Angewandte Computer-
und Biowissenschaften

HOCHSCHULE
MITTWEIDA
University of Applied Sciences

BACHELOR THESIS

Mr
Rasem Soufi

Introduction to Linear Codes & Binary Hamming Codes

2021

Faculty of **Applied Computer Sciences & Biosciences**

BACHELOR THESIS

Introduction to Linear Codes & Binary Hamming Codes

Author:
Rasem Soufi

Study Programme:
Applied Mathematics

Seminar Group:
MA18w1-B

First Referee:
Prof. Dr. rer. nat. Klaus Dohmen

Second Referee:
Prof. Dr. rer. nat. Peter Tittmann

Mittweida, September 2021

I. Preface

Since its foundation as an application of algebra, coding theory is obtaining a day by day increasing importance. For instance, any communication system needs the concepts of coding theory to function efficiently. In this thesis, reader will find an introductory explanation to linear codes and binary hamming codes including some of the algebraic tools devised in their applications. All the described software applications are verified using `SageMath` 9.0 using Hochschule Mittweida's JupyterHub.

Acknowledgment

First and forever, I am very grateful to God Almighty for His graces and blessings. I thank my small and big family to their endless support and encouragement.

I would love to specially thank

> Prof. Dohmen, for his support and guidance to achieve this work and other projects by his supervision.
>
> Prof. Tittmann, for his support, encouragement and guidance which I will be all of my life grateful for.

I would like to thank all my teachers (Prof. Viellmann, Prof. Baaske, Prof. Zaussinger, Dr. Nebel, Dr. Lange-Geisler, Mr. Stockmann and Mrs. Reader) for the priceless effort they were giving to afford me and my colleges lectures in the best quality. I would like to thank all the teaching and managerial staff of faculty of applied computer sciences & biosciences and of Hochschule Mittweida, you were all part of my chance to catch up again with my dreams.

To my daughter Ruba.

II. Contents

Preface I

Contents 7

1 Coding Theory 9

1.1 Minimum distance decoding 11

1.2 Equivalent Codes 13

2 Linear Codes 17

2.1 Construction of Finite Fields 17

2.1.1 Software Applications 18

2.2 Structure of Vector Spaces over Finite Fields 20

2.2.1 Software Application 20

2.3 Construction of Linear Codes 21

2.4 Encoding & Parity Check Matrix 25

2.5 Minimum Distance in Linear Codes 26

2.6 Short Note regarding Cosets 27

2.7 Syndrome Decoding 28

2.8 Software Application 33

3 Binary Hamming Codes 37

3.1 Construction of Binary Hamming Codes 37

3.2 Software Application 40

Bibliography 45

1 Coding Theory

Coding theory is the study of codes which apply the concepts of algebra fascinatingly to correct errors that may occur by transmitting messages through noisy channels. It is an applied mathematics subject that occurs in many scientific branches as information theory, computer sciences and electrical engineering. The main mission of applying coding theory is formulating error correcting codes to encode and decode the messages in a manner that we overcome the errors which may occur by the transmission through noisy channels. We will first take a look at a communication system in Figure 1.1.

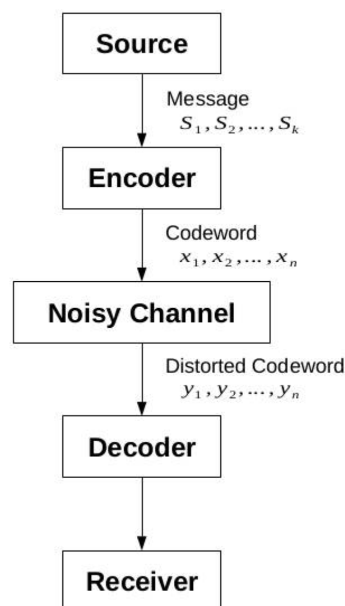

Figure 1.1: Communication System [Wel04]

Definition 1.1 (cf. [Wel04]) Consider a finite set Σ which will be denoted *alphabet* and its elements are *symbols*. *Messages* are sequences of symbols. A *code* $\mathscr{C}$ over Σ is a collection of sequences of symbols from Σ. The members of $\mathscr{C}$ are *codewords*. We will deal with codes in which the codewords are all of the same length. This type of codes is called *block codes*. If the codewords of $\mathscr{C}$ have length n and $|\Sigma| = q$, then the code is described as a *q-ary code of length* n (*binary* when $q = 2$, $\Sigma = \{0, 1\}$, and *ternary* when $q = 3$, $\Sigma = \{0, 1, 2\}$). The set of all n-sequences of symbols from Σ is denoted $\mathbf{V}_n(\Sigma)$. Elements of $\mathbf{V}_n(\Sigma)$ are called *vectors* or *words*. Where it is more important to remind the reader that Σ has q symbols we write $\mathbf{V}_n(\Sigma)$ as $\mathbf{V}_n(q)$, while, $\mathbf{V}_n$ stands for the set of binary words of length n. A vector $\mathbf{x} \in \mathbf{V}_n$ will be denoted $\mathbf{x} = (x_1, \cdots, x_n)$ or for brevity $\mathbf{x} = x_1 x_2 \cdots x_n$. The errors we will consider are the swap of symbols in the transmitted

codewords.

Definition 1.2 The source is supposed to produce sequences of *block messages* which are elements of a non-empty already defined finite set of messages B. We will consider that the block messages are all of the same length. This assumption is meant to ease the encoding and decoding processes.

Definition 1.3 (cf. [Wel04]) For all $\mathbf{x},\mathbf{y} \in \mathbf{V}_n(\Sigma)$ we define the *Hamming[1] distance* or *distance* $d(\mathbf{x},\mathbf{y})$ between $\mathbf{x}$ and $\mathbf{y}$ to be the number of places (digits) in which $\mathbf{x}$ and $\mathbf{y}$ differ. It can be described as

$$d(\mathbf{x},\mathbf{y}) = |\{i \mid 1 \le i \le n \text{ and } x_i \neq y_i\}|$$

where $\mathbf{x} = (x_1,\cdots,x_n)$ and $\mathbf{y} = (y_1,\cdots,y_n)$ [HK03].

The *minimum distance* of a code $\mathscr{C}$ written $d(\mathscr{C}) = \min d(\mathbf{c}_i,\mathbf{c}_j)$ where the minimum is taken over all pairs of distinct codewords in $\mathscr{C}$. Suppose $0 \in \Sigma$, we define the *weight* $w(\mathbf{x})$ of a word $\mathbf{x}$ to be the number of nonzero digits in $\mathbf{x}$. Clearly, $w(\mathbf{x}) = d(\mathbf{x},\mathbf{0}_n)$ where $\mathbf{0}_n = (0,0,\cdots,0) \in \mathbf{V}_n(\Sigma)$.

In the following theorem we prove that the hamming distance satisfies the axioms of distance function.

Theorem 1.4 *Let $\mathbf{x},\mathbf{y}$ and $\mathbf{z}$ be words from $\mathbf{V}_n(\Sigma)$, then*

1. $d(\mathbf{x},\mathbf{y}) \ge 0$;
2. $d(\mathbf{x},\mathbf{y}) = 0$ *if and only if* $\mathbf{x} = \mathbf{y}$;
3. $d(\mathbf{x},\mathbf{y}) = d(\mathbf{y},\mathbf{x})$;
4. $d(\mathbf{x},\mathbf{y}) \le d(\mathbf{x},\mathbf{z}) + d(\mathbf{z},\mathbf{y})$.

Proof: For (1), the number of different digits between $\mathbf{x}$ and $\mathbf{y}$ must be a non-negative integer. For (2), the words $\mathbf{x}$ and $\mathbf{y}$ don't differ in any digit if and only if $\mathbf{x} = \mathbf{y}$. The (3) is obvious.

For (4), suppose $d(\mathbf{x},\mathbf{y}) = r$, then $\mathbf{x}$ and $\mathbf{y}$ differ in r digits. Now suppose that r' of these r digits differ between $\mathbf{x}$ and $\mathbf{z}$ (then $(r-r')$ of these digits correspond), which implies that $d(\mathbf{x},\mathbf{z}) \ge r'$ since they may differ in other positions too. Then $(r-r')$ of the r digits

[1] Named after **Richard Wesley Hamming** (February 11, 1915 – January 7, 1998) was an American mathematician whose work had many implications for computer engineering and telecommunications [https://en.wikipedia.org/wiki/Richard_Hamming].

must differ between $\mathbf{y}$ and $\mathbf{z}$ which implies that $d(\mathbf{y},\mathbf{z}) \geq r - r'$. Combining these two inequalities, we obtain $d(\mathbf{x},\mathbf{z}) + d(\mathbf{z},\mathbf{y}) \geq r = d(\mathbf{x},\mathbf{y})$.

□

Definition 1.5 (cf. [Wel04]) The *e-sphere* surrounding a word $\mathbf{x}$ denoted $S_e(\mathbf{x})$ is defined by

$$S_e(\mathbf{a}) = \{\mathbf{y} \mid d(\mathbf{x},\mathbf{y}) \leq e\}.$$

Definition 1.6 The *encoder* is a bijective map E from the set of the block messages B into the code $\mathscr{C}$

$$E : B \to \mathscr{C}$$

We will denote the process of mapping a block message $\mathbf{b}$ into a codeword $\mathbf{c}$ by *encoding* $\mathbf{b}$ into $\mathbf{c}$.

Definition 1.7 The *decoder* has got two steps process. First step is connecting a received word $\mathbf{y} \in \mathbf{V}_n(\Sigma)$ to a codeword in $\mathscr{C}$ according to a relation $R \subseteq \mathbf{V}_n(\Sigma) \times \mathscr{C}$ in which a received word cannot be connected with two different codewords. However, in general we cannot assume that each received word can be connected to a codeword. The best case is when the relation R can be considered a map such that the decoder can *decode* each received word into a unique codeword. Second step is decoding the codeword into a block message according to E^{-1}. The exact meaning of the word *decoding* should be understood from the syntax.

Since it is an information theoretic topic, we will not discuss deeply the transmission channels with noise. We will assume that the noise of the channel may change some symbols of the transmitted codeword. Thus, we will receive a word in $\mathbf{V}_n(\Sigma)$ in the end of the transmission of a codeword. Further information are described in [Wel04].

1.1 Minimum distance decoding

Definition 1.8 (cf. [Wel04]) The *minimum distance decoding* means comparing the received word $\mathbf{y}$ with all the possible codewords and then decode it into a codeword $\mathbf{c}$ such that

$$d(\mathbf{y},\mathbf{c}) < d(\mathbf{y},\mathbf{c}') \text{ for all } \mathbf{c}' \in \mathscr{C} \text{ where } \mathbf{c} \neq \mathbf{c}'.$$

The following theorem gives an idea about the importance of the minimum distance decoding.

Theorem 1.9 (cf. [Wel04]) *If a code has a minimum-distance d, the minimum distance decoding scheme will correct up to $\frac{1}{2}(d-1)$ errors.*

Proof: (cf. [Wel04]) Take $e = \lfloor \frac{1}{2}(d-1) \rfloor$ and consider e-sphere surrounding $\mathbf{x}$. Because of the minimum distance hypothesis, if $\mathbf{x}$ and $\mathbf{y}$ are distinct codewords, then

$$S_e(\mathbf{x}) \cap S_e(\mathbf{y}) = \emptyset$$

Hence, minimum-error decoding will correct up to e errors. □

If a code has M codewords of length n and has minimum distance d, then it is denoted by (n, M, d)-code [Wel04].

Example 1.10 The *repetition code* repeats the symbol several times before transmitting it through the channel [Moo05]. For example, consider a binary repetition code of length 3.

block message	**codeword**	**1-sphere**
0	$\mathbf{c}_1 = 000$	$\{000, 100, 010, 001\}$
1	$\mathbf{c}_2 = 111$	$\{111, 110, 101, 011\}$

Notice that $d(\mathbf{c}_1, \mathbf{c}_2) = 3$. This code is a $(3, 2, 3)$-code.

If the message to be transmitted is 101. It will be encoded into 111000111. The decoder may receive 110010011 from the channel. First we break the received message in words of length 3 to obtain $110 \mid 010 \mid 011$ then we decode each word after comparing the distances with each codeword

word $\mathbf{x}$	$d(\mathbf{x}, \mathbf{c}_1)$	$d(\mathbf{x}, \mathbf{c}_2)$
110	2	1
010	1	2
011	2	1

to obtain $111 \mid 000 \mid 111$. Now we can decode ito the main message 101.

Decoding mistakes may occur when more than one error effects the words of length 3. For instance, if the received message in the previous transmission is 110010001. The decoder will break it in words of length 3 to obtain $110 \mid 010 \mid 001$ and will decode it into 100 which is not the intended message.

We may notice a special property in the code used in the example 1.10. Each possible word is an element of exactly one of the 1-spheres of the codewords. This idea motivates the following definition.

Definition 1.11 (cf. [Wel04]) A *perfect code* $\mathscr{C}$ over $\mathbf{V}_n(\Sigma)$ for which, there exists $t > 0$ such that the t-spheres of the codewords are disjoint and their union contains each possible word of $\mathbf{V}_n(\Sigma)$.

Example 1.12 Consider the binary repetition code of length 4. we will encode $0 \to 0000$ and $1 \to 1111$. This code has got a minimum distance of 4. If the decoder receives the word 1010 which is equidistant from the two codewords. The minimum distance decoding scheme will fail to connect this word to any of the codewords (as mentioned in Definition 1.7).

1.2 Equivalent Codes

Definition 1.13 (cf. [Wel04]) Suppose that we have an (n,M,d)-code $\mathscr{C}$. The natural way to present it is by an $M \times n$ array whose rows are the distinct codewords.

Now suppose that π is any permutation of $(1,2,\cdots,n)$ and that for each codeword $\mathbf{c} \in \mathscr{C}$, we apply the transformation $\pi : \mathbf{c} \to \mathbf{c}'$ defined by

$$c'_i = c_{\pi(i)} \quad (1 \leq i \leq n).$$

We call such a transformation a *positional permutation*.

In the same way, if μ is any permutation of the symbols in Σ, we say that μ induces a *symbol permutation* of $\mathscr{C}$ if, for some i, with $1 \leq i \leq n$, and for each codeword $\mathbf{c} \in \mathscr{C}$, we transform by $\mathbf{c} \to \mathbf{c}'$, where $\mathbf{c}'$ is defined by

$$c'_j = c_j \quad (1 \leq j \leq n,\ j \neq i), \qquad c'_i = \mu(c_i).$$

Definition 1.14 (cf. [Wel04]) If a code $\mathscr{C}'$ can be obtained from a code $\mathscr{C}$ by a sequence of positional or symbol permutations, then $\mathscr{C}'$ is called an *equivalent* code.

Theorem 1.15 (cf. [Wel04]) *If $\mathscr{C}$ and $\mathscr{C}'$ are equivalent codes, then the set of distances between the codewords of $\mathscr{C}$ is identical with that between the codewords of $\mathscr{C}'$.*

Proof: We will show that both positional and symbol permutations do not effect distances, hence no sequence of them does.

Suppose $\mathbf{x} = (x_1,x_2,\cdots,x_n), \mathbf{y} = (y_1,y_2,\cdots,y_n)$ are two codewords of (n,M,d)-code $\mathscr{C}$ with distance $d(\mathbf{x},\mathbf{y})$. By applying positional permutation we obtain $\mathbf{x}'$ and $\mathbf{y}'$ where

$$x'_i = x_{\pi(i)} \qquad y'_i = y_{\pi(i)}$$

Since π is a permutation, $x'_i = y'_i$ if and only if $x_i = y_i$, and $x'_i \neq y'_i$ if and only if $x_i \neq y_i$ which implies that $d(\mathbf{x}', \mathbf{y}') = d(\mathbf{x}, \mathbf{y})$.

by applying a symbol permutation μ on the position i we obtain $\mathbf{x}''$ and $\mathbf{y}''$ where

$$x''_j = x_j \quad y''_j = y_j \quad \text{for} \quad (1 \leq j \leq n,\ j \neq i) \quad \text{and} \quad x''_i = \mu(x_i) \quad y''_i = \mu(y_i)$$

Since μ is a permutation, $x''_i = y''_i$ if and only if $x_i = y_i$, and $x''_i \neq y''_i$ if and only if $x_i \neq y_i$ which implies that $d(\mathbf{x}'', \mathbf{y}'') = d(\mathbf{x}, \mathbf{y})$. □

Theorem 1.16 (cf. [Wel04]) *If $\mathscr{C}$ is an (n,M,d)-code and $\mathbf{u}$ is any n-vector over the same alphabet, then there exists a code $\mathscr{C}'$ that contains $\mathbf{u}$ and is equivalent to $\mathscr{C}$.*

Proof: (cf. [Wel04]) The first codeword $\mathbf{c}_1$ of $\mathscr{C}$ can be transformed into $\mathbf{u}$ by at most n symbol permutations. □

Example 1.17 Lets denote the code mentioned in Example 1.10 by $\mathscr{C}$. We can write it in a matrix manner as follows

$$\mathscr{C} = \begin{pmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}.$$

The code $\mathscr{C}'$ obtained from $\mathscr{C}$ by applying symbol permutation $\pi(1) = 0, \pi(0) = 1$ on the first column of the matrix of $\mathscr{C}$,

$$\mathscr{C}' = \begin{pmatrix} \mathbf{c}'_1 \\ \mathbf{c}'_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

is equivalent to $\mathscr{C}$. By applying the positional permutation $\pi(1) = 2, \pi(2) = 1, \pi(3) = 3$ on the rows of the matrix of $\mathscr{C}'$, we obtain the equivalent code $\mathscr{C}''$ with the codewords

$$\mathscr{C}'' = \begin{pmatrix} \mathbf{c}''_1 \\ \mathbf{c}''_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}.$$

We can recognise that the distance between the two codewords of any of the previous codes is the same

Example 1.18 Now we consider a more exciting example. Consider the ternary code $\mathscr{C}$ consisting of the codewords

$$\mathscr{C} = \begin{pmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \\ \mathbf{c}_3 \\ \mathbf{c}_4 \\ \mathbf{c}_5 \\ \mathbf{c}_6 \\ \mathbf{c}_7 \\ \mathbf{c}_8 \\ \mathbf{c}_9 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 2 \\ 1 & 0 & 2 & 1 & 0 \\ 1 & 1 & 0 & 1 & 2 \\ 0 & 2 & 2 & 0 & 1 \\ 2 & 0 & 1 & 2 & 0 \\ 2 & 2 & 0 & 2 & 1 \\ 1 & 2 & 1 & 1 & 1 \\ 2 & 1 & 2 & 2 & 2 \end{pmatrix}.$$

We may apply the positional permutation $\pi(3) = 5, \pi(5) = 3, \pi(1) = 2, \pi(2) = 4, \pi(4) = 1$ on the rows of the previous matrix to obtain the equivalent code $\mathscr{C}'$ represented by the following matrix

$$\mathscr{C}' = \begin{pmatrix} \mathbf{c}'_1 \\ \mathbf{c}'_2 \\ \mathbf{c}'_3 \\ \mathbf{c}'_4 \\ \mathbf{c}'_5 \\ \mathbf{c}'_6 \\ \mathbf{c}'_7 \\ \mathbf{c}'_8 \\ \mathbf{c}'_9 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 & 1 \\ 1 & 1 & 0 & 0 & 2 \\ 1 & 1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 2 & 2 \\ 2 & 2 & 0 & 0 & 1 \\ 2 & 2 & 1 & 2 & 0 \\ 1 & 1 & 1 & 2 & 1 \\ 2 & 2 & 2 & 1 & 2 \end{pmatrix}.$$

We may apply the symbol permutation $\pi(0) = 2, \pi(1) = 0, \pi(2) = 1$ on the second column of the matrix of code $\mathscr{C}'$ to obtain the code $\mathscr{C}''$ represented by the following matrix

$$\mathscr{C}'' = \begin{pmatrix} \mathbf{c}''_1 \\ \mathbf{c}''_2 \\ \mathbf{c}''_3 \\ \mathbf{c}''_4 \\ \mathbf{c}''_5 \\ \mathbf{c}''_6 \\ \mathbf{c}''_7 \\ \mathbf{c}''_8 \\ \mathbf{c}''_9 \end{pmatrix} = \begin{pmatrix} 0 & 2 & 0 & 0 & 0 \\ 0 & 2 & 2 & 1 & 1 \\ 1 & 0 & 0 & 0 & 2 \\ 1 & 0 & 2 & 1 & 0 \\ 0 & 2 & 1 & 2 & 2 \\ 2 & 1 & 0 & 0 & 1 \\ 2 & 1 & 1 & 2 & 0 \\ 1 & 0 & 1 & 2 & 1 \\ 2 & 1 & 2 & 1 & 2 \end{pmatrix}.$$

It is easy to verify that $d(\mathbf{c}_i, \mathbf{c}_j) = d(\mathbf{c}'_i, \mathbf{c}'_j) = d(\mathbf{c}''_i, \mathbf{c}''_j)$ for all $i, j \in \{1, \cdots, 9\}$.

2 Linear Codes

Linear Codes overcome the problem of computing the distances between a received word and each of the codewords of a code, specially when the code contains a huge number of codewords [Wel04]. In this chapter we will discuss the general idea of linear codes. However, we need first to be familiar with the algebraic structures which are used in the construction of linear codes.

2.1 Construction of Finite Fields

Theorem 2.1 (cf. [WJu18]) *For every prime p and every positive integer n, there exists a finite field F with p^n elements.*

Theorem 2.1 has got another part, which shows that finite fields of the same size are isomophic (similar in structure). Introducing the proof of this theorem requires algebraic concepts that are out of the range of this work. Reader can find more details in [WJu18]. We will move to the algorithm of constructing finite fields of p^n elements where p is prime and n is a positive integer.

Algorithm 1 (cf. [WJu18]) *To construct a field of p^n elements*

(1) We search for an irreducible polynomial $f(x) \in F_p[x]$ of degree n,

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n.^2$$

(2) We assume that α is a zero of the polynomial, $f(\alpha) = 0$.

(3) The set of the elements of F_{p^n} is

$$F_{p^n} = \{b_0 + b_1\alpha + b_2\alpha^2 + \cdots + b_{n-1}\alpha^{n-1} \quad | \quad b_i \in F_p \quad \text{and} \quad f(\alpha) = 0\}.$$

Addition and Multiplication are defined modulo p.

Example 2.2 We construct the field F_{3^2}. The polynomial $f(x) = x^2 + 1$ is of degree 2 and irreducible in $F_3[x]$ since $f(0) = 1$ and $f(1) = f(2) = 2$. Now we assume that α is

[2] Notice that the set of all polynomials with coefficients in a field F and a variable x is denoted $F[x]$.

a zero of $f(x)$, in other words $f(\alpha) = 0$. Then the elements we are searching for are

$$F_{3^2} = \{0, 1, 2, \alpha, 2\alpha, 1+\alpha, 1+2\alpha, 2+\alpha, 2+2\alpha\}.$$

The operations addition and multiplication modulo 3 - denoted $\oplus, \odot$ respectively - with respect to $\alpha^2 = 2$ are obtained as in the following tables.

$\oplus$	0	1	2	α	2α	$1+\alpha$	$1+2\alpha$	$2+\alpha$	$2+2\alpha$
0	0	1	2	α	2α	$1+\alpha$	$1+2\alpha$	$2+\alpha$	$2+2\alpha$
1	1	2	0	$1+\alpha$	$1+2\alpha$	$2+\alpha$	$2+2\alpha$	α	2α
2	2	0	1	$2+\alpha$	$2+2\alpha$	α	2α	$1+\alpha$	$1+2\alpha$
α	α	$1+\alpha$	$2+\alpha$	2α	0	$1+2\alpha$	1	$2+2\alpha$	2
2α	2α	$1+2\alpha$	$2+2\alpha$	0	α	1	$1+\alpha$	2	$2+\alpha$
$1+\alpha$	$1+\alpha$	$2+\alpha$	α	$1+2\alpha$	1	$2+2\alpha$	2	2α	0
$1+2\alpha$	$1+2\alpha$	$2+2\alpha$	2α	1	$1+\alpha$	2	$2+\alpha$	0	α
$2+\alpha$	$2+\alpha$	α	$1+\alpha$	$2+2\alpha$	2	2α	0	$1+2\alpha$	1
$2+2\alpha$	$2+2\alpha$	2α	$1+2\alpha$	2	$2+\alpha$	0	α	1	$1+\alpha$

$\odot$	0	1	2	α	2α	$1+\alpha$	$1+2\alpha$	$2+\alpha$	$2+2\alpha$
0	0	0	0	0	0	0	0	0	0
1	0	1	2	α	2α	$1+\alpha$	$1+2\alpha$	$2+\alpha$	$2+2\alpha$
2	0	2	1	2α	α	$2+2\alpha$	$2+\alpha$	$1+2\alpha$	$1+\alpha$
α	0	α	2α	2	1	$2+\alpha$	$1+\alpha$	$2+2\alpha$	$1+2\alpha$
2α	0	2α	α	1	2	$1+2\alpha$	$2+2\alpha$	$1+\alpha$	$2+\alpha$
$1+\alpha$	0	$1+\alpha$	$2+2\alpha$	$2+\alpha$	$1+2\alpha$	2α	2	1	α
$1+2\alpha$	0	$1+2\alpha$	$2+\alpha$	$1+\alpha$	$2+2\alpha$	2	α	2α	1
$2+\alpha$	0	$2+\alpha$	$1+2\alpha$	$2+2\alpha$	$1+\alpha$	1	2α	α	2
$2+2\alpha$	0	$2+2\alpha$	$1+\alpha$	$1+2\alpha$	$2+\alpha$	α	1	2	2α

2.1.1 Software Applications

Fortunately, one of the applications of mathematics software systems is to help us build complicated mathematical structures. For instance, we do not need to go through the previous process each time we want to construct a finite field. To do so, we can use `SageMath`.

In the beginning, we need to construct our finite field F using the routine `GF(q,'var')`. In the following example, we will form F_{3^2} using variable a.

```
1 sage: F = GF(9,'a');F
2       Finite Field in a of size 3^2
```

The arguments we passed to the routine GF are GF(P^n,variable). It is convenient to pass the number of elements we need in F. However, it is more practical to pass the variable you want the routine to use in the construction polynomial which we can check as follows.

```
1 sage: F.polynomial()
2       a^2 + 2*a + 2
```

Now we can list the elements of our field.

```
1 sage: L = list(F);L
2       [0, a, a + 1, 2*a + 1, 2, 2*a, 2*a + 2, a + 2, 1]
```

To assign an element of F to a variable, we can do as follows.

```
1 sage: x = F('a+1')
2 sage: y = F('2*a')
3 sage: z = F(1)
4 sage: b = F(0)
```

Now we can apply addition and multiplication $\pmod{p}$.

```
1 sage: x + y
2       1
3 sage: x * y
4       a + 2
```

We can also calculate the additive and the multiplicative inverses.

```
1 sage: -x
2       2*a + 2
3 sage: y^(-1)
4       s*a + 1
```

Even subtraction and division are defined.

```
1 sage: x-y
2       2*a + 1
3 sage: x/y
4       2*a
```

You can even consider multiplication and addition over equivalence classes of the elements of F. The result will be reduced to the default element of F.

```
1 sage: F('4*a^2') * F('5*a^3')
2       a
3 sage: F('4*a^2') + F('5*a^3')
4       2*a
```

Even though elements of F are of a special type in SageMath

```
1 sage: type(F(1))
2       <class 'sage.rings.finite_rings.element_givaro.FiniteField_givaroElement'
3 sage: type(1)
4       <class 'sage.rings.integer.Integer'>
```

addition, multiplication, subtraction and division operators are overloaded to handle operations as follows.

```
1 sage: F(2) + 17
2       1
3 sage: F(2*a^2) * 5
4       a + 1
```

The integer which occurs in such operations is taken $\pmod{p}$. That is why we should be careful specially with division.

```
1 sage: F(2) / 6
2       ZeroDivisionError: division by zero in finite field
```

For further explanation, consider [SMb].

2.2 Structure of Vector Spaces over Finite Fields

The axioms of vector space definition did not specify a special type of fields to be used in the construction [WJu18]. With appropriate vector space operations (vector addition and scalar multiplication) we may construct vector spaces over finite fields.

Consider the set $F_{p^n}^m$ where p is prime and m,n are positive integers. It can be presented as

$$F_{p^n}^m = \{(x_1,\cdots,x_m) \quad | \quad x_i \in F_{p^n} \quad \text{for} \quad i = 1,\ldots,m\}.$$

For each two elements $\mathbf{x} = (x_1,\ldots,x_m), \mathbf{y} = (y_1,\ldots,y_m) \in F_{p^n}^m$ we define the vector addition by

$$\mathbf{x}+\mathbf{y} = (x_1+y_1 \pmod{p},\cdots,x_m+y_m \pmod{p}).$$

For each $\alpha \in F_{p^n}$ and $\mathbf{x} \in F_{p^n}^m$ we define the scalar multiplication by

$$\alpha\mathbf{x} = (\alpha x_1 \pmod{p},\cdots,\alpha x_m \pmod{p}).$$

It is easy to show that these operations fulfil the vector space axioms.

2.2.1 Software Application

We will discuss some routines and definitions of vector spaces in `SageMath`. To define a vector space, we need to know the field over which it is defined and its dimension. We pass these arguments to the routine `VectorSpace()`. Lets consider the vector space V of dimension 5 over F_7.

```
1 sage: V = VectorSpace(GF(7),5)
```

Now we can deal with vectors of V.

```
1 sage: X = V([2,3,1,0,1])
```

The constructor is going to take the values $\pmod{p}$.

```
1 sage: Y = V([9,19,-1,1,8]);Y
2       (2, 5, 6, 1, 1)
```

Now we can apply addition and subtraction between vectors.

```
1 sage: X+Y
2       (4, 1, 0, 1, 2)
3 sage: X-Y
4       (0, 5, 2, 6, 0)
```

We can also multiply the vector by a scalar

```
1 sage: 7*X
2       (0, 0, 0, 0, 0)
```

We may choose the scalar to be any element of our field.

```
1 sage: GF(7)(4^(-1))*X
2       (4, 6, 2, 0, 2)
```

In addition, we can define a subspace of V. We need only the vectors that span the subspace W.

```
1 sage: W = V.subspace([V([1,1,0,3,-1]),V([2,0,5,-5,0])]);W
2       Vector space of degree 5 and dimension 2
3       over Finite Field of size 7
4       Basis matrix:
5       [1 0 6 1 0]
6       [0 1 1 2 6]
```

Note that the basis computed by `SageMath` is row reduced. If the vectors we choose are not linearly independent, we still can span a subspace using the same method.

```
1 sage: S = V.subspace([V([1,1,0,3,-1]),V([4,4,0,12,-4]),
2             V([2,0,5,3,5])]); S
3       Vector space of degree 5 and dimension 2
4       over Finite Field of size 7
5       Basis matrix:
6       [1 0 6 5 6]
7       [0 1 1 5 0]
```

The documentation [SMc] is considered for further reading to apply linear algebra in sagemath.

2.3 Construction of Linear Codes

The only condition to be able to construct and use linear codes is that the cardinality of the alphabet Σ is a prime power p^m such that we can consider Σ the finite field F_{p^m} and the set $\mathbf{V}_n(\Sigma)$ is the vector space of dimension n over F_{p^m}, a typical member of which will be denoted by $\mathbf{x} = (x_1, x_2, \cdots, x_n)$, which, for brevity, will be sometimes written $\mathbf{x} = x_1 \cdots x_n$ where $x_i \in F_{p^m}$ [Wel04]. Lets consider the block messages which we want to encode to be k-sequences of symbols of Σ. Then, we can expand the block messages to be the elements of the vector space $\mathbf{V}_k(\Sigma)$ [Wel04].

Definition 2.3 (cf. [Wel04]) A *linear code* $\mathscr{C}$ over Σ is defined to be any subspace of $\mathbf{V}_n(p^m)$. If $\mathscr{C}$ is a k-dimensional subspace, then we denote it by $[n,k]$-code.

Definition 2.4 (cf. [Wel04]) We define a *generator matrix* for a linear $[n,k]$-code $\mathscr{C}$ to be any $k \times n$ matrix whose rows are k linearly independent codewords of $\mathscr{C}$.

Theorem 2.5 (cf. [Wel04]) *Suppose G is a generator matrix of $\mathscr{C}$ and G' is any other matrix obtained from G by any finite sequence of operations of the following types:*

1. *Permuting rows.*
2. *Multiplying a row by a nonzero scalar.*
3. *Adding to a row a scalar multiple of another row.*
4. *Permuting columns.*
5. *Multiplying any column by a nonzero multiple.*

Then G' is a generator matrix of a code $\mathscr{C}'$ which is equivalent to $\mathscr{C}$.

Proof: Let $\mathscr{C}$ be a linear $[n,k]$-code generated by the $k \times n$ matrix G all defined over the field F. Denote the row vectors of G by $\mathbf{r}_i$ for $i = 1, 2, \cdots, k$. We want to show that each of the operations $(1)-(5)$ gives a matrix G' which is the generator matrix of an equivalent code $\mathscr{C}'$.

For (1), the vector space spanned by the row vectors of G will not be changed by any permutation of them.

For (2), let $\alpha \in F$ be a nonzero element. Multiplying a row vector $\mathbf{r}_i$ of G by α is replacing $\mathbf{r}_i$ by $\mathbf{r}_i' = \alpha \mathbf{r}_i$ to obtain G'. Any vector of the span of the row vectors of G' is obtained by

$$\begin{array}{lcl} \alpha_1 \mathbf{r}_1 + \alpha_2 \mathbf{r}_2 + \cdots + & \alpha_i \mathbf{r}_i' & + \cdots + \alpha_k \mathbf{r}_k = \\ \alpha_1 \mathbf{r}_1 + \alpha_2 \mathbf{r}_2 + \cdots + & \alpha_i(\alpha \mathbf{r}_i) & + \cdots + \alpha_k \mathbf{r}_k = \\ \alpha_1 \mathbf{r}_1 + \alpha_2 \mathbf{r}_2 + \cdots + & (\alpha_i \alpha)\mathbf{r}_i & + \cdots + \alpha_k \mathbf{r}_k \in \mathscr{C} \end{array}$$

where $\alpha_1, \cdots, \alpha_k \in F$.

What we actually do by applying (3) is replacing a row vector $\mathbf{r}_i$ by a linear combination $\mu \mathbf{r}_i + \beta \mathbf{r}_j = r_i'$ to obtain G'. In the same manner as in the operation (2), any vector in the span of the row vectors of G' is obtained by

$$\begin{array}{lllll}
\alpha_1\mathbf{r}_1+\cdots+ & \alpha_j\mathbf{r}_j & +\cdots+ & \alpha_i\mathbf{r}'_i & +\cdots+\alpha_k\mathbf{r}_k = \\
\alpha_1\mathbf{r}_1+\cdots+ & \alpha_j\mathbf{r}_j & +\cdots+ & \alpha_i(\mu\mathbf{r}_i+\beta\mathbf{r}_j) & +\cdots+\alpha_k\mathbf{r}_k = \\
\alpha_1\mathbf{r}_1+\cdots+ & (\alpha_j+\alpha_i\beta)\mathbf{r}_j & +\cdots+ & (\alpha_i\mu)\mathbf{r}_i & +\cdots+\alpha_k\mathbf{r}_k \in \mathscr{C}
\end{array}$$

where $\alpha_1,\cdots,\alpha_k \in F$.

For instance, any finite sequence of the previous three operations will give us a $k\times n$ matrix G' that generates the code $\mathscr{C}'$ which is identical to $\mathscr{C}$.

Now we will show that operation (4) is equivalent to the positional permutation in the full matrix of the code. Let G be

$$\begin{pmatrix}
g_{11} & g_{12} & \cdots & g_{1i} & \cdots & g_{1j} & \cdots & g_{1n} \\
g_{21} & g_{22} & \cdots & g_{2i} & \cdots & g_{2j} & \cdots & g_{2n} \\
\vdots & \vdots & & \vdots & & \vdots & & \vdots \\
g_{k1} & g_{k2} & \cdots & g_{ki} & \cdots & g_{kj} & \cdots & g_{kn}
\end{pmatrix}.$$

Since the code $\mathscr{C}$ is the span of the row vectors of G, an element in $\mathscr{C}$ can be described as

$$\begin{array}{llllllll}
\mathbf{c} = (& \alpha_1 g_{11} & + & \alpha_2 g_{21} & + & \cdots & + & \alpha_k g_{k1}, \\
& \alpha_1 g_{12} & + & \alpha_2 g_{22} & + & \cdots & + & \alpha_k g_{k2}, \\
& & & \cdots & & & & \\
& \alpha_1 g_{1i} & + & \alpha_2 g_{2i} & + & \cdots & + & \alpha_k g_{ki}, \\
& & & \cdots & & & & \\
& \alpha_1 g_{1j} & + & \alpha_2 g_{2j} & + & \cdots & + & \alpha_k g_{kj}, \\
& & & \cdots & & & & \\
& \alpha_1 g_{1n} & + & \alpha_2 g_{2n} & + & \cdots & + & \alpha_k g_{kn})
\end{array}$$

where $\alpha_1,\cdots,\alpha_k \in F$. Without loss of generatlity, assume that we permute the columns i and j in G to obtain G'. An element of $\mathscr{C}'$ will be obtained as follows,

$$\begin{array}{llllllll}
\mathbf{c}' = (& \alpha_1 g_{11} & + & \alpha_2 g_{21} & + & \cdots & + & \alpha_k g_{k1}, \\
& \alpha_1 g_{12} & + & \alpha_2 g_{22} & + & \cdots & + & \alpha_k g_{k2}, \\
& & & \cdots & & & & \\
& \alpha_1 g_{1j} & + & \alpha_2 g_{2j} & + & \cdots & + & \alpha_k g_{kj}, \\
& & & \cdots & & & & \\
& \alpha_1 g_{1i} & + & \alpha_2 g_{2i} & + & \cdots & + & \alpha_k g_{ki}, \\
& & & \cdots & & & & \\
& \alpha_1 g_{1n} & + & \alpha_2 g_{2n} & + & \cdots & + & \alpha_k g_{kn}).
\end{array}$$

Thus, columns permutation in the generator matrix is equivalent to positional permutation of the full matrix of the code.

For (5) and by the same manner, let us multiply the column i of G by the nonzero

element $\alpha \in F$. We obtain

$$G' = \begin{pmatrix} g_{11} & g_{12} & \cdots & \alpha g_{1i} & \cdots & g_{1n} \\ g_{21} & g_{22} & \cdots & \alpha g_{2i} & \cdots & g_{2n} \\ \vdots & \vdots & & \vdots & & \vdots \\ g_{k1} & g_{k2} & \cdots & \alpha g_{ki} & \cdots & g_{kn} \end{pmatrix}.$$

The position i in a codeword $\mathbf{c}' \in \mathscr{C}'$ will be $c'_i = \alpha(c_i)$ which is equivalent to symbol permutation in position i. □

Theorem 2.6 (cf. [Wel04]) *Let G be any $k \times n$ matrix whose rows are linearly independent. Then, by applying a sequence of operations of type $(1)-(5)$ of Theorem 2.5 to G, it is possible to transform G into a matrix of type $[I_k, A]$, where I_k is the $k \times k$ identity matrix.*

Proof: suppose

$$G = \begin{pmatrix} g_{11} & g_{12} & \cdots & g_{1n} \\ g_{21} & g_{22} & \cdots & g_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k1} & g_{k2} & \cdots & g_{kn} \end{pmatrix}.$$

We will denote the rows by $\mathbf{r}_i$ for $1 \le i \le k$ and denote columns by $\mathbf{c}_j$ for $1 \le j \le n$. We will start with $g_{11} \in \mathbf{r}_1$ and then we repeat the steps on the elements $g_{ll} \in \mathbf{r}_l$ for $l = 2, 3, \cdots, k$ in order.

(Step1) If $g_{11} = 0$ exchange $\mathbf{c}_1$ with $\mathbf{c}_j$ such that $g_{1j} \neq 0$,

(Step2) In the new matrix, transform $\mathbf{r}_1$ into $g_{11}^{-1}\mathbf{r}_1$ to obtain $\mathbf{r}_1 = (1, g_{12}^{(1)}, \cdots, g_{1n}^{(1)})$,

(Step3) Transform each row $\mathbf{r}_i$ into $\mathbf{r}_i + (-g_{i1})\mathbf{r}_1$ for $1 \le i \le k$ and $i \neq 1$ to obtain $\mathbf{r}_i = (0, g_{i2}^{(1)}, \cdots, g_{in}^{(1)})$,

we obtain the following matrix

$$G^{(1)} = \begin{pmatrix} 1 & g_{12}^{(1)} & \cdots & g_{1n}^{(1)} \\ 0 & g_{22}^{(1)} & \cdots & g_{2n}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & g_{k2}^{(1)} & \cdots & g_{kn}^{(1)} \end{pmatrix}.$$

Now we repeat the previous three steps on each new matrix with respect to the elements g_{ll} for $l = 2, 3, \cdots, k$ in order. This is possible since the rank of G is k. The matrix $G^{(k)}$ is of the desired form. □

2.4 Encoding & Parity Check Matrix

Now we are ready to discuss the encoding process of a linear code. Suppose that $\mathscr{C}$ is a linear $[n,k]$-code over $F_{p^m} = \Sigma$, and it has generator matrix G defined as

$$G = \begin{pmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \\ \vdots \\ \mathbf{r}_k \end{pmatrix} = [I_k, A]$$

where the $\mathbf{r}_i$ are n-vectors over F_{p^m} and A is a $k \times (n-k)$ matrix (cf. [Wel04]). The codewords of $\mathscr{C}$ are all the $(p^m)^k$ vectors of length n of the form

$$\sum_{i=1}^{k} a_i \mathbf{r}_i \quad \text{where} \quad a_i \in F_{p^m} \qquad \text{(cf. [Wel04]).}$$

The basic idea of encoding is as follows. If the block message is $\mathbf{s} = (s_1, \cdots, s_k)$, we encode $\mathbf{s}$ into the codeword $\mathbf{c} = (c_1, \cdots, c_n)$ where the c_i are given by the rule

$$c_i = s_i \quad (1 \le i \le k), \tag{2.1}$$

$$(c_{k+1}, \cdots, c_n) = \mathbf{s}A \qquad \text{(cf. [Wel04]).} \tag{2.2}$$

In other words,

$$(c_1, \cdots, c_n) = (s_1, \cdots, s_k)G.$$

The equations (2.1) and (2.2) can be written in the form

$$\begin{gathered}
(s_1, \cdots, s_k)A = (c_1, \cdots, c_k)A = (c_{k+1}, \cdots, c_n) \\
(-c_1, -c_2, \cdots, -c_k, \underbrace{0, \cdots, 0}_{n-k}) \begin{bmatrix} -A \\ I_{n-k} \end{bmatrix} = (\underbrace{0, \cdots, 0}_{k}, c_{k+1}, \cdots, c_n) \begin{bmatrix} -A \\ I_{n-k} \end{bmatrix} \\
(c_1, \cdots, c_n) \begin{bmatrix} -A \\ I_{n-k} \end{bmatrix} = (\underbrace{0, \cdots, 0}_{n-k}) = \mathbf{0}_{n-k}
\end{gathered}$$

so that they are equivalent to

$$[-A^{\mathsf{T}}, I_{n-k}]\mathbf{c}^{\mathsf{T}} = \mathbf{0}_k^{\mathsf{T}}.$$

The matrix

$$H = [-A^{\mathsf{T}}, I_{n-k}] \tag{2.3}$$

is called the *parity-check matrix* of the code (cf. [Wel04]).

What we have shown above is that a vector $\mathbf{z}$ is a codeword of $\mathscr{C}$ if and only if $H\mathbf{z}^{\mathsf{T}} = \mathbf{0}_k^{\mathsf{T}}$ [Wel04]. In other words, the parity check matrix defines a linear map in which the kernel is our code [WJu18]. It owes its name to the fact that what we are doing is adding on

some check digits to correct errors in the received words [Wel04]. For that reason, in an $[n,k]$-code, the first k digits of a codeword are often called the *message digits* and the remaining $n-k$ are the *check* digits [Wel04].

Example 2.7 The code $\mathscr{C}$ used in Example 1.18 is a linear code over the alphabet $F_3 = \{0,1,2\}$, generated by the matrix

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

The codewords $\mathbf{c}_1, \cdots, \mathbf{c}_9$ are generated as $\mathbf{c}_i = \mathbf{x}_i G$ where $\mathbf{x}_i$ for $i = 1, \cdots, 9$ are the 9 different elements of F_3^2 . In the generator matrix, we recognise the following matrix

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

Now we can construct the parity check matrix

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

It can be easily verified that

$$H\mathbf{c}_i^{\mathsf{T}} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \quad \text{for} \quad i = 1, \cdots, 9$$

Note that in the previous example, 2 codewords, namely

$$\begin{pmatrix} \mathbf{c}_2 \\ \mathbf{c}_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 2 & 1 & 0 \\ 0 & 1 & 1 & 0 & 2 \end{pmatrix},$$

were sufficient to generate the whole code consisting of 9 codewords. This is one of the advantages of linear codes. They allow k codewords to describe a code consisting of $(p^m)^k$ codewords [Wel04]. This is derived from the vector spaces property that a k dimensional subspace is completely described when we have k linearly independent vectors from it [Wel04].

2.5 Minimum Distance in Linear Codes

After the construction of the linear code, we will discuss its minimum distance.

Lemma 2.8 *Let* $\mathbf{x}, \mathbf{y}$ *and* $\mathbf{z}$ *be vectors from* $\mathbf{v}_n(p^m)$*, we obtain*

$$d(\mathbf{x}+\mathbf{z}, \mathbf{y}+\mathbf{z}) = d(\mathbf{x}, \mathbf{y}).$$

Proof: Let $\mathbf{x} = (x_1, x_2, \cdots, x_n)$, $\mathbf{y} = (y_1, y_2, \cdots, y_n)$ and $\mathbf{z} = (z_1, z_2, \cdots, z_n)$. Then $\mathbf{x}+\mathbf{z} = (x_1+z_1, \cdots, x_n+z_n)$ and $\mathbf{y}+\mathbf{z} = (y_1+z_1, \cdots, y_n+z_n)$. It follows,

$$\left.\begin{array}{lcl} x_i+z_i = y_i+z_i & \Leftrightarrow & x_i = y_i \\ x_i+z_i \neq y_i+z_i & \Leftrightarrow & x_i \neq y_i \end{array}\right\} \quad \text{for } i = 1, \cdots, n.$$

□

Theorem 2.9 (cf. [Wel04]) *The minimum distance of a linear code* $\mathscr{C}$ *is the minimum weight of a nonzero vector in* $\mathscr{C}$*.*

Proof: (cf. [Wel04]) Let d be the minimum distance of an $[n,k]$-code $\mathscr{C}$ and suppose $\mathbf{x}$ and $\mathbf{y}$ are codewords with $d(\mathbf{x}, \mathbf{y}) = d$. Since $\mathscr{C}$ is a linear subspace, the vector $(\mathbf{x}-\mathbf{y})$ is also a codeword of $\mathscr{C}$. Thus

$$w(\mathbf{x}-\mathbf{y}) = d(\mathbf{x}-\mathbf{y}, \mathbf{0}_n) = d(\mathbf{x}-\mathbf{y}+\mathbf{y}, \mathbf{0}_n+\mathbf{y}) = d(\mathbf{x}, \mathbf{y}) = d,$$

implies that the minimum weight is at most d. However, it cannot be strictly less than d. Assume that $\mathbf{z} \in \mathscr{C}$ and $\mathbf{z} \neq \mathbf{0}_n$ with $w(\mathbf{z}) < d$, then we would obtain

$$w(\mathbf{z}) = d(\mathbf{z}, \mathbf{0}_n) < d$$

which is a contradiction since $\mathbf{0}_n \in \mathscr{C}$. □

What we actually do by defining and encoding using linear codes is that we transform our block messages vector space to a higher dimensional vector space where we can distribute the codewords in a manner that we obtain greater distances between them [WJu18]. If we know the minimum distance d of an $[n,k]-$code, we denote it $[n,k,d]-$code. Notice that an $[n,k,d]-$code over F_{p^m} is denoted in general form by $(n, (p^m)^k, d)$ [Wel04].

2.6 Short Note regarding Cosets

We will deal with the concept of cosets to construct the decoding scheme of our linear code.

Definition 2.10 (cf. [WJu18]) Let G be a group and H be a subgroup of G. Define a *left coset* of H with *representative* $g \in G$ to be the set $gH = \{gh \mid h \in H\}$. *right coset* can be defined similarly by $Hg = \{hg \mid h \in H\}$.

In our work, our linear code $\mathscr{C}$ is a subspace, which will be considered a subgroup of the commutative additive group $(\mathbf{V}_n(P^m),+)$. because of commutativity, left and right cosets of $\mathscr{C}$ coincide. Thus, to work with a *coset*, we do not need the notation left or right [WJu18]. Furthermore, we will describe cosets additively.

Lemma 2.11 *(cf. [WJu18]) Let H be a subgroup of an additive group G and suppose that $g_1,g_2 \in G$. The following conditions are equivalent.*

1. $g_1+H = g_2+H$;
2. $g_2 \in g_1+H$;
3. $g_1+(-g_2) \in H$.

Proof: $((1) \Rightarrow (2))$ If $g_1+H = g_2+H$ then there exist $h_1,h_2 \in H$ such that $g_1+h_1 = g_2+h_2$. By cancellation law $g_2 = g_1+(h_1+(-h_2))$. Let $h_1+(-h_2) = h$. It must be an element of H. Thus, $g_2 = g_1+h \in H$.

$((2) \Rightarrow (3))$ If g_2 is an element of g_1+H, then, there exists $h \in H$ such that $g_2 = g_1+h$. By cancellation law, $g_1+(-g_2) = -h$. Since H is a subgroup, $-h \in H$ and thus $g_1+(-g_2) \in H$.

$((3) \Rightarrow (1))$ Consider $g_1+(-g_2) \in H$ where $g_1,g_2 \in G \setminus H$. Then there exists $h \in H$ such that

$$\begin{aligned}
g_1+(-g_2) &= h \\
g_1+(-g_2) &= h+h_1+(-h_1) && \text{for } h_1 \in H \\
g_1+h_1 &= g_2+(h+h_1) && \text{by cancellation law} \\
g_1+h_1 &= g_2+h_2. && \text{for } h+h_1 = h_2 \in H
\end{aligned}$$

Hence, $g_1+H = g_2+H$. □

2.7 Syndrome Decoding

By transmitting a codeword $\mathbf{c} = (c_1,\cdots,c_n)$ through the channel, the decoder may receive the word $\mathbf{y} = (y_1,\cdots,y_n)$. Lets take a deeper look at the digits of $\mathbf{y}$ and $\mathbf{c}$. Since $y_i,c_i \in F_{p^m}$, then there exists $e_i \in F_{p^m}$ such that $y_i = c_i+e_i$ for $i = 1,\cdots,n$ and

$$\begin{aligned}
e_i = 0 &\quad \text{if and only if} \quad y_i = c_i \\
e_i \neq 0 &\quad \text{if and only if} \quad y_i \neq c_i
\end{aligned}$$

By gathering the values e_i in one vector, we obtain the *error vector* $\mathbf{e} = (e_1,\cdots,e_n)$ where $\mathbf{y} = \mathbf{c}+\mathbf{e}$. Our mission is to find the vector $\mathbf{e}$ such that we can decode $\mathbf{y}$ into

$\mathbf{c} = \mathbf{y} + (-\mathbf{e})$.

Lemma 2.12 (cf. [Wel04]) *If $\mathbf{y}$ is the received vector, then the set of possible error vectors is the coset of $\mathscr{C}$ that contains the vector $\mathbf{y}$.*

Proof: (cf. [Wel04]) If $\mathbf{y}$ is received, then $\mathbf{e}$ is an error vector of $\mathbf{y}$ if and only if there exists a codeword $\mathbf{c} \in \mathscr{C}$ such that $\mathbf{e} = \mathbf{y} - \mathbf{c}$. Since $\mathscr{C}$ is a subspace, if $\mathbf{c} \in \mathscr{C}$, then $-\mathbf{c} \in \mathscr{C}$ and thus $\mathbf{e} = \mathbf{y} + \mathbf{c}'$ (with $\mathbf{c}' = -\mathbf{c}$) and $\mathbf{e} \in \mathbf{y} + \mathscr{C}$. □

Definition 2.13 (cf. [Wel04]) For each coset $\mathbf{a} + \mathscr{C}$, we call a vector $\mathbf{z}_0 \in \mathbf{a} + \mathscr{C}$ a *coset leader* if that coset contains no other vector with smaller weight. Unfortunately, the coset leader is not always uniquely determined.

Theorem 2.14 (cf. [Wel04]) *Two vectors $\mathbf{y}_1$ and $\mathbf{y}_2$ are in the same coset with respect to $\mathscr{C}$ if and only if*

$$H\mathbf{y}_1^T = H\mathbf{y}_2^T.$$

Proof: The vectors $\mathbf{y}_1$ and $\mathbf{y}_2$ are in the same coset if and only if there exists some codeword $\mathbf{c}$ such that

$$\begin{aligned}
\mathbf{y}_1 &= \mathbf{y}_2 + \mathbf{c} \\
\mathbf{y}_1^\mathsf{T} &= (\mathbf{y}_2 + \mathbf{c})^\mathsf{T} = \mathbf{y}_2^\mathsf{T} + \mathbf{c}^\mathsf{T} \\
H\mathbf{y}_1^\mathsf{T} &= H(\mathbf{y}_2^\mathsf{T} + \mathbf{c}^\mathsf{T}) \\
&= H\mathbf{y}_2^\mathsf{T} + H\mathbf{c}^\mathsf{T}.
\end{aligned}$$

However, $H\mathbf{c}^\mathsf{T} = \mathbf{0}_k^\mathsf{T}$. □

Definition 2.15 (cf. [Wel04]) The *syndrome* of the coset $\mathbf{a} + \mathscr{C}$ is the vector $H\mathbf{a}^\mathsf{T}$. By Theorem 2.14, it is well defined.

Now we are ready to construct our decoding algorithm.

Algorithm 2 (cf. [Wel04])

Step(1): Formulate a look up table in which for each coset $\mathbf{a} + \mathscr{C}$ we assign the syndrome and we choose the coset leader.

Step(2): On receiving $\mathbf{y}$, calculate its syndrome $H\mathbf{y}^T$.

Step(3): From the above look up table, read off the corresponding coset leader $\mathbf{z_0}$.

Step(4): Decode $\mathbf{y}$ *as the vector* $\mathbf{y}+(-\mathbf{z}_0)$.

Theorem 2.16 (cf. [Wel04]) *Algorithm 2 is a minimum-distance decoding scheme for the linear code* $\mathscr{C}$.

Proof: (cf. [Wel04]) First note that any received vector $\mathbf{y}$ is decoded as a codeword. This is because $\mathbf{y}$ and $\mathbf{z}_0$ are in the same coset. Hence $\mathbf{y}+(-\mathbf{z}_0) \in \mathscr{C}$

Suppose that there is a codeword $\mathbf{c}$ with

$$d(\mathbf{y}, \mathbf{y}+(-\mathbf{z}_0)) > d(\mathbf{y}, \mathbf{c}).$$

Then, equivalently,

$$d(\mathbf{z}_0, \mathbf{0}_n) > d(\mathbf{y}+(-\mathbf{c}), \mathbf{0}_n)$$

This means that the weight $w(\mathbf{z}_0) > w(\mathbf{y}+(-\mathbf{c}))$. However,

$$H(\mathbf{y}+(-\mathbf{c}))^{\mathsf{T}} = H\mathbf{y}^{\mathsf{T}} + (-H\mathbf{c}^{\mathsf{T}}) = H\mathbf{y}^{\mathsf{T}}$$

since $\mathbf{c}$ is a codeword, so $\mathbf{y}-\mathbf{c}$ has the same syndrome as $\mathbf{y}$ and according to Theorem 2.14, belongs to the same coset as $\mathbf{y}$ and has weight strictly less than that of the coset leader $\mathbf{z}_0$, which is a contradiction. □

Example 2.17 Consider the binary code $\mathscr{C}$ generated by the matrix

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

The parity check matrix is

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

The block messages are the elements of F_2^3. We generate our code by multiplying each

block message by the generator matrix from the left.

$$\begin{array}{lll} 000 & \rightarrow & \\ 100 & \rightarrow & \\ 010 & \rightarrow & \\ 001 & \rightarrow & \\ 110 & \rightarrow & \\ 101 & \rightarrow & \\ 011 & \rightarrow & \\ 111 & \rightarrow & \end{array} \quad \mathscr{C} = \left\{ \begin{array}{c} 000000, \\ 100110, \\ 010110, \\ 001011, \\ 110011, \\ 101110, \\ 011101, \\ 111000 \end{array} \right\}$$

As mentioned earlier, a codeword $\mathbf{c}$ consists of three message digits and three check digits as follows

$$\mathbf{c} = \underbrace{c_1c_2c_3}_{\text{message digits}} \quad \underbrace{c_4c_5c_6}_{\text{check digits}} .$$

Now we can formulate our look up table

Coset Leader	Coset	Syndrome
$\mathbf{v}_1 = 000000$	$\{000000, 100101, 010110, 001011, 110011, 101110, 011101, 111000\}$	$\begin{pmatrix}0\\0\\0\end{pmatrix}$
$\mathbf{v}_2 = 100000$	$\{100000, 000101, 110110, 101011, 010011, 001110, 111101, 011000\}$	$\begin{pmatrix}1\\0\\1\end{pmatrix}$
$\mathbf{v}_3 = 010000$	$\{010000, 110101, 000110, 011011, 100011, 111110, 001101, 101000\}$	$\begin{pmatrix}1\\1\\0\end{pmatrix}$
$\mathbf{v}_4 = 001000$	$\{001000, 101101, 011110, 000011, 111011, 100110, 010101, 110000\}$	$\begin{pmatrix}0\\1\\1\end{pmatrix}$
$\mathbf{v}_5 = 000100$	$\{000100, 100001, 010010, 001111, 110111, 101010, 011001, 111100$	$\begin{pmatrix}1\\0\\0\end{pmatrix}$
$\mathbf{v}_6 = 000010$	$\{000010, 100111, 010100, 001001, 110001, 101100, 011111, 111010\}$	$\begin{pmatrix}0\\1\\0\end{pmatrix}$
$\mathbf{v}_7 = 000001$	$\{000001, 100100, 010111, 001010, 110010, 101111, 011100, 111001\}$	$\begin{pmatrix}0\\0\\1\end{pmatrix}$
$\mathbf{v}_8 = 001100$	$\{001100, 101001, 011010, 000111, 111111, 100010, 010001, 110100\}$	$\begin{pmatrix}1\\1\\1\end{pmatrix}$

Suppose that we want to send the message 001101110. To encode this message, first we break it in blocks of three digits

$$001 \mid 101 \mid 110.$$

We obtain the corresponding codewords by multiplying each block by the generator matrix from the left. The result is

$$001011 \mid 101110 \mid 110011$$

which we can transmit.

Lets consider that the decoder receives 101011101110111111. First we break the received word into blocks of length 6 to obtain

$$\mathbf{y}_1 = 101011 \mid \mathbf{y}_2 = 101110 \mid \mathbf{y}_3 = 111111.$$

We calculate the syndromes of these vectors

$$H\mathbf{y}_1^\intercal = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \quad H\mathbf{y}_2^\intercal = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} H\mathbf{y}_3^\intercal = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

Now we can decode the vectors $\mathbf{y}_1, \mathbf{y}_2$ and $\mathbf{y}_3$ as follows

$$\begin{aligned} \mathbf{y}_1 &\to \mathbf{y}_1 + (-\mathbf{v}_2) = 001011 \\ \mathbf{y}_2 &\to \mathbf{y}_2 + (-\mathbf{v}_1) = 101110 \\ \mathbf{y}_3 &\to \mathbf{y}_3 + (-\mathbf{v}_8) = 110011. \end{aligned}$$

Finally, we decode each codeword into the sequence consisting of the first three digits to obtain the main message.

Notice that we may choose the vector $\mathbf{v}_8$ differently. For instance, if we choose $\mathbf{v}_8' = 010001$ then we would have decoded the previous received message as follows

$$\mathbf{y}_3 \to \mathbf{y}_3 + (-\mathbf{v}_8') = 101110$$

which is not the transmitted codeword. However, the minimum distance of our code is $d(\mathscr{C}) = 3$ thus, it is not expected to correct more than one error.

To overcome the scenario in which we will not be able to decode a received word uniquely, we may choose the coset leader arbitrary from the many coset elements with minimum weight as we have done in the previous example. Regardless this disadvantage, linear codes are widely used since they ease the minimum distance decoding scheme by using the syndrome decoding scheme [Wel04].

2.8 Software Application

In this section, we will implement a linear code using `SageMath` where numerous routines are defined regarding coding theory.

In the beginning, we need to define the finite field over which we will define our code. In our example, the finite field is F_3.

```
1 sage: F = GF(3)
```

By this point, we already know that we need a $k \times n$ matrix over F_{p^m} of linearly independent rows to be the generator matrix of a code which is meant to encode block messages of length k into codewords of length n. In our example, we will encode 3 digits block messages into 11 digits codewords that we obtain a large minimum distance. Now we form the row vectors of our matrix.

```
1 sage: u = vector(F, [1,2,0,1,0,0,3,1,1,0,0])
2 sage: v = vector(F, [0,2,1,0,1,2,0,2,0,1,0])
3 sage: w = vector(F, [2,2,0,0,0,0,1,0,2,2,1])
```

Then we define our matrix as follows.

```
1 sage: M = matrix([u,v,w])
```

Now we can define our code.

```
1 sage: C = LinearCode(M)
2 sage: C
3       [11, 3] linear code over GF(3)
4 sage: type(C)
5       <class 'sage.coding.linear_code.LinearCode_with_category'>
```

Now we will see the behaviour of some attributes of the linear code class.

```
1  sage: C.generator_matrix()
2        [1 2 0 1 0 0 0 1 1 0 0]
3        [0 2 1 0 1 2 0 2 0 1 0]
4        [2 2 0 0 0 0 1 0 2 2 1]
5  sage: C.parity_check_matrix()
6        [1 0 0 0 0 0 0 0 2 0 0]
7        [0 1 0 0 0 0 0 0 1 1 0]
8        [0 0 1 0 0 0 0 0 0 2 2]
9        [0 0 0 1 0 0 0 0 2 0 2]
10       [0 0 0 0 1 0 0 0 0 2 2]
11       [0 0 0 0 0 1 0 0 0 1 1]
12       [0 0 0 0 0 0 1 0 0 0 2]
13       [0 0 0 0 0 0 0 1 2 1 0]
```

We can even print the codewords of our code.

```
1 sage: for i in C: print(i)
```

Maybe the most important information to know about our code is its minimum distance.

```
1 sage: C.minimum_distance()
2       5
```

Now we know that our code can correct up to two errors.

There is a default attribute to standardize the generator matrix of a code. However, a matrix over a finite field is in general an immutable data type. Thus, we will not be able to use this attribute. In some cases, we will be able to use it (we will do so in another example). To standardize our code, we shall initialize the row vectors of our matrix before defining it. The following routines are supposed to help us formulate a standardized matrix using the operations from Theorem 2.5.

```
1  sage: def vectors_initializer(i,Mat):
2  sage:     if Mat[i][i] == 0:
3  sage:         L = []
4  sage:         LL= []
5  sage:         index = 0
6  sage:         for j in range(len(Mat[i])):
7  sage:             L.append(0)
8  sage:         for j in range(len(Mat)):
9  sage:             LL.append(copy(L))
10 sage:         for j in range(i,len(Mat[i])):
11 sage:             if Mat[i][j] != 0:
12 sage:                 index = j
13 sage:         for j in range(len(Mat)):
14 sage:             LL[j][index] = Mat[j][i]
15 sage:             LL[j][i]=Mat[j][index]
16 sage:         for j in range(len(Mat)):
17 sage:             for k in range(len(Mat[i])):
18 sage:                 if k != i:
19 sage:                     if k!= index:
20 sage:                         LL[j][k]= Mat[j][k]
21 sage:         for j in range(len(LL)):
22 sage:             LL[j] = vector(F,LL[j])
23 sage:         Mat = LL
24 sage:     if Mat[i][i] != 1:
25 sage:         scalar = 1/Mat[i][i]
26 sage:         for j in range(i,len(Mat[i])):
27 sage:             Mat[i][j] = Mat[i][j] * scalar
28 sage:     for k in range(len(Mat)):
29 sage:         if k != i:
30 sage:             if Mat[k][i] != 0 :
31 sage:                 Vector = (-Mat[k][i]) * Mat[i]
32 sage:                 Mat[k] = Mat[k] + Vector
33 sage:     return(Mat)
34 sage: def initialate (List):
35 sage:     for i in range(len(List)):
36 sage:         List = vectors_initializer(i,List)
37 sage:     return(List)
38 sage: def initialized_matrix(List):
39 sage:     L = copy(List)
40 sage:     L = initialate(L)
41 sage:     return(matrix(L))
```

A standardized matrix, which we will use to build a code equivalent to our code, is obtained by passing the row vectors of the main matrix as a list to the routine `initialized_matrix()`.

```
1  sage: MM = initialized_matrix([u,v,w])
```

Now we can use the previous matrix to build a code.

```
1  sage: CC = LinearCode(MM)
```

```
2  sage: CC.generatormatrix()
3        [1 0 0 2 0 0 1 2 1 2 1]
4        [0 1 0 1 0 0 1 1 0 2 1]
5        [0 0 1 1 1 2 1 0 0 0 1]
6  sage: CC.parity_check_matrix()
7        [1 0 0 0 0 0 0 0 2 0 0]
8        [0 1 0 0 0 0 0 0 1 1 0]
9        [0 0 1 0 0 0 0 0 0 2 2]
10       [0 0 0 1 0 0 0 0 2 0 2]
11       [0 0 0 0 1 0 0 0 0 2 2]
12       [0 0 0 0 0 1 0 0 0 1 1]
13       [0 0 0 0 0 0 1 0 0 0 2]
14       [0 0 0 0 0 0 0 1 2 1 0]
```

Maybe comparing the distances between the codewords of the codes `C` and `CC` is a good practice. In our example, we will compare their weights. Further comparisons are left for the reader. We define the following routines.

```
1  sage: def zeros_number (Vector):
2  sage:     number = 0
3  sage:     for i in Vector:
4  sage:         if i==0 :
5  sage:             number += 1
6  sage:     return(number)
7  sage: def number_nonzeros (Vector):
8  sage:     return(len(Vector) - zeros_number(Vector))
9  sage: def Hamming_weights (C):
10 sage:     L = []
11 sage:     for i in C:
12 sage:         L.append(len(i) - zeros_number(i))
13 sage:     L.sort()
14 sage:     return(L)
```

Now we can pass a code `C` to `Hamming_weights()` to obtain a sorted list of the weights of the codewords of `C`.

```
1  sage: CWeights = Hamming_weights(C); print(Cweights)
2  sage: CCWeights = Hamming_weights(CC); print(CCweights)
3        [0, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 8,
4                       8, 8, 8, 9, 9, 9, 9, 9, 9, 10, 10, 10, 10]
5        [0, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 8,
6                       8, 8, 8, 9, 9, 9, 9, 9, 9, 10, 10, 10, 10]
```

We recognize that both codes have the same list of weights.

Lets try to put our code `CC` in use. First we need to construct the channel which we use to transmit codewords. To do so, there are many constructors. In our example, we will construct a channel regarding the codewords vector space and the number of errors per codeword.

```
1  sage: n_err = 2
2  sage: G = F^11
3  sage: Chan = channels.StaticErrorRateChannel(G,n_err)
```

We can use this channel to transmit codewords. Thus we want to encode a random block message (since receiver is not supposed to know the main message until the end

of decoding process) into a codeword.

```
1 sage: message = vector(F,[randint(1,3),randint(1,3),randint(1,3)])
```

Now, we use the default encoder and we transmit the produced codeword. To transmit a word, we just need to pass it as an argument to the channel which we have defined.

```
1 sage: CW = CC.encode(message)
2 sage: received = Chan(CW)
3 sage: print(received)
4       (0, 0, 2, 0, 2, 1, 1, 1, 2, 0, 1)
```

We can check if the received message is an element of our code.

```
1 sage: received in CC
2       False
```

Now we move to decoding the received word. We can decode a word into a codeword or directly into the block message.

```
1 sage: decoded = CC.decode_to_code(received)
2 sage: print(decoded)
3       (2, 0, 2, 0, 2, 1, 1, 1, 2, 1, 1)
4 sage: possible_message = CC.decode_to_message(received)
5 sage: possible_message == message
6       True
```

By comparing the received word and the output of our decoder, we recognize that the errors occurred in the first and the tenth positions. We can try to transmit a message with number of errors more than that our code can correct.

```
1 sage: Errors = 3
2 sage: WorseChannel = channels.StaticErrorRateChannel(G,Errors)
```

Now by rerunning the following block of statements several times, sometimes we will obtain a `True` and sometimes we will obtain a `False`.

```
1 sage: MESSAGE = vector(F,[randint(1,3),randint(1,3),randint(1,3)])
2 sage: CodeWord = CC.encode(MESSAGE)
3 sage: RECEIVED = WorseChannel(CodeWord)
4 sage: DECODED = CC.decode_to_message(RECEIVED)
5 sage: DECODED == MESSAGE
```

For further readings regarding applying linear codes in `SageMath` we may read [SMa].

3 Binary Hamming Codes

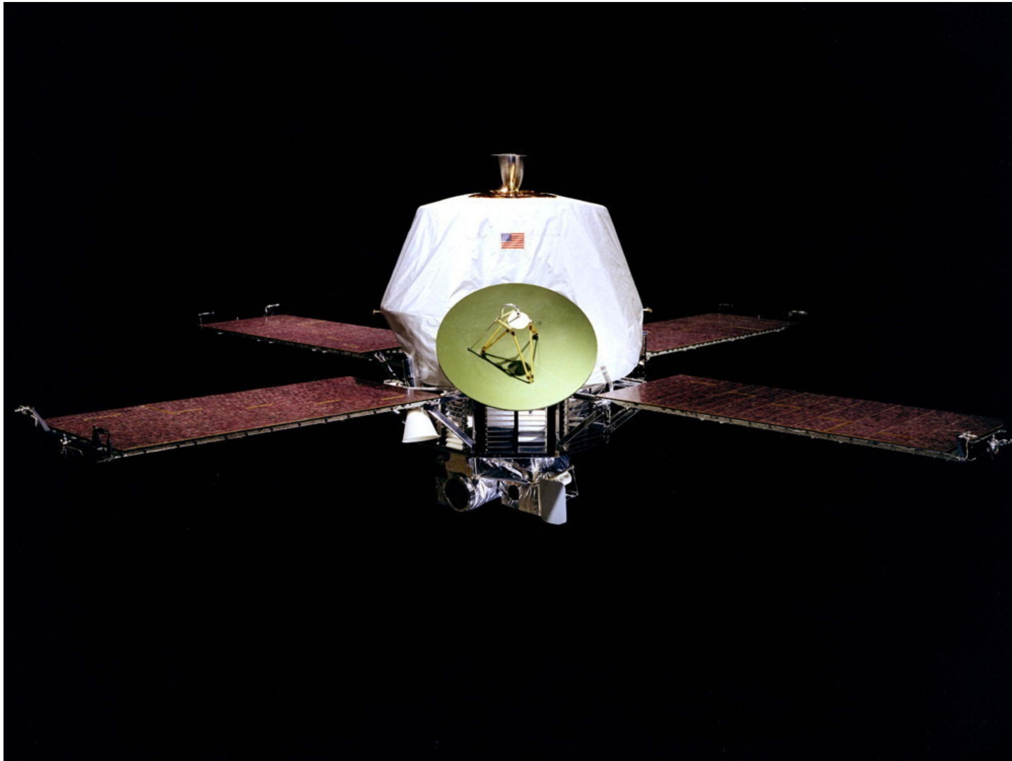

Figure 3.1: Mariner 9 [NASA]. A generalized hamming code was part of the concatenated code used in Mariner Mars mission [DM70]

The Hamming code, conceived by Richard Hamming at Bell laboratories in 1950, is the first error-correcting code to be used in applications [MMR08]. A Hamming code is substantially a perfect linear code able to correct a single error [MMR08].

3.1 Construction of Binary Hamming Codes

Since our work is an introduction, we will restrict to the binary case. The method we will consider in creating binary hamming codes can be understood as an inverse of the method we used by the general form of linear codes. Suppose r is a positive integer (this is going to be the number of check digits). Let $n = 2^r - 1$, this is going to be the length of the words and codewords. The block messages are binary messages of length $k = n - r$. Now formulate the matrix H in which the column vectors are the transpose of the distinct nonzero vectors of $\mathbf{V}_r$. In other words

$$H = [\mathbf{a}_1^\mathsf{T}, \cdots \mathbf{a}_n^\mathsf{T}]$$

where

$$\mathbf{a}_i \in \mathbf{V}_r \setminus \{\mathbf{0}_r\} \text{ for } i = 1, \cdots, n \quad \text{and} \quad \mathbf{a}_i \neq \mathbf{a}_j \text{ for } i \neq j.$$

Now we can consider H to be a parity check matrix of an $[n,k]$-code. This code is called a *binary* $[n,k]$*-hamming code* (for brevity, we will call it an $[n,k]$-hamming code). Notice that an $[n,k]$-hamming code is not uniquely defined. To follow the general form we used in defining linear codes, we will rearrange H into the form $[-A^{\mathsf{T}}, I_r]$. Notice that $-A^{\mathsf{T}} = A^{\mathsf{T}}$ since its an $r \times k$ binary matrix. Now we can obtain the generator matrix $G = [I_k, A]$. This method is obtained by combining the descriptions of [MMR08] and [Wel04]. In the following theorem, the properties of hamming codes are summed up.

Theorem 3.1 (cf. [Wel04]) *Any hamming code is a perfect single-error-correcting code.*

Proof: We first show that the minimum distance of such a Hamming code $\mathscr{C}$ with a parity check matrix H is at least 3. Denote the columns of H by $\mathbf{a}_i$ for $i = 1, \cdots, n$. Because $\mathscr{C}$ is a linear code, we know from Theorem 2.9 that the minimum distance $d(\mathscr{C})$ equals the minimum weight of a vector in $\mathscr{C}$.

Suppose that $\mathscr{C}$ has a codeword $\mathbf{u}$ of weight 1 with a nonzero entry in the ith place. Then

$$H\mathbf{u}^{\mathsf{T}} = \mathbf{a}_i = \mathbf{0}_r^{\mathsf{T}},$$

which is obviously not true.

Suppose $\mathscr{C}$ has a codeword $\mathbf{v}$ of weight 2, with nonzero entries in the ith and jth digits. Then

$$H\mathbf{v}^{\mathsf{T}} = \mathbf{a}_i + \mathbf{a}_j = \mathbf{0}_r^{\mathsf{T}}.$$

However, this means that H has 2 identical columns which is not true. Thus, $d(\mathscr{C}) \geq 3$.

Since the columns of H are the transposes of the distinct non-zero vectors of $\mathbf{V}_r$ which is a vector space and a commutative additive finite group, the sum of two different columns of H is another column of H. Lets formulate a codeword $\mathbf{w} \in \mathscr{C}$ which contains three 1's in positions i, j and m such that $\mathbf{a}_i + \mathbf{a}_j = \mathbf{a}_m$. Then $H\mathbf{w}^{\mathsf{T}} = \mathbf{0}_r^{\mathsf{T}}$. Thus, we will have codewords of length 3 and the minimum distance of $\mathscr{C}$ is 3.

To show that $\mathscr{C}$ is perfect just notice that the 1-sphere surrounding any codeword $\mathbf{x} \in \mathscr{C}$ will contain $1 + n = 2^r$ vectors. Since $\mathscr{C}$ contains $2^k = 2^{(n-r)}$ codewords, the union of these 1-spheres is the complete set of 2^n vectors of V_n which completes the proof. □

Before formulating an encoding/decoding algorithm, we need to discuss the following short note. Consider an $[n,k]$-hamming code $\mathscr{C}$ with parity check matrix H with columns denoted $\mathbf{a}_i$ for $i = 1, \cdots, n$. If the maximum number of errors in a received word is 1. We can represent a received word by

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

where $\mathbf{c}$ is a codeword in $\mathscr{C}$

$$\mathbf{e} = \mathbf{0}_n \quad \text{if } \mathbf{y} \in \mathscr{C}$$
$$\mathbf{e} = \mathbf{e}_i \text{ is an element of the standard basis of } \mathbf{V}_n \quad \text{if } \mathbf{y} \notin \mathscr{C}$$

Regarding the syndrome of $\mathbf{y}$, we write

$$\begin{aligned} H\mathbf{y}^{\mathsf{T}} &= H(\mathbf{c}+\mathbf{e})^{\mathsf{T}} \\ &= H\mathbf{c}^{\mathsf{T}} + H\mathbf{e}^{\mathsf{T}} \\ &= H\mathbf{e}^{\mathsf{T}}. \end{aligned}$$

If $\mathbf{y}$ is a codeword, then $H\mathbf{e} = H\mathbf{0}_n = \mathbf{0}_r^{\mathsf{T}}$ and $\mathbf{y}$ is encoded into $\mathbf{y}$. Otherwise, we conclude that $\mathbf{e} = \mathbf{e}_i$ has got a 1 in position i. We can obtain this position by comparing the syndrome of the received word with the columns of H. In other words, i is the position of $\mathbf{a}_i$ such that $H\mathbf{y}^{\mathsf{T}} = \mathbf{a}_i$. This short note is a combination of the descriptions of [Wel04] and [MMR08]. Now we are ready for the following elegant decoding algorithm for Hamming codes.

Algorithm 3

Step(1) Rearrange H such that we obtain a parity check matrix of the standard form $H = [-A^T, I_r]$. denote the columns of H by $\mathbf{a}_i$ for $i = 1, \cdots, n$

Step(2) Now we can formulate our generator matrix $G = [I_{n-r}, -(-A^T)^T] = [I_k, A]$.

Step(3) When a vector $\mathbf{y}$ is received, calculate its syndrome $\mathbf{v} = H\mathbf{y}^T$.

Step(4) Assuming only a single error, the syndrome decoding scheme gives:

(a) If $\mathbf{v} = H\mathbf{y}^T = \mathbf{0}_r^T$, then no error occurred, and $\mathbf{y}$ is a codeword.

(b) If $\mathbf{v} = H\mathbf{y}^T \neq \mathbf{0}_r^T$, decode $\mathbf{y}$ by assuming an error in the ith position for which $\mathbf{v} = \mathbf{a}_i$.

Example 3.2 The repetition code $\mathscr{C}$ which occurs in Example 1.10 is a $[3,1]$-hamming code constructed by taking $r = 2$ and $n = 2^2 - 1 = 3$ with parity check matrix

$$H = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

and generator matrix

$$G = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$$

Example 3.3 The $[7,4]$-Hamming code has parity-check matrix

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

and generator matrix

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

Suppose we receive the vector $\mathbf{y} = (1010110)$, so that $H\mathbf{y}^{\mathsf{T}} = (011)^{\mathsf{T}}$. Then, on the assumption that there is not more than a single error, we assume the error occurs in the first position and hence decode $\mathbf{y}$ as $\mathbf{c} = (0010110)$.

Notation *Even though binary hamming codes cannot correct more than* 1 *error per word, the advantage is that they have a high rate of efficiency. In other word, they need few number of check digits to decode correctly words of high length. We demonstrate this in the following table.*

Word Length n	***Check Digits*** r	***Message Digits*** k
1	*1*	*0*
3	*2*	*1*
7	*3*	*4*
15	*4*	*11*
31	*5*	*26*
63	*6*	*57*
⋮	⋮	⋮

Notice that to safely transmit 57 *digits, we need only* 6 *check digits.*

Because of the advantages of hamming codes, we find them used widely, specially in the case when errors occur rarely as in computer devises (as flash memories [MMR08]). In the following section we will enjoy applying hamming codes.

3.2 Software Application

In this section we will see a demonstration of binary hamming code in `SageMath`. For instance, we will define a $[15,11]-$hamming code `HC` and then check how encoding and decoding works. First we will define the $[15,11]-$hamming code over the field F_2.

```
1 sage: F = GF(2)
2 sage: HC = codes.HammingCode(F,4)
```

```
sage: HC
      [15, 11] Hamming Code over GF(2)
```

We can obtain the generator matrix and the parity check matrix as follows.

```
sage: GM = HC.generator_matrix();GM
      [1 0 0 0 0 0 0 0 0 0 0 0 0 1 1]
      [0 1 0 0 0 0 0 0 0 0 0 0 1 0 1]
      [0 0 1 0 0 0 0 0 0 0 0 0 1 1 0]
      [0 0 0 1 0 0 0 0 0 0 1 0 0 0 1]
      [0 0 0 0 1 0 0 0 0 0 1 0 0 1 0]
      [0 0 0 0 0 1 0 0 0 0 1 0 1 0 0]
      [0 0 0 0 0 0 1 0 0 0 1 0 1 1 1]
      [0 0 0 0 0 0 0 1 0 0 1 0 1 1 0]
      [0 0 0 0 0 0 0 0 1 0 1 0 1 0 1]
      [0 0 0 0 0 0 0 0 0 1 1 0 0 1 1]
      [0 0 0 0 0 0 0 0 0 0 0 1 1 1 1]
sage: PCM = HC.parity_check_matrix(); PCM
      [1 0 1 0 1 0 1 0 1 0 1 0 1 0 1]
      [0 1 1 0 0 1 1 0 0 1 1 0 0 1 1]
      [0 0 0 1 1 1 1 0 0 0 0 1 1 1 1]
      [0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
```

Notice that the generator matrix slightly differs from the one we defined in the construction method. The previous parity check matrix differs completely from ours. For this code, we can use the default decoders and encoder as we have done by the linear codes. However, we will build our encoding/decoding scheme. We will depend on some default routines. Thus, the encoding decoding scheme will not be identical to what we discussed earlier. Lets first transform our code into the standard form.

```
sage: HCSF = HC.standard_form()
sage: HCSF = HCSF[0]
sage: HCSF
      [15, 11] linear code over GF(2)
```

According to the knowledge of `SageMath`, what we obtained is a linear code in the general form (not a hamming code). However, we already know that it is a member of the hamming codes family.

```
sage: SFGM = HCSF.generator_matrix();SFGM
      [1 0 0 0 0 0 0 0 0 0 0 0 0 1 1]
      [0 1 0 0 0 0 0 0 0 0 0 0 1 0 1]
      [0 0 1 0 0 0 0 0 0 0 0 0 1 1 0]
      [0 0 0 1 0 0 0 0 0 0 0 1 0 0 1]
      [0 0 0 0 1 0 0 0 0 0 0 1 0 1 0]
      [0 0 0 0 0 1 0 0 0 0 0 1 1 0 0]
      [0 0 0 0 0 0 1 0 0 0 0 1 1 1 1]
      [0 0 0 0 0 0 0 1 0 0 0 1 1 1 0]
      [0 0 0 0 0 0 0 0 1 0 0 1 1 0 1]
      [0 0 0 0 0 0 0 0 0 1 0 1 0 1 1]
      [0 0 0 0 0 0 0 0 0 0 1 0 1 1 1]
sage: SFPCM = HCSF.parity_check_matrix();SFPCM
      [1 0 1 0 1 0 1 0 1 0 0 1 1 0 1]
      [0 1 1 0 0 1 1 0 0 1 0 1 0 1 1]
      [0 0 0 1 1 1 1 0 0 0 1 0 1 1 1]
      [0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
```

Notice that the parity check matrix differs from the one we built. However, the generator matrix is of the form which is familiar for us. This is going to help us in decoding the codewords into block messages. Now we will define a routine to help us generate block messages.

```
1  sage: def message_builder (K,d):
2  sage:     message = []
3  sage:     for i in range (0,d):
4  sage:         message.append(randint(0,1))
5  sage:     return(vector(K,message))
```

Here is another routine to help encode a block message into a codeword.

```
1  sage: def encode (vector, M):
2  sage:     return(vector*M)
```

Here is the decoding routine which decodes a received word into a codeword using a hamming code parity check matrix.

```
1  sage: def decode (vect, PCM):
2  sage:     Corrector = []
3  sage:     for i in PCM.columns():
4  sage:         Corrector.append(0)
5  sage:     position = -1
6  sage:     for i in PCM.columns():
7  sage:         position += 1
8  sage:         if C.syndrome(vect) == i:
9  sage:             Corrector[position]= 1
10 sage:     print("Corrector = ", Corrector)
11 sage:     return(vect+vector(F,Corrector))
```

In addition, we need a routine which decodes a codeword into a block message depending on the number of check digits. Notice that in our decoding process, we are considering the codeword to be generated by a generator matrix of the standard form.

```
1  sage: def message_decoder (K,check,vect):
2  sage:     vect = list(vect)
3  sage:     vect = vect[0:2^check-1-check]
4  sage:     return(vector(K,vect))
```

We will define our channel and then put all of these routines in use.

```
1  sage: n_err = 1
2  sage: G = F^(r^2-1)
3  sage: Chan = channels.StaticErrorRateChannel(G,n_err)
```

We generate a random block message which is unknown for us.

```
1  sage: message = message_builder(F,2^r-1-r)
```

Now we encode our block message into a codeword using our generator matrix in standard form.

```
1  sage: encoded = encode(message,SFGM)
```

We transmit the encoded message through the channel which we defined.

```
1  sage: received = Chan(encoded);received
```

```
2       (0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0)
```

We decode the received word into a codeword.

```
1  sage: decoded = decode(received,SFPCM)
2        Corrector = [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
3  sage: decoded
4        (0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0)
```

The decoded word can be decoded into a block message.

```
1  sage: Exp_message = message_decoder(F,r,decoded)
2  sage: Exp_message == message
3        True
4  sage: Exp_message
5        (0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1)
```

For further readings [SMa].

Bibliography

[DM70] Bernbard Dorscb and Warner H. Miller. *NASA Technical Note, Error Controling Using a Concatenated Code.* National Aeronautics and Space Adminstration, 1970, pp. 12,13.

[HK03] Toyokazu Hiramatsu and Günter Köhler. *Coding Theory and Number Theory.* Springer Science+Business Media Dordrecht, 2003, pp. 1-7.

[MMR08] Rino Micheloni, Alessia Marelli, and Roberto Ravasio. *Error Correcting Codes for Non-Volatile Memories.* Springer Science+Business Media B.V., 2008, pp. 35-38, 145-165.

[Moo05] Todd K. Moon. *Error Correcting Coding, Mathematical Methods and Algorithms.* John Wiley & Sons, Inc, 2005, p. 28.

[NASA] National Aeronautics and Space Adminstration. Mariner 9 Image. `https://www.jpl.nasa.gov/missions/mariner-9-mariner-i`. Accessed: 5. Sept. 2021.

[SMa] The Sage Development Team. *Sage 9.3 Reference Manual: Coding Theory.* 2021, pp. 3-31.

[SMb] The Sage Development Team. *Sage 9.4 Reference Manual: Finite Rings, Release 9.4.* 2021, pp. 39-61.

[SMc] The Sage Development Team. *Sage Constructions, Release 9.4.* 2021, pp. 31-41.

[Wel04] Dominic Welsh. *Codes and Cryptography.* Oxford Science Publications, 2004, pp. 1-63.

[WJu18] Tohman W. Judson. *Abstract Algebra Theory and Application.* Orthogonal Publishing l3c, 2018, pp. 88-94, 108-136, 233-257, 313-321, 322-363.

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 14. September 2021