recognises that some symbols of the received word have been corrupted by the channel. This is easy to do thanks to redundancy: valid code words are only q^k of the q^n possible words of the vectorial space $\mathcal{V}_n^{(q)} \doteq GF^n(q)$. If the received code word does not belong to the code dictionary, then some errors must have changed the original (valid) word. Remember that some very unlucky cases may happen: if the noisy channel modifies the sent codeword so that to change it into another valid codeword, the code will not be able to detect any error. In order to avoid this eventuality, we must ensure a certain difference between the q^n code words, which increases with the amount of reduncancy: the more symbols we add

the less probably a corrupted code word will become exactly another valid word.

A *linear block code* is able to *properly* detect an error only if the number of corrupted symbols t is smaller than d_{min}. For greater weight error vectors we have no warranty of correctly decoding the received word, since symbol errors may be so many as to transform the word into another valid code word. Furthermore there is no way to know the error vector: we can only act on the code in order to make probability of having $t \geq d_{min}$ errors very remote. The only precaution we can take is trying to make this kind of events very unlikely, or, equivalently, to require a minumum Hamming distance as great as possible. Once a wrong codeword is detected, the receiver can ask the sender for a re-transmission.

3.2.2 Error correction

A code $\mathcal{C}$ *corrects* an error when it recognises that some symbols of the received word have been corrupted by the channel, but it manages to guess the original sent word. If the input words are all equally likely, the reasoning is still very naive: if the received word differs from a word $\mathbf{u}$ for a number of symbols smaller than from any other word, then it is more likely that $\mathbf{u}$ has been sent. Of course we are assuming that the system is "well-done", that is a small number of errors is more probable than a great amount of distortion.

Any word $\mathbf{v}$ in $GF^n(q)$ not belonging to $\mathcal{C}$ is nearer, in terms of Hamming distance, to one or more code words than to others. To be sure to properly interpret the sent word we must have the assurance that the decoded code word $\mathbf{c}$ is nearer to received word $\mathbf{v}$ than any other code word. This is trivially true if the distance between the two words is less than $\frac{d_{min}}{2}$. As a matter of fact if t symbols, with $t \geq \frac{d_{min}}{2}$, are different from the choosen codeword, there might be another code word with less distance; while, if $t < \frac{d_{min}}{2}$ we are sure that $d(\mathbf{v}, \mathbf{c})$ is the least possible distance. The following theorem formalises that, being $\mathbf{v}$ the received word, if $\mathbf{v}$ differs from the code word $\mathbf{c}$ for less than $\frac{d_{min}}{2}$ symbols, then $\mathbf{v}$ is nearer to $\mathbf{c}$ than to any other code word.

Theorem 11. *$d(\mathbf{v}, \mathbf{c}) < \lfloor \frac{d_{min}}{2} \rfloor$ and $d(\mathbf{w}, \mathbf{c}) \geq d_{min} \Rightarrow d(\mathbf{v}, \mathbf{c}) < d(\mathbf{v}, \mathbf{w})$.*

Proof. Because of the tringular inequality for distances we have:

$$d(\mathbf{w}, \mathbf{c}) \leq d(\mathbf{w}, \mathbf{v}) + d(\mathbf{v}, \mathbf{c})$$

and therefore:

$$d(\mathbf{w}, \mathbf{c}) - d(\mathbf{v}, \mathbf{c}) \leq d(\mathbf{w}, \mathbf{v})$$

In the worst case, when d_{min} is even and thus $\lfloor \frac{d_{min}}{2} \rfloor$ is as great as possible:

$$d(\mathbf{w}, \mathbf{c}) - d(\mathbf{v}, \mathbf{c}) = d_{min} - \left(\frac{d_{min}}{2} - 1 \right) = \frac{d_{min}}{2} + 1$$

and, in general:

$$d(\mathbf{w}, \mathbf{c}) - d(\mathbf{v}, \mathbf{c}) \geq \frac{d_{min}}{2} + 1$$

It follows that:

$$d(\mathbf{w}, \mathbf{v}) \geq d(\mathbf{w}, \mathbf{c}) - d(\mathbf{v}, \mathbf{c}) \geq \frac{d_{min}}{2} + 1 > \lfloor \frac{d_{min}}{2} \rfloor > d(\mathbf{v}, \mathbf{c})$$

In conclusion:

$$d(\mathbf{v}, \mathbf{c}) < d(\mathbf{w}, \mathbf{v}) \quad \forall\ \mathbf{w} \neq \mathbf{v}$$

□

A *linear block code* is able to *properly* correct an error only if the number of corrupted symbols t is smaller than $\lfloor \frac{d_{min}}{2} \rfloor$.

3.3 Cyclic codes

Cy clic codes are *linear block codes* with the property that, given a code word, every vector obtained by shifting its components on the left or on the right of any number of positions is still a code word. That is:

$$[c_0\ c_1\ \ldots\ c_{n-1}] \in \mathcal{C}\ \Rightarrow\ [c_l\ c_{l+1}\ \ldots\ c_{n-1}\ c_0\ c_1\ \ldots c_{l-1}] \in \mathcal{C}$$

Let's consider the polynomial ring defined as $GF(q)[x]/(x^n - 1)$. In this set, multiplying by x corresponds to shifting the coefficients to the right of one position:

$$v(x) \in GF(q)[x]/(x^n - 1)$$

$$\begin{aligned}
x \cdot v(x) &= \mathcal{R}_{x^n-1}[x \cdot v(x)] \\
&= \mathcal{R}_{x^n-1}[x \cdot (v_0 + v_1 x + \ldots + v_{n-1}x^{n-1})] \\
&= \mathcal{R}_{x^n-1}[v_0 x + v_1 x^2 + \ldots + v_{n-1}x^n] \\
&= \mathcal{R}_{x^n-1}[v_0 x + v_1 x^2 + \ldots + v_{n-1}x^n - v_{n-1} + v_{n-1}] \\
&= \mathcal{R}_{x^n-1}[v_0 x + v_1 x^2 + \ldots + v_{n-1}(x^n - 1) + v_{n-1}] \\
&= \mathcal{R}_{x^n-1}[v_{n-1} + v_0 x + v_1 x^2 + \ldots + v_{n-2}x^{n-1} + v_{n-1}(x^n - 1)] \\
&= v_{n-1} + v_0 x + v_1 x^2 + \ldots + v_{n-2}x^{n-1}
\end{aligned}$$

We can obtain a *cyclic code* by considering a subset of polynomials in $GF(q)[x]/(x^n - 1)$: if we choose one or more polynomials in this set and we also take every polynomial got by shifting the original one, the vectors of their coefficients represent a *cyclic code*. Furthermore it is worth noting that we could associate any of the possible code words to an element of the *Galois Field* $GF(q^n)$.

It is possible to prove that codes of this type are totally identified by a generator polynomial $g(x)$. Every code word is obtained by multiplying the input word for the generator polynomial, but, before talking about it, we first need to introduce some basic results.

Theorem 12. *$\mathcal{C}$ is a* cyclic code $\Rightarrow \exists!\ g(x) \in \mathcal{C}$*, monic polynomial of least degree with coefficient of $x^0 = 1$: $g_0 \neq 0$.*

Proof. We will prove it by contradiction, assuming there are two monic polynomials $g(x)$ and $f(x)$ both with minimum degree m:

$$\begin{aligned} g(x) &= g_0 + g_1 x + \ldots + g_{m-1}x^{m-1} + x^m \\ f(x) &= f_0 + f_1 x + \ldots + f_{m-1}x^{m-1} + x^m \end{aligned}$$

Then, since *cyclic codes* are also *linear codes* by definition, $g(x) - f(x)$ must belong to $\mathcal{C}$:

$$\begin{aligned} g(x) - f(x) &= (g_0 + g_1 x + \ldots + x^m) - (f_0 + f_1 x + \ldots + x^m) \\ &= (g_0 - f_0) + (g_1 - f_1)x + \ldots (g_{m-1} - f_{m-1})x^{m-1} \end{aligned}$$

which has got degree less than m. The contradiction lies in the fact that m is the minimum degree, then $g(x)$ must be the only minimum degree monic polynomial in $\mathcal{C}$.

To prove that $g_0 \neq 0$ let's still proceed by contradiction, imposing $g_0 = 0$:

$$\begin{aligned} g(x) &= g_0 + g_1 x + \ldots + g_{m-1}x^{m-1} + x^m \\ &= g_1 x + \ldots + g_{m-1}x^{m-1} + x^m \\ &= x(g_1 + g_2 x + \ldots + g_{m-1}x^{m-2} + x^{m-1}) \\ &= x \cdot \tilde{g}(x) \end{aligned}$$

Multiplying $g(x)$ by x^{n-1} gives as result another code polynomial, in particular:

$$\begin{aligned} x^{n-1} \cdot g(x) &= \mathcal{R}_{x^n-1}[x^{n-1}g(x)] \\ &= \mathcal{R}_{x^n-1}[x^{n-1}x\tilde{g}(x)] = \mathcal{R}_{x^n-1}[x^n\tilde{g}(x)] \\ &= \mathcal{R}_{x^n-1}[x^n(g_1 x + \ldots + g_{m-1}x^{m-1})] \end{aligned}$$

Multiplying $\tilde{g}(x)$ by x^n means shifting its coefficients of n positions, that is equivalent to make an entire circle arriving to the initial configuration. Then:

$$x^{n-1} \cdot g(x) = g_1 x + \ldots + g_{m-1} x^{m-1}$$

which has got degree lower than m, hence the contradiction: g_0 must not be null. □

Theorem 13. *$\mathcal{C} = GF(q^n)$ is a* cyclic code *and $g(x)$ is its monic polynomial of least degree with $g_0 \neq 0$, then $v(x)$ on $GF(q)$ with degree smaller than n is a polynomial code $\Leftrightarrow$ $v(x)$ is a multiple of $g(x)$.*

Proof. $\Leftarrow$) The generic polynomial $p(x)$ can be seen as a linear combination of powers of x; then, multiplying $p(x)g(x)$ means shifting $g(x)$ in any way and sum all these different new words; the resulting word still belongs to $\mathcal{C}$ because it is linear. Thus if $v(x) = p(x)g(x)$ is a multiple of $g(x)$, it is also a code polynomial.

$\Rightarrow$) Let's write $v(x)$ as it follows:

$$v(x) = q(x)g(x) + r(x)$$

where $deg[r(x)] < deg[q(x)]$ and $deg[r(x)] < deg[g(x)]$. Then:

$$v(x) - q(x)g(x) = r(x)$$

Since $\mathcal{C}$ is also a *linear block code* and $q(x)g(x)$ belongs to $\mathcal{C}$, as we proved in the first part of this demonstration, $v(x) - q(x)g(x)$ must still belong to $\mathcal{C}$ and thus $r(x)$ must belong to $\mathcal{C}$ too. We know that $deg[r(x)] < deg[g(x)]$, but, since $g(x)$ is the minimum degree polynomial, $r(x)$ can only be the null polynomial: 0. Thus:

$$r(x) = 0 \Rightarrow v(x) = q(x)g(x)$$

that is $v(x)$ is a multiple of $g(x)$. □

The polynomial $g(x)$ introduced in Theorems 12 and 13 is the *generator polynomial* of $\mathcal{C}$: it uniquely identifies the code and through it we can build the whole $\mathcal{C}$. Let's see other interesting properties of $g(x)$.

Theorem 14. *$g(x)$ is the least degree monic polynomial with $g_0 \neq 0$ of a* cyclic code *$\mathcal{C}$ with length n, then it is the generator polynomial of $\mathcal{C}$ $\Leftrightarrow$ $g(x)$ divides $x^n - 1$.*

Proof. $\Rightarrow$) It certainly exists a way to write $x^n - 1$ as it follows:

$$x^n - 1 = q(x)g(x) + r(x)$$

where $deg[r(x)] < deg[q(x)]$ and $deg[r(x)] < deg[g(x)]$. Therefore:

$$\begin{aligned} \mathcal{R}_{x^n-1}[x^n - 1] &= \mathcal{R}_{x^n-1}[q(x)g(x) + r(x)] \\ 0 &= \mathcal{R}_{x^n-1}[q(x)g(x)] + \mathcal{R}_{x^n-1}[r(x)] \end{aligned}$$

and hence:

$$\mathcal{R}_{x^n-1}[q(x)g(x)] = -r(x)$$

$\mathcal{R}_{x^n-1}[q(x)g(x)]$ is deinitely a code word because $g(x)$ is the generator polynomial, then $r(x)$ must be a code polynomial as well; however, since $deg[r(x)] < deg[g(x)]$, it must be: $r(x) = 0$, and then $x^n - 1 = q(x)g(x)$. In other words, $g(x)$ divides $x^n - 1$.

$\Leftarrow$) Let's choose a code polynomial: $v(x) = v_0 + v_1x + \ldots + v_{n-1}x^{n-1}$. By definition this polynomial is divisible by $g(x)$. Let's shift $v(x)$ of one position to the right by multiplying it for x:

$$\begin{aligned} x \cdot v(x) &= x(v_0 + v_1x + \ldots + v_{n-1}x^{n-1}) \\ &= v_0x + v_1x^2 + \ldots + v_{n-1}x^n \\ &= v_0x + v_1x^2 + \ldots + v_{n-1}x^n - v_{n-1} + v_{n-1} \\ &= v_{n-1} + v_0x + v_1x^2 + \ldots + v_{n-1}(x^n - 1) \\ &= (v_{n-1} + v_0x + v_1x^2 + \ldots + v_{n-2}x^{n-1}) + v_{n-1}(x^n - 1) \\ &= \tilde{v}(x) + v_{n-1}(x^n - 1) \end{aligned}$$

It follows that:

$$\begin{aligned} \tilde{v}(x) &= xv(x) - v_{n-1}(x^n - 1) \\ \tilde{v}(x) &= xa(x)g(x) - v_{n-1}b(x)g(x) \\ \tilde{v}(x) &= [xa(x) - v_{n-1}b(x)]g(x) \end{aligned}$$

Last equalities hold because, being $v(x)$ a code polynomial, it is divisible by $g(x)$ and $g(x)$ divides x^n-1 for hypotheses. Then $\tilde{v}(x)$, which was obtained by shifting a code word, is still a code polynomial as it is a multiple of $g(x)$. This means that $g(x)$ is a generator polynomial. □

Since $g(x)$ has got degree m, with $m < n$, we can multiply for $g(x)$ only polynomials with degree less than or equal to $n-m-1$ in order to get a code polynomial with degree less than n. Input polynomials have to be formed by

up to $n-m$ coefficients and therefore we have q^{n-m} valid input polynomials; this is the number of vectors we can make up by filling $n-m$ positions with q elements. Notice that, for larger degrees, we will not obtain further resulting polynomials, thanks to multiplication modulo (x^n-1).

It is clear that, in order to build a *cyclic code*, we have first to find a proper generator polynomial. Theorem 14 tells us that, if we need a n-symbol code, we should look for $g(x)$ among polynomials on $GF(q)$ which divide x^n-1. Let's define the set of prime polynomials which are divisors of x^n-1:

$$\mathcal{F} = \{f_1(x), f_2(x), \ldots, f_M(x)\}$$

such that:

$$x^n - 1 = \prod_{l=1}^{M} f_l(x) \tag{3.1}$$

The product of every subset of polynomials taken from $\mathcal{F}$ forms a possible generator polynomial: we could build 2^M-2 non-banal generator polynomials (excluding "0" and x^n-1 itself).

3.3.1 Primitive cyclic codes

Let's limit our range of research: we will be interested only on codes whose symbols are taken from $GF(q)$ and whose length is $n = q^m-1$ for some $m \in \mathbb{N}$. This kind of codes are called *primitive cyclic codes*. This restriction brings to the following result, which will reveal to be very comfortable:

$$x^n - 1 = x^{q^m-1} - 1$$

Let's consider the *finite field* $GF(q^m)$, extension of $GF(q)$. It follows from *Fermat Theorem* 4 that:

$$x^{q^m-1} - 1 = \prod_{i=1}^{q^m-1} (x - \alpha_i)$$

where α_i is a non-zero element of $GF(q)$. Using equation (3.1) we see that a generic prime polynomial $f_l(x)$ on $GF(q)$ can be factored in $GF(q^m)$ as a product of some $(x-\alpha_i)$ factors. In other words: $f_l(x)$, monic polynomial in $GF(q)$, becomes zero if evaluated in any element α_i of the extension $GF(q^m)$. That is to say that $f_l(x)$ is a *minimal polynomial* on $GF(q)$ with respect to a certain element α_i. We can, thus, build a generator polynomial by selecting its zeros among the elements of the extended field $GF(q^m)$ as it follows:

1. choose the zeros of $g(x)$: $\beta_1, \beta_2, ..., \beta_\mu \in GF(q^m)$

2. find the minimal polynomials: $f_{\beta_1}(x), f_{\beta_2}(x), ..., f_{\beta_\mu}(x)$
3. evaluate $g(x) = LCM(f_{\beta_1}(x), f_{\beta_2}(x), ..., f_{\beta_\mu}(x))$

The question is now about how to compute minimal polynomials. Before introducing another important theorem, we need to have a look to two relevant lemmas.

Theorem 15. *Let $GF(q) = GF(p^n)$ be a* finite field. *For all $s(x)$ and $t(x)$ on $GF(q)$ and for every integer $\nu \geq 1$, it holds that:*

$$[s(x) + t(x)]^{p^\nu} = [s(x)]^{p^\nu} + [t(x)]^{p^\nu}$$

Proof. Let's prove it by induction. The base case is $\nu = 1$:

$$[s(x) + t(x)]^p = [s(x)]^p + [t(x)]^p$$

We can use the binomial equation of Newton to get:

$$\begin{aligned}[s(x) + t(x)]^p &= \sum_{k=0}^{p} \binom{p}{k} [s(x)]^k [t(x)]^{p-k} \\ &= [s(x)]^p + \sum_{k=1}^{p-1} \binom{p}{k} [s(x)]^k [t(x)]^{p-k} + [t(x)]^p \end{aligned} \tag{3.2}$$

Let's have a look to the summation term:

$$\binom{p}{k} = \frac{p!}{k!(p-k)!} = \frac{p(p-1)!}{k!(p-k)!}$$

Since p is prime for hypotheses and $\binom{p}{k}$ is an integer, then $k!(p-k)!$ must divide $(p-1)!$, because neither k or $p-k$ can divide p, for $1 \leq k \leq p-1$. Thus $\binom{p}{k} = p\lambda$. For any k of the summation, we will get:

$$\binom{p}{k} [s(x)]^k [t(x)]^{p-k} = p\lambda [s(x)]^k [t(x)]^{p-k} = [p \cdot r(x)]\lambda$$

In $GF(p)$ the sum of p equal terms gives zero as result, because of addition modulo p. As a matter of fact, being $a \in GF(p)$, $a + a + \ldots + a$ p times is equal to $\mathcal{R}_p[pa] = 0$. Then, remembering that elements of $GF(p^n)$ can be associated to polynomials in $GF(q)$ with degree lower than n, also in $GF(p^n)$ summing p times the same elements gives zero as result, because we are erasing the relative polynomials coefficient by coefficient. This means that the summation of the equation (3.2) is always null, and then:

$$[s(x) + t(x)]^p = [s(x)]^p + [t(x)]^p$$

If we suppose the sentence has been verified for $\nu = i > 1$, for the $(i+1)$-th step it holds that:

$$\begin{aligned}[s(x)+t(x)]^{p^{i+1}} &= [s(x)+t(x)]^{p^i p} = \left[[s(x)+t(x)]^{p^i}\right]^p \\ &= \left[[s(x)]^{p^i} + [t(x)]^{p^i}\right]^p \\ &= [s(x)]^{p^{i+1}} + [t(x)]^{p^{i+1}}\end{aligned}$$

Since we proved that the sentence holds for $\nu = i+1$ as well, we can affirm that it holds for every $\nu > 0$. □

Theorem 16. *Let $GF(p^n)$ be a* finite field. *For every polynomial $f(x)$ with degree h on $GF(q)$ and for every integer $\nu \geq 1$, it holds that:*

$$[f(x)]^{p^\nu} = \sum_{i=0}^{h} f_i^{p^\nu} x^{ip^\nu}$$

Proof. Let $f(x)$ be the generic polynomial

$$f(x) = f_0 + f_1 x + f_2 x^2 + \ldots + f_{h-1}x^{h-1} + f_h x^h$$

where h is arbitrary. Then:

$$\begin{aligned}[f(x)]^{p^\nu} &= \left[\sum_{i=0}^{h} f_i x^i\right]^{p^\nu} \\ &= \left[f_h x^h + \sum_{i=0}^{h-1} f_i x^i\right]^{p^\nu} \\ &= \left[f_h x^h\right]^{p^\nu} + \left[\sum_{i=0}^{h-1} f_i x^i\right]^{p^\nu}\end{aligned}$$

where last equality holds for Theorem 15. Iterataing this reasoning we will

get:

$$\begin{aligned}
[f(x)]^{p^\nu} &= \left[f_h x^h\right]^{p^\nu} + \left[\sum_{i=0}^{h-1} f_i x^i\right]^{p^\nu} \\
&= \left[f_h x^h\right]^{p^\nu} + \left[f_{h-1}x^{h-1} + \sum_{i=0}^{h-2} f_i x^i\right]^{p^\nu} \\
&= \left[f_h x^h\right]^{p^\nu} + \left[f_{h-1}x^{h-1}\right]^{p^\nu} + \left[\sum_{i=0}^{h-2} f_i x^i\right]^{p^\nu} \\
&\vdots \\
&= \left[f_h x^h\right]^{p^\nu} + \left[f_{h-1}x^{h-1}\right]^{p^\nu} + \ldots + [f_1 x]^{p^\nu} + [f_0]^{p^\nu} \\
&= \sum_{i=0}^{h} \left[f_i x^i\right]^{p^\nu} \\
&= \sum_{i=0}^{h} f_i^{p^\nu} x^{ip^\nu}
\end{aligned}$$

and this concludes the proof. □

Theorem 17. *Let be $\beta \in GF(q^m)$ and $f_\beta(x)$ minimal polynomial of β on $GF(q) \Rightarrow f_\beta(x)$ is also the minimal polynomial on $GF(q)$ of β^q.*

Proof. Our aim is proving that $f_\beta(\beta^q) = 0$. Using Theorem 16 and supposing $f_\beta(x)$ has got degree h, we obtain:

$$[f_\beta(x)]^q = \sum_{i=0}^{h} f_i^q x^{iq}$$

because $q = p^\nu$ with $\nu = n$. Since f_i is an element of $GF(q)$, $f_i^q = f_i$ and then:

$$[f_\beta(x)]^q = \sum_{i=0}^{h} f_i (x^q)^i = f_\beta(x^q)$$

For hypotheses $f_\beta(\beta) = 0$, thus $[f_\beta(\beta)]^q = f_\beta(\beta^q) = 0$. Since $f_\beta(x)$ is a prime polynomial on $GF(q)$, it is the minimal polynomial on $GF(q)$ for β^q. □

Two elements in $GF(q^m)$ that have got the same minimal polynomial on $GF(q)$ are called *conjugate* with respect to $GF(q)$.

Thanks to Theorem 17 it is easy to recognise that

$$\mathcal{B}_q \doteq \{\beta, \beta^q, \beta^{q^2}, ..., \beta^{q^{r-1}}\}$$

is a set of conjugate elements with respect to $GF(q)$, where $q^r - 1$ is the multiplicative order of β.

Theorem 18. *$\beta \in GF(q^m) \Rightarrow$ the minimal polynomial of β on $GF(q)$ is:*

$$f_\beta(x) = \prod_{i=0}^{r-1}(x - \beta^{q^i})$$

where r is the lowest integer such that $\beta^{q^r} = \beta$.

Proof. First of all notice that, thanks to Theorem 17, we know that every element of $\mathcal{B}_q$ is a root of the polynomial $f_\beta(x)$; then the factors of the polynomial must contain every of the elements of $\mathcal{B}_q$ at least once.

Let's introduce the polynomial $f(x) = \prod_{i=0}^{r-1}(x - \beta^{q^i})$ and let's prove that $f(x) = f_\beta(x)$, that is $f(x)$ is the minimal polynomial of β. Because of its definition we already know that $f(x)$ is monic. We have to verify that its coefficients belong to $GF(q)$.

$$\begin{aligned}
[f(x)]^q &= \left[\prod_{i=0}^{r-1}(x - \beta^{q^i})\right]^q \\
&= \left[(x-\beta)(x-\beta^q)(x-\beta^{q^2}) \cdot \ldots \cdot (x-\beta^{q^{r-1}})\right]^q \\
&= (x-\beta)^q(x-\beta^q)^q(x-\beta^{q^2})^q \cdot \ldots \cdot (x-\beta^{q^{r-1}})^q \\
&= (x^q-\beta^q)(x^q-\beta^{q^2})(x^q-\beta^{q^3}) \cdot \ldots \cdot (x^q-\beta^{q^r}) \\
&= (x^q-\beta^q)(x^q-\beta^{q^2})(x^q-\beta^{q^3}) \cdot \ldots \cdot (x^q-\beta)
\end{aligned}$$

where we made use of Theorem 15 because $q = p^\nu$ with $\nu = n$. We can write the last expression as:

$$[f(x)]^q = \prod_{i=0}^{r-1}(x^q - \beta^{q^i})$$

and thus :

$$[f(x)]^q = f(x^q) = \sum_{i=0}^{r} f_i(x^q)^i = \sum_{i=0}^{r} f_i x^{iq} \tag{3.3}$$

On the other hand, for Theorem 16, we know that:

$$[f(x)]^q = \sum_{i=0}^{r-1} f_i^q x^{iq} \tag{3.4}$$

Comparing equations (3.3) and (3.4) we find that $f_i = f_i^q$ and then $f_i \in GF(q)$ for $i = 1, 2, \ldots, r$. □

We now have all the knowledge necessary to properly build a *primitive cyclic code*:

1. Choose a finite field $GF(q)$ and select $n = q^m - 1$ for any m.
2. Choose the zeros of $g(x)$: $\beta_1, \beta_2, ..., \beta_\mu \in GF(q^m)$.
3. Find out all the conjugate elements of each chosen zero β.
4. Compute the minimal polynomial, $f_\beta(x)$, for each chosen zero by multiplying all the factors $(x - \beta^{q^s})$ related to the conjugate elements of a zero: $\beta^{q^s} \in \mathcal{B}_q$.
5. Evaluate $g(x) = LCM\{f_{\beta_1}(x), f_{\beta_2}(x), ..., f_{\beta_\mu}(x)\}$.

3.4 BCH codes

BCH codes are a subclass of *cyclic codes* and are characterized by their particular construction through minimal polynomials. The construction of this kind of code is very similar to the ordinary proceeding for generating *cyclic codes*, with the only difference that, choosing the zeros of $g(x)$, we have to consider elements of $GF(q^m)$ described by consecutive powers of the primitive element α.

Theorem 19. *$\mathcal{C}$ is a* cyclic code *on $GF(q)$ with length n; $\beta \in GF(q^m)$ with multiplicative order n. **If** among the zeros of $g(x)$, generator polynomial of $\mathcal{C}$, there are $\gamma \geq 1$ consecutive powers of β, that is $g(\beta^{\xi_0+1}) = g(\beta^{\xi_0+2}) = ... = g(\beta^{\xi_0+\gamma}) = 0$, where ξ_0 is a generic offset, **then** $d_{min}(\mathcal{C}) \geq \gamma + 1$.*

Proof. Let's take any code polynomial $v(x) = v_0 + v_1 x + \ldots + v_{n-1}x^{n-1}$ that satisfies hypotheses. It must hold that:

$$v(\beta^{\xi_0+1}) = v(\beta^{\xi_0+2}) = ... = v(\beta^{\xi_0+\gamma}) = 0$$

Any of these polynomials can be expressed in the form:

$$\begin{aligned} v(\beta^{\xi_0+k}) &= v_0 + v_1(\beta^{\xi_0+k}) + v_2(\beta^{\xi_0+k})^2 + \ldots + v_{n-1}(\beta^{\xi_0+k})^{n-1} \\ &= v_0 + v_1\beta^{\xi_0+k} + v_2\beta^{2(\xi_0+k)} + \ldots + v_{n-1}\beta^{(n-1)(\xi_0+k)} \end{aligned}$$

with $1 \leq k \leq \gamma$. It is, thus, possible writing all of these polynomials in a matrix form, introducing the matrix $\mathbf{H}$, defined as it follows:

$$\mathbf{H} \doteq \begin{bmatrix} 1 & \beta^{\xi_0+1} & \beta^{2(\xi_0+1)} & \dots & \beta^{(n-1)(\xi_0+1)} \\ 1 & \beta^{\xi_0+2} & \beta^{2(\xi_0+2)} & \dots & \beta^{(n-1)(\xi_0+2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \beta^{\xi_0+\gamma} & \beta^{2(\xi_0+\gamma)} & \dots & \beta^{(n-1)(\xi_0+\gamma)} \end{bmatrix}$$

By defining also the vector related to $v(x)$: $\mathbf{v} = [v_0\ v_1\ \dots\ v_{n-1}]$, all the polynomials above said can be computed by the matrix product:

$$\mathbf{v} \cdot \mathbf{H}^T = [v_0\ v_1\ \dots\ v_{n-1}] \begin{bmatrix} 1 & 1 & \dots & 1 \\ \beta^{\xi_0+1} & \beta^{\xi_0+2} & \dots & \beta^{\xi_0+\gamma} \\ \vdots & \vdots & \ddots & \vdots \\ \beta^{(n-1)(\xi_0+1)} & \beta^{(n-1)(\xi_0+2)} & \dots & \beta^{(n-1)(\xi_0+\gamma)} \end{bmatrix} =$$

$$= \begin{bmatrix} v_0 + v_1\beta^{\xi_0+1} + \dots + v_{n-1}\beta^{(n-1)(\xi_0+1)} \\ v_0 + v_1\beta^{\xi_0+2} + \dots + v_{n-1}\beta^{(n-1)(\xi_0+2)} \\ \vdots \\ v_0 + v_1\beta^{\xi_0+\gamma} + \dots + v_{n-1}\beta^{(n-1)(\xi_0+\gamma)} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Let's proceed by contradiction, assuming that the minimum distance of the code $\mathcal{C}$ is not greater than γ: $d_{min}(\mathcal{C}) \leq \gamma$. Then, for Theorem 9, there exists a code word $\mathbf{w}$ with *Hamming weight* less than or equal to γ: $w(\mathbf{w}) = h \leq \gamma$. If we call $p_1, p_2, \dots, p_h$ its non-zero components, the code word can be written as:

$$\mathbf{w} = [w_{p_1}\ w_{p_2}\ \dots\ w_{p_h}]$$

Since $\mathbf{w}$ is a code word, it vanishes in all of the zeros of $g(x)$, thus:

$$\mathbf{w} \cdot \mathbf{H}^T = [w_{p_1}\ w_{p_2}\ \dots\ w_{p_h}] \begin{bmatrix} \beta^{p_1(\xi_0+1)} & \beta^{p_1(\xi_0+2)} & \dots & \beta^{p_1(\xi_0+h)} \\ \beta^{p_2(\xi_0+1)} & \beta^{p_2(\xi_0+2)} & \dots & \beta^{p_2(\xi_0+h)} \\ \vdots & \vdots & \ddots & \vdots \\ \beta^{p_h(\xi_0+1)} & \beta^{p_h(\xi_0+2)} & \dots & \beta^{p_h(\xi_0+h)} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Product between a vector and a matrix is a way to get a linear combination of matrix rows. If every of these linear combinations gives zero as result, then we can deduce that all the rows are linearly dependent and thus the determinant of the matrix must be zero. Let's re-write the previous matrix

as it follows:

$$\mathbf{M} = \begin{bmatrix} \beta^{p_1(\xi_0+1)} & \beta^{p_1(\xi_0+2)} & \dots & \beta^{p_1(\xi_0+h)} \\ \beta^{p_2(\xi_0+1)} & \beta^{p_2(\xi_0+2)} & \dots & \beta^{p_2(\xi_0+h)} \\ \vdots & \vdots & \ddots & \vdots \\ \beta^{p_h(\xi_0+1)} & \beta^{p_h(\xi_0+2)} & \dots & \beta^{p_h(\xi_0+h)} \end{bmatrix}$$

$$= \begin{bmatrix} \beta^{p_1(\xi_0+1)} & \beta^{p_1(\xi_0+1)}\beta^{p_1} & \dots & \beta^{p_1(\xi_0+1)}\beta^{p_1(h-1)} \\ \beta^{p_2(\xi_0+1)} & \beta^{p_2(\xi_0+1)}\beta^{p_2} & \dots & \beta^{p_2(\xi_0+1)}\beta^{p_2(h-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \beta^{p_h(\xi_0+1)} & \beta^{p_h(\xi_0+1)}\beta^{p_h} & \dots & \beta^{p_h(\xi_0+1)}\beta^{p_h(h-1)} \end{bmatrix}$$

$$= \begin{bmatrix} \beta^{p_1(\xi_0+1)} & 0 & \dots & 0 \\ 0 & \beta^{p_2(\xi_0+1)} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \beta^{p_h(\xi_0+1)} \end{bmatrix} \begin{bmatrix} 1 & \beta^{p_1} & \dots & \beta^{p_1(h-1)} \\ 1 & \beta^{p_2} & \dots & \beta^{p_2(h-1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \beta^{p_h} & \dots & \beta^{p_h(h-1)} \end{bmatrix}$$

and, evaluating the determinants of the two matrices:

$$det(\mathbf{M}) = \beta^{(\xi_0+1)(p_1+p_2+\dots+p_h)} det \begin{bmatrix} 1 & \beta^{p_1} & \dots & \beta^{p_1(h-1)} \\ 1 & \beta^{p_2} & \dots & \beta^{p_2(h-1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \beta^{p_h} & \dots & \beta^{p_h(h-1)} \end{bmatrix}$$

For what above said, $det(\mathbf{M})$ must be zero.

We are going to prove by induction that the determinant of the second matrix, which is said to be in *Vandermonde* form, cannot be null if elements $\beta^{p_1}, \beta^{p_2}, \dots, \beta^{p_h}$ are different one from each other and this is guaranteed by the choice of β: it has got order n by hypotheses and the code polynomials have maximum degree $n-1$; then we consider powers of β only from 0 to $n-1$ and these values must all be different. The base case is simply verified for a matrix of dimension 2. Let $\mathbf{V}_{\varphi-1}$ be a *Vandermonde* matrix of dimension $\varphi - 1$ in the form:

$$\begin{bmatrix} 1 & 1 & \dots & 1 \\ X_1 & X_2 & \dots & X_{\varphi-2} \\ \vdots & \vdots & \ddots & \vdots \\ X_1^{\varphi-2} & X_2^{\varphi-2} & \dots & X_{\varphi-2}^{\varphi-2} \end{bmatrix}$$

For inductive hypotheses we will assume that, if $X_1, X_2, \dots, X_{\varphi-2}$ are all different, then $\mathbf{V}_{\varphi-1}$ is non-singular. Let's consider now $\mathbf{V}_\varphi$ of dimension φ

and replace the element X_φ with the variable x. We will get:

$$\mathbf{V}_\varphi(x) = \begin{bmatrix} 1 & 1 & \dots & 1 \\ X_1 & X_2 & \dots & x \\ \vdots & \vdots & \ddots & \vdots \\ X_1^{\varphi-1} & X_2^{\varphi-1} & \dots & x^{\varphi-1} \end{bmatrix}$$

Evaluating the deteminant by expanding along the last column:

$$det(\mathbf{V}_\varphi(x)) = d_0 + d_1 x + \ldots + d_{\varphi-1} x^{\varphi-1}$$

where $d_0, d_1, \ldots, d_{\varphi-1}$ are determinants of matrices with dimension $\varphi - 1$, but only $d_{\varphi-1}$ is the determinant of a *Vandermonde* matrix, which is non-singular because of inductive hypotheses; this guarantees that $det(\mathbf{V}_\varphi(x))$ has got degree $\varphi - 1$, thus it vanishes for at most $\varphi - 1$ values. It is easy to see that $X_1, X_2, \ldots, X_{\varphi-1}$ are $\varphi - 1$ distinct solutions of the equations, because, if $x = X_i$, $\mathbf{V}_\varphi(x)$ would have two equal columns and the determinant would be determinely zero. Then, we can write that:

$$det(\mathbf{V}_\varphi(X_{\varphi-1})) = det(\mathbf{V}_\varphi) = d_{\varphi-1} \prod_{i=1}^{\varphi-1} (X_\varphi - X_i)$$

which is not null if and only if $X_1, X_2, \ldots, X_\varphi$ are all different one from each other.

Since the *Vandermonde* matrix cannot have null determinant, then it should be $\beta^{(\xi_0+1)(p_1+p_2+\ldots+p_h)} = 0$, but this is possible only if $\beta = 0$, which is in contradiciton with the hypotheses that β has got order n. We have proved that a code word with *Hamming weigth* smaller than or equal to γ cannot exist, and then $d_{min}(\mathcal{C}) \geq \gamma + 1$. □

It should be now clear why we introduced the constraint of choosing the zeros as consecutive powers of β: it allows us to fix the minimum *Hamming distance* of the code $\mathcal{C}$, and therefore we can decide the maximum number of errors we can detect or correct. As a matter of fact, if we are required to correct t errors, we have to select $2t$ consecutive powers, so that to obtain a minimum distance $2t+1$, which allows us to correct up to $\lfloor \frac{d_{min}}{2} \rfloor = \lfloor \frac{2t+1}{2} \rfloor = \lfloor t + \frac{1}{2} \rfloor = t$ corrupted symbols.

3.5 Reed-Solomon codes

In the previous sections we have investigated the powerful classes of *primitive cyclic codes* and *BCH codes*, defined on $GF(q^r)$, whose length is

$n = q^r - 1$. *Reed-Solomon codes* are a subclass of non-binary *BCH codes* on $GF(q^r)$, where $r = 1$. Let's see what it does mean. Having $r = 1$ obviously implies that $GF(q^r) = GF(q)$; suppose we have to assure to correct up to t errors, then we have to choose $2t$ consecutive powers of the primitive element α in $GF(q)$. Here comes the first advantage: we can build $GF(q)$ as an extension of a *Galois Field* $GF(p)$, $GF(q) = GF(p^m)$, with p prime. We can do that, once we have fixed p, evaluating $GF(q) = GF(p^m) = GF(p)[x]/p(x)$, with $p(x)$ primitive polynomial of degree m. In this way we can consider elements of $GF(q)$ as polynomials and, by choosing $p(x)$ a primitive polynomial in $GF(p)$ of degree m, we are sure that $\alpha = x$ is the primitive polynomial in $GF(q)$. For sake of computational simplicity it is usual to take $p = 2$. Elements of $GF(q)$ are thus simply expressed as binary polynomials, whose coefficients can only be either 0 or 1. This provides a very easy digital implementation and even *finite field* operations are trivially computable: addition modulo 2 is made by *XOR* function between two vectors and also multiplication is simplified by the use of the *shift* operator.

Once we have chosen $2t$ consecutive powers of the primitive element, that is powers of $x \in GF(2^m)$, we have to compute all their minimal polynomials. A minimal polynomial of a on $GF(q)$ is, by definition, the minimum degree monic polynomial with coefficients in $GF(q)$, such that $a \in GF(q^m)$ is a zero of it. Here is a second benefit: in the case that $GF(q^m) = GF(q)$, the required polynomial is trivially $x - a$. Therefore, for each consecutive power of α, the minimal polynomial will be $x - \alpha^i$ and the generator polynomial can be easily evaluated by multiplying all of these elementary factors:

$$g(x) = \prod_{i=1}^{2t} (x - \alpha^{\xi_0 + i}) \tag{3.5}$$

where ξ_0 is an arbitrary offset. Note that *Reed-Solomon codes* are easier to encode than generic *BCH codes* beacuse they do not require the computation of minimal polynomials of every zero we choose.

Reed-Solomon codes are *minimum distance codes*, that is a *Reed-Solomon code* of the type (n, k) (hereafter we will refer to it as $RS(n, k)$) has got the smallest possible *Hamming distance*: $d_{min}(RS(n, k)) = n - k + 1$. This is easy to see because, thanks to construction process and Theorem 19:

$$d_{min}(RS(n, k)) \geq 2t + 1$$

From equation (3.5) we note that *Reed-Solomon* generator polynomial has got degree $2t$, but, as a generator polynomial of a code (n, k), its degree must

also be $n-k$; then: $n-k=2t$ and

$$d_{min}\left(RS(n,k)\right) \geq 2t+1 = n-k+1$$

On the other hand, *Singleton bound* of Theorem 10 tells us that

$$d_{min}\left(RS(n,k)\right) \leq n-k+1$$

Putting together the two inequalities we find:

$$d_{min}\left(RS(n,k)\right) = n-k+1$$

which is the smallest possible.

The here presented *Reed-Solomon codes* definition is quite different from the one given in section (1.3). These two formulation are equivalent and it can be proved through discrete-time Fourier transform; we will not explore this topic in more detail, but a comprehensive description can be found in [6, pp. 181-188]

3.5.1 *RS(255, 223)* code

We are going to introduce one of the most commonly used instances of *Reed-Solomon codes*: *RS(255, 223)*. This code maps 223 length words in 255 length code words. Let's inspect its parameters:

- $k = 223$
- $n = 255$
  Since in *Reed-Solomon codes* $n = q-1$, where q is the numerosity of the employed *Galois Field*, whose elements represent the code symbols, we deduce that the *finite field* we need is $GF(q) = GF(256)$. As discussed above, choosing the *finite field* as an extension of $GF(2)$ makes implementation much easier; thus, it is very comfortable to see $GF(256)$ as $GF(2^8)$. The number of bits required to represent in binary form a symbol is, then, $m = 8$.
- $m = 8$
  m is the degree of the prime polynomial $p(x)$ which characterizes $GF(p^m) = GF(2^8) = GF(2)[x]/p(x)$. Every element of $GF(256)$ is uniquely associated to a poloynomial with coefficients in $GF(2)$ with smaller degree than 8 and it can be, thus, represented as a binary vector of length 8.

  As primitive polynomial we will use $p(x) = 1 + x^2 + x^3 + x^4 + x^8$.

- $t = 16$
  t is the greatest number of symbol errors *RS(255, 223)* can correct:

  $$t = \left\lfloor \frac{d_{min}}{2} \right\rfloor = \left\lfloor \frac{n-k+1}{2} \right\rfloor = \left\lfloor \frac{255-223+1}{2} \right\rfloor = \left\lfloor \frac{33}{22} \right\rfloor = \lfloor 16.5 \rfloor = 16$$

  Since the code corrects symbols, which are made up by 8 bits each, the capability of *Reed-Solomon codes* to correct errors is much higher than it can seem: if we use a binary transmission on the channel, this can corrupt up to $16 \times 8 = 128$ bits, but, unless more than 16 symbols are involved, the word can still be properly corrected.

Chapter 4

Reed-Solomon Encoder

In this chapter the general encoding proceeding for *Reed-Solomon codes* will be exposed. Every theorical description will be followed by the realisation for a *RS(255, 223)* code. For the definition of *Reed-Solomon codes* we know that $n = q-1$, where $q = p^m$ is the numerosity of the *Galois Field* from which symbols that made up a code word are taken. It is usual to choose $p = 2$ to have an easier digital implementation: binary polynomial notation makes it very easy to perform operations in the *finite field*, even in an electrical way; for example the polynomial sum can be implemented just by a *XOR* function.

4.1 Building the *Galois Field*

To make up the set of symbols that will form our alphabet, we have to compute all the polynomials of $GF(p)[x]/p(x)$, where $p(x)$ is a polynomial with degree m. Every m-degree polynomial can be chosen, but, for practical reasons, m-degree primitive polynomials are usually taken. In this document we will not see how to find a primitive polynomial, but in Appendix A a precompiled table is provided for some primitive polynomials with coefficients in $GF(2)$. Using a primitive polynomial allows us to indicate every element in $GF(q)$ with a power of the primitive element $\alpha = x$ and it is very simple to build the whole field starting from the primitive element: since this one generates every other element, we have just to multiply x by itself n times and evaluate the reminder modulo $p(x)$ of the obtained polynomial.

RS(255, 223) As we have to build a $GF(256) = GF(2^8)$ *finite field*, the primitive polynomial we will adopt is: $p(x) = 1+x^2+x^3+x^4+x^8$. Using this polynomial the primitive element α of $GF(2^8)$ will be x, thus, 256 consecutive

powers of x will cover the whole *finite field* and will form every disposition of "1" and "0" in 8 positions.

Let's start by the primitive element $\alpha = x$ to form the whole *field*: the first two elements are, of course, $0 = \alpha^{-\infty}$ and $x = \alpha^1$. Then, we move to evaluate $\alpha^2 = x^2$, which, having less degree than $m = 8$, is already a valid polynomial and so on. At the generic i-th step, we have to evaluate $\alpha^i = \mathcal{R}_{p(x)}[x^i]$. If everything has been properly computed, α^q, that is $\alpha^{(n-1)} = \alpha^{255}$, should correspond to 1.

4.2 Creating the generator polynomial

Once the alphabet is defined we shall dedicate to build the generator polynomial $g(z)$ of the code (to avoid confusion we will use the variable "x" to work with polynomials referring to *Galois Field* elements, and the variable "z" to deal with input and code polynomials, whose coefficients are elements of $GF(q)$). To correct t errors, the generator polynomial $g(z)$ must be formed by at least $2t$ factors. For sake of simplicity we will take exactly $2t$ factors:

$$g(z) = (z - \beta_1)(z - \beta_2) \cdot \ldots \cdot (z - \beta_{2t})$$

where $\beta_1, \beta_2, \ldots, \beta_{2t}$ are powers of the primitive element α. It is not important from where we start considering the powers, but we have to guarantee that powers are consecutive.

RS(255, 223) Let's take the first $2t = 2 \cdot 16 = 32$ powers:

$$\beta_1 = \alpha,\ \beta_2 = \alpha^2,\ \ldots,\ \beta_{2t} = \alpha^{2t} = \alpha^{32}$$

If we multiply consecutively each of the factors $(z - \beta_1)(z - \beta_2) \ldots (z - \beta_{32})$ we will obtain the 32-degree generator polynomial:

$$\begin{aligned} g(z) \quad &= 45 \quad +216z \quad +239z^2 \quad +24^3 \quad +253z^4 \quad +104z^5 + \\ &+27z^6 \quad +40z^7 \quad +107z^8 \quad +50z^9 \quad +163z^{10} \quad +210z^{11} + \\ &+227z^{12} \quad +134z^{13} \quad +224z^{14} \quad +158z^{15} \quad +119z^{16} \quad +13z^{17} + \\ &+158z^{18} \quad +z^{19} \quad +238z^{20} \quad +164z^{21} \quad +82z^{22} \quad +43z^{23} + \\ &+15z^{24} \quad +232z^{25} \quad +246z^{26} \quad +142z^{27} \quad +50z^{28} \quad +189z^{29} + \\ &+29z^{30} \quad +232z^{31} \quad +z^{32} \end{aligned}$$

4.3 Encoding

Once the generator polynomial $g(z)$ has been evaluated, we are ready to encode an input word: each of the q^k possible k length input words can be

transformed into a polynomial of degree $k-1$ using each of its k symbols as a coefficient:

$$[u_0, u_1, u_2, ..., u_{k-1}] \;\rightarrow\; u(z) = u_0 + u_1 z + u_2 z^2 + ... + u_{k-1} z^{k-1}$$

Now, by multiplying the $(k-1)$-degree polynomial $u(z)$ for the $(n-k)$-degree generator polynomial $g(z)$, we will obtain a code polynomial $c(z)$, whose degree is:

$$(k-1) + (n-k) = n-1$$

RS(223, 255) Our code words are 255-length vectors or, equivalently, 254-degree polynomials. Of course the most significant coefficients must be null and we could have a lower degree polynomial, but we can never exceed the maximum degree: 254.

4.3.1 Systematic encoding

Encoding as above illustrated is not yet the most commonly adopted: we would often like to guess the original input word just by looking at the produced code word. In other words, we might want to get a *systematic code* (see section (3.2)).

Since an input word is made by k symbols and a codeword by n symbols, $n-k$ redundancy symbols have been added to protect the message. Then, one could wonder if it is possible to leave the k symbols of the input word unalterated in the code word and to add the $n-k$ further redundancy symbols in the beginning (or in the end) of the code word. This is perfectly possible and, indeed, it does not require much more computation than non-systematic algorithm does.

If we want to gather together all the input symbols (without changing their order, of course!) in the end of the code word, we could simply multiplying the input polynomial for z^{n-k}. The resulting polynomial is a $(n-1)$-degree polynomial with the most significant coefficients all equal to the coefficients of the input polynomial:

$$\begin{aligned} u(z) \cdot z^{n-k} &= (u_0 + u_1 z + u_2 z^2 + ... + u_{k-1} z^{k-1}) \cdot z^{n-k} \\ &= u_0 z^{n-k} + u_1 z^{n-k+1} + u_2 z^{n-k+2} + ... + u_{k-1} z^{n-1} \end{aligned}$$

which corresponds to the binary word:

$$[\underbrace{0\ 0\ \ldots\ 0}_{n-k}\ \underbrace{u_0\ u_1\ u_2\ \ldots\ u_{k-1}}_{k}]$$

Code thus achieved, however, might not be still a *Reed-Solomon code* or even a *linear code*. To be a *Reed-Solomon code*, every polynomial built in this way should be divisible for the generator polynomial $g(z)$, but this is not always true. Let's suppose:

$$u(z) \cdot z^{n-k} = q(z) \cdot g(z) + r(z) \tag{4.1}$$

with $deg[r(z)] < deg[q(z)]$ and $deg[r(z)] < deg[g(z)]$. Then, we can observe that:

$$\begin{aligned} \mathcal{R}_{g(z)}[u(z) \cdot z^{n-k}] &= \mathcal{R}_{g(z)}[q(z) \cdot g(z) + r(z)] \\ &= \mathcal{R}_{g(z)}[q(z) \cdot g(z)] + \mathcal{R}_{g(z)}[r(z)] \\ &= r(z) \end{aligned} \tag{4.2}$$

because $r(z)$ has got lower degree than $g(z)$ and the first term, clearly divisible by $g(z)$, has null reminder. Therefore, putting together expressions (4.1) and (4.2):

$$u(z) \cdot z^{n-k} - \mathcal{R}_{g(z)}[u(z) \cdot z^{n-k}] = q(z) \cdot g(z)$$

Polynomial $u(z) \cdot z^{n-k} - \mathcal{R}_{g(z)}[u(z) \cdot z^{n-k}]$ is obviously divisible by $g(z)$ because it is equal to something divisible by $g(z)$ and since:

$$deg[r(z)] < deg[g(z)] = 2t = n - k$$

powers of z in $r(z)$ go from z^0 to z^{n-k-1} and there is not any coefficient that is added to the coefficients of the polynomial $u(z) \cdot z^{n-k}$ before evaluated, whose powers go from z^{n-k} to z^{n-1}. Thus, the resulting polynomial $u(z) \cdot z^{n-k} - \mathcal{R}_{g(z)}[u(z) \cdot z^{n-k}]$ has got the k most significant coefficients equal to the coefficients of the input polynomial $u(z)$; moreover, it is divisible by $g(z)$, that makes it a valid code polynomial:

$$c(z) = u(z) \cdot z^{n-k} - \mathcal{R}_{g(z)}[u(z) \cdot z^{n-k}]$$

This method allows us to build a *Reed-Solomon systematic code*.

RS(255, 223) The equivalent encoding formula for a *RS(255, 223)* code is:

$$c(z) = u(z) \cdot z^{32} - \mathcal{R}_{g(z)}[u(z) \cdot z^{32}]$$

where $u(z)$ is the input word.

Chapter 5

Reed-Solomon Decoder

In this chapter we will introduce a decoding algorithm described by Berlekemp in [2] and by Massey in [15]. We will also compare the efficiency of this strategy with the one proposed by Peterson, Gorenstein and Zierler ten years before. It is interesting making this comparison because, before the invention of the *Berlekamp-Massey algorithm* the only available solution of the decoding problem was represented by the far more expensive technique proposed by Peterson.

In this chapter, for the sake of clearness, theorical explanation will be followed by examples using an $RS(7, 3)$ code. It would be impossible making any simple demonstration with a $RS(255, 223)$ code.

First of all, let's define the notation we will use in the following:

- $\mathbf{u}$ is the original k-symbol input word
- $\mathbf{c}$ is the n-symbol code word generated by $\mathbf{u}$
- $\tilde{\mathbf{c}}$ is the received code word, after the corruption of some symbols because of the noisy channel
- $\tilde{\mathbf{u}}$ is the decoded word. If some errors have occurred during the transmission, this vector might be different from $\mathbf{u}$

We can model the corruption of the codeword by adding an error vector $\mathbf{e}$ to the sent word $\mathbf{c}$:

$$\tilde{\mathbf{c}} = \mathbf{c} + \mathbf{e}$$

It is obvious that, if we knew $\mathbf{e}$, it would be very easy to subtract it from the received word and then recover the correct code word. Unfortunately, we cannot have any idea about the nature of $\mathbf{e}$, because channel noise is a

consequence of unpredictable factors. We will see, however, that, unless too many errors have occurred during the transmission, we can deduce $\mathbf{e}$ thanks to redundancy symbols. For a more detailed illustration of noisy channels and distortion the reader is addressed to [22, ch. 3, 4, 5].

It will be very useful to consider vectors above said as polynomials, with the leftmost component the less significant. This holds:

$$\tilde{c}(x) = c(x) + e(x) \tag{5.1}$$

with $e(x) = e_0 + e_1 x + \ldots + e_{n-1}x^{n-1}$. In most cases, not all the components of $\mathbf{c}$ will be modified by the noise, thus it is worth defining ν as the number of the non-zero coefficients of $e(x)$ and re-writing it as:

$$e(x) = e_{p_1}x^{p_1} + e_{p_2}x^{p_2} + \ldots + e_{p_\nu}x^{p_\nu} \tag{5.2}$$

where p_i represents the power of the variable x, and thus the position of the components ($0 \leq p_i \leq n-1$), while e_{p_i} represents the value of the non-zero coefficient in the p_i position, with $1 \leq i \leq \nu$. In order to completely identify $e(x)$ we have to know:

- the number ν of the non-zero coefficients
- the ν positions of the non-zero coefficients $p_1, p_2, \ldots, p_\nu$
- the ν values of the non-zero coefficients $e_{p_1}, e_{p_2}, \ldots, e_{p_\nu}$.

RS(7, 4) Let's assume that the information word $\mathbf{u} = [6\ 6\ 1]$ has to be transmitted. Using a systematic encoding we will obtain the codeword $\mathbf{c} = [5\ 2\ 1\ 2\ 6\ 6\ 1]$. Let's suppose, also, that channel corrupts some symbols, transforming $\mathbf{c}$ into $\tilde{\mathbf{c}} = [\mathbf{1}\ 2\ \mathbf{5}\ 2\ 6\ 6\ 1]$. Our aim is going back to the original $\mathbf{c}$.

5.1 Syndromes

The first step to decode the received word is computing *syndromes*, that is the results obtained by evaluating the word in the roots of the generator polynomial. Every code word is created so that it can be divided by the generator polynomial $g(x)$ and in section (3.3) (see Theorem 13) we have already seen that every polynomial with degree lower than n with this characteristic is a valid code word. Therefore, if we evaluate a word in any of the roots of $g(x)$, we will obtain 0 as result only if it is a code word. For as the

generator polynomial has been created, we know its $2t$ roots: $\alpha, \alpha^2, \ldots, \alpha^{2t}$, with α primitive element of $GF(2^m)$. The i-th syndrome is defined as:

$$\begin{aligned} S_i \doteq \tilde{c}(\beta_i) &= c(\beta_i) + e(\beta_i) = 0 + e(\beta_i) = e(\alpha^i) \\ &= e_{p_1}(\alpha^i)^{p_1} + e_{p_2}(\alpha^i)^{p_2} + \ldots + e_{p_\nu}(\alpha^i)^{p_\nu} \\ &= e_{p_1}(\alpha^{p_1})^i + e_{p_2}(\alpha^{p_2})^i + \ldots + e_{p_\nu}(\alpha^{p_\nu})^i \end{aligned} \tag{5.3}$$

Since we dispose of $2t$ different zeros β, we can find $2t$ syndromes: $S_1, \ldots, S_{2t}$. If every syndrome is null, we are sure that the code word is valid and we just have to extrapolate the information word from the righmost k values, if we are using a systematic form. Otherwise, if just a syndrome is different by zero, the code word does not have one of the roots of $g(x)$ as its root, it cannot be divided by $g(x)$ and, therefore, it is not valid. In this case we have *detected* an error. Remember that there is the risk that occurred symbol errors modify the code word into another eligible one, so that we are not able to recognise the transformation and the error is not detected. In order to avoid this eventuality, we take care to provide the largest possible minimum distance d_{min} of the code. As seen in section (3.5), *Reed-Solomon codes* already have the maximum d_{min} allowed for a code (n, k), thus, in order to reach a higher robustness, we have to increase the amount of redundancy, by choosing a greater n.

Of course we are interested in the worst case, when syndromes are not all zero and we have to go back to the original word analysing $S_1, S_2, \ldots, S_{2t}$. For the sake of clearness we will replace in every of the $2t$ syndrome expressions (5.3) the generic term e_{p_j} with the variable Y_j and the generic term α^{p_j} with the variable X_j, where j is an integer between 1 and ν. This brings to the following non-linear system, made of $2t$ equations and 2ν variables:

$$\begin{cases} S_1 = Y_1X_1 + Y_2X_2 + \ldots + Y_\nu X_\nu \\ S_2 = Y_2X_2^2 + Y_2X_2^2 + \ldots + Y_\nu X_\nu^2 \\ \vdots \\ S_{2t} = Y_1X_1^{2t} + Y_2X_2^{2t} + \ldots + Y_\nu X_\nu^{2t} \end{cases} \tag{5.4}$$

Notice that the above system can be solved only if $2\nu < 2t$, that is if the number of symbol errors is less than the maximum number of errors that can be corrected by the code. If the probability of having more than t errors is too high, we should use a more powerful code in order to guarantee a reliable communication.

RS(7, 4) One can easily see that roots of the generator polynomial are $\alpha = 2$, $\alpha^2 = 4$, $\alpha^3 = 3$ and $\alpha^4 = 6$. Evaluating the received polynomial,

$\tilde{c}(x) = 1 + 2x + 5x^2 + 2x^3 + 6x^4 + 6x^5 + x^6$, in such elements we will get:

$$\begin{aligned} S_1 &= \tilde{c}(\alpha) = \alpha \\ S_2 &= \tilde{c}(\alpha^2) = 1 \\ S_3 &= \tilde{c}(\alpha^3) = \alpha^4 \\ S_4 &= \tilde{c}(\alpha^4) = \alpha^5 \end{aligned}$$

Since there is at least one non-zero syndrome, we know that the received one cannot be a valid code word.

5.2 Error locator polynomial

We are not able to find a solution in a quick way for the non-linear system (5.4), but things improve if we make use of an auxiliary polynomial $\Lambda(x)$ which has got $X_1^{-1}, X_2^{-1}, \ldots, X_\nu^{-1}$ as roots: the *error locator polynomial*; as said by its name, this polynomial will permit us to find locations of the occurred alterations. We can build the *error locator polynomial* as it follows:

$$\Lambda(x) \doteq \prod_{j=1}^{\nu}(1 - X_j x) = \Lambda_\nu x^\nu + \Lambda_{\nu-1} x^{\nu-1} + \ldots + \Lambda_1 x + 1 \tag{5.5}$$

It is clear that our aim is re-writing the syndrome system as a linear system by some parametric replacements. Let's see how we can do that. For $1 \leq j \leq \nu$ we know that $\Lambda(X_j^{-1}) = 0$ and thus it is licit to write: $\Lambda(X_j^{-1}) Y_j X_j^{l+\nu} = 0$ with l an arbitrary integer. Estimating the previous equation we get:

$$\begin{aligned} \Lambda(X_j^{-1}) Y_j X_j^{l+\nu} &= [\Lambda_\nu (X_j^{-1})^\nu + \Lambda_{\nu-1}(X_j^{-1})^{\nu-1} + \ldots + \Lambda_1 (X_j^{-1}) + 1] Y_j X_j^{l+\nu} \\ &= [\Lambda_\nu X_j^{-\nu} X_j^{l+\nu} + \Lambda_{\nu-1} X_j^{1-\nu} X_j^{l+\nu} + \ldots + \Lambda_1 X_j^{-1} X_j^{l+\nu} + X_j^{l+\nu}] Y_j \\ &= [\Lambda_\nu X_j^{l} + \Lambda_{\nu-1} X_j^{l+1} + \ldots + \Lambda_1 X_j^{l+\nu-1} + X_j^{l+\nu}] Y_j \\ &= \Lambda_\nu Y_j X_j^{l} + \Lambda_{\nu-1} Y_j X_j^{l+1} + \ldots + \Lambda_1 Y_j X_j^{l+\nu-1} + Y_j X_j^{l+\nu} = 0 \end{aligned}$$

We can now notice some more similarities with the expression of a syndrome (5.3). Let's see what happens if we sum the above expression for j that goes

from 1 to ν:

$$\begin{aligned}\sum_{j=1}^{\nu}\Lambda(X_j^{-1})Y_jX_j^{l+\nu} &= (\Lambda_\nu Y_1X_1^l+\Lambda_{\nu-1}Y_1X_1^{l+1}+\ldots+\Lambda_1Y_1X_1^{l+\nu-1}+Y_1X_1^{l+\nu})+\\ &+(\Lambda_\nu Y_2X_2^l+\Lambda_{\nu-1}Y_2X_2^{l+1}+\ldots+\Lambda_1Y_2X_2^{l+\nu-1}+Y_2X_2^{l+\nu})+\ldots+\\ &+(\Lambda_\nu Y_\nu X_\nu^l+\Lambda_{\nu-1}Y_\nu X_\nu^{l+1}+\ldots+\Lambda_1Y_\nu X_\nu^{l+\nu-1}+Y_\nu X_\nu^{l+\nu})=\\ &=\Lambda_\nu(Y_1X_1^l+\ldots+Y_\nu X_\nu^l)+\\ &+\Lambda_{\nu-1}(Y_1X_1^{l+1}+\ldots+Y_\nu X_\nu^{l+1})+\ldots+\\ &+(Y_1X_1^{l+\nu}+\ldots+Y_\nu X_\nu^{l+\nu})=\\ &=\Lambda_\nu S_l+\Lambda_{\nu-1}S_{l+1}+\ldots+S_{l+\nu}=0\end{aligned}$$

Forcing l to be equal to $1,2,\ldots,\nu$ we will obtain the following linear system in ν equations and ν variables $(\Lambda_1,\Lambda_2,\ldots,\Lambda_\nu)$:

$$\begin{cases}\Lambda_\nu S_1+\Lambda_{\nu-1}S_2+\ldots+S_{\nu+1}=0\\ \Lambda_\nu S_2+\Lambda_{\nu-1}S_3+\ldots+S_{\nu+2}=0\\ \qquad\qquad\vdots\\ \Lambda_\nu S_\nu+\Lambda_{\nu-1}S_{\nu+1}+\ldots+S_{2\nu}=0\end{cases}\tag{5.6}$$

which is much easier to solve. To find a solution for the system we can think about it in a matrix form:

$$\begin{bmatrix}S_1 & S_2 & \cdots & S_\nu\\ S_2 & S_3 & \cdots & S_{\nu+1}\\ \vdots & \vdots & \ddots & \vdots\\ S_\nu & S_{\nu+1} & \cdots & S_{2\nu-1}\end{bmatrix}\begin{bmatrix}\Lambda_\nu\\ \Lambda_{\nu-1}\\ \vdots\\ \Lambda_1\end{bmatrix}=\begin{bmatrix}-S_{\nu+1}\\ -S_{\nu+2}\\ \vdots\\ -S_{2\nu}\end{bmatrix}\tag{5.7}$$

or, in compact form:

$$\mathbf{S}_\nu\mathbf{\Lambda}=-\mathbf{S}(\nu+1,2\nu)$$

If $\mathbf{S}_\nu$ is invertible, then the system solution is:

$$\mathbf{\Lambda}=-\mathbf{S}_\nu^{-1}\mathbf{S}(\nu+1,2\nu)\tag{5.8}$$

Once we have found out the coefficients of $\Lambda(x)$, we can go back to $X_1,X_2,\ldots,X_\nu$ and thus make the system of syndromes (5.4) linear; we could now solve it to evaluate $Y_1,Y_2,\ldots,Y_\nu$ and eventually recompose the polynomial $e(x)$ to subtract from $\tilde{c}(x)$ in (5.1).

In order to solve the system we should invert matrix $\mathbf{S}_\nu$, but this may be very expensive in terms of time, especially if this operation has to be performed several times, as in *Peterson algorithm* [18], also described in [4,

pp. 166-174]. Berlekamp proposed a much more efficient algorithm to solve system (5.4) in an inductive way: at the r-th step we look for a solution only the first r syndromes, $S_1, S_2, \ldots, S_r$, ignoring the remaining ones; then we consider a syndrome related to a zero not yet used and we check whether the *error locator polynomial* found at the previous passage still works for the current system. If it fits, we can hold the current $\Lambda(x)$ and add another syndrome, otherwise we have to compute $\mathbf{\Lambda}$ again. Let's suppose we have already find the vector $\mathbf{\Lambda}$ for this r-th step in any way and let's call it $\mathbf{\Lambda}^{(r)}$. The corresponding polynomial is:

$$\Lambda^{(r)}(x) = \Lambda_{l_r}^{(r)} x^{l_r} + \Lambda_{l_r-1}^{(r)} x^{l_r-1} + \ldots + \Lambda_1^{(r)} x + 1 \tag{5.9}$$

where l_r indicates the maximum degree the polynomial could have at this step. We do not still know the real weight of the error polynomial $e(x)$, ν, we can only know that, for r syndromes a polynomial of degree at most l_r, with $l_r \leq \nu$, is sufficient to solve actual r equations system. It is important to understand that l_r may not be the degree of $\Lambda^{(r)}(x)$, but it is an upper bound for that. One could wonder what is the real meaning of such a parameter: it becomes quite clearer by investigating *linear feedback shift register* theory [8, ch. 8]; in this work we will not go deeper into this matter because of the wide extent of this topic, whose relation with algebraic decoding was largely developed by Massey [15]; we will limit ourselves to say that l_r represents the length of a shift register [4, pp. 177]. A brief, but exhaustive, view of relation between *shift registers* and decoding algorithms can be found in [6, pp. 188-214] and [7, pp. 245-252].

Coefficients found for the r-th step satisfy the system:

$$\begin{cases} \Lambda_{l_r} S_1 + \Lambda_{l_r-1} S_2 + \ldots + \Lambda_1 S_{l_r} + S_{l_r+1} = 0 \\ \Lambda_{l_r} S_2 + \Lambda_{l_r-1} S_3 + \ldots + \Lambda_1 S_{l_r+1} + S_{l_r+2} = 0 \\ \qquad \vdots \\ \Lambda_{l_r} S_{r-l_r} + \Lambda_{l_r-1} S_{r-l_r+1} + \ldots + \Lambda_1 S_{r-1} + S_r = 0 \end{cases} \tag{5.10}$$

or, in a shorter form:

$$-S_i = \sum_{h=1}^{l_r} \Lambda_h^{(r)} S_{i-h} \tag{5.11}$$

for $i = l_r + 1, l_r + 2, \ldots, r$ and, since $\Lambda_0^{(r)} = 1$ for every r, it can also be seen as:

$$\sum_{h=0}^{l_r} \Lambda_h^{(r)} S_{i-h} = 0 \tag{5.12}$$

As observed and deeply investigated in [24, p. 145], [7, pp. 245-252] and [6, pp. 188-191], expressions (5.11) and (5.12) are exactly output equations for a *shift register*.

Let $\Lambda^{(r)}$ be one of the polynomials with the lowest l_r such that it satisfies expressions (5.11) and (5.12), then we call the couple $\{\Lambda^{(r)}(x), l_r\}$ the *minimum solution* for the first r syndromes. Of course, polynomials verifying system (5.10) could have any degree, but we are interested in finding the solution with the least l_r because it corresponds to the lower degree *error locator polynomial* and, thus, to the least number of errors occurred in the transmitted word. This is an as strict as reasonable condition, because we are assuming we are using a code such that our symbol error probability is quite low and the channel corrupts only a few symbols for each word. If these hypotheses did not hold, all our efforts to achieve a reliable communication would be quite useless, because we could not guarantee any proper correction of transmission errors. Hence we will consider the most likely event the one such that number of symbol errors is smaller than or equal to t. For a less naive treatment of this topic, the reader is addressed to other texts, as [12], [1] or [22, ch. 3, 4, 5].

Once we know $\{\Lambda^{(r)}(x), l_r\}$, we are ready to find $\{\Lambda^{(r+1)}(x), l_{r+1}\}$ such that:

$$\sum_{h=0}^{l_{r+1}} \Lambda_h^{(r+1)} S_{i-h} = 0 \tag{5.13}$$

for $i = l_{r+1} + 1, l_{r+1} + 2, \ldots, r + 1$.

First of all we have to check whether the solution for the r-th step still fits for the $(r+1)$-th one, thus all we have to do is verifying if the polynomial $\Lambda^{(r)}(x)$ solves the $(r + 1)$-th equation, related to the $(r + 1)$-th syndrome. We have to evaluate the syndrome S_{r+1} thorugh the $(r + 1)$-th root of $g(x)$ and compare it to the syndrome computed by $\Lambda^{(r)}(x)$. Let's define the latter, recalling (5.11), as:

$$\tilde{S}_{r+1} \doteq -\sum_{h=1}^{l_r} \Lambda_h^{(r)} S_{r+1-h} \tag{5.14}$$

and the r-th difference between real syndrome and equivalent syndrome as:

$$\begin{aligned}\delta_r &\doteq S_{r+1} - \tilde{S}_{r+1} \\ &= S_{r+1} - (-\sum_{h=1}^{l_r} \Lambda_h^{(r)} S_{r+1-h}) = S_{r+1} + \sum_{h=1}^{l_r} \Lambda_h^{(r)} S_{r+1-h} \\ &= S_{r+1} + \Lambda_1^{(r)} S_r + \Lambda_2^{(r)} S_{r-1} + \ldots + \Lambda_{l_r}^{(r)} S_{r+1-l_r} \\ &= \sum_{h=0}^{l_r} \Lambda_h^{(r)} S_{r+1-h} \end{aligned} \tag{5.15}$$

If $\delta_r = 0$, then $\{\Lambda^{(r)}(x), l_r\}$ is a minimum solution still at the $(r+1)$-th step and $\{\Lambda^{(r)}(x), l_r\} = \{\Lambda^{(r+1)}(x), l_{r+1}\}$. If $\delta_r \neq 0$ we have to find a new minimum solution. In order to understand how to look for it, we have first to introduce two lemmas and a theorem.

Theorem 20 (Lemma)**.** *Let $\{\Lambda^{(r)}(x), l_r\}$ be the minimum solution for the first r syndromes and let $\delta_r \neq 0$; let $\{\Lambda^{(m)}(x), l_m\}$ be a solution (not necessarily minimum) for the first m syndromes, with $1 \leq m < r$ and $\delta_m \neq 0$. Then*

$$\{\Lambda^{(r+1)}(x), l_{r+1}\} \quad \textit{with} \quad \begin{array}{l} \Lambda^{(r+1)}(x) \doteq \Lambda^{(r)}(x) - \delta_r \delta_m^{-1} x^{r-m} \Lambda^{(m)}(x) \\ l_{r+1} \doteq max\{l_r, l_m + r - m\} \end{array}$$

is a solution for the first $r+1$ syndromes.

Proof. First of all we can notice that for hypotheses it holds that:

$$\sum_{h=0}^{l_r} \Lambda_h^{(r)} S_{i-h} = \begin{cases} 0 & i = l_r + 1, l_r + 2, \ldots, r \\ \delta_r \neq 0 & i = r+1 \end{cases}$$
$$\sum_{h=0}^{l_m} \Lambda_h^{(m)} S_{i-h} = \begin{cases} 0 & i = l_m + 1, l_m + 2, \ldots, m \\ \delta_m \neq 0 & i = m+1 \end{cases}$$

$\{\Lambda^{(r+1)}(x), l_{r+1}\}$ is a solution for $S_1, S_2, \ldots, S_{r+1}$ if

$$\sum_{h=0}^{l_{r+1}} \Lambda_h^{(r+1)} S_{i-h} = 0 \quad \text{for } i = l_{r+1} + 1, l_{r+1} + 2, \ldots, r+1$$

If we re-write the (5.13) using definitions given in the hypotheses, we will

get:

$$\begin{aligned}\sum_{h=0}^{l_{r+1}} \Lambda_h^{(r+1)} S_{i-h} &= \sum_{h=0}^{l_{r+1}} S_{i-h}[\Lambda_h^{(r)} - \delta_r \delta_m^{-1} x^{r-m} \Lambda_h^{(m)}] \\ &= \sum_{h=0}^{l_{r+1}} S_{i-h}\Lambda_h^{(r)} - \sum_{h=0}^{l_{r+1}} S_{i-h}\delta_r\delta_m^{-1}x^{r-m}\Lambda_h^{(m)} \\ &= \sum_{h=0}^{l_r} S_{i-h}\Lambda_h^{(r)} - \sum_{h=0}^{l_m} S_{i-h}\delta_r\delta_m^{-1}x^{r-m}\Lambda_h^{(m)} = 0\end{aligned}$$

Third equality holds because for definition polynomials $\Lambda^{(r)}(x)$ and $\Lambda^{(m)}(x)$ have maximum degree respectively l_r and l_m and thus $\Lambda_{l_r+1} = \Lambda_{l_r+2} = \ldots = 0$ and $\Lambda_{l_m+1} = \Lambda_{l_m+2} = \ldots = 0$.

Since multiplying a polynomial for x^k means traslating its coefficients of k steps towards the most significant positions, coefficients $\Lambda_0^{(m)}, \Lambda_1^{(m)}, \ldots, \Lambda_{l_m}^{(m)}$ become $\Lambda_{r-m}^{(m)}, \Lambda_{r-m+1}^{(m)}, \ldots, \Lambda_{r-m+l_m}^{(m)}$. Therefore we can omit the factor x^{r-m} and hold the same index h going from 0 to l_m just by changing the subscripts of the coefficients into $h-(r-m)$. The equivalent expression is:

$$\sum_{h=0}^{l_r} S_{i-h}\Lambda_h^{(r)} - \delta_r\delta_m^{-1} \sum_{h=r-m}^{l_m+(r-m)} S_{i-h}\Lambda_{h-(r-m)}^{(m)}$$

and, applying the parametric substitution $j \doteq h-(r-m)$:

$$\sum_{h=0}^{l_r} S_{i-h}\Lambda_h^{(r)} - \delta_r\delta_m^{-1} \sum_{j=0}^{l_m} S_{i-j-(r-m)}\Lambda_j^{(m)}$$

As above seen, the first summation vanishes for $l_r + 1 \leq i \leq r$, while the second one vanishes for $l_m+1 \leq i-(r-m) \leq m$, or rather: $l_m+(r-m)+1 \leq i \leq r$. Therefore, summations vanish for $i \leq r$, but they are both zero only if $i \geq max\{l_r, l_m + r - m\} + 1$. This means that, under the condition exposed in the hypotheses

$$\sum_{h=0}^{l_{r+1}} \Lambda_h^{(r+1)} S_{i-h} = 0 \quad \text{for } max\{l_r, l_m + r - m\} + 1 \leq i \leq r$$

This conludes the proof that $\{\Lambda^{(r+1)}(x), l_{r+1}\}$, found as above described, is still a solution for the first r syndromes, but what about the $(r+1)$-th

one? For $i = r + 1$ we will get:

$$\begin{aligned}\sum_{h=0}^{l_{r+1}} \Lambda_h^{(r+1)} S_{i-h} &= \sum_{h=0}^{l_r} \Lambda_h^{(r)} S_{r+1-h} - \delta_r \delta_m^{-1} \sum_{j=0}^{l_m} \Lambda_j^{(m)} S_{m+1-j} \\ &= \delta_r - \delta_r \delta_m^{-1} \delta_m \\ &= \delta_r - \delta_r = 0\end{aligned}$$

where the (5.15) has been used in the second equality. We can now assert that $\{\Lambda^{(r+1)}(x), l_{r+1}\}$ is a solution also for the $(r+1)$-th syndrome, but it may not be the minimum one: since $\{\Lambda^{(m)}(x), l_m\}$ is not a minimum solution, l_m could be greater than necessary and, by computing $l_{r+1} = max\{l_r, l_m + r - m\}$ we could find a value that is not the minimum allowed. □

Lemma 20 tells us that, given a minimum solution for the r-th step and a generic solution for one of the previous stages, we are able to find a solution, not always minimum, for the $(r + 1)$-th step.

Theorem 21 (Lemma)**.** *Let $\{\Lambda^{(r)}(x), l_r\}$ be the minimum solution for the first r syndromes, with $\delta_r \neq 0$; let $\{\Lambda^{(r+1)}(x), l_{r+1}\}$ be any solution for the first $r + 1$ syndromes (not necessarily minimum); $\{\Lambda^{(m)}(x), l_m\}$, with $1 \leq m < r$ such that:*

$$\begin{aligned}&ax^{r-m}\Lambda^{(m)}(x) = \Lambda^{(r+1)}(x) - \Lambda^{(r)}(x) \quad a \neq 0, \Lambda_0^{(m)} = 1 \\ &l_m = l_{r+1} - (r - m)\end{aligned}$$

is a solution for the first m syndromes and $\delta_m = 0$.

Proof. For the hypotheses it holds that:

$$\begin{aligned}&\sum_{h=0}^{l_r} \Lambda_h^{(r)} S_{i-h} = \begin{cases} 0 & i = l_r + 1, l_r + 2, \ldots, r \\ \delta_r \neq 0 & i = r + 1 \end{cases} \\ &\sum_{h=0}^{l_{r+1}} \Lambda_h^{(r+1)} S_{i-h} = 0 \quad i = l_{r+1} + 1, l_{r+1} + 2, \ldots, r + 1\end{aligned}$$

with $l_r \leq l_{r+1}$ because $\{\Lambda^{(r)}(x), l_r\}$ is a minimum solution.

Subtracting the two above expressions member to member:

$$\sum_{h=0}^{l_{r+1}} \left[\Lambda_h^{(r+1)} - \Lambda_h^{(r)}\right] S_{i-h} = \begin{cases} 0 & i = l_{r+1} + 1, l_{r+1} + 2, \ldots, r \\ -\delta_r & i = r + 1 \end{cases} \tag{5.16}$$

because both $\Lambda^{(r)}(x)$ and $\Lambda^{(r+1)}(x)$ provide a solution for the first r syndromes, while the second result is given by (5.15).

Let's call m the integer such that $r-m$ is the number of consecutive zero coefficients $\left[\Lambda_h^{(r+1)} - \Lambda_h^{(r)}\right]$ of the syndromes S_{i-h} in (5.16); we can be sure there is at least one zero coefficient because, for $h = 0$, we have:

$$S_i[\Lambda_0^{(r+1)} - \Lambda_0^{(r)}] = S_i[1-1] = S_i \cdot 0$$

Moreover, let's call a the first non-zero coefficient. Then we will get:

$$\Lambda_0^{(r+1)} - \Lambda_0^{(r)} = \begin{cases} 0 & h = 0, 1, \ldots, r-m-2, r-m-1 \\ a & h = r-m \end{cases}$$

and then:

$$\sum_{h=0}^{l_{r+1}} \left[\Lambda_h^{(r+1)} - \Lambda_h^{(r)}\right] S_{i-h} = \begin{cases} 0 & i = l_{r+1}+1, l_{r+1}+2, \ldots, r \\ -\delta_r & i = r+1 \end{cases}$$

which, after applying the parametric substitution $k \doteq h-(r-m)$, $j \doteq i-(r-m)$ and $l_m \doteq l_{r+1}-(r-m)$, becomes:

$$\begin{aligned}
&\sum_{k=0}^{l_{r+1}-(r-m)} S_{i-k-(r-m)} \left[\Lambda_{k+(r-m)}^{(r+1)} - \Lambda_{k+(r-m)}^{(r)}\right] = \begin{cases} 0 & i = l_{r+1}+1, l_{r+1}+2, \ldots, r \\ -\delta_r & i = r+1 \end{cases} \\
&\sum_{k=0}^{l_{r+1}-(r-m)} S_{j+(r-m)-k-(r-m)} \left[\Lambda_{k+(r-m)}^{(r+1)} - \Lambda_{k+(r-m)}^{(r)}\right] = \begin{cases} 0 & j+(r-m) = l_{r+1}+1, \ldots, r \\ -\delta_r & j+(r-m) = r+1 \end{cases} \\
&\sum_{k=0}^{l_{r+1}-(r-m)} S_{j-k} \left[\Lambda_{k+(r-m)}^{(r+1)} - \Lambda_{k+(r-m)}^{(r)}\right] = \begin{cases} 0 & j = l_{r+1}+1-(r-m), \ldots, r-(r-m) \\ -\delta_r & j = r+1-(r-m) \end{cases} \\
&\sum_{k=0}^{l_m} S_{j-k} \left[\Lambda_{k+(r-m)}^{(r+1)} - \Lambda_{k+(r-m)}^{(r)}\right] = \begin{cases} 0 & j = l_m+1, l_m+2, \ldots, m \\ -\delta_r & j = m+1 \end{cases}
\end{aligned} \tag{5.17}$$

We have, thus, defined a new polynomial $\Lambda^{(m)}(x)$ with $\Lambda_k^{(m)} = a^{-1}[\Lambda_{k+(r-m)}^{(r+1)} - \Lambda_{k+r-m}^{(r)}]$ and we can re-write the (5.17) as

$$\sum_{k=0}^{l_m} S_{j-k}\Lambda_k^{(m)} = \begin{cases} 0 & j = l_m+1, l_m+2, \ldots, m \\ -a^{-1}\delta_r & j = m+1 \end{cases}$$

Since $a \neq 0$, the couple $\{\Lambda^{(m)}(x), l_m\}$ represents a solution for the first m syndromes, with $\delta_m \neq 0$. □

Lemma 21 has not got a great relevance if considered alone, but it is fundamental in the proof of the following result.

Theorem 22. *Let $\{\Lambda^{(r)}(x), l_r\}$ be a minimum solution for the first r syndromes. Among minimum solutions of the previous steps $(1, 2, \ldots, r-1)$ let $\{\Lambda^{(m)}(x), l_m\}$, with $1 \leq m < r$, be one with $\delta_m \neq 0$ and the one with the largest value $m - l_m$. Then,*

if $\delta_r \neq 0$ the couple $\{\Lambda^{(r+1)}(x), l_{r+1}\}$ with

$$\begin{aligned} \Lambda^{(r+1)}(x) &\doteq \Lambda^{(r)}(x) - \delta_r \delta_m^{-1} x^{r-m} \Lambda^{(m)}(x) \\ l_{r+1} &\doteq max\{l_r, l_m + r - m\} \end{aligned}$$

is a minimum solution for the first $r+1$ syndromes;

if $\delta_r = 0$ the couple $\{\Lambda^{(r+1)}(x), l_{r+1}\}$ with $\Lambda^{(r+1)}(x) = \Lambda^{(r)}(x)$ and $l_{r+1} = l_r$ is a minimum solution for the first $r+1$ syndromes.

Proof. Since all the hypotheses of Lemma 20 are verified, then $\Lambda^{(r+1)}(x)$ is a solution for the first $r+1$ syndromes. Our purpose is proving that it is a minimum solution, too. Let's distinguish two cases for the event $\delta_r \neq 0$: $l_r \geq l_m + r - m$ and $l_r \leq l_m + r - m$.

$\mathbf{l_r \geq l_m + r - m}$ In this case $l_{r+1} = l_r$, thus since $\{\Lambda^{(r)}(x), l_r\}$ is a minimum solution for the first r syndromes and l_{r+1} can't be lower than l_r, also $\{\Lambda^{(r+1)}(x), l_{r+1}\}$ must be a minimum solution for the first $r+1$ syndromes.

$\mathbf{l_r \leq l_m + r - m}$ In this case $l_{r+1} = l_m + r - m$. Let's proceed by contradiction, affirming that there exists a solution $\{\tilde{\Lambda}^{(r+1)}(x), \tilde{l}_{r+1}\}$ with $\tilde{l}_{r+1} < l_{r+1}$ for the first $r+1$ syndromes. Therefore, Lemma 21 tells us that it must exist an integer $\tilde{m}$, $1 \leq \tilde{m} < r$, such that there is a solution $\{\Lambda^{(\tilde{m})}(x), l_{\tilde{m}}\}$ for the first $\tilde{m}$ syndromes with $\delta_{\tilde{m}} \neq 0$ and $l_{\tilde{m}} = \tilde{l}_{r+1} - (r - \tilde{m})$.

For the hypotheses, $\tilde{m} - l_{\tilde{m}} \leq m - l_m$, then

$$\tilde{l}_{r+1} = l_{\tilde{m}} + (r - \tilde{m}) = r - (\tilde{m} - l_{\tilde{m}}) \geq r - (m - l_m) = l_{r+1}$$

that is: $\tilde{l}_{r+1} \geq l_{r+1}$, which is in contradiction with the above condition $\tilde{l}_{r+1} < l_{r+1}$. This refutes the absurd statement saying that there is a minimum solution for the first $r+1$ syndrome different from $\{\Lambda^{(r+1)}(x), l_{r+1}\}$.

On the other hand, if $\delta_r = 0$, then $\{\Lambda^{(r)}(x), l_r\}$ is already a solution for the first $r+1$ syndromes and, being it minimum for the first r, it must be minimum for the first $r+1$ as well. □

Theorem 22 gives us a way to find a minimum solution at the r-th step just by knowing the minimum solution for one of the previous steps. Now, all we need is a starting point.

To start our algorithm we need an initial minimum solution and a minimum solution for one of the previous passages. We may assume that the first step is the one at which we have not yet considered any syndrome and the related minumum solution can be referred to as: $\{\Lambda^{(0)}, l_0\}$. Note that, as defined in Theorem 22, degree of the *error locator polynomial* can only increase or remain equal between two consecutive steps, but it can never decrease. Thus, if we chose $\Lambda^{(0)}(x)$ with degree greater than 1, it would have at least one zero, and then there would be at least a non-zero component of the error vector $\mathbf{e}$, no matter the step or the found syndromes. This brings to a contradiction in the case of all null syndromes, that is if the received word is a code word, because the final *error locator polynomial* would find an error even if the sent word has not been corrupted. Therefore, $\Lambda^{(0)}(x)$ must have degree 0, that is $\Lambda^{(0)}(x)$ is a non-zero constant: $l_0 = 0$ and $\Lambda^{(0)}(x) = \kappa$, with $\kappa \in GF(2^m),\ \kappa \neq 0$. However, since $\Lambda^{(0)}(x)$ is an *error locator polynomial*, for definition given in (5.5) it must have 1 as known term. Thus, we can set: $\Lambda^{(0)}(x) = 1$. From a simple computation we can find δ_0 to be:

$$\delta_0 = \sum_{h=0}^{l_0} \Lambda_h^{(0)} S_{0+1-h} = \sum_{h=0}^{0} \Lambda_h^{(0)} S_{1-h} = \Lambda_0^{(0)} S_1 = S_1$$

Through the same reasoning, and following what exposed in [16, p. 159] and [13, pp. 155-156], we can take as previous minimum solution the couple $\{\Lambda^{(-1)}(x), l_{-1}\}$, where, for what just said, $\Lambda^{(-1)}(x) = 1$ and $l_{-1} = 0$. In order to use this couple for the algorithm presented in Theorem 22, δ_{-1} must be different from zero; we can assume $\delta_{-1} = 1$. Our initial conditions are:

$$\begin{array}{lllll} \mathbf{m}: & \Lambda^{(-1)}(x) = 1 & l_{-1} = 0 & \delta_{-1} = 1 \\ \mathbf{r}: & \Lambda^{(0)}(x) = 1 & l_0 = 0 & \delta_0 = S_1 \end{array}$$

Thanks to this starting point we can perform the whole *Berlekamp-Massey algorithm* and find the *error locator polynomial*: $\Lambda(x)$. Its degree will corresponds to ν, number of symbol errors.

Once the *error locator polynomial* is defined, we have to find its roots in order to know error locations. Since we are dealing with *finite fields*, the easiest way to discover the zeros is evaluating $\Lambda(x)$ in each of the elements of the *finite field* $GF(q)$. We will found ν roots, that are: $X_1^{-1}, X_2^{-1}, \ldots, X_\nu^{-1}$. Their relative inverse elements point the ν error positions: $X_1 = e_{p_1}, X_2 =$

$e_{p_2}, \ldots, X_\nu = e_{p_\nu}$. At this point we know both how many errors have occurred and where such errors have modified the transmitted word. Algorithm introduced in the next section gives us a way to compute errors magnitutes.

RS(7, 4) In order to execute the *Berlekamp-Massey algorithm*, we need a starting point: an actual minimum solution for the current step and a minimum solution for one of the previous steps. Let's assume the current step is that at which we consider 0 syndromes, that is $r = 0$, where r is our stage-counter. Of course there is not any real previous step, but we can assume we know that at -1-th step $\{\Lambda^{(-1)}(x), l_{-1}\}$ was a minimum solution, with $\delta_{-1} = 1$. Then -1 will become our initial m, that is a minimum solution for one of the previous passages. At stage $r = 0$ we have:

$$\begin{array}{lllll} \mathbf{m}: & \Lambda^{(-1)}(x) = 1 & l_{-1} = 0 & -1 - l_{-1} = -1 - 0 = -1 & \delta_{-1} = 1 \\ \mathbf{r}: & \Lambda^{(0)}(x) = 1 & l_0 = 0 & 0 - l_0 = 0 - 0 = 0 & \end{array}$$

Evaluating δ_0 we will find:

$$\delta_0 = \sum_{h=0}^{l_0} \Lambda_h^{(0)} S_{0+1-h} = \sum_{h=0}^{0} \Lambda_h^{(0)} S_{1-h} = \Lambda_0^{(0)} S_1 = S_1$$

and, since $S_1 = \alpha \neq 0$ we must find a new polynomial fitting also for the $(r+1)$-th step. By using Theorem 22:

$$\Lambda^{(1)}(x) = \Lambda^{(0)}(x) - \delta_0 \delta_{-1}^{-1} x^{0-(-1)} \Lambda^{(-1)}(x) = 1 + \alpha x$$

and $l_1 = max\{l_0, l_{-1} + 0 - (-1)\} = max\{0, 1\} = 1$. Eventually, as $0 - l_0 > -1 - l_{-1}$, from next step $\Lambda^{(0)}(x)$ will become the new minimum solution for a previous passage. Therefore, at step $r = 1$ we have:

$$\begin{array}{lllll} \mathbf{m}: & \Lambda^{(0)}(x) = 1 & l_0 = 0 & 0 - l_0 = 0 - 0 = 0 & \delta_0 = \alpha \\ \mathbf{r}: & \Lambda^{(1)}(x) = 1 + \alpha x & l_1 = 1 & 1 - l_1 = 1 - 1 = 0 & \end{array}$$

Let's check whether $\Lambda^{(1)}(x)$ is a suitable solution for $r + 1$ by evaluating δ_1:

$$\delta_1 = \sum_{h=0}^{l_1} \Lambda_h^{(1)} S_{1+1-h} = \sum_{h=0}^{1} \Lambda_h^{(1)} S_{2-h} = \Lambda_0^{(1)} S_2 + \Lambda_1^{(1)} S_1 = S_2 + \alpha S_1 = 1 + \alpha^2 = \alpha^6 \neq 0$$

Using the algorithm we will find: $\Lambda^{(2)}(x) = 1 + \alpha^6 x$ and $l_2 = 1$. Since now we have two minimum solutions with the same value of $m - l_m$, that is $0 - l_0 = 1 - l_1 = 0$, we can choose any one the two solution to carry on; even

though the choice of two different m will bring to different passages, it can be proved that the final solution will always be the same.

Next passage, for $r = 2$, has:

$$\begin{array}{lllll} \mathbf{m}: & \Lambda^{(0)}(x) = 1 & l_0 = 0 & 0 - l_0 = 0 - 0 = 0 & \delta_0 = \alpha \\ \mathbf{r}: & \Lambda^{(2)}(x) = 1 + \alpha^6 x & l_2 = 1 & 2 - l_2 = 2 - 1 = 1 & \end{array}$$

with $\delta_2 = \alpha^3 \neq 0$. Therefore we have to proceed with the algorithm and, after finding $\Lambda^{(3)}(x) = 1 + \alpha^6 x + \alpha^2 x^2$, $l_3 = 2$ and the new $m = 1$, step $r = 3$ will be:

$$\begin{array}{lllll} \mathbf{m}: & \Lambda^{(2)}(x) = 1 + \alpha^6 x & l_2 = 1 & 2 - l_2 = 2 - 1 = 1 & \delta_0 = \alpha^3 \\ \mathbf{r}: & \Lambda^{(3)}(x) = 1 + \alpha^6 x + \alpha^2 x^2 & l_3 = 2 & 3 - l_3 = 3 - 2 = 1 & \end{array}$$

Let's see what happens now. Evaluating δ_3 we will find:

$$\delta_3 = \sum_{h=0}^{2} \Lambda_h^{(3)} S_{4-h} = \alpha^5 + \alpha^6 \alpha^4 + \alpha^2 = 0$$

This means that solution for the third step still fits for the forth one and the *error locator polynomial* does not need to be updated. We could expect something like this, because, knowing that only two symbols have been corrupted, we could imagine that $\Lambda(x)$ would have only two roots. It would have been possible, however, that $\Lambda(x)$ changed coefficients without increasing its degree.

Finally we have found $\Lambda(x) = 1 + \alpha^6 x + \alpha^2 x^2 = 1 + 5x + 4x^2$, from which we deduce $\nu = 2$. We have, now, to discover its roots. Testing every element of $GF(2^3)$ we realise that the only values that make it vanish are: $X_1^{-1} = 1$ and $X_2^{-1} = \alpha^5$, whose inverse elements are $X_1 = 1$ and $X_2 = \alpha^2$, which are related to powers 0 and 2. Hence, we have found that the received polynomial $\tilde{c}(x)$ has got corrupted coefficients of the known term and x^2. The error polynomial has the following guise:

$$\mathbf{e} = [X\ 0\ X\ 0\ 0\ 0\ 0]$$

5.3 Forney Algorithm

Once the *error locator polynomial* and its ν roots have been found, we know the number of components of the error vector $\mathbf{e}$ and their positions. Last stage is finding their values, that is $Y_1 = e_{p_1}, Y_2 = e_{p_2}, \ldots, Y_\nu = e_{p_\nu}$. Let's define the *syndrome polynomial*:

$$S(x) \doteq \sum_{i=1}^{2t} S_i x^{i-1} = S_1 + S_2 x + S_3 x^2 + \ldots + S_{2t-1} x^{2t-2} + S_{2t} x^{2t-1} \quad (5.18)$$

Remembering that:

$$S_i = \sum_{l=1}^{\nu} Y_l X_l^i$$

we can re-write the (5.18) as:

$$S(x) \doteq \sum_{i=1}^{2t} S_i x^{i-1} = \sum_{i=1}^{2t} \left[x^{i-1} \sum_{l=1}^{\nu} Y_l X_l^i \right] = \sum_{i=1}^{2t} \sum_{l=1}^{\nu} Y_l X_l^i x^{i-1} \tag{5.19}$$

and, using *error locator polynomial* defined in (5.5), we can finally introduce the polynomial $\Omega(x)$:

$$\Omega(x) \doteq \mathcal{R}_{x^{2t}}[S(x)\Lambda(x)] \tag{5.20}$$

This polynomial, which has no coefficient with degree greater than $2t$, will permit us to find error components $Y_1, Y_2, \ldots, Y_\nu$.

First of all, we can re-write $\Omega(x)$ using expressions (5.19) and (5.5):

$$\begin{aligned}
\Omega(x) &= \mathcal{R}_{x^{2t}} \left[\left(\sum_{i=1}^{2t} \sum_{l=1}^{\nu} Y_l X_l^i x^{i-1} \right) \left(\prod_{j=1}^{\nu} (1 - X_j x) \right) \right] \\
&= \mathcal{R}_{x^{2t}} \left[\left(\sum_{l=1}^{\nu} Y_l X_l \sum_{i=1}^{2t} X_l^{i-1} x^{i-1} \right) \left(\prod_{j=1}^{\nu} (1 - X_j x) \right) \right] \\
&= \mathcal{R}_{x^{2t}} \left[\left(\sum_{l=1}^{\nu} Y_l X_l \sum_{i=1}^{2t} (X_l x)^{i-1} \right) \left(\prod_{j=1}^{\nu} (1 - X_j x) \right) \right] \\
&= \mathcal{R}_{x^{2t}} \left[\left(\sum_{l=1}^{\nu} Y_l X_l \sum_{i=1}^{2t} (X_l x)^{i-1} \right) (1 - X_l x) \left(\prod_{j=1, j\neq l}^{\nu} (1 - X_j x) \right) \right] \\
&= \mathcal{R}_{x^{2t}} \left[\sum_{l=1}^{\nu} Y_l X_l (1 - X_l x) \sum_{i=1}^{2t} (X_l x)^{i-1} \prod_{j=1, j\neq l}^{\nu} (1 - X_j x) \right]
\end{aligned}$$

Term $(1 - X_l x) \sum_{i=1}^{2t} (X_l x)^{i-1}$ represents a factorization of the expression $1 - (X_l x)^{2t}$; as a matter of fact, considering the easier expression $1 - z^y$, it holds that:

$$\begin{aligned}
1 - z^y &= 1 + (z + z^2 + \ldots + z^{y-1}) - (z + z^2 + \ldots + z^{y-1}) - z^y \\
&= (1 + z + z^2 + \ldots + z^{y-1}) - z(1 + z + z^2 + \ldots + z^{y-1}) \\
&= (1 - z)(1 + z + z^2 + \ldots + z^{y-1}) \\
&= (1 - z) \sum_{i=1}^{y} z^{i-1}
\end{aligned}$$

Therefore, $\Omega(x)$ can be expressed as:

$$\begin{aligned}\Omega(x) &= \mathcal{R}_{x^{2t}}\left[\underbrace{\left(\sum_{l=1}^{\nu} Y_l X_l\right)}_{\text{constant } \lambda}\underbrace{\left(1-(X_l x)^{2t}\right)}_{\text{A(x)}}\underbrace{\left(\prod_{j=1,j\neq l}^{\nu}(1-X_j x)\right)}_{\text{B(x)}}\right] \\ &= \mathcal{R}_{x^{2t}}\left[\lambda A(x)B(x)\right] \\ &= \lambda\mathcal{R}_{x^{2t}}\left[\mathcal{R}_{x^{2t}}\left[A(x)\right]\mathcal{R}_{x^{2t}}\left[B(x)\right]\right]\end{aligned}$$

Now, analyzing $A(x)$, we can see that: $\mathcal{R}_{x^{2t}}\left[A(x)\right] = \mathcal{R}_{x^{2t}}\left[1-(X_l x)^{2t}\right] = 1$ and thus:

$$\Omega(x) = \lambda\mathcal{R}_{x^{2t}}\left[B(x)\right] = \lambda B(x) = \sum_{l=1}^{\nu} Y_l X_l \prod_{j=1,j\neq l}^{\nu}(1-X_j x) \tag{5.21}$$

because $B(x)$ has degree $\nu \leq t < 2t$. Evaluating equation (5.21) in a generic X_k^{-1}, that is in one of the roots of the *error locator polynomial* $\Lambda(x)$, we will get:

$$\Omega(X_k^{-1}) = \sum_{l=1}^{\nu} Y_l X_l \prod_{j=1,j\neq l}^{\nu}(1-X_j X_k^{-1})$$

Every term of the summation vanishes when $j = k$. The only term that does not become null is the one for $l = k$, because the term of the product that should vanish is skipped, the subscript going from 1 to ν, avoiding l. Then the expression becomes:

$$\Omega(X_k^{-1}) = Y_k X_k \prod_{j=1,j\neq k}^{\nu}(1-X_j X_k^{-1})$$

and therefore:

$$Y_k = \frac{X_k^{-1}\Omega(X_k^{-1})}{\prod_{j=1,j\neq k}^{\nu}(1-X_j X_k^{-1})} \tag{5.22}$$

By using equation (5.22) we can evaluate $Y_k = e_{p_k}$ for $k = 1, 2, \ldots, \nu$. There is, however, an easier equation to compute Y_k, that will be here illustrated. Let's start by deriving the *error locator polynomial* $\Lambda(x)$ using the

chain rule

$$\begin{aligned}\Lambda'(x) &= \frac{d}{dx}\prod_{i=1}^{\nu}(1-X_i x)\\ &= -X_1(1-X_2x)\cdot\ldots\cdot(1-X_\nu x)+(1-X_1x)\frac{d}{dx}[(1-X_2x)\cdot\ldots\cdot(1-X_\nu x)]\\ &= \ldots\\ &= -\sum_{l=1}^{\nu}X_l\prod_{i=1,i\neq l}^{\nu}(1-X_i x) \qquad (5.23)\end{aligned}$$

and then, evaluating the (5.23) in a generic X_k we will find:

$$\Lambda'(X_k^{-1}) = -\sum_{l=1}^{\nu}X_l\prod_{i=1,i\neq l}^{\nu}(1-X_iX_k^{-1}) = -X_k\prod_{i=1,i\neq k}^{\nu}(1-X_iX_k^{-1}) \qquad (5.24)$$

and, in conclusion, putting together (5.24) and (5.22):

$$Y_k = -\frac{\Omega(X_k^{-1})}{\Lambda'(X_k^{-1})} \qquad (5.25)$$

Notice that, since $\Omega(x)$ is the remainder of the division of a polynomial by an only polynomial term x^{2t}, this allows a very fast computation of the remainder, because we can see the dividend polynomial as it follows:

$$\begin{aligned}p(x) &= k_nx^n+k_{n-1}x^{n-1}+\ldots+k_{2t+1}x^{2t+1}+k_{2t}x^{2t}+k_{2t-1}x^{2t-1}+\ldots+k_1x+k_0\\ &= (k_nx^{n-2t}+k_{n-1}x^{n-2t-1}+\ldots+k_{2t+1}x+k_{2t})x^{2t}+k_{2t-1}x^{2t-1}+\ldots+k_1x+k_0\end{aligned}$$

and the remainder modulo x^{2t} of this polynomial is clearly:

$$\mathcal{R}_{x^{2t}}[p(x)] = k_{2t-1}x^{2t-1}+\ldots+k_1x+k_0$$

Then, in order to evaluate $\Omega(x)$ we just have to find the polynomial $S(x)\Lambda(x)$ and then consider only the term with power smaller than $2t$.

5.3.1 Finite field polynomial derivative

As explained in important algebra texts as [3, pp. 465-466] or [8, pp. 119-121], polynomial derivative with coefficients in a *Galois Field* can be evaluated as in any other domain:

$$\begin{aligned}p(x) &= k_nx^n+k_{n-1}x^{n-1}+\ldots+k_1x+k_0\\ p'(x) &= nk_nx^{n-1}+(n-1)k_{n-1}x^{n-1}+\ldots+k_1 \qquad (5.26)\end{aligned}$$

In *finite fields* in the form $GF(2^m)$ this brings to several semplifications; the generic term $k_n x^n$ becomes:

$$nk_n x^{n-1} = \underbrace{(k_n + k_n + \ldots + k_n)}_{n \text{ times}} x^{n-1}$$

If n is even we can group the k_n in couples whose terms erase each other, because adding an element to itself always gives zero as result (see section (2.5.1)). Thus, every term with even power disappears in the first derivative of $p(x)$. On the other hand, for odd powers we can make the same reasoning: if we group the k_n in couples these will all vanishes, leaving an only coefficient. Thus, terms with odd power hold the same coefficient, but their degrees decrease by one.

Example: derivative Suppose we have to derive $p(x) = \alpha^4 z^3 + \alpha z^2 + \alpha^2 z + \alpha^3$ with α primitive element for a certain field $GF(2^m)$. Following definition (5.26) we will get:

$$\begin{aligned} p'(x) &= 3\alpha^4 z^2 + 2\alpha z + \alpha^2 \\ &= (\alpha^4 + \alpha^4 + \alpha^4)z^2 + (\alpha + \alpha)z + \alpha^2 \\ &= \alpha^4 z^2 + \alpha^2 \end{aligned}$$

RS(7, 4) We will use expression (5.25) to compute coefficients of the error polynomial. First of all let's briskly find $\Omega(x)$ and $\Lambda'(x)$.

$$S(x)\Lambda(x) = (\alpha + x + \alpha^4 x^2 + \alpha^5 x^3)(1 + \alpha^6 x + \alpha^2 x^2) = \alpha + \alpha^3 x^4 + x^5$$

and, evaluated modulo $x^{2t} = x^4$:

$$\Omega(x) = \alpha$$

Using what exposed in section (5.3.1) we quickly find that:

$$\Lambda'(x) = \alpha^6$$

For pure coincidence both these polynomials are constants, thus the coefficients will have the same magnitude:

$$Y_1 = Y_2 = -\frac{\Omega(X_{1,2}^{-1})}{\Lambda'(X_{1,2}^{-1})} = \frac{\alpha}{\alpha^6} = \alpha^2 = 4$$

The error vector is:

$$\mathbf{e} = [\alpha^2\ 0\ \alpha^2\ 0\ 0\ 0\ 0]$$

and, as subtraction is equivalent to addition in *finite fields*, summing it to the received word we will get, in decimal form:

$$\hat{\mathbf{c}} = [5\ 2\ 1\ 2\ 6\ 6\ 1]$$

which is equal to the original code word $\mathbf{c}$.

Chapter 6

Matlab Implementation

In addition to this paper, a *Matlab* implementation for a *Reed-Solomon* systematic encoder and a *Berlekemp-Massey* decoder has been realised for a *RS(255, 223)* code. In this chapter we will introduce a brief presentation of the code and a short explanation of how it was thought and made.

For reasons of computational efficiency every function has been included in the same file: `rs.m`. We will browse the whole file focusing on the functioning and the logic behind every piece of code, investigating how encoding and decoding functions work. For the sake of clearness we can imagine to divide the file in several sections, each dedicated to a particular process. The file does not actually respect this division, but, for every section, we will report the relative piece of code.

6.1 Encoder initialization

The first step to the implementation of the encoder is the construction of an appropriate *Galois Field*. In order to achieve a reasonable computational speed and efficiency we will make use of several look-up tables: a moderate waste of memory allows significant improvements in terms of evaluating time for encoding functions. Five look-up tables have been built: two to store elements of $GF(256)$ in different notations and briskly reach one from another (`polTab` and `logTab`) and three to save time during the execution of fundamental operations in the *finite field* (`sumTab`, `mulTab` and `divTab`).

There are three notations to indicate *Galois Field* elements: their effective decimal value (from 0 to 255), polynomial notation, as polynomials with coefficients in $GF(2)$ ($\{0, 1\}$) with smaller degree than 8, and eventually logarithmic notation, as powers of the primitive element, which will be called α in the following. Furthermore, a convention will be used hereafter: even

though both α^0 and α^{255} refers to element "1", in this context we will use only α^{255}, while exponent "0" will refer to the null element, which would actually has power "$-\infty$".

6.1.1 Parameters

In the very beginning of the file we define and store the fundamental parameters and variables characterizing our *RS(255, 223)* code:

```
Section 1:  parameters
n = 255;
k = 223;
m = 8;
t = 16;
primPol = [1 0 1 1 1 0 0 0 1];
```

Let's have a look to each variable:

`n` : number of symbols of a code word ($n = 2^m - 1$).

`k` : number of symbols of an input word.

`m` : number of bits used to express a symbol in binary notation.

`t` : maximum number of symbol errors the code can correct.

`primPol` : primitive polynomial of degree m (it has been found in a proper table, see Appendix A). In the case of a *RS(255, 223)* code we will use: $f(x) = 1 + x^2 + x^3 + x^4 + x^8$.

6.1.2 `polTab` and `logTab`

The table of the elements, called `polTab` in the code, is a matrix of dimension $2^m \times m$, that is 256×8, in which every row represents an element of $GF(2^8)$ written in the form of a polynomial with coefficients in $GF(2)$ of maximum degree 8 (the less significant coefficient is the leftmost). The elements are ordered by successive powers of the primitive element: $0, \alpha, \alpha^2, \alpha^3, \ldots, \alpha^{255}$. The first row represents the zero element, and thus it contains only a sequence of 8 zeros; the second one represents the primitive element $\alpha = x$, so it is in the form: [0 1 0 ... 0]; the last one represents $\alpha^{255} = 1$, thus it is in the form: [1 0 ... 0]. Every cell of the vector called `logTab` contains the exponent that the primitive element α should have to

produce its relative row index. In this way, the effective value of the element can be use as index to recover its logarithmic representation.

Let's see how these tables are built: after defining the primitive polynomial of degree $m = 8$, an empty matrix of dimension $2^8 \times 8$ and an empty vector of dimension 2^8 are initialised.

Section 2: initialization of polTab and logTab

```
polTab = zeros(2^m, m);
logTab = zeros(1, 2^m);
```

All the elements will be obtained by multiplying the primitive element by itself, starting by the second row: $\alpha^2, \alpha^3, \ldots$. In the generic i-th step we have already evaluated the i-th row of the matrix and we have to compute the $(i+1)$-th one. Multiplying $a_0 + a_1 x + \ldots + a_{m-1} x^{m-1}$ by x gives as result $a_0 x + a_1 x^2 + \ldots + a_{m-1} x^m$, which is equivalent to shift all the components of the i-th row vector to the right of one position. We have now to evaluate the new polynomial modulo $f(x)$. If the rightmost element of the new $(m+1)$-th position is a "0", then the new polynomial has degree less than $m = 8$ and it is already good to be inserted in the $(i+1)$-th row. If the rightmost component is a "1", otherwise, we have to divide this polynomial by $f(x)$ and consider only the remainder. It is easy to see that the polynomial got by $x \cdot v_i(x)$, where $v_i(x)$ denotes the i-th row vector, has as maximum degree $m = 8$ and, thus, we can add and subtract the terms of degree less than 8 which compose the primitive polynomial, remembering that, since we are using polynomials with binary coefficients, adding and subtracting is equivalent to adding twice the term. In the final polynomial we can group all the terms that compose the primitive polynomial: the other ones represent $v_i(x)$ modulo $f(x)$.

Example Let be $v_i(x) = 1 + x^2 + x^3 + x^6 + x^7$. After evaluating $x \cdot v_i(x)$ we will find:

$$\tilde{v}_{i+1}(x) = x + x^3 + x^4 + x^7 + x^8$$

which is equivalent to:

$$\begin{aligned}
\tilde{v}_{i+1}(x) &= x + x^3 + x^4 + x^7 + x^8 + (1 + x^2) - (1 + x^2) \\
&= x + x^3 + x^4 + x^7 + x^8 + (1 + x^2) + (1 + x^2) \\
&= (1 + x^2 + x^3 + x^4 + x^8) + (1 + x + x^2 + x^7) \\
&= f(x) + (1 + x + x^2 + x^7)
\end{aligned}$$

$\tilde{v}_{i+1}(x)$ modulo $f(x)$ is clearly $v_{i+1}(x) = 1 + x + x^2 + x^7$. The $(i+1)$-th row will, thus, be [1 1 1 0 0 0 0 1].

All we have to do in order to obtain $\tilde{v}_{i+1}(x)$ modulo $f(x)$ is only adding primitive polynomial terms which do not appear in $\tilde{v}_{i+1}(x)$ and remove the ones that are already present. The easiest way to do that is computing a *XOR* function between vectors of the primitive polynomial and the corresponding vector of $\tilde{v}_{i+1}(x)$.

Example Revisiting the previous example, it is sufficient to make the said operation:

$$\begin{aligned} \tilde{v}_{i+1}(x) &\rightarrow [0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1]\ XOR \\ f(x) &\rightarrow [1\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 1]\ = \\ v_{i+1}(x) &\rightarrow [1\ 1\ 1\ 0\ 0\ 0\ 0\ 1\ 0] \end{aligned}$$

Every polynomial is stored in the row whose index represents the exponent of the relative power of the primitive element. This allows a fast passage from logarithmic notation to polynomial notation. Once the binary polynomial is complete, we can convert it into decimal form as it will explained below and save the current logarithmic value at the index given by the decimal value. By iterating this proceeding for all the $2^m = 256$ elements we will fill the whole matrix `polTab` and the whole vector `logTab` with all the elements of $GF(256)$. In this way by using `polTab` we can quickly pass from logarithmic notation to polynomial notation and by using `logTab` from decimal value to logarithmic notation. Furthermore, one can easily pass from polynomial notation to the effective value acting a transformation from binary to decimal.

We have decided not to use the *Matlab* native function `bi2de()` to convert a binary vector into a decimal integer because it takes a lot of time in order to check many necessary conditions before executing the conversion. After several empirical tests, this has revealed to be the most expensive piece of code of the whole program in terms of time, also because it is called hundreds of thousands of times. Therefore, we have settled to omit any correctness check, assuming there are no errors in the rest of the code which could make this operation fail. The conversion is made simply by multiplying the binary vector by a precompiled vertical vector (`binTab`) created in the very beginning of the file and containing all the powers of 2, from 2^0 to 2^8. Matrix product has been preferred to nested *for* cycles to execute this operation. Transformation from binary to decimal closes the circle which allows us to quickly pass from one notation to another using only two matrices.

Here is the complete piece of code providing for the creation of the elements of *GF(256)*:

Section 3: filling polTab and logTab

```
binTab = zeros(m, 1);
for i = 1 : m
    binTab(i) = 2^(i - 1);
end

% ...

x = zeros(1, m + 1);
x(1) = 1;
for i = 2 : 2^m
   x = circshift(x, [1, 1]);
   if x(m + 1) == 1
       x = xor(primPol, x);
   end
   polTab(i, :) = x(1:m);
   dec = (x(1:m))*binTab;
   logTab(dec + 1) = i - 1;
end
```

6.1.3 sumTab, mulTab and divTab

In this section we create the three fundamental look-up tables for *finite field* operations. To create `sumTab` every element i of $GF(256)$ is summed to every other element j and the result is then stored in a matrix of dimension 256×256 at indexes $(i+1, j+1)$: *Matlab* indexes go from 1 to 256, but *Galois Field* elements go from 0 to 255. In order to sum two values we need first to bring them into binary polynomial form and then apply the *XOR* function between the vectors. This operation also provides for subtraction since, thanks to operations modulo 2 in $GF(2)$: $1 + 1 = 1 - 1 = 0$. To execute the sum between two elements a and b is then sufficient to recover the element at coordinates $(a+1, b+1)$.

Here is the code:

Section 4: creation of sumTab

```
sumTab = zeros(n + 1, n + 1);
for i = 1 : n + 1
    p1 = polTab(logTab(i) + 1, :);
    for j = i : n + 1
        p2 = polTab(logTab(j) + 1, :);
        sumTab(i, j) = (xor(p1, p2))*binTab;
```

```
        if i ~= j
            sumTab(j, i) = (xor(p1, p2))*binTab;
        end
    end
end
```

To create `mulTab` every element i of $GF(256)$ is multiplied by every other element j. The result is then stored in a matrix of dimension 256×256 at indexes $(i+1, j+1)$. In order to multiply two elements, first the two values are brought to logarithmic form and then, since $\alpha^i \cdot \alpha^j = \alpha^{i+j}$, we can add the two values to get the index of `polTab` at which the resulting element is stored. If the resulting exponent is greater than $2^m - 1 = 255$, we have just to subtract from it the integer 255 until it decreases under the maximum allowed value: 255 itself.

Example

$$\alpha^{134} \cdot \alpha^{195} = \alpha^{134+195} = \alpha^{329} = \alpha^{255+74} = \alpha^{255} \cdot \alpha^{74} = 1 \cdot \alpha^{74} = \alpha^{74}$$

As stated in the beginning of this chapter (see section (6.1)) by convention "0" represents the null element and "255" the unity element, otherwise it would be impossible to quickly obtain the zero element from the tables. To execute multiplication between two elements a and b is then sufficient to recover the element at coordinates $(a+1, b+1)$.

Here is the code:

Section 5: creation of mulTab

```
mulTab = zeros(n + 1, n + 1);
for i = 1 : n + 1
    p1 = logTab(i);
    for j = i : n + 1
        p2 = logTab(j);
        if p1 == 0 || p2 == 0
            mulTab(i, j) = 0;
            mulTab(j, i) = 0;
        else
            res = p1 + p2;
            while res >= 2^m
                res = res - n;
            end
            mulTab(i, j) = (polTab(res + 1, :))*binTab;
            mulTab(j, i) = (polTab(res + 1, :))*binTab;
        end
```

```
    end
end
```

To create `divTab` every element i of $GF(256)$ is divided by every other element j. The result is then stored in a matrix of dimension 256×256 at indexes $(i+1, j+1)$. The proceeding to divide two elements is very similar to the multiplication one and we will make use of logarithmic form again. Dividing by an element is equivalent to multiplying by its inverse and the inverse of an element is the value such that, multiplied by the element itself, gives 1 as result. Thus, in order to find the inverse of an element it is sufficient to compute its logarithmic notation as $255 - i$ where i is the logarithmic notation of the original value:

$$\alpha^j \div \alpha^i = \alpha^j \cdot \alpha^{-i} = \alpha^j \cdot \alpha^{255-i} = \alpha^{255+j-i}$$

From this point forward we can evaluate the expression just as it was a multiplication, following what exposed above. To execute division between two elements a and b is then sufficient to recover the element at coordinates $(a+1, b+1)$. During the building of this matrix we store also some forbidden resultss, like division by zero; assuming that such prohibited computations will be never performed in this code, we have decided to store value -1 in cells corresponding to $\alpha^j \div 0$ so that to get a non-sense value in the case this operation was executed: we preferred a program failure to a valid result obtained by impossible operations and, in this way, we are sure that code will fail if cells like these are called. Here is the code:

Section 6: creation of divTab

```
divTab = zeros(n + 1, n + 1);
for i = 1 : n + 1
    for j = 1 : n + 1
        if j == 1
            divTab(i, j) = -1;
        else
            if i == 1
                divTab(i, j) = 0;
            else
                invJ = logTab(j);
                if invJ ~= n
                    invJ = n - invJ;
                end
                p1 = logTab(i);
                res = p1 + invJ;
                while res >= 2^m
```

```
                    res = res - n;
                end
                divTab(i, j) = (polTab(res + 1, :))*binTab;
            end
        end
    end
end
```

6.2 Generator polynomial

After completing the creation of the *Galois Field*, we have now to build the appropriate generator polynomial for the code. The polynomial is evaluated by multiplying $2t = 32$ factors in the form $(z + \beta)$, where β are consecutive powers of the primitive element α. Initially the generator polynomial is $z+2$, since the primitive element corresponds to the vector [0 1 0 ... 0] and it can be always identified with the element "2" after a conversion from binary to decimal. By using a *for* cycle with $2t - 1$ iterations the polynomial is completed: at the generic i-th step the successive factor $(z+\alpha^i)$ is evaluated and multiplied by the current generator polynomial, given by the product of the first $i - 1$ factors. After $2t - 1$ iterations the polynomial is the product of $2t$ consecutive factors.

To multiply two polynomials with coefficients in $GF(2^m)$, it is comfortable first writing polynomials as two vectors with leftmost element the less significant. The first step is initializing an empty vector which will be the result. If the two original vectors has got respectively x and y elements, the polynomials have degree $x - 1$ and $y - 1$, thus the final polynomial will have degree at most $(x - 1) + (y - 1)$. In order to contain a s-degree polynomial, a vector must have $s + 1$ positions, then our final vector must have $(x - 1) + (y - 1) + 1 = x + y - 1$ positions.

To compute the multiplication we have to scan the two whole vectors and multiply every element of the first by every element of the second; this is implemented through two nested *for* cycles, one for each vector. In every step we can evaluate the power related to the current coefficient and then add it in the proper position of the final vector. Since multiplication between different couples of terms could give as result coefficients related to the same power, once we have computed the product between two terms we do not have to replace the corresponding position in the final vector, but to sum it to the coefficient before found. The code is the following:

Section 7: creation of the generator polynomial

```
genPol = [2 1]; % polynomial (alpha + x)
for i = 1 : 2*t - 1
    p1 = genPol;
    succ = (polTab((i + 1) + 1, :))*binTab;
    p2 = [succ 1];
    l1 = length(p1);
    l2 = length(p2);
    res = zeros(1, l1 + l2 - 1);
    for l = 1 : l1
        for j = 1 : l2
            index = l + j - 1;
            c1 = p1(l);
            c2 = p2(j);
            val = mulTab(c1 + 1, c2 + 1);
            res(index) = sumTab(res(index) + 1, val + 1);
        end
    end
    genPol = res;
end
```

6.3 Input word generation

The purpose of this section is creating a random input word of $k = 223$ symbols: first an empty vector is allocated, then, through a *for* cycle, every position is filled with a value between 0 and 255. The proceeding adopted to generate a random value is described in the following:

- through *Matlab* function `rand()` a value between 0 and 1 made by three decimals is randomly generated
- the random value is multiplied by 1000. This guarantees that the current value can be greater than 255
- the ceiling of the actual value is evaluated, so that to deal with integer numbers
- the division modulo $n + 1 = 256$ is performed on the integer, so as to get a value between 0 and 255

Section 8: creation of a random input word

```
inWord = zeros(1, k);
```

```
for i = 1 : k
    inWord(i) = mod(ceil(rand(1, 1) * 1000), (n + 1));
end
```

6.4 Systematic encoding

This sections performs the real encoding: it receives a vector of 223 symbols as argument, the input word, and produces a code word of 255 symbols. First of all, input word is multiplied by $z^{n-k} = z^{32}$; this is equivalent to shifting the input word of 32 positions to the right, which is nimbly realised by filling the rightmost 32 positions of a new 255 length vector with the input word. Then, we have to add polynomial $\mathcal{R}_{g(z)}[z^{32}u(z)]$ to this vector. We can compute this reminder polynomial making use of the division algorithm; eventually, by using the polynomial addition algorithm we can sum the two polynomials together to form the final code word. Addition and division algorithms are exposed below.

6.4.1 Addition between polynomials

The polynomials are used as two vectors with leftmost element the less significant. Addition between polynomials is very easy: we only have to sum couples of coefficients corresponding to the same power of the variable. We extract the two coefficients in the i-th position, find their sum by using the matrix `sumTab` and then store the new coefficient in i-th position of an empty vector. Notice that, in order to properly work, the function need to deal with vectors of the same length. Thus, before executing any addition, the function provides to add some zeros after the most significant component of the shorter vector in order to reach the length of the longer one.

6.4.2 Division between polynomials

Polynomials are used as two vectors with leftmost element the less significant. At every step we find a dividend of lower and lower degree and we evaluate a reminder polynomial. When the reminder polynomial has degree less than the divisor one, algorithm stops. Let's see what happens at each step:

1. Divide the coefficient of the maximum degree term of the dividend by the coefficient of the maximum degree term of the divisor: the result

will be the coefficient of the next term of the quotient. This first step is computed by using the `divTab` matrix.

2. Multiply every term of the divisor by the just found quotient term and, for each of the obtained values, replace the coefficient with its opposite (inverse with respect to addition). Since every element is representable by a polynomial with coefficients in $GF(2)$, every element is the inverse of itself.

3. Subtract the polynomial thus obtained from the actual dividend. The resulting polynomial is the reminder of this step of the division. If its degree is still higher than the divisor one, this polynomial becomes the new dividend and the cycle starts again from the beginning.

The following is the code for the systematic encoding function (in this piece of code working with vector with leftmost coefficient the most significant has been more comfortable):

Section 9: systematic encoding

```
longInWord = [zeros(1, n - k) inWord];
dvdn = wrev(longInWord);
dvs = wrev(genPol);
degdvdn = length(dvdn) - 1;
degdvs = length(dvs) - 1;
quot = zeros(1, degdvdn - degdvs + 1);
while degdvdn >= degdvs
    coeffDvdn = dvdn(1);
    coeffDvs = dvs(1);
    coeffQuot = divTab(coeffDvdn + 1, coeffDvs + 1);
    quotIndex = length(quot) - (degdvdn - degdvs);
    quot(quotIndex) = coeffQuot;
    aux = zeros(1, degdvdn - degdvs + 1);
    aux(1) = coeffQuot;

    l1 = length(aux);
    l2 = length(dvs);
    res = zeros(1, l1 + l2 - 1);
    for i = 1 : l1
        for j = 1 : l2
            index = i + j - 1;
            c1 = aux(i);
            c2 = dvs(j);
            val = mulTab(c1 + 1, c2 + 1);
```

```
            res(index) = sumTab(res(index) + 1, val + 1);
        end
    end

    long = dvdn;
    short = res;
    if length(dvdn) < length(res)
        long = res;
        short = wrev(dvdn);
    end
    short = [short zeros(1, length(long) - length(short))];

    l = length(long);
    newDvdn = zeros(1, l);
    for i = 1 : l
        c1 = short(i);
        c2 = long(i);
        sum = sumTab(c1 + 1, c2 + 1);
        newDvdn(i) = sum;
    end

    firstNonZero = 0;
    for i = 1 : length(newDvdn)
        if newDvdn(i) == 0
            firstNonZero = firstNonZero + 1;
        else
            break;
        end
    end
    newDvdn = newDvdn(firstNonZero + 1 : length(newDvdn));
    dvdn = newDvdn;
    degdvdn = length(dvdn) - 1;
end

dvdn = wrev(dvdn);
long = dvdn;
short = longInWord;
if length(dvdn) < length(longInWord)
    long = longInWord;
    short = dvdn;
end
short = [short zeros(1, length(long) - length(short))];
l = length(long);
```

```
codeword = zeros(1, l);
for i = 1 : l
    c1 = long(i);
    c2 = short(i);
    sum = sumTab(c1 + 1, c2 + 1);
    codeword(i) = sum;
end
```

6.5 Noise generation

In this section we generate an error vector which will corrupt the code word. The random process we use to create such a vector does not have any claim to model a real noisy channel: all we intend to do is simply distorting the transmitted vector to test the decoder and the robustness of the code.

To generate random noise we create a n length zero vector which we will fill with random values in random positions. First we find a random number of error components `numErr`; then, if the number of errors is not null (`numErr` $\neq 0$), we choose a random position for every error component: if in that position there is a "0", we fill it with a random value from 1 to 255. Once the error vector is created, we sum it to the transmitted code word:

Section 10: noise generation

```
errCW = codeword;
noise = zeros(1, n);
numErr = mod(ceil(rand(1, 1) * 100), maxNumErr + 1);
if numErr ~= 0
    for i = 1 : numErr
        errPosition = mod(ceil(rand(1, 1) * 10000), n) + 1;
        while noise(errPosition) ~= 0
            errPosition = mod(ceil(rand(1, 1) * 10000), n)
   + 1;
        end
        noise(errPosition) = mod(ceil(rand(1, 1) * 10000),
   n) + 1;
    end

    for i = 1 : n
        errCW(i) = sumTab(errCW(i) + 1, noise(i) + 1);
    end
end
```

6.6 Decoder initialization and syndromes

This section has the only aim of preparing some vectors and tables fundamental for the decoding process. The very first table we build is the matrix of the powers of all the element of $GF(q)$: every column is reserved to one of the $n+1$ elements, ordered by logarithmic notation, that is: $0, \alpha, \alpha^2, \ldots, \alpha^n$. Each row represents a power from 0 to $n-1$: since we will not use any vector longer than n positions, polynomial we will deal with will not have degree greater than $n-1$.

Section 11: powers table

```
powers = zeros(n, n + 1);
for i = 2 : n + 1
    currElem = polTab(i, :)*binTab;
    power = 1;
    for j = 1 : n
        powers(j, i) = power;
        power = mulTab(currElem + 1, power + 1);
    end
end
```

To properly decode we must, obviously, know what code we are using, that is parameters like n, m, k and t must be known by the receiver, who can find the number of roots of the generator polynomial: $2t$. This number is fundamental to allow us to evaluate all the necessary syndromes. For as the generator polynomial has been done, we know its roots are the first $2t$ powers of the primitive element α. To compute the $2t$ syndromes we have to evaluate the received polynomial, $\tilde{\mathbf{c}}(x)$, in every of the generator polynomial roots: we first extrapolate the proper powers from the table `powers`:

Section 12: generator polynomial roots table

```
numRoots = 2*t;
rootsPowers = powers(:, 2 : numRoots + 1);
```

and then execute the multiplication between the vector `errCW` and one of the columns of the roots powers:

Section 13: syndromes evaluation

```
syndrome = zeros(1, numRoots);
for i = 1 : numRoots
    res = 0;
    for j = 1 : n
```

```
        res = sumTab(res + 1, mulTab(errCW(j) + 1,
    rootsPowers(j, i) + 1) + 1);
     end
     syndrome(i) = res;
end
```

If all the syndromes are null we are sure that the received word belongs to the set of the valid codeword. Since we are dealing only with positive integers we can make this check only by summing every syndrome component; if the sum is zero, then every syndrome is null, the word is a good code word and we can extrapolate the information word from the rightmost k symbols:

Section 14: syndorme check

```
 if sum(syndrome) == 0
        decInWord = codeword(n - k + 1: n);
        disp('no errors occurred');
        decWord = codeword;
```

Otherwise, if there is at least one non-zero syndrome we have to find out what and how many errors have occurred.

6.7 Error locator polynomial

This section performs a large part of the decoding process. After these operation we will know the number ν of occurred errors and positions of non-zero elements of the error vector.

The first thing to do is defining initial conditions. We will use $r = 0$ and $m = -1$ as exposed in section (5.2):

Section 15: Berlekamp-Massey algorithm initialization

```
M = -1;
lambdaR = [1];
lambdaM = [1];
lr = 0;
lm = 0;
dm = 1;
```

Through a *for* cycle we will increase the value of r and update *error locator polynomial* and its related parameters (l_r, δ_r and $r - l_r$). Once we have begun the cycle we have to compute δ_r by using (5.15) to check whether the solution at the previous step since fits for the next one:

Section 16: discrepancy

```
for R = 0 : 2*t - 1
    dr = 0;
    for h = 0 : lr
        dr = sumTab(dr + 1, mulTab(lambdaR(h + 1) + 1,
   syndrome(R + 1 - h) + 1) + 1);
    end
```

If δ_r is not null we can move on to compute $\Lambda^{(r+1)}(x)$ according to Theorem 22. We will find that in several passages. First we look for δ_m^{-1}:

Section 17: finding δ_m^{-1}

```
dmLogInv = -1;
if dm == 0
    disp('Error!');
else
    dmLogInv = logTab(dm + 1);
    if dmLogInv ~= n
        dmLogInv = n - dmLogInv;
    end
end
dmInv = polTab(dmLogInv + 1, :)*binTab;
```

then we evaluate $-\delta_r\delta_m^{(-1)}x^{r-m}\Lambda^{(m)}(x)$:

Section 18: finding $-\delta_r\delta_m^{(-1)}x^{r-m}\Lambda^{(m)}(x)$

```
trasLambdaM = [zeros(1, R - M) lambdaM];
drdm = mulTab(dr + 1, dmInv + 1);
for i = 1 : length(trasLambdaM)
    trasLambdaM(i) = mulTab(trasLambdaM(i) + 1, drdm + 1);
end
```

and eventually we sum together $\Lambda^{(r)}(x)$ with the just found polynomial:

Section 19: error locator polynomial at $(r+1)$-th step

```
long = lambdaR;
short = trasLambdaM;
if length(long) < length(short)
    long = trasLambdaM;
    short = lambdaR;
end
short = [short zeros(1, length(long) - length(short))];
```

```
l = length(long);
lambdaRR = zeros(1, l);
for i = 1 : l
    c1 = long(i);
    c2 = short(i);
    lambdaRR(i) = sumTab(c1 + 1, c2 + 1);
end
```

To find l_{r+1} we have to know which one between l_r and $l_m + r - m$ is the greater one:

Section 20: shift register length

```
lrr = -1;
if lr > lm + R - M
    lrr = lr;
else
    lrr = lm + R - M;
end
```

Notice that we have saved the new *error locator polynomial* $\Lambda^{(r+1)}(x)$ and the new shift register length l_{r+1} in two temporary variables `lambdaRR` and `lrr` because we still need current values of `lambdaR` and `lr` to understand whether r would be a suitable m for next stages of the algorithm:

Section 21: new previous minimum solution

```
if R - lr > M - lm
    M = R;
    lambdaM = lambdaR;
    lm = lr;
    dm = dr;
end

lambdaR = lambdaRR;
lr = lrr;
```

Once the *for* cycle ends the vector `lambdaR` contains the coefficients of the *error locator polynomial* and, since there is at least one non-null syndrome, and thus at least one error, the polynomial must have at least degree 1 and the relative vector at least two components. The variable `nu` indicates the *locator polynomial* degree. Now we have to find the roots of this polynomial by simply evaluating $\Lambda^{(r)}(x)$ in every elements: every time the polynomial vanishes we write the related element in logarithmic form in a vector called `invPositions`. In `positions` we save the inverse of each

root stored in `invPositions`; these values represent the positions of the non-null components of the error vector.

Section 22: error locator polynomial roots

```
nu = length(lambdaR) - 1;
index = length(lambdaR);
while lambdaR(index) == 0
    nu = nu - 1;
    index = index - 1;
end

invPositions = zeros(1, nu);
index = 1;
for i = 1 : n
    res = 0;
    for j = 1 : length(lambdaR)
        res = sumTab(res + 1, mulTab(lambdaR(j) + 1, powers
   (j, i + 1) + 1) + 1);
    end
    if res == 0
        invPositions(index) = i;
        index = index + 1;
    end
end

positions = zeros(1, nu);
for i = 1 : nu
    pos = invPositions(i);
    if pos == 0
        disp('Error!');
    else
        if pos ~= n
                pos = n - pos;
        end
    end
    positions(i) = pos;
end
```

6.8 Forney algorithm

The last step the complete the decoding is evaluating error component magnitudes through the *Forney algorithm* exposed in section (5.3).

We compute the polynomial product $S(x)\Lambda(x)$ and, after the vector is formed, we consider only the $2t$ leftmost components to take the polynomial modulo x^{2t}:

Section 23: evaluating $S(x)\Lambda(x)$

```
l1 = length(lambdaR);
l2 = length(syndrome);
ltot = l1 + l2 - 1;
synLam = zeros(1, ltot);
for i = 1 : l1
    for j = 1 : l2
            index = i + j - 1;
        synLam(index) = sumTab(synLam(index) + 1, mulTab(
   lambdaR(i) + 1, syndrome(j) + 1) + 1);
    end
end
synLam = synLam(1 : 2*t);
```

We now create a vector of the numerators we will use in expression (5.25) by evaluating $\Omega(x)$ in every of the values X_k^{-1} contained in `invPositions`:

Section 24: numerators

```
lambdaRootsPow = zeros(n, nu);
for i = 1 : nu
    lambdaRootsPow(:, i) = powers(:, invPositions(i) + 1);
end

% ...

numerators = zeros(1, nu);
for i = 1 : nu
    res = 0;
    for j = 1 : length(numerators)
        res = sumTab(res + 1, mulTab(lambdaRootsPow(j, i) +
    1, synLam(j) + 1) + 1);
    end
    numerators(i) = res;
end
```

To get denominators we first compute the first derivative of $\Lambda(x)$ and then evaluate it in every of the elements of `invPositions`. For what exposed in section (5.3.1) we can find the derivative by transposing the vector `lambdaR` one step on the left, so that to decrease the power of each term. Coefficients of odd powers will remain unchanged, but even terms are all imposed to zero. Being the derivative computed, we evaluate it on all the elements of `invPositions` to get the vector `denominators`:

Section 25: derivative of $\Lambda(x)$ and denominators

```
derLambda = lambdaR(2 : length(lambdaR));
for i = 1 : floor(length(derLambda) / 2)
    derLambda(2*i) = 0;
end

denominators = zeros(1, nu);
for i = 1 : nu
    res = 0;
    for j = 1 : length(derLambda)
        res = sumTab(res + 1, mulTab(lambdaRootsPow(j, i) +
    1, derLambda(j) + 1) + 1);
    end
    denominators(i) = res;
end
```

We are now ready to compute every magnitude Y by dividing every numerator by the corrispondent denominator:

Section 26: magnitudes

```
Y = zeros(1, nu);
for i = 1 : nu
    Y(i) = divTab(numerators(i) + 1, denominators(i) + 1);
end
```

We have all the information to build the error vector: we scan the whole vector `positions` and, for every index, we save in a zero vector the correspondent value of the coefficient. We must care that α^n does not refer to the n-th power, which does not exists in a n length vector, but to the known term, that is to power x^0. The last step is adding the error vector to the received code word:

Section 27: error vector and decoded word

```
error = zeros(1, n);
```

```
for i = 1 : length(positions)
    pos = positions(i);
    if pos == n
        error(1) = Y(i);
    else
        error(pos + 1) = Y(i);
    end
end

decWord = zeros(1, n);
for i = 1 : n
    decWord(i) = sumTab(errCW(i) + 1, error(i) + 1);
end
```

Appendix A

Primitive polynomials on $GF(2)$

Here is a table of some primitive polynomials with coefficients in $GF(2)$, that is in $\{0, 1\}$, whose degrees go from 2 to 24. The table gathers some of the primitive polynomials illustrated in [4, p. 79].

Degree	Polynomial
2	$1 + x + x^2$
3	$1 + x + x^3$
4	$1 + x + x^4$
5	$1 + x^2 + x^5$
6	$1 + x + x^6$
7	$1 + x^3 + x^7$
8	$1 + x^2 + x^3 + x^4 + x^8$
9	$1 + x^4 + x^9$
10	$1 + x^3 + x^{10}$
11	$1 + x^2 + x^{11}$
12	$1 + x + x^4 + x^6 + x^{12}$
13	$1 + x + x^3 + x^4 + x^{13}$
14	$1 + x + x^6 + x^{10} + x^{14}$
15	$1 + x + x^{15}$
16	$1 + x + x^3 + x^{12} + x^{16}$
17	$1 + x^3 + x^{17}$
18	$1 + x^7 + x^{18}$
19	$1 + x + x^2 + x^5 + x^{19}$
20	$1 + x^3 + x^{20}$
21	$1 + x^2 + x^{21}$
22	$1 + x + x^{22}$
23	$1 + x^5 + x^{23}$
24	$1 + x + x^2 + x^7 + x^{24}$

APPENDIX A. PRIMITIVE POLYNOMIALS ON $GF(2)$

Bibliography

[1] N. AMBRAMSON *Information Theory and Coding*, 1963, McGraw-Hill, USA.

[2] E.R. BERLEKAMP *Algebraic Coding Theory*, 1968, McGraw-Hill, New York (USA).

[3] G. BIRKHOFF and S. Mac LANE *A Survey of Modern Algebra*, 1953, Macmillan, New York, (USA).

[4] R.E. BLAHUT *Theory and Practice of Error Control Codes*, 1983, Addison-Wesley Publishing Company.

[5] R.C. BOSE and D.K. RAY-CHAUDHURI *On a Class of Error-Correcting Binary Group Codes*, 1960, Inform. Contr., vol.3, pp. 68-79.

[6] G.C. CLARK JR. and J. BIBB CAIN *Error-Correction Coding for Digital Communications*, 1981, Plenum Press, New York (USA).

[7] R.G. GALLAGER *Information Theory and Reliable Communication*, 1968, Wiley, USA.

[8] L. GÅRDING and T. TAMBOUR *Algebra for Computer Science*, 1988, Springer-Verlag, Berlin (GER).

[9] R.W. HAMMING *Error Detecting and Error Correcting Codes*, 1950, Bell System Technical Journal, vol.29, pp. 147-150.

[10] R.V.L. HARTLEY *Transmission of Information*, 1928, Bell System Technical Journal, vol.7, num.3.

[11] A. HOCQUENGHEM *Codes Correcteurs d'Erreurs*, 1959, Chiffres, vol.2, pp. 147-156.

[12] S. IHARA *Information Theory for Continuous Systems*, 1993, World Scientific Publishing.

[13] S. LIN and D.J. COSTELLO JR. *Error Control Coding: Fundamentals and Applications*, 1983, Prentice-Hall, Englewood Cliffs, New Jersey (USA).

[14] J.H. van LINT *Introduction to Coding Theory*, 1982, Springer, Berlin (GER).

[15] J.L. MASSEY *Shift-Register Synthesis and BCH Decoding*, 1969, IEEE Trans. Inform. Theory, vol.IT-15, num.1, pp. 122-127.

[16] C.M. MONTI *Teoria dei Codici: Codici a Blocco*, 1995, Libreria Progetto, Padova (ITA).

[17] H. NYQUIST *Certain Topics in Telegraph Transmission Theory*, 1928, Transactions of the American Institute of Electrical Engineers, vol.47.

[18] W.W. PETERSON *Encoding and Error-Correction Procedures for the Bose-Chaudhuri Codes*, 1960, IRE Trans. Inform. Theory, vol.IT-6.

[19] J.R. PIERCE *An Introduction to Information Theory. Symbols, Signals and Noise*, 1980, Dover Publications, New York (USA).

[20] E. PRANGE *Cyclic Error Correcting Codes in two Symbols*, 1957, Tech. Note AFCRC-TN-57-103, Air Force Cambridge Research Center, Cambridge, MA (USA).

[21] I.S. REED and G. SOLOMON *Polynomial Codes over Certain Finite Fields*, 1960, Journal of the Society for Industrial and Applied Mathematics, SIAM, vol.8, pp. 300-304.

[22] T. RICHARDSON and R. URBANKE *Modern Coding Theory*, 2008, Cambridge University Press, New York (USA).

[23] C.E. SHANNON *A Mathematical Theory of Communication*, 1948, Bell System Technical Journal, vol.27, pp.379-423.

[24] P. SWEENEY *Error Control Coding*, 2002, Wiley, Chichester, West Sussex (ENG).

[25] B.L. van der WAERDEN *Modern Algebra*, 1950, Frederick Ungar, New York (USA).

[26] S.B. WICKER and V.K. BHARGAVA *Reed-Solomon Codes and their Applications*, 1994, IEEE Press, New York (USA).