UNIVERSIDAD
DE GRANADA

Facultad de Ciencias
E.T.S. Ingenierías Informática y de Telecomunicación

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO DE FIN DE GRADO

# El criptosistema de McEliece

Presentado por:
Antonio Merino Gallardo

Tutor:
Pedro A. García Sánchez
*Departamento de Álgebra*

Curso académico 2021-2022

# El criptosistema de McEliece

Antonio Merino Gallardo

Antonio Merino Gallardo *El criptosistema de McEliece*.

Trabajo de fin de Grado. Curso académico 2021-2022.

**Responsable de tutorización**

Pedro A. García Sánchez
*Departamento de Álgebra*

Doble Grado en Ingeniería Informática y Matemáticas

Facultad de Ciencias
E.T.S. Ingenierías Informática y de Telecomunicación

Universidad de Granada

# Índice general

# Introducción

La confidencialidad e integridad de la mayoría de las comunicaciones que se llevan a cabo por internet está asegurada por sistemas criptográficos que serán inseguros en el momento en que se disponga de ordenadores cuánticos de alta capacidad. Es el caso de criptosistemas tan extendidos como RSA, DSA y ECDSA, que quedarían totalmente comprometidos por el algoritmo cuántico de Shor [Sho94]. Es aún incierto cuándo se construirían ordenadores cuánticos de alta capacidad en el caso de que finalmente se consiga. En cualquier caso, la gran amenaza que plantean hace necesario estar preparados con suficiente margen para dicho escenario.

Con dicho objetivo es con el que nació el campo de la criptografía post-cuántica, dedicada al diseño de criptosistemas que no sean vulnerables a los ordenadores cuánticos. Uno de los criptosistemas que, tras un exhaustivo criptoanálisis, no se ha conseguido romper ni con algoritmos tradicionales ni con algoritmos cuánticos es el que propuso McEliece en 1978 [McE78]. Es en este criptosistema en el que se fundamenta la reciente propuesta Classic McEliece [BCL+17] que busca convertirlo en un estándar.

El criptosistema de McEliece toma como base la teoría de códigos correctores de errores. Esta teoría estudia cómo controlar los errores que se producen al transmitir un mensaje a través de un canal con ruido. La idea es añadir redundancia a los mensajes antes de mandarlos, de un cierto modo que permita que el receptor del mensaje pueda corregir hasta un determinado número de errores, pero sin comprometer en exceso la eficiencia en la comunicación. El estudio de esta teoría es clave para comprender el funcionamiento del criptosistema de McEliece.

De este modo, una importante parte de este trabajo se dedicará a los códigos correctores de errores, centrándose en las familias de los códigos lineales, códigos cíclicos y códigos Goppa. Se describirá la estructura matemática que presentan y cómo se emplean para la corrección de errores. Las principales fuentes consultadas a este propósito son [Hil86], [Rom92] y [VL82].

Además del estudio teórico de los códigos, se hará uso del software matemático SageMath para trabajar con ejemplos de los códigos lineales y cíclicos, y se realizará una implementación de los códigos Goppa binarios en Python usando librerías de SageMath. SageMath [Tea21] es un software matemático de código abierto que se construye sobre muchos otros paquetes de código abierto como NumPy, SciPy, matplotlib, Sympy, Maxima o GAP. Su amplia variedad de funcionalidades matemáticas puede ser accedida mediante un lenguaje de programación basado en Python, lo que permite trabajar con él de forma cómoda.

Además, todos los ejemplos implementados serán plasmados en cuadernos de Jupyter [KRKP+16], otro software libre que permite integrar el código desarrollado con explicaciones a modo de cuaderno, lo que resulta en una potente herramienta a nivel didáctico y que permite promover la reproducibilidad de los experimentos.

Una vez desarrollada la teoría de códigos correctores de errores, se explicará cómo se usa para construir el sistema criptográfico diseñado por McEliece. Se analizará su seguridad y una

de las principales variantes propuestas hasta ahora, el criptosistema de Niederreiter [Nie86]. La clase implementada para los códigos Goppa binarios será la base para implementar los procesos de cifrado y descifrado, tanto con el criptosistema de McEliece, como con el de Niederreiter.

Todo el código desarrollado, del que se han extraído los ejemplos que aparecen a lo largo de este documento, puede encontrarse en GitHub:

`https://github.com/amerigal/criptosistema-de-mceliece`.

## Objetivos iniciales y resultados alcanzados

Los principales objetivos que se plantearon fueron los siguientes:

1. estudiar la teoría de códigos correctores de errores, con especial énfasis en los códigos lineales, códigos cíclicos y códigos Goppa,
2. emplear algún software matemático para trabajar de forma práctica con los conceptos presentados sobre códigos correctores de errores,
3. estudiar el criptosistema de McEliece, analizando su seguridad y ataques que se le han realizado e
4. implementar el criptosistema de McEliece.

La consecución de los dos primeros objetivos queda reflejada en el desarrollo de los Capítulos 1 a 4. Como software matemático se ha elegido SageMath, por ser una de las opciones de código abierto más versátiles y con la que se puede trabajar de forma cómoda mediante un lenguaje basado en Python.

El tercer objetivo se resuelve en el Capítulo 5.

Finalmente, el último objetivo queda cubierto con la implementación de los códigos Goppa binarios descrita en el Capítulo 4 y con la implementación del proceso de cifrado y descifrado del criptosistema de McEliece que se encuentra en el Capítulo 5. Además, se ha implementado también la versión del criptosistema propuesta por Niederreiter.

Los conocimientos previos más relevantes para la consecución de estos objetivos han sido sobre álgebra lineal, cuerpos finitos, anillos de polinomios y clases de complejidad computacional.

# Resumen

## Breve resumen

Este trabajo desarrolla la teoría de códigos correctores de errores en que se fundamenta el criptosistema de McEliece y se apoya en ella para el estudio de este sistema criptográfico.

De este modo, se formalizan primeramente conceptos clave en la teoría de codificación de la información como son el de alfabeto, código o capacidad de corrección de errores. Estos conceptos sentarán las bases sobre las que desarrollar el resto la teoría.

El siguiente paso es dotar de una estructura matemática a los códigos, construyéndolos así sobre cuerpos finitos. Esto lleva al estudio en profundidad de los códigos lineales, cubriendo importantes conceptos como los de matriz generadora o matriz de paridad así como los procedimientos de codificación y decodificación.

Seguidamente, el estudio se centra en los códigos cíclicos, una familia de códigos lineales con interesantes propiedades matemáticas que nos introduce en el uso de polinomios para representar y trabajar con las palabras de un alfabeto.

El desarrollo de la teoría de códigos correctores de errores finaliza presentando la familia de códigos lineales en la que se fundamenta el criptosistema de McEliece: los códigos Goppa. Se describe la base teórica de estos códigos y se estudia el algoritmo de decodificación de Patterson.

Todo este desarrollo permite describir de forma sencilla y precisa el criptosistema de McEliece, explicando cómo los códigos correctores de errores permiten la construcción de un esquema criptográfico. Se cubren cuestiones clave como la principal modificación propuesta para este criptosistema, el análisis de su seguridad o la reciente propuesta que aboga por su estandarización.

**Palabras clave:** códigos correctores de errores, códigos lineales, códigos Goppa, criptografía post-cuántica, criptosistema de McEliece

## Extended summary in English

This document develops the theory of error correcting codes and explains how they can be used to construct a cryptographic scheme, studying the McEliece cryptosystem.

In this sense, the two main areas that this work covers are coding theory and cryptography, both lying at the intersection of computer science and mathematics. The first three chapters have more of a theoretical and mathematical approach while exemplifying how software can be used to work with error correcting codes. Chapter 4 combines the mathematical description of Goppa codes with the practical implementation of this family of codes along with Patterson's algorithm to decode them. Finally, Chapter 5 is the one where cryptography

is studied, again with a practical approach describing and implementing the cryptographic schemes and analysing their security.

A more in depth description of the contents of each chapter is included below.

**Chapter 1:** This chapter presents key concepts of coding theory. An alphabet is a set of symbols, words are made up of symbols and codes are sets of words called codewords. A block code is a code in which all the words have the same length. The Hamming distance is introduced as a way of measuring the distance between words of the same length by counting the number of positions in which they differ. This Hamming distance of words leads to the notion of the minimum distance of a code, which is the least distance between its words.

Next, the focus is on the capacity of codes to detect and correct errors. When a codeword is transmitted through a channel, errors can occur, in the sense that certain positions may end up with a different value. The capacity of a code to detect and correct those errors is proved to be strictly related to its minimum distance. The task of decoding will be that of correcting errors.

Finally, alphabets are set to be finite fields, allowing to introduce the weight of a word and the weight of a code as well as an alternative way to express the occurrence of errors in the transmission of a word.

**Chapter 2:** This chapter addresses linear codes. If an alphabet is set to be a finite field, words of a given length are elements of the vector space that is obtained by taking the cartesian product of multiple copies of the alphabet. In this sense, a linear code is a linear subspace of that cartesian product. A generator matrix for a linear code is a matrix for which the rows are a basis of the code. The task of encoding consists in multiplying by this matrix.

An equivalence relation that preserves the capacity to detect and correct errors is introduced for these codes. This relation is key in the sense that every code is proved to be equivalent to another code in which the generator matrix has a simplified form called systematic form.

Another crucial concept is that of the dual of a code, consisting of the words that are orthogonal to all of the words of a code. The dual code is a linear code whose generator matrix is said to be a partity check matrix for the original code. The parity check matrix allows us to obtain important results related to the minimum distance of a code.

Lastly, a method for decoding linear codes is presented. It is based on the concept of the syndrome of a word, which is obtained by multiplying by the parity check matrix.

Through all this chapter, the coding library of SageMath is used to exemplify the different concepts presented for linear codes.

**Chapter 3:** This chapter focuses on the study of cyclic codes. Cyclic codes are linear codes in which every circular shift of a codeword is also a codeword. These codes can be viewed as ideals in a polynomial ring. In this sense, words are considered to be polynomials with coefficients in a finite field. The polynomial generator is a generator of the ideal satisfying certain properties. An interesting result is that there is a bijection between the cyclic codes of a given length $n$ and the monic divisors of $x^n - 1$.

The generator matrix can be directly obtained from the generator polynomial. Another important polynomial related to these codes is the check polynomial, from which the parity check matrix can be obtained.

The tasks of encoding and decoding with these codes are simplified by means of the polyno-

mial expression of words and the generator polynomial.

As in the previous chapter, the coding library of SageMath is used to show different examples of cyclic codes.

**Chapter 4:** This chapter introduces Goppa codes. The mathematical construction of Goppa codes is explained in detail. Given a non-prime finite field, the key elements for defining a Goppa code are the Goppa polynomial, which is a polynomial with coefficients in the field, and the defining set, which is a subset of the field such that none of its elements is a root of the Goppa polynomial. A Goppa code is said to be irreducible when the Goppa polynomial is irreducible. Goppa codes are linear by construction.

In this line, the construction of a parity check matrix is described. This matrix is the key to obtain a result that gives bounds for the dimension and minimum distance of Goppa codes.

Next, the focus is set on binary Goppa codes, which is the family of codes on which the McEliece cryptosystem is based. For irreducible binary Goppa codes, a better bound for the minimum distance is obtained. In particular, if the irreducible Goppa polynomial has degree t, then the minimum distance of the code is $d \geq 2t + 1$. This implies that such codes can correct at least $t$ errores. The decoding of these codes is described with Patterson's algorithm. This algorithm is based on two polynomials: the syndrome polynomial and the error locator polynomial. The coefficients of the syndrome polynomial can be obtained by multiplying a word by the parity check matrix. Regarding the error locator polynomial, its roots indicate the positions of the word in which an error has occurred. Some of its steps involve the application of the extended Euclidean algorithm and the calculation of square roots in a quotient ring. Patterson's algorithm provides us with an efficient method for decoding binary Goppa codes.

Along with this theoretical study of Goppa codes, the implementation of binary Goppa codes is described. It has been implemented in Python and using certain libraries provided by SageMath. A class for this family of codes is defined. The code is constructed by providing the Goppa polynomial and the defining set. Its methods provide the basic functionalities needed to work with these codes, such as the obtention of the parity check matrix or the syndrome polynomial of a word. Another important method is the decoding one, which basically consists in the implementation of Patterson's algorithm. Besides from its description, this class is used in this chapter to show several examples of the concepts presented.

**Chapter 5:** This chapter focuses on the study and implementation of the McEliece cryptosystem, a public-key cryptosystem based on error correcting codes. In particular, it uses binary Goppa codes. The idea is that the generator matrix of an irreducible binary Goppa code is obfuscated by multiplying it by two regular matrices. The result of this product is the matrix that will constitute the public key along with the degree of the Goppa polynomial. Regarding the private key, it consists of the Goppa polynomial and the regular matrices indicated previously. Encryption consists in encoding the message with the matrix of the public key and adding as many errors as the degree of the Goppa polynomial. The information contained in the private key allows to correct the errors on the word by applying Patterson's algorithm and, thus, it allows the decryption of a given ciphertext.

The main modification of the McEliece cryptosystem is later presented: the Niederreiter cryptosystem. It is a dual version of the McEliece cryptosystem in the sense that it uses a parity check matrix instead of a generator matrix. Niederreiter proposed the use of a different family of error correcting codes, but it was later proved to be insecure. However, his scheme is secure when using binary Goppa codes. The Niederreiter cryptosystem results in a more

efficient version of the McEliece cryptosystem with equivalent security when used with binary Goppa codes.

Both the McEliece and Niederreiter cryptosystems are implemented taking advantage of the previously implemented class for binary Goppa codes.

The security of these cryptosystems is next analysed. They are both closely related to the problem of decoding a general linear code, which is known to be $NP$-complete. The most effective technique to attack them is called information-set decoding. A basic approach is explained and several more advanced proposals are presented, with Stern's attack being the model.

Lastly, the Classic McEliece proposal is introduced. It is a proposal based on the McEliece cryptosystem presented for the NIST call for cryptosystems that resist quantum computers. Some of its main features are described.

**Keywords:** error correcting codes, linear codes, Goppa codes, post-quantum cryptography, McEliece cryptosystem

# 1 Códigos correctores de errores

La transmisión de información por un canal puede estar sujeta a la ocurrencia de errores que modifiquen el mensaje enviado dificultando su interpretación por parte del receptor. Para lidiar con esta situación, se puede incluir información redundante en el mensaje. Por ejemplo, si quisiéramos mandar un mensaje en binario, podríamos mandar cada bit dos veces, esto es, reemplazando los 0s por 00s y los 1s por 11s. De este modo, si se produjera la alteración de un solo bit al transmitir el mensaje, el receptor sería capaz de reconocer que ha ocurrido un error identificando una pareja con bits distintos. Sin embargo, podría no ser capaz de determinar cuál era la pareja original, no pudiendo averiguar entonces cuál era el mensaje mandando originalmente.

Un enfoque simple para abordar esto sería repetir los bits tres veces en vez de dos, reemplazando los 0s por 000s y los 1s por 111s. En este caso, si solo se produjera la alteración de un bit, el receptor sería capaz no solo de identificar la tripleta afectada, sino también de deducir cuál era el mensaje transmitido originalmente. Para ello, habría que identificar las tripletas inválidas con la correspondiente tripleta válida más *cercana*, esto es, identificando las tripletas 001, 010 y 100 con la 000 y las tripletas 110, 101 y 011 con la 111. Sin embargo, si se produjera la alteración de dos bits de una misma tripleta, este esquema no permitiría obtener el mensaje original, pues se decodificaría erróneamente la tripleta afectada.

La repetición de símbolos es solo un caso particular de las transformaciones que estudiaremos para añadir redundancia a un mensaje. En general, la idea es diseñar una manera de codificar las palabras de un alfabeto de partida en palabras del mismo u otro alfabeto que serán las que finalmente se transmitan. Se buscará que la decodificación sea sencilla y tenga buenas propiedades de detección y corrección de errores, pero tratando de mantener la longitud del mensaje codificado lo más pequeña posible.

Para la elaboración de este capítulo se han consultado ([Hil86], Capítulo 1) y ([Rom92], Capítulo 4).

## 1.1. Codificación de la información

Introduciremos primeramente los conceptos fundamentales sobre los que construir la teoría de codificación de la información, a saber, el de *alfabeto* y el de *código*. Abordaremos también una cuestión clave como es el problema de conseguir un código *decodificable de manera única* para el que proporcionaremos una conveniente solución.

**Definición 1.1** (Alfabeto)**.** Un *alfabeto* es un conjunto finito $\mathcal{A} = \{a_1, ..., a_q\}$ a cuyos elementos llamaremos *símbolos*. Una *palabra* de longitud $n$ sobre $\mathcal{A}$ es un elemento de $\mathcal{A}^n$, esto es, una $n$-upla de símbolos de $\mathcal{A}$ que representaremos tanto en notación vectorial como por yuxtaposición.

De este modo, $a = (a_{i_1}, a_{i_2}, ..., a_{i_k})$ con $a_{i_k} \in \mathcal{A}$ es una palabra de longitud $k$ que también notaremos por $a = a_{i_1} a_{i_2} \dots a_{i_k}$.

Dado un alfabeto $\mathcal{A}$, notaremos por $\mathcal{A}^*$ al conjunto de las palabras que se pueden formar con los símbolos de $\mathcal{A}$ incluyendo la palabra vacía y $\mathcal{A}^+$ si excluimos la palabra vacía. Formalmente,

$$\mathcal{A}^* = \bigcup_{n \geq 0} \mathcal{A}^n \quad y \quad \mathcal{A}^+ = \bigcup_{n > 0} \mathcal{A}^n.$$

A la hora de codificar la información para su posterior transmisión, tendremos un alfabeto de partida $\mathcal{A}_1$ y tomaremos otro alfabeto $\mathcal{A}_2$. La idea será definir una aplicación $C : \mathcal{A}_1 \to \mathcal{A}_2^+$ inyectiva que a cada símbolo de $\mathcal{A}_1$ le asocie una palabra no nula formada por símbolos de $\mathcal{A}_2$. En general, no estaremos tan interesados por la regla de asignación en sí, sino por su imagen, que será la que nos permita estudiar las propiedades del esquema de codificación. Dicha imagen podrá ser un subconjunto no vacío de $\mathcal{A}_2^+$ arbitrario.

**Definición 1.2** (Código)**.** Un *código* sobre un alfabeto $\mathcal{A}$ es un subconjunto $\mathcal{C} \subset \mathcal{A}^+$ no vacío. A los elementos de $\mathcal{C}$ los llamaremos *palabras código*.

**Ejemplo 1.1.** En el primer caso comentado, consistente en la duplicación de bits, los alfabetos serían $\mathcal{A}_1 = \mathcal{A}_2 = \{0, 1\}$, la aplicación de codificación $C : \{0, 1\} \to \{0, 1\}^+$ dada por $C(0) = 00$ y $C(1) = 11$, y el código $\mathcal{C} = \{00, 11\}$.

Si un alfabeto tiene $q$ símbolos, los códigos construidos sobre este se llaman *códigos q-arios*. En el caso del alfabeto $\{0, 1\}$ se llaman *códigos binarios* y son los más extendidos.

La codificación de una palabra se basará en la codificación de cada uno de sus símbolos. En este sentido, podemos extender de manera natural la aplicación $C : \mathcal{A}_1 \to \mathcal{A}_2^+$ que codifica cada símbolo del alfabeto a $C : \mathcal{A}_1^* \to \mathcal{A}_2^*$ que codifica cada palabra sobre el alfabeto $\mathcal{A}_1$ mediante la codificación de cada uno de sus símbolos y que notaremos de igual forma. La inyectividad de dicha aplicación extendida a $\mathcal{A}_1^*$ es una propiedad esencial que buscamos en un código, pues significará que es *decodificable de manera única*. Para obtener dicha inyectividad no es suficiente la inyectividad de la aplicación que codifica los símbolos. Podemos considerar, por ejemplo, $C : \{0, 1, 2\} \to \{0, 1\}^+$ dada por $C(0) = 0$, $C(1) = 1$ y $C(2) = 00$. Pese a ser inyectiva, no lo es su extensión a todas las palabras sobre $\{0, 1, 2\}$, pues la palabra $00 \in \{0, 1\}^+$ puede decodificarse como 00 o como 2.

La condición suficiente más sencilla que podemos plantear para obtener un código decodificable de manera única es que todas las palabras del código tengan la misma longitud. De esta manera, la decodificación se simplificaría a separar el mensaje en bloques de longitud la de las palabras código y decodificar cada uno de esos bloques, no habiendo posibilidad entonces a ambigüedad.

**Definición 1.3** (Código de bloque)**.** Sea $\mathcal{A}$ un alfabeto y $\mathcal{C}$ un código sobre $\mathcal{A}$ con $|\mathcal{C}| = M$. Se dice que $\mathcal{C}$ es un *código de bloque con parámetros* $(n, M)$ o que $\mathcal{C}$ es un $(n, M)$*-código* si todas las palabras código de $\mathcal{C}$ tienen una longitud fija $n$.

En este sentido, cuando a partir de ahora se considere un código, nos estaremos refiriendo implícitamente a un código de bloque.

Resuelta la unicidad de decodificación, las propiedades que nos interesa estudiar son las de detección y corrección de errores. Para hablar de ellas es conveniente introducir el concepto

de distancia, que nos dará una noción de *proximidad* entre las palabras construidas sobre un alfabeto.

## 1.2. Distancia de Hamming

De entre las múltiples distancias que podemos definir sobre un alfabeto para medir la proximidad entre palabras de una misma longitud, nos centraremos en la distancia de Hamming. Como observaremos cuando discutamos la tarea de decodificación, el uso de esta distancia nos permitirá decodificar con la mayor probabilidad bajo ciertas suposiciones.

En este sentido, formalizaremos dicha distancia y veremos las propiedades que satisface. Todo esto nos llevará a introducir un concepto clave que trata sobre la proximidad de las palabras dentro de un código y sobre el que se construyen interesantes resultados sobre la capacidad de detección y corrección de errores.

**Definición 1.4** (Distancia de Hamming)**.** Sea $\mathcal{A}$ un alfabeto. Llamaremos distancia de Hamming a la aplicación:

$$\mathrm{d} : \mathcal{A}^n x \mathcal{A}^n \to \mathbb{N}$$

$$\mathrm{d}(x, y) = \#\{i \in \{1, ..., n\} : x_i \neq y_i\},$$

esto es, al número de posiciones en que $x, y \in \mathcal{A}^n$ difieren.

**Ejemplo 1.2.** Sean $x = (1, 0, 0, 0),\ y = (1, 0, 2, 3) \in \{0, 1, 2, 3\}^4$. Entonces, $\mathrm{d}(x, y) = 2$.

La distancia de Hamming es, de hecho, una distancia en $\mathcal{A}^n$.

**Proposición 1.1.** *La distancia de Hamming* $\mathrm{d} : \mathcal{A}^n x \mathcal{A}^n \to \mathbb{N}$ *verifica las siguientes propiedades:*

1. $\mathrm{d}(x, y) \geq 0$ *para todo* $x, y \in \mathcal{A}^n$ *con* $\mathrm{d}(x, y) = 0$ *si, y solo si,* $x = y$ *(no negatividad).*
2. $\mathrm{d}(x, y) = \mathrm{d}(y, x)$ *para todo* $x, y \in \mathcal{A}^n$ *(simetría).*
3. $\mathrm{d}(x, z) \leq \mathrm{d}(x, y) + \mathrm{d}(y, z)$ *para todo* $x, y, z \in \mathcal{A}$ *(desigualdad triangular).*

*Demostración.* Las propiedades 1 y 2 se siguen directamente de la definición de distancia de Hamming. Probemos la propiedad 3.

Sean $x, y, z \in \mathcal{A}$. Para cada $i \in \{1, ..., n\}$ tal que $x_i \neq z_i$, debe ser $x_i \neq y_i$ o $y_i \neq z_i$, pudiendo verificarse las dos. Por lo tanto,

$$\{i \in \{1, ..., n\} : x_i \neq z_i\} \subseteq (\{i \in \{1, ..., n\} : x_i \neq y_i\} \cup \{i \in \{1, ..., n\} : y_i \neq z_i\}),$$

de modo que,

$$\begin{aligned} \#\{i \in \{1, ..., n\} : x_i \neq z_i\} &\leq \#(\{i \in \{1, ..., n\} : x_i \neq y_i\} \cup \{i \in \{1, ..., n\} : y_i \neq z_i\}) \\ &\leq \#\{i \in \{1, ..., n\} : x_i \neq y_i\} + \#\{i \in \{1, ..., n\} : y_i \neq z_i\} \end{aligned}$$

esto es,

$$\mathrm{d}(x, z) \leq \mathrm{d}(x, y) + \mathrm{d}(y, z). \qquad \square$$

**Definición 1.5** (Distancia mínima de un código)**.** Dado un código $\mathcal{C}$, se define su distancia mínima $\mathrm{d}(\mathcal{C})$ como la menor distancia no nula entre sus palabras, esto es,

$$\mathrm{d}(\mathcal{C}) = \min\{\mathrm{d}(x,y) : \ x,y \in \mathcal{C}, \ \ x \neq y\}.$$

**Ejemplo 1.3.** El código $\mathcal{C} = \{000, 111\} \subseteq \{0,1\}^3$ tiene distancia mínima $\mathrm{d}(\mathcal{C}) = 3$.

Es esta distancia mínima de un código la que nos permitirá establecer cotas del mínimo de capacidad de detección y corrección de errores que tiene un código.

## 1.3. Capacidad de detección y corrección de errores

Formalizaremos los conceptos de detección y corrección de errores con la idea de obtener importantes resultados acerca de la capacidad que posee un código en relación a ellos. Para presentar dichos conceptos contemplaremos las modificaciones que puede sufrir una palabra código en su transmisión y su relación con el resto de palabras código. En este sentido, dichos conceptos no dependen del alfabeto de partida ni de la regla de asignación, pero sí del código $\mathcal{C}$ construido sobre un alfabeto $\mathcal{A}$.

Llamaremos *palabra recibida* a la palabra $c' \in \mathcal{A}^+$ resultante de transmitir una cierta palabra código $c \in \mathcal{C}$. Cabe notar que $c$ y $c'$ tendrán la misma longitud, pero, por la posible ocurrencia de errores en la transmisión, no tiene por qué verificarse $c' = c$ y no necesariamente pertenecerá $c'$ al código $\mathcal{C}$.

Será necesario primero precisar cómo cuantificar los errores producidos en una transmisión.

**Definición 1.6.** Sea $\mathcal{C}$ un código sobre $\mathcal{A}$ y $c \in \mathcal{C}$ una palabra código. Se dice que *se han cometido $t$ errores* al transmitir $c$ si la palabra recibida $c' \in \mathcal{A}^+$ verifica $\mathrm{d}(c,c') = t$.

**Definición 1.7.** Se dice que un código $\mathcal{C}$ *detecta $t$* errores si cuando se cometen como mucho $t$ errores al transmitir una palabra código, la palabra recibida no es una palabra código.

Podemos observar que si $\mathcal{C}$ detecta $t$ errores, entonces es claro que también detecta $r$ errores para todo $0 \leq r \leq t$.

**Definición 1.8** (Código $t$-detector)**.** Se dice que un código $\mathcal{C}$ es *$t$-detector* si detecta $t$ errores pero no detecta $t+1$ errores.

**Definición 1.9.** Se dice que un código $\mathcal{C}$ *corrige $t$ errores* si cuando se cometen como mucho $t$ errores al transmitir una palabra código, es posible determinar unívocamente la palabra código correcta como aquella que se encuentra a distancia mínima de la palabra recibida.

De manera análoga a la detección de errores observamos que si $\mathcal{C}$ corrige $t$ errores, entonces también corrige $r$ errores para todo $0 \leq r \leq t$.

**Definición 1.10** (Código $t$-corrector)**.** Se dice que un código $\mathcal{C}$ es *$t$-corrector* si corrige $t$ errores pero no corrige $t+1$ errores.

Como habíamos anticipado, podemos caracterizar las capacidades de detección y corrección de errores de un código mediante su distancia mínima.

**Proposición 1.2.** *Sea $\mathcal{C} \subseteq \mathcal{A}$ un código y sea $t = \mathrm{d}(\mathcal{C}) - 1$. Entonces $\mathcal{C}$ es un código $t$-detector.*

*Demostración.* Sea $a \in \mathcal{A}^+$ la palabra recibida al transmitir una palabra código $c \in \mathcal{C}$ en cuya transmisión se han producido como mucho $t$ errores, esto es, $\mathrm{d}(c, a) \leq \mathrm{t} < \mathrm{d}(\mathcal{C})$. Entonces, por definición de $\mathrm{d}(\mathcal{C})$, deducimos que $a \notin C$, luego $\mathcal{C}$ detecta $t$ errores.

Queda ver que $\mathcal{C}$ no detecta $t+1$ errores. Por definición de $\mathrm{d}(\mathcal{C})$, existen $c_1, c_2 \in \mathcal{C}$ tales que $\mathrm{d}(c1, c2) = \mathrm{d}(\mathcal{C})$. Entonces, si al transmitir $c_1$ se producen $\mathrm{d}(\mathcal{C})$ errores, se podría recibir $c_2$, que es también una palabra código, luego $\mathcal{C}$ no detecta $\mathrm{d}(\mathcal{C})$ errores y obtenemos que $\mathcal{C}$ es $(\mathrm{d}(\mathcal{C}) - 1)$-detector. □

**Notación.** *Dado $q \in \mathbb{Q}$, representaremos por $[q]$ a la parte entera de $q$.*

**Proposición 1.3.** *Sea $\mathcal{C} \subseteq \mathcal{A}$ un código y sea $t = \left[\frac{\mathrm{d}(\mathcal{C})-1}{2}\right]$. Entonces $\mathcal{C}$ es un código $t$-corrector.*

*Demostración.* Sea $a \in \mathcal{A}^+$ la palabra recibida al transmitir una palabra código $c \in \mathcal{C}$ en cuya transmisión se han producido como mucho t errores, esto es, $\mathrm{d}(a, c) \leq t$.

Veamos que $c$ es la palabra código a menor distancia de $a$. En caso contrario, existiría $c' \in \mathcal{C}$ tal que $\mathrm{d}(a, c') \leq \mathrm{d}(a, c)$.

Por la desigualdad triangular tendríamos:

$$\mathrm{d}(c, c') \leq \mathrm{d}(c, a) + \mathrm{d}(a, c') \leq 2\,\mathrm{d}(c, a) \leq 2t = 2\left[\frac{\mathrm{d}(\mathcal{C}) - 1}{2}\right] \leq \mathrm{d}(\mathcal{C}) - 1,$$

esto es, $\mathrm{d}(c, c') < \mathrm{d}(\mathcal{C})$, lo cual es una contradicción, de modo que $\mathcal{C}$ corrige $\left[\frac{\mathrm{d}(\mathcal{C})-1}{2}\right]$ errores.

Por otro lado, veamos que $\mathcal{C}$ no corrige $t+1$ errores. Sean $c_1, c_2 \in \mathcal{C}$ tales que $\mathrm{d}(c_1, c_2) = \mathrm{d}(\mathcal{C})$. Modificando $t+1$ posiciones de $c_1$ por el correspondiente valor de $c_2$ de entre las $\mathrm{d}(\mathcal{C})$ posiciones en que $c_1$ y $c_2$ difieren, obtenemos $a \in \mathcal{A}^+$ tal que:

$$\begin{aligned}\mathrm{d}(a, c_2) &= \mathrm{d}(c_1, c_2) - (t+1) = \mathrm{d}(\mathcal{C}) - \left(\left[\frac{\mathrm{d}(\mathcal{C}) - 1}{2}\right] + 1\right) \leq \\ &\leq 2\left[\frac{\mathrm{d}(\mathcal{C}) - 1}{2}\right] + 2 - \left(\left[\frac{\mathrm{d}(\mathcal{C}) - 1}{2}\right] + 1\right) = \left[\frac{\mathrm{d}(\mathcal{C}) - 1}{2}\right] + 1 = t + 1 = \mathrm{d}(a, c_1),\end{aligned}$$

donde hemos usado que si $b \in \mathbb{N}$ entonces $\frac{b}{2} \leq \left[\frac{b+1}{2}\right]$.

Hemos obtenido que $\mathrm{d}(a, c_2) \leq \mathrm{d}(a, c_1)$, luego $\mathcal{C}$ no corrige $t+1$ errores pero sí $t$, de modo que es $t$-corrector. □

## 1.4. Decodificación

La tarea de decodificación es la de corrección de errores. Sea $r \in \mathcal{A}^+$ la palabra recibida al transmitir cierta palabra código. Decodificar $r$ consistirá en corregir los errores que se han cometido en la transmisión para encontrar la palabra código enviada originalmente. Para ello, se considerará como correcta la palabra código que esté a menor distancia de $r$. En este sentido, el esquema de decodificación que seguiremos es el de decodificación por vecino más cercano.

Como hemos visto en la Proposición 1.3, está garantizado que dicho sistema de decodificación será correcto para una cierta cantidad de errores cometidos en la transmisión. En cualquier caso, suponiendo que los errores ocurren de manera equiprobable en todas las posiciones de las palabras y que es más probable que se cometan $t \geq 0$ errores a que se comentan $t+1$ errores, entonces este esquema de decodificación por vecino más cercano con la distancia de Hamming será también una decodificación por mayor probabilidad.

Hasta ahora, hemos considerado como alfabeto un conjunto $\mathcal{A}$ finito arbitrario. Dotando de estructura a dicho alfabeto, podremos construir códigos con propiedades interesantes que simplifiquen los procesos de codificación y decodificación.

## 1.5. Códigos sobre cuerpos finitos

Sea $\mathcal{A}$ un alfabeto y $\mathcal{C}$ un $(n, M)$-código sobre $\mathcal{A}$. Si tomamos $\mathcal{A} = \mathbb{F}_q$ el cuerpo finito de $q$ elementos, obtenemos una estructura que nos permite operar con los símbolos del alfabeto y que induce una manera natural de operar con las palabras construidas sobre dicho alfabeto. El producto cartesiano $\mathbb{F}_q^n$, que identificamos con las palabras de longitud $n$, adquiere de forma canónica una estructura de espacio vectorial.

Para un alfabeto con dicha estructura podemos introducir nuevos conceptos que nos serán prácticos para deducir propiedades sobre los códigos.

**Definición 1.11** (Peso de una palabra)**.** Dado $x \in \mathbb{F}_q^n$, definimos el *peso* de $x$ y denotamos por $\mathrm{w}(x)$ al número de coordenadas no nulas de $x$, esto es,

$$\mathrm{w}(x) = \#\{i \in \{1, ..., n\} : x_i \neq 0\}.$$

Es claro entonces que $\mathrm{w}(x) = 0$ si, y solo si, $x = 0$.

**Ejemplo 1.4.** Dado $x = 102304 \in \mathbb{F}_5^6$, tenemos que $\mathrm{w}(x) = 4$.

Definido dicho operador sobre las palabras de longitud $n$ se induce de manera natural un operador sobre los $(n, M)$-códigos construidos sobre $\mathbb{F}_q$.

**Definición 1.12** (Peso de un código)**.** Dado $\mathcal{C}$ un $(n, M)$-código sobre $\mathbb{F}_q$, definimos el *peso* de $\mathcal{C}$ y denotamos por $\mathrm{w}(\mathcal{C})$ al menor de los pesos de las palabras no nulas que contiene, esto es,

$$\mathrm{w}(\mathcal{C}) = \min\{\mathrm{w}(x) : x \in \mathcal{C} \backslash \{0\}\}.$$

Este peso que hemos definido para palabras y códigos es el llamado peso de Hamming y tiene una relación evidente con la distancia de Hamming antes definida.

**Proposición 1.4.** *Sean $x, y \in \mathbb{F}_q^n$. Entonces,*

$$\mathrm{d}(x, y) = \mathrm{w}(x - y).$$

Dicho peso de Hamming nos proporciona otro enfoque para cuantificar los errores cometidos en la transmisión de una palabra código.

**Proposición 1.5.** *Sea $\mathcal{C}$ un $(n, M)$-código sobre $\mathbb{F}_q$ y $c \in \mathcal{C}$ una palabra código. Sea $c' \in \mathbb{F}_q^n$ la palabra recibida al transmitir $c$. Entonces, se han cometido $t$ errores al transmitir $c$ si, y solo si, $c'$ es de la forma $c' = c + e$ con $\mathrm{w}(e) = t$. Diremos que $e$ es el patrón de error y que se ha cometido un error de peso $t$.*

Estamos ya en condiciones de presentar una de las familias de códigos más importantes: los códigos lineales.

# 2 Códigos lineales

Supongamos que disponemos de un mensaje construido sobre el alfabeto $\mathbb{F}_q$ que queremos transmitir. La idea es separar dicho mensaje en bloques de $k$ símbolos y codificar cada bloque transformándolo en otro bloque de $n \geq k$ símbolos de $\mathbb{F}_q$, que será el que finalmente se transmita. De este modo, el código resultante será un código de bloque con palabras código de longitud $n$ y la aplicación de codificación será de la forma: $C : \mathbb{F}_q^k \to \mathbb{F}_q^n$. Notemos que este esquema sigue siendo consistente con el planteamiento inicial sin más que considerar como alfabeto de partida $\mathcal{A}_1 = \mathbb{F}_q^k$.

La clave de esta construcción es que el código $\mathcal{C}$ pasa a ser un subconjunto del espacio vectorial $\mathbb{F}_q^n$, de modo que podemos dotarlo de una cierta estructura que simplifique su representación y las tareas de codificación y decodificación.

Seguiremos el planteamiento de ([Rom92], Capítulo 5).

**Definición 2.1** (Código lineal)**.** Un *código lineal q-ario de longitud n y rango k* es un subespacio vectorial $\mathcal{L} \subseteq \mathbb{F}_q^n$ de dimensión $k$. Diremos también que $\mathcal{L}$ es un $[n,k]_q$-código lineal.

Para el caso binario simplificaremos la notación diciendo que $\mathcal{L}$ es un $[n,k]$-código lineal.

**Ejemplo 2.1.** El código $\mathcal{C} = \{00, 11\} \subseteq \mathbb{F}_2^2$ es un $[2,1]$-código lineal.

**Ejemplo 2.2.** Consideremos la aplicación lineal $f : \mathbb{F}_3^2 \to \mathbb{F}_3^4$ dada por

$$f(a,b) = (a, b, a+b, a-b).$$

Entonces tenemos que su imagen $\mathrm{Im}(f) = \{(0,0,0,0), (0,1,1,2), (0,2,2,1), (1,0,1,1), (1,1,2,0), (1,2,0,2), (2,0,2,2), (2,1,0,1), (2,2,1,0)\} \subseteq \mathbb{F}_3^4$ es un subespacio vectorial de $\mathbb{F}_3^4$ de dimensión 3, luego $\mathrm{Im}(f)$ es un $[4,3]_3$-código lineal.

*Observación* 2.1. El número de palabras código en un $[n,k]_q$-código lineal es $M = q^k$, ya que todo espacio vectorial de dimensión $k$ sobre $\mathbb{F}_q$ es isomorfo a $\mathbb{F}_q^k$. Sin embargo, es claro que no todo subconjunto de tamaño $M = q^k$ es un subespacio vectorial de dimensión $k$ de $\mathbb{F}_q$. En el caso binario, dado que los escalares por los que se puede multiplicar a las palabras de $\mathbb{F}_2^n$ son únicamente el 0 y el 1, una condición necesaria y suficiente para que un subconjunto $\mathcal{C} \subseteq \mathbb{F}_2^n$ de tamaño $|\mathcal{C}| = 2^k$ sea un $[n,k]$-código lineal es que la suma de cualesquiera dos palabras de $\mathcal{C}$ quede dentro de $\mathcal{C}$.

**Ejemplo 2.3.** El código $\mathcal{C} = \{0000, 0010, 1101, 1111\} \subseteq \mathbb{F}_2^4$ es un $[4,2]$-código lineal pues la suma de cualesquiera dos palabras de $\mathcal{C}$ queda dentro de $\mathcal{C}$.

## 2.1. Matriz generadora

Dado $\mathcal{C} \subseteq \mathbb{F}_q^n$ un $[n,k]_q$-código lineal, por tratarse de un subespacio vectorial de dimensión $k$ de $\mathbb{F}_q^n$, podemos encontrar una base $B = \{c_1, ..., c_k\} \subseteq \mathbb{F}_q^n$ de $\mathcal{C}$ que permite expresar toda palabra código de manera única como combinación lineal de los elementos de $B$. En este sentido, dada $w \in \mathcal{C}$, existen unos únicos $\lambda_1, ..., \lambda_k \in \mathbb{F}_q$ tales que

$$w = \lambda_1 c_1 + \cdots \lambda_k c_k.$$

Si consideramos dichos elementos de $\mathcal{C}$ escritos con las coordenadas respecto de la base canónica de $\mathbb{F}_q^n$ como $w = (w_1, ..., w_n)$, $c_i = (c_{i1}, ..., c_{in})$ con $i \in \{1, ..., k\}$, podemos expresar la anterior combinación lineal de forma matricial:

$$(w_1, ..., w_n) = (\lambda_1, ..., \lambda_k) \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{k1} & c_{k2} & \dots & c_{kn} \end{pmatrix}.$$

Variando los parámetros $\lambda_1, ..., \lambda_k \in \mathbb{F}_q$, obtenemos las $q^k$ palabras de $\mathcal{C}$.

A la matriz $(c_{ij})$ la llamaremos matriz generadora del código lineal y será fundamental para el trabajo con el código. En particular, veremos que es clave en la tarea de codificación.

**Definición 2.2** (Matriz generadora)**.** Dado un $[n,k]_q$-código lineal $\mathcal{C}$, una *matriz generadora* de $\mathcal{C}$ es una matriz $G \in M_{k\times n}(\mathbb{F}_q)$ cuyas filas son las coordenadas de los elementos de una base de $\mathcal{C}$.

*Observación* 2.2. Si $G \in M_{k\times n}(\mathbb{F}_q)$ es una matriz generadora de un $[n,k]_q$-código lineal $\mathcal{C}$, entonces $\mathcal{C} = \{aG \mid a \in \mathbb{F}_q^k\}$.

**Ejemplo 2.4.** El software matemático SageMath nos será de utilidad para ilustrar algunos ejemplos en este y los siguientes capítulos. Como acabamos de ver, los códigos lineales vienen determinados por una matriz generadora suya.

En este sentido, podemos definir un código lineal a partir de su matriz generadora.

```
sage> F = GF(2)
sage> M = matrix(F, [[1, 1, 0, 1, 0, 0],\
                     [1, 0, 0, 0, 1, 1],\
                     [0, 1, 1, 0, 1, 0],\
                     [0, 0, 1, 0, 1, 1]])
sage> C = codes.LinearCode(M)
sage> C
[6, 4] linear code over GF(2)
```

Una vez construido el código podemos comprobar que la matriz generadora es la introducida.

```
sage> C.generator_matrix()
[1 1 0 1 0 0]
[1 0 0 0 1 1]
[0 1 1 0 1 0]
[0 0 1 0 1 1]
sage>C.generator_matrix() == M
True
```

También podemos comprobar que, efectivamente, las filas de la matriz constituyen una base del código.

```
sage> C.basis()
[
(1, 1, 0, 1, 0, 0),
(1, 0, 0, 0, 1, 1),
(0, 1, 1, 0, 1, 0),
(0, 0, 1, 0, 1, 1)
]
```

La matriz generadora es además el fundamento del esquema de codificación. Sea $\mathcal{C}$ un $[n,k]_q$-código lineal con matriz generadora $G$. Supongamos que queremos codificar $a \in \mathbb{F}_q^k$. Su palabra código asociada será $w \in \mathbb{F}_q^n$ dada por:

$$w = aG.$$

Efectivamente, por construcción de $G$ tenemos que $w \in \mathcal{C}$.

De este modo, la codificación se basa en multiplicar el bloque de longitud $k$ que queremos codificar por la matriz generadora para obtener un bloque de longitud $n$ que será su palabra código asociada. En este sentido, no es necesario almacenar explícitamente cuál es la palabra código asociada a cada bloque de longitud $k$, sino que basta con almacenar la matriz generadora para calcular dicha asociación sin más que una multiplicación. Esto puede suponer un importante ahorro de espacio para valores grandes de $n$ y $k$.

**Ejemplo 2.5.** Veamos un ejemplo de codificación con el código lineal recién definido en el Ejemplo 2.4.

```
sage> x = vector(GF(2), [0, 1, 0, 1])
sage> x
(0, 1, 0, 1)
sage> x*M
(1, 0, 1, 0, 0, 0)
sage> C.encode(x)
(1, 0, 1, 0, 0, 0)
```

Alternativamente, podemos definir un codificador a partir del código $\mathcal{C}$.

```
sage> E = codes.encoders.LinearCodeGeneratorMatrixEncoder(C)
sage> E
Generator matrix-based encoder for [6, 4] linear code over GF(2)
sage> E.generator_matrix()
[1 1 0 1 0 0]
[1 0 0 0 1 1]
[0 1 1 0 1 0]
[0 0 1 0 1 1]
sage> E.encode(x)
(1, 0, 1, 0, 0, 0)
```

## 2.2. Códigos lineales equivalentes

Podemos observar que un $[n,k]_q$-código lineal $\mathcal{C}$ no tiene una única matriz generadora pues, salvo casos triviales, un subespacio vectorial tiene más de una base. Dada una base podemos obtener todas las demás operando los elementos que la constituyen, lo que se traduce en operaciones elementales entre filas en una matriz generadora dada. Dadas dos matrices generadoras distintas de $\mathcal{C}$, las reglas de asignación resultantes serán también distintas. Sin embargo, recordamos que las propiedades de los códigos en las que estamos interesados no dependen de la regla de asignación, sino de su imagen, que en este caso sería el mismo código $\mathcal{C}$.

Por otro lado, podemos disponer de dos códigos lineales distintos que esencialmente tengan las mismas propiedades de detección y corrección de errores. Es el caso que se da, por ejemplo, entre un código lineal y el resultante de aplicar una permutación fija de las posiciones en todas sus palabras código. Esta transformación se traduce en una permutación de las columnas de una matriz generadora del código.

**Definición 2.3** (Códigos equivalentes)**.** Diremos que dos $[n,k]_q$-códigos lineales $\mathcal{C}_1$ y $\mathcal{C}_2$ son *equivalentes* si $\mathcal{C}_2$ se obtiene a partir de $\mathcal{C}_1$ aplicando una permutación fija a todas sus palabras código.

Es claro que dicha relación entre los códigos es, en efecto, una relación de equivalencia.

Por las observaciones antes hechas sobre cómo se traducen en las matrices generadoras los cambios de base y la permutación de posiciones, deducimos el siguiente resultado.

**Proposición 2.1.** *Los códigos generados por dos matrices generadoras $G_1, G_2 \in M_{k\times n}(\mathbb{F}_q)$ son equivalentes si, y solo si, $G_2$ se obtiene a partir de $G_1$ mediante operaciones elementales en las filas y permutaciones de las columnas.*

**Ejemplo 2.6.** Apliquemos una serie de operaciones a la matriz generadora de un código para obtener otro equivalente.

```
sage> M1 = matrix(GF(7), [[3, 1, 0, 1, 0, 6],\
                          [1, 0, 2, 3, 0, 4],\
                          [0, 1, 4, 0, 1, 0],\
                          [0, 3, 4, 0, 5, 3]])
sage> C1 = codes.LinearCode(M1)
sage> C1
[6, 4] linear code over GF(7)
```

Definimos la matriz que intercambia columnas 2 y 4.

```
sage> I1 = elementary_matrix(GF(7),6,col1=2,col2=4)
sage> I1
[1 0 0 0 0 0]
[0 1 0 0 0 0]
[0 0 0 0 1 0]
[0 0 0 1 0 0]
[0 0 1 0 0 0]
[0 0 0 0 0 1]
```

Definimos la matriz que realiza la operación Fila1 = Fila1 + 3 × Fila3.

```
sage> I2 = elementary_matrix(GF(7),4,row1=1,row2=3,scale=3)
sage> I2
[1 0 0 0]
[0 1 0 3]
[0 0 1 0]
[0 0 0 1]
```

Aplicamos operaciones a la matriz generadora y comprobamos que obtenemos un código equivalente.

```
sage> M2 = I2*M1*I1
[3 1 0 1 0 6]
[1 2 1 3 0 6]
[0 1 1 0 4 0]
[0 3 5 0 4 3]
sage> C2 = codes.LinearCode(M2)
[6, 4] linear code over GF(7)
sage> C2.is_permutation_equivalent(C1)
True
```

En este sentido, aplicando operaciones de ese tipo a las matrices generadoras obtenemos códigos que tienen esencialmente las mismas propiedades. Esto nos lleva a buscar para cada código lineal la matriz más simple que sea matriz generadora de un código lineal equivalente al original.

## 2.3. Códigos sistemáticos

Supongamos que el esquema de codificación de un $[n,k]_q$-código lineal $\mathcal{C}$ es tal que al codificar cualquier palabra $a \in \mathbb{F}_q^k$, la palabra código resultante $w_a \in \mathcal{C} \subseteq \mathbb{F}_q^n$ es de la forma $w_a = aa'$ con $a' \in \mathbb{F}_q^{n-k}$. En ese caso, los primeros $k$ bits de todas las palabras código corresponden a la palabra de $\mathbb{F}_q^k$ que codifican. Esto simplifica considerablemente las tareas de codificación y decodificación. La matriz generadora de $\mathcal{C}$ que proporciona tal esquema de codificación será de la forma:

$$G = \left(\begin{array}{cccc|cccc} 1 & 0 & \dots & 0 & a_{11} & a_{12} & \dots & a_{1n-k} \\ 0 & 1 & \dots & 0 & a_{21} & a_{22} & \dots & a_{2n-k} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & a_{k1} & a_{k2} & \dots & a_{kn-k} \end{array}\right) = (\mathrm{Id}_k \mid A)\,,$$

siendo $\mathrm{Id}_k$ la matriz identidad de orden $k$ y con $A \in M_{k\times(n-k)}(\mathbb{F}_q)$.

**Definición 2.4** (Código sistemático)**.** Un $[n,k]_q$-código lineal es *sistemático* si tiene una matriz generadora de la forma $G = (\mathrm{Id}_k \mid A)$ con $A \in M_{k\times(n-k)}(\mathbb{F}_q)$. Si la matriz $G$ tiene dicha expresión, diremos que está en *forma estándar*.

**Proposición 2.2.** *Todo código lineal es equivalente a un código lineal sistemático.*

*Demostración.* Sea $\mathcal{C}$ es un $[n,k]_q$-código lineal y sea $G$ una matriz generadora suya. Entonces $G$ tiene rango $k$ y aplicando operaciones elementales a sus filas se puede obtener una matriz en forma normal de Hermite por filas. Permutando las columnas podemos obtener una matriz de la forma $(\mathrm{Id}_k \mid A)$, siendo el código que genera un código lineal sistemático equivalente a $\mathcal{C}$. □

**Ejemplo 2.7.** Dado un código lineal, SageMath nos permite obtener una matriz generadora suya que está en forma estándar salvo por la reordenación de las columnas. Esta se obtiene aplicando operaciones por filas a la matriz generadora proporcionada.

```
sage> M = matrix(GF(2), [[1, 1, 0, 1, 0, 0],\
                         [1, 1, 1, 0, 0, 0],\
                         [0, 0, 0, 0, 1, 0],\
                         [0, 0, 1, 0, 0, 1]])
sage> C = codes.LinearCode(M)
sage> C.systematic_generator_matrix()
[1 1 0 0 0 1]
[0 0 1 0 0 1]
[0 0 0 1 0 1]
[0 0 0 0 1 0]
```

También podemos obtener el código lineal sistemático equivalente al código dado junto con la reordenación de columnas requerida para su obtención.

```
sage> Cs = C.standard_form()
sage> Cs
([6, 4] linear code over GF(2), [1, 3, 4, 5, 2, 6])
sage> Cs[0].systematic_generator_matrix()
[1 0 0 0 1 1]
[0 1 0 0 0 1]
[0 0 1 0 0 1]
[0 0 0 1 0 0]
sage> Cs[0].is_permutation_equivalent(C)
True
```

## 2.4. Código dual y matriz de paridad

Estudiemos ahora el conocido como código dual de un código lineal, cuyas matrices generadoras nos darán interesante información del código de partida. En particular, nos permitirán obtener resultados relativos a la distancia mínima del código.

**Definición 2.5** (Código dual)**.** Dado un $[n,k]_q$-código lineal $\mathcal{C}$ definimos su código dual $\mathcal{C}^\perp$ por

$$\mathcal{C}^\perp = \{y \in \mathbb{F}_q^n \mid \langle x, y\rangle = 0 \ \text{ para todo } x \in \mathcal{C}\},$$

donde $\langle x, y\rangle = \sum_{i=1}^{n} x_i y_i$ denota al producto escalar en $\mathbb{F}_q^n$.

*Observación* 2.3. $(\mathcal{C}^\perp)^\perp = \mathcal{C}$.

El código dual $\mathcal{C}^\perp$ no es más que el subespacio ortogonal a $\mathcal{C}$, que es un subespacio vectorial de dimensión $n-k$. Por lo tanto, $\mathcal{C}^\perp$ es un $[n, n-k]_q$-código lineal.

**Definición 2.6** (Matriz de paridad)**.** Llamaremos *matriz de paridad* de $\mathcal{C}$ a una matriz generadora de $\mathcal{C}^\perp$.

Del mismo modo que las matrices generadoras, las matrices de paridad no serán únicas salvo casos triviales.

**Ejemplo 2.8.** Si definimos un $[6,4]$-código lineal, su dual será un $[6,2]$-código lineal y podremos observar la equivalencia entre las respectivas matrices de paridad y generadora.

```
sage> M = matrix(GF(2), [[1, 1, 0, 1, 0, 0],\
                         [1, 0, 0, 0, 1, 1],\
                         [0, 1, 1, 0, 1, 0],\
                         [0, 0, 1, 0, 1, 1]])
sage> C = codes.LinearCode(M)
sage> C
[6, 4] linear code over GF(2)
sage> Cd = C.dual_code()
sage> Cd
[6, 2] linear code over GF(2)
sage> C.parity_check_matrix()
[1 0 1 1 1 0]
[0 1 0 1 1 1]
sage> Cd.generator_matrix()
[1 0 1 1 1 0]
[0 1 0 1 1 1]
```

Al igual que la matriz generadora, la matriz de paridad determina de forma unívoca al código.

**Proposición 2.3.** *Sea $H$ una matriz de paridad para el $[n,k]_q$-código lineal $\mathcal{C}$. Entonces,*

$$\mathcal{C} = \{x \in \mathbb{F}_q^n \mid xH^T = 0\} = \{x \in \mathbb{F}_q^n \mid Hx^T = 0\}.$$

*Demostración.* Por ser $H$ una matriz generadora de $\mathcal{C}^\perp$, tenemos que $\mathcal{C}^\perp = \{yH \mid y \in \mathbb{F}_q^n\}$. Por otro lado,

$$\mathcal{C} = (\mathcal{C}^\perp)^\perp = \{x \in \mathbb{F}_q^n \mid \langle x, z\rangle = 0 \ \textit{ para todo } z \in \mathcal{C}^\perp\}.$$

Por lo tanto,

$$\mathcal{C} = \{x \in \mathbb{F}_q^n \mid \langle x, yH\rangle = 0 \ \textit{ para todo } y \in \mathbb{F}_q^n\} = \{x \in \mathbb{F}_q^n \mid xH^T = 0\} = \{x \in \mathbb{F}_q^n \mid Hx^T = 0\}.$$

□

En los casos en los que disponemos de la matriz generadora expresada en forma estándar podemos obtener directamente la matriz de paridad. Si $G = (\mathrm{Id}_k \mid A)$ es una matriz generadora de $\mathcal{C}$, entonces $H = \left(-A^T \,\middle|\, \mathrm{Id}_{n-k}\right)$ es una matriz de paridad ya que, multiplicando por

cajas,

$$GH^T = (\mathrm{Id}_k \mid A)\begin{pmatrix} -A \\ \hline \mathrm{Id}_{n-k} \end{pmatrix} = -A + A = 0.$$

La matriz de paridad nos proporciona un método para determinar la distancia mínima de un código lineal. Para obtenerlo haremos uso de un resultado que nos da la equivalencia entre la distancia mínima de un código lineal y su peso.

**Proposición 2.4.** *Sea $\mathcal{C}$ un código lineal. Entonces,*

$$\mathrm{d}(\mathcal{C}) = \mathrm{w}(\mathcal{C}).$$

*Demostración.* Tenemos que

$$\begin{aligned}\mathrm{d}(\mathcal{C}) &= \min\{\mathrm{d}(x,y) :\ x,y \in \mathcal{C},\ \ x \neq y\} = \min\{\mathrm{w}(x-y) :\ x,y \in \mathcal{C},\ \ x \neq y\} = \\ &= \min\{\mathrm{w}(x) :\ x \in \mathcal{C}\} = \mathrm{w}(\mathcal{C}),\end{aligned}$$

donde hemos usado que $\mathcal{C}$ es un subespacio vectorial. □

**Teorema 2.1.** *Sea $H$ una matriz de paridad de un código lineal $\mathcal{C}$. Entonces,*

$$\mathrm{d}(\mathcal{C}) = \min\{s \in \mathbb{N} |\ H \text{ tiene } s \text{ columnas linealmente dependientes}\}.$$

*Demostración.* Sea $c \in \mathcal{C}$ una palabra código del $[n,k]_q$-código lineal $\mathcal{C}$ con $\mathrm{w}(c) = \mathrm{w}(\mathcal{C})$. Por la Proposición 2.3 tenemos que $Hc^T = 0$, lo que significa que hay $\mathrm{w}(c)$ columnas de $H$ linealmente dependientes. Por lo tanto,

$$\mathrm{d}(\mathcal{C}) = \mathrm{w}(\mathcal{C}) = \mathrm{w}(c) \geq \min\{s \in \mathbb{N} |\ H \text{ tiene } s \text{ columnas linealmente dependientes}\}.$$

Por otro lado, si hay $\mathrm{d}(\mathcal{C}) - 1$ columnas de $H$ linealmente dependientes, existe $x \in \mathbb{F}_q^n$ con $\mathrm{w}(x) = \mathrm{d}(\mathcal{C}) - 1$ tal que $Hx^T = 0$. Pero por la Proposición 2.3, obtendríamos que $x \in \mathcal{C}$ y además $\mathrm{w}(x) = \mathrm{d}(\mathcal{C}) - 1 = \mathrm{w}(\mathcal{C}) - 1$, luego $\mathrm{w}(x) < \mathrm{w}(\mathcal{C})$ y llegamos a una contradicción. □

Este resultado nos permite obtener una sencilla cota de la distancia mínima de los códigos lineales, conocida como cota de Singleton.

**Proposición 2.5.** *Dado un $[n,k]_q$-código lineal $\mathcal{C}$ tenemos que*

$$\mathrm{d}(\mathcal{C}) \leq n - k + 1.$$

*Demostración.* Tomemos $H$ una matriz de paridad de $\mathcal{C}$. Sabemos que su dimensión es $(n-k) \times n$, luego tendrá como mucho $n-k$ columnas linealmente independientes. Por el teorema que acabamos de demostrar, si tomamos $\mathrm{d}(\mathcal{C}) - 1$ columnas de $H$, estas serán linealmente independientes, luego $\mathrm{d}(\mathcal{C}) - 1 \leq n - k$. □

Otro de los puntos claves de la matriz de paridad es que nos permite introducir el concepto de síndrome, que resulta clave para las tareas de detección y corrección de errores en los códigos lineales.

## 2.5. Decodificación por síndrome

Como vimos al presentar el concepto de decodificación para un código genérico, esta se basa en la búsqueda de la palabra código más cercana a cierta palabra dada $x$. En este sentido, un posible enfoque para decodificar sería el de medir la distancia de $x$ a todas las palabras código y quedarnos con la que nos de la distancia mínima (o con una de ellas en caso de no haber unicidad). Es claro que se trata de un método efectivo, pero que puede resultar poco eficiente, especialmente cuando se trabaja con códigos de tamaño considerable.

De este modo, para los códigos lineales se puede aplicar un procedimiento de decodificación más eficiente que se construye sobre el concepto de síndrome de una palabra y sobre el espacio cociente que induce un código lineal como subespacio vectorial. El síndrome también será una interesante herramienta para la detección de errores.

**Definición 2.7** (Síndrome de una palabra)**.** Sea $H$ una matriz de paridad de un $[n,k]_q$-código lineal $\mathcal{C}$ y $x \in \mathbb{F}_q^n$. El *síndrome* de $x$ se define como $xH^T$.

*Observación* 2.4. Por la Proposición 2.3, es claro que una palabra está en el código si, y solo si, su síndrome es 0.

**Ejemplo 2.9.** Calculemos el síndrome de un par de palabras para un $[8,3]$-código lineal. Definimos primero el código.

```
sage> M = matrix(GF(2), [[1, 1, 0, 0, 0, 0, 0, 1],\
                         [0, 0, 1, 1, 0, 0, 1, 0],\
                         [0, 0, 0, 0, 1, 1, 0, 1]])
sage> C = codes.LinearCode(M)
sage> C
[8, 3] linear code over GF(2)
```

Obtenemos la matriz de paridad del código.

```
sage> H = C.parity_check_matrix()
sage> H
[1 0 0 0 0 1 0 1]
[0 1 0 0 0 1 0 1]
[0 0 1 0 0 0 1 0]
[0 0 0 1 0 0 1 0]
[0 0 0 0 1 1 0 0]
```

Tomamos un vector que no está en el código y calculamos su síndrome.

```
sage> r = vector(GF(2), (1,0,0,0,1,0,1,0))
sage> r
(1, 0, 0, 0, 1, 0, 1, 0)
sage> r in C
False
sage> H*r
(1, 0, 1, 1, 1)
sage> C.syndrome(r)
(1, 0, 1, 1, 1)
```

Tomamos ahora un vector que sí está en el código y calculamos igualmente su síndrome.

```
sage> r2 = vector(GF(2), (1,1,1,1,1,1,1,0))
sage> r2
(1,1,1,1,1,1,1,0)
sage> r2 in C
True
sage> H*r2
(0, 0, 0, 0, 0)
sage> C.syndrome(r2)
(0, 0, 0, 0, 0)
```

**Proposición 2.6.** *El síndrome de una palabra no depende de la palabra código de la que proceda, sino únicamente del patrón de error que ha ocurrido.*

*Demostración.* Sea $x \in \mathbb{F}_q^n$ una palabra recibida con un patrón de error $e \in \mathbb{F}_q^n$, esto es, $x = c + e$ siendo $c \in \mathcal{C}$ la palabra código transmitida originalmente. Entonces,

$$xH^T = (c+e)H^T = cH^T + eH^T = 0 + eH^T = eH^T \qquad \square$$

El síndrome nos proporciona un sencillo método para detectar muchos patrones de error, consistente en comprobar si el síndrome de la palabra es distinto de 0. De hecho, aunque la capacidad detectora de un código queda determinada por su distancia mínima, pudiéndose detectar todos los patrones de error de peso menor que dicha distancia, hay muchos otros patrones de error de peso mayor que también se podrán detectar.

**Proposición 2.7.** *Un $[n,k]_q$-código lineal detecta $q^n - q^k$ patrones de error.*

*Demostración.* Sea $H$ una matriz de paridad de un $[n,k]_q$-código lineal $\mathcal{C}$. Sean $c \in \mathcal{C}$ una palabra código, $e \in \mathbb{F}_q^n$ un patrón de error y $x = c + e$ la palabra recibida al transmitir $c$. En la demostración de la proposición anterior hemos visto que $xH^T = eH^T$, luego $xH^T = 0$ si, y solo si, $e \in \mathcal{C}$.

Esto significa que, de los $q^n - 1$ posibles patrones de error, los únicos indetectables son aquellos que coinciden con una de las $q^k - 1$ palabras código no nulas. $\square$

En lo que se refiere a la corrección de errores, un método efectivo es el de la decodificación por síndrome, que aprovecha la estructura de espacio cociente de $\mathbb{F}_q^n$ sobre un $[n,k]_q$-código $\mathcal{C}$. Dado $x \in \mathbb{F}_q^n$ notamos por $[x] \in \mathbb{F}_q^n/\mathcal{C}$ a la clase de $x$. Recordamos que $[x] = [y]$ si, y solo si, $x - y \in \mathcal{C}$.

**Proposición 2.8.** *Sea $\mathcal{C}$ un $[n,k]_q$-código lineal, $H$ una matriz de paridad y $x, y \in \mathbb{F}_q^n$. Entonces,*

$$[x] = [y] \;\; \textit{si, y solo si,} \;\; xH^T = yH^T.$$

*Demostración.* La igualdad

$$xH^T = yH^T$$

es equivalente a

$$(x-y)H^T = 0.$$

Por la Proposición 2.3, esto significa que

$$x - y \in \mathcal{C},$$

esto es,

$$[x] = [y]. \qquad \square$$

De este modo, los elementos de una misma clase de equivalencia son aquellos que tienen el mismo síndrome. En particular, por la Proposición 2.6, una palabra y su patrón de error están en la misma clase de equivalencia. En este sentido, la decodificación por síndrome funciona de la siguiente manera: dado $x \in \mathbb{F}_q^n$ consideraremos que su patrón de error asociado $e$ es uno de los elementos con menor peso en la clase de $x$, esto es, de entre los elementos que tienen el mismo síndrome que $x$. De este modo, decodificaremos $x$ como $x - e \in \mathcal{C}$. Por tomar $e$ de entre los de menor peso, la palabra $x - e$ es una de las palabras código a menor distancia de $x$ y, en consecuencia, seguimos el esquema de decodificación por vecino más cercano.

**Ejemplo 2.10.** A partir de un código lineal, SageMath nos permite construir un decodificador por síndrome. Partimos del código que utilizamos para calcular varios síndromes en el Ejemplo 2.9.

```
sage> D = codes.decoders.LinearCodeSyndromeDecoder(C)
sage> D
Syndrome decoder for [8, 3] linear code over GF(2) handling errors of weight up to 3
```

Este decodificador trabaja con una tabla de síndromes con la asociación entre los síndromes y los respectivos representantes de menor peso.

```
sage> D.syndrome_table()
{(0, 0, 0, 0, 0): (0, 0, 0, 0, 0, 0, 0, 0),
  (1, 0, 0, 0, 0): (1, 0, 0, 0, 0, 0, 0, 0),
  (0, 1, 0, 0, 0): (0, 1, 0, 0, 0, 0, 0, 0),
  (0, 0, 1, 0, 0): (0, 0, 1, 0, 0, 0, 0, 0),
  (0, 0, 0, 1, 0): (0, 0, 0, 1, 0, 0, 0, 0),
  (0, 0, 0, 0, 1): (0, 0, 0, 0, 1, 0, 0, 0),
  (1, 1, 0, 0, 1): (0, 0, 0, 0, 0, 1, 0, 0),
  (0, 0, 1, 1, 0): (0, 0, 0, 0, 0, 0, 1, 0),
  (1, 1, 0, 0, 0): (0, 0, 0, 0, 0, 0, 0, 1),
  (1, 0, 1, 0, 0): (1, 0, 1, 0, 0, 0, 0, 0),
  (1, 0, 0, 1, 0): (1, 0, 0, 1, 0, 0, 0, 0),
  (1, 0, 0, 0, 1): (1, 0, 0, 0, 1, 0, 0, 0),
  (0, 1, 0, 0, 1): (1, 0, 0, 0, 0, 1, 0, 0),
  (1, 0, 1, 1, 0): (1, 0, 0, 0, 0, 0, 1, 0),
  (0, 1, 1, 0, 0): (0, 1, 1, 0, 0, 0, 0, 0),
  (0, 1, 0, 1, 0): (0, 1, 0, 1, 0, 0, 0, 0),
  (0, 1, 1, 1, 0): (0, 1, 0, 0, 0, 0, 1, 0),
  (0, 0, 1, 0, 1): (0, 0, 1, 0, 1, 0, 0, 0),
  (1, 1, 1, 0, 1): (0, 0, 1, 0, 0, 1, 0, 0),
  (1, 1, 1, 0, 0): (0, 0, 1, 0, 0, 0, 0, 1),
  (0, 0, 0, 1, 1): (0, 0, 0, 1, 1, 0, 0, 0),
  (1, 1, 0, 1, 1): (0, 0, 0, 1, 0, 1, 0, 0),
  (1, 1, 0, 1, 0): (0, 0, 0, 1, 0, 0, 0, 1),
  (0, 0, 1, 1, 1): (0, 0, 0, 0, 1, 0, 1, 0),
  (1, 1, 1, 1, 1): (0, 0, 0, 0, 0, 1, 1, 0),
  (1, 1, 1, 1, 0): (0, 0, 0, 0, 0, 0, 1, 1),
```

```
  (1, 0, 1, 0, 1): (1, 0, 1, 0, 1, 0, 0, 0),
  (0, 1, 1, 0, 1): (1, 0, 1, 0, 0, 1, 0, 0),
  (1, 0, 0, 1, 1): (1, 0, 0, 1, 1, 0, 0, 0),
  (0, 1, 0, 1, 1): (1, 0, 0, 1, 0, 1, 0, 0),
  (1, 0, 1, 1, 1): (1, 0, 0, 0, 1, 0, 1, 0),
  (0, 1, 1, 1, 1): (1, 0, 0, 0, 0, 1, 1, 0)}
```

De este modo, podemos emplearlo para decodificar un vector de $\mathbb{F}_2^8$.

```
sage> r = vector(GF(2), (1,0,0,0,1,0,1,0))
sage> r
(1, 0, 0, 0, 1, 0, 1, 0)
sage> C.syndrome(r)
(1, 0, 1, 1, 1)
sage> D.decode_to_code(r)
(0, 0, 0, 0, 0, 0, 0, 0)
```

# 3 Códigos cíclicos

Los códigos cíclicos constituyen una familia de códigos lineales con una interesante estructura matemática y que al mismo tiempo son prácticos, en el sentido de que las tareas de codificación y decodificación con estos códigos se pueden implementar de forma eficiente con el uso de registros de desplazamiento. Estudiaremos una cómoda representación para trabajar con estos códigos así como su relación con la forma de trabajar con los códigos lineales que ya hemos visto. Se ha seguido la estructura de ([Rom92], Sección 7.4) y ([Hil86], Capítulo 12).

**Definición 3.1** (Código cíclico)**.** Un $[n,k]_q$-código lineal $\mathcal{C}$ es *cíclico* si cualquier desplazamiento cíclico de una palabra código es también una palabra código, esto es, si $c_0c_1\cdots c_{n-1}\in\mathcal{C}$, entonces $c_{n-1}c_0c_1\cdots c_{n-2}\in\mathcal{C}$. En dicho caso, diremos que $\mathcal{C}$ es un $[n,k]_q$-*código cíclico*.

**Ejemplo 3.1.** El $[2,1]_3$-código lineal $\mathcal{C}=\{00,\ 21,\ 12\}$ es trivialmente un código cíclico, pues desplazando cíclicamente toda palabra código obtenemos otra palabra código.

Una manera cómoda de trabajar con estos códigos es representando sus palabras código mediante polinomios. Para ello, consideremos el anillo $R_n=\frac{\mathbb{F}_q[x]}{\langle x^n-1\rangle}$, formado por los polinomios con coeficientes en el cuerpo $\mathbb{F}_q$ reducidos módulo el polinomio $x^n-1$.

Dicho anillo está formado por todos los polinomios de grado menor que $n$, con los que podemos identificar de manera natural las palabras de un $[n,k]_q$-código lineal $\mathcal{C}$ vía la aplicación:

$$\begin{aligned}\phi:\mathcal{C}&\to R_n\\ c_0c_1\cdots c_{n-1}&\mapsto c_0+c_1x+\cdots+c_{n-1}x^{n-1}.\end{aligned}$$

Es claro que $\phi$ es un isomorfismo de espacios vectoriales entre $\mathcal{C}$ y su imagen $\phi(\mathcal{C})$. En este sentido y por mayor comodidad, trataremos también a los códigos lineales como subconjuntos de $R_n$.

Dicho isomorfismo nos proporciona una práctica caracterización de los códigos cíclicos.

Recordamos que un conjunto $\mathcal{I}\subseteq R_n$ es un ideal si, y solo si,

(1). $\mathcal{I}$ es un subgrupo aditivo de $R_n$, esto es, si $a(x),\ b(x)\in\mathcal{I}$, entonces $a(x)-b(x)\in\mathcal{I}$.

(2). Si $a(x)\in\mathcal{I}$, $r(x)\in R_n$, entonces $r(x)a(x)\in\mathcal{I}$.

**Teorema 3.1.** *Un código lineal $\mathcal{C}\subseteq R_n$ es cíclico si, y solo si, $\mathcal{C}$ es un ideal de $R_n$.*

*Demostración.* Supongamos que $\mathcal{C}$ es un código cíclico. La condición (1) se sigue de que $\mathcal{C}$ es un código lineal. Para la condición (2), tomemos $a(x)\in\mathcal{C}$, $r(x)=r_0+r_1x+\cdots r_{n-1}x^{n-1}\in R_n$. Observamos que la multiplicación por $x$ se corresponde con un desplazamiento cíclico

en una posición, luego $xa(x) \in \mathcal{C}$ e inductivamente $x^i a(x) \in \mathcal{C}$ para todo $i \in \{1, 2, \ldots, n-1\}$. Por lo tanto, el producto

$$r(x)a(x) = r_0 a(x) + r_1 xa(x) + \cdots + r_{n-1} x^{n-1} a(x)$$

está también en $\mathcal{C}$ pues cada uno de los sumandos está.

Recíprocamente, si $\mathcal{C}$ satisface (1) y (2), la condición (1) junto con la (2) tomando $r(x)$ un escalar implican que $\mathcal{C}$ es un código lineal. Por otro lado, tomando $r(x) = x$ obtenemos que todo desplazamiento cíclico queda dentro de $\mathcal{C}$. □

## 3.1. Polinomio generador

Introducimos ahora un concepto clave para los códigos cíclicos: el polinomio generador. No solo determinará al código de forma unívoca, sino que también será práctico para la obtención de otros elementos del código como las matrices generadora y de paridad, así como para la tarea de codificación.

Dado $p(x) \in R_n$, el ideal generado por $p(x)$, notado por $\langle p(x) \rangle$ vendrá dado por

$$\langle p(x) \rangle = \{r(x)p(x) \mid r(x) \in R_n\},$$

donde los productos se realizan módulo $x^n - 1$. Por el teorema anterior, el ideal $\langle p(x) \rangle$ es un código cíclico. De hecho, veremos a continuación que todo código cíclico es de esa forma.

**Teorema 3.2.** *Sea $\mathcal{C} \subseteq R_n$ un ideal no nulo, esto es, un código cíclico no nulo. Entonces,*

*(1). existe un único polinomio mónico $g(x)$ en $\mathcal{C}$ con grado mínimo,*

*(2). $\mathcal{C} = \langle g(x) \rangle$,*

*(3). $g(x)$ es un factor de $x^n - 1$.*

*Demostración.* Todo ideal del anillo $\mathbb{F}_q[x]$ es principal por tratarse de un dominio euclideo, luego también todo ideal del anillo cociente $R_n$ es principal. Además, los ideales tendrán como generador a un polinomio de grado mínimo, que es único si lo tomamos mónico. Con esto quedan probadas las afirmaciones (1) y (2). La afirmación (3) se sigue de que los ideales de $R_n$ se corresponden con los ideales de $\mathbb{F}_q$ que contienen a $\langle x^n - 1 \rangle$ y de que si $\langle x^n - 1 \rangle \subseteq \langle g(x) \rangle$, entonces $g(x) | x^n - 1$. □

**Corolario 3.1.** *El conjunto de los códigos cíclicos sobre $R_n$ está en biyección con el conjunto de los divisores mónicos de $x^n - 1$. De este modo, la factorización del polinomio $x^n - 1$ nos permite describir todos los códigos cíclicos de una longitud dada $n$.*

**Definición 3.2** (Polinomio generador)**.** Sea $\mathcal{C}$ un código cíclico no nulo. Al polinomio $g(x)$ dado por el Teorema 3.2 lo llamaremos *polinomio generador* de $\mathcal{C}$.

*Observación* 3.1. Aunque el llamado *polinomio generador* de un código cíclico $\mathcal{C}$ sea único, el código $\mathcal{C}$ puede tener más polinomios que lo generen. Es el caso del siguiente ejemplo.

**Ejemplo 3.2.** El software SageMath nos será también de utilidad para trabajar con códigos cíclicos. Podemos construir un código cíclico indicando su polinomio generador y el valor de $n$. El polinomio generador se definirá indicando el cuerpo en el que tomar los coeficientes, siendo en este caso $\mathbb{F}_2$.

```
sage> n = 3
sage> F.<x> = GF(2)[]
sage> g = x+1
sage> g.divides(x^n-1)
True
sage> C = codes.CyclicCode(generator_pol=g, length=n)
sage> C
[3, 2] Cyclic Code over GF(2)
sage> C.generator_polynomial()
x + 1
```

Podemos listar sus elementos, observando que todo desplazamiento cíclico de las palabras queda dentro del código.

```
sage> list(C)
[(0, 0, 0), (1, 1, 0), (0, 1, 1), (1, 0, 1)]
```

Del mismo modo, podemos ver que el polinomio generador no es el único que genera el código, pues cualquiera de los polinomios no nulos del código genera a los otros tres, esto es, $\mathcal{C} = \langle x+1 \rangle = \langle x^2+x \rangle = \langle x^2+1 \rangle$.

```
sage> list(p*(x+1)%(x^n-1) for p in [0,1,x,x+1])
[0, x + 1, x^2 + x, x^2 + 1]
sage> list(p*(x^2+x)%(x^n-1) for p in [0,1,x,x+1])
[0, x^2 + x, x^2 + 1, x + 1]
sage> list(p*(x^2+1)%(x^n-1) for p in [0,1,x,x+1])
[0, x^2 + 1, x + 1, x^2 + x]
```

**Ejemplo 3.3.** El Corolario 3.1 nos proporciona un procedimiento para construir todos los códigos cíclicos sobre $R_4$ trabajando con polinomios sobre $\mathbb{F}_3[x]$ consistente en tomar como polinomios generadores todos los divisores mónicos de $x^4 - 1$.

```
sage> n = 4
sage> F.<x> = GF(3)[]
sage> for factor in divisors(x^n-1):
        C = codes.CyclicCode(generator_pol=factor, length=n)
        print((C, C.generator_polynomial()))
([4, 4] Cyclic Code over GF(3), 1)
([4, 3] Cyclic Code over GF(3), x + 1)
([4, 3] Cyclic Code over GF(3), x + 2)
([4, 2] Cyclic Code over GF(3), x^2 + 1)
([4, 2] Cyclic Code over GF(3), x^2 + 2)
([4, 1] Cyclic Code over GF(3), x^3 + x^2 + x + 1)
([4, 1] Cyclic Code over GF(3), x^3 + 2*x^2 + x + 2)
([4, 0] Cyclic Code over GF(3), x^4 + 2)
```

## 3.2. Matriz generadora

Por ser códigos lineales, los códigos cíclicos admiten también matrices generadoras que los determinan y con todas las características vistas en el capítulo anterior. Hemos visto que los códigos cíclicos están generados por su polinomio generador, luego es claro que este determina de alguna forma las matrices generadoras que admite un código. Previo al resultado que concreta dicha relación, introducimos un lema referente a una propiedad que presentan los polinomios generadores.

**Lema 3.1.** *Sea $g(x) = g_0 + g_1x + g_2x^2 + \cdots + g_rx^r$ el polinomio generador de un código cíclico $\mathcal{C} \subseteq R_n$. Entonces, $g_0 \neq 0$.*

*Demostración.* Podemos suponer sin pérdida de generalidad que $g_r \neq 0$, es decir, que $g(x)$ tiene grado $r$. Por ser $\mathcal{C}$ un código cíclico y como $g(x) \in \mathcal{C}$, entonces $x^{n-1}g(x) \in \mathcal{C}$, esto es,

$$x^{n-1}g(x) = g_0x^{n-1} + g_1x^n + g_2x^{n+1} + \cdots + g_rx^{n+r-1} = g_0x^{n-1} + g_1 + g_2x + \cdots + g_rx^{r-1} \in \mathcal{C}.$$

De este modo, si $g_0 = 0$, entonces $x^{n-1}g(x)$ es un polinomio de grado $r-1$ que pertenece a $\mathcal{C}$, luego se contradice la minimalidad del grado de $g(x)$ como polinomio generador. □

**Teorema 3.3.** *Sea $\mathcal{C}$ un código cíclico con polinomio generador $g(x) = g_0 + g_1x + \cdots + g_rx^r$ de grado $r$. Entonces, la dimensión de $\mathcal{C}$ es $n - r$ y una matriz generadora de $\mathcal{C}$ es*

$$G = \begin{pmatrix} g_0 & g_1 & g_2 & \cdots & g_r & 0 & 0 & \cdots & 0 \\ 0 & g_0 & g_1 & g_2 & \cdots & g_r & 0 & \cdots & 0 \\ 0 & 0 & g_0 & g_1 & g_2 & \cdots & g_r & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & & \ddots & 0 \\ 0 & 0 & \cdots & 0 & g_0 & g_1 & g_2 & \cdots & g_r \end{pmatrix}.$$

*Demostración.* ([Hil86], Teorema 12.12) La matriz $G$ descrita tiene dimensión $(n-r) \times n$, luego si probamos que en efecto es matriz generadora de $\mathcal{C}$ obtendremos que la dimensión de $\mathcal{C}$ es $n - r$.

Para ver que $G$ es una matriz generadora debemos comprobar que sus filas, vistas como elementos de $R_n$, constituyen una base de $\mathcal{C}$. Estos elementos conforman el conjunto

$$\mathcal{B} = \{g(x),\ xg(x),\ x^2g(x), ..., x^{n-r-1}g(x)\}.$$

Por un lado, es claro que son linealmente independientes por ser $G$ escalonada con $g_0 \neq 0$.

Por otro lado, veamos que $\mathcal{B}$ es un conjunto de generadores de $\mathcal{C}$. Dado $a(x) \in \mathcal{C}$, veamos que se puede expresar como combinación lineal de los elementos de $\mathcal{B}$.

En la prueba del Teorema 3.2 (2) vimos que se puede expresar

$$a(x) = q(x)g(x),$$

sin tener que realizar reducción módulo $x^n - 1$. Como $a(x) \in R_n$ tenemos que $deg(a(x)) < n$ y como $deg(g(x)) = r$ debe ser $deg(q(x)) < n - r$.

En este sentido, se podrá expresar $q(x)$ como

$$q(x) = q_0 + q_1 x + \cdots + q_{n-r-1} x^{n-r-1}.$$

De este modo, obtenemos la combinación lineal que buscábamos

$$\begin{aligned} a(x) = q(x)g(x) &= (q_0 + q_1 x + \cdots + q_{n-r-1} x^{n-r-1})g(x) \\ &= q_0 g(x) + q_1 x g(x) + \cdots + q_{n-r-1} x^{n-r-1} g(x). \end{aligned}$$

□

**Ejemplo 3.4.** Calculemos la matriz generadora de un $[6,3]_7$-código cíclico.

```
sage> n = 6
sage> F.<x> = GF(7)[]
sage> g = x^3 + 4*x^2 + 6*x + 3
sage> C = codes.CyclicCode(generator_pol=g, length=n)
sage> C
[6, 3] Cyclic Code over GF(7)
sage> C.generator_polynomial()
x^3 + 4*x^2 + 6*x + 3
sage> C.generator_matrix()
[3 6 4 1 0 0]
[0 3 6 4 1 0]
[0 0 3 6 4 1]
```

## 3.3. Matriz y polinomio de paridad

Nos centramos ahora en obtener una matriz de paridad para un código cíclico. Dado que la matriz generadora que hemos encontrado no está en forma estándar, no nos será cómodo partir de ella para la obtención de la matriz de paridad. Recurriremos en su lugar al conocido como *polinomio de paridad* que definiremos a continuación y cuya existencia se basa en el hecho de que el polinomio generador es un divisor del polinomio $x^n - 1$.

**Definición 3.3** (Polinomio de paridad)**.** Sea $\mathcal{C}$ un $[n,k]_q$-código cíclico con polinomio generador $g(x)$. Llamaremos *polinomio de paridad* de $\mathcal{C}$ al polinomio $h(x) \in R_n$ de grado $k$ que verifica:

$$x^n - 1 = g(x)h(x).$$

**Ejemplo 3.5.** Obtengamos el polinomio de paridad del código cíclico recién introducido en el Ejemplo 3.4.

```
sage> h = C.check_polynomial()
sage> h
x^3 + 3*x^2 + 3*x + 2
sage> g*h == x^n-1
True
```

**Teorema 3.4.** *Sea $\mathcal{C}$ un $[n,k]_q$-código cíclico con polinomio de paridad $h(x) = h_0 + h_1 x + \cdots + h_k x^k$. Entonces,*

*(1). el código $\mathcal{C}$ puede describirse como*

$$\mathcal{C} = \{p(x) \in R_n \mid p(x)h(x) \equiv 0\},$$

*(2). una matriz de paridad para $\mathcal{C}$ viene dada por*

$$H = \begin{pmatrix} h_k & h_{k-1} & h_{k-2} & \dots & h_0 & 0 & 0 & \dots & 0 \\ 0 & h_k & h_{k-1} & h_{k-2} & \dots & h_0 & 0 & \dots & 0 \\ 0 & 0 & h_k & h_{k-1} & h_{k-2} & \dots & h_0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & & \ddots & 0 \\ 0 & 0 & \dots & 0 & h_k & h_{k-1} & h_{k-2} & \dots & h_0 \end{pmatrix},$$

*(3). el código dual $\mathcal{C}^\perp$ es un $[n, n-k]$-código cíclico con polinomio generador*

$$h^\perp(x) = h_0^{-1}(h_k + h_{k-1}x + \cdots + h_0x^k).$$

*Demostración.* ([Rom92], Teorema 7.4.4)

(1). Sea $g(x)$ el polinomio generador de $\mathcal{C}$. Tomando $p(x) \in \mathcal{C} = \langle g(x) \rangle$, debe ser $p(x) = f(x)g(x)$ para cierto $f(x) \in R_n$. Por lo tanto,

$$p(x)h(x) = f(x)g(x)h(x) = f(x)(x^n - 1) \equiv 0 \pmod{x^n - 1}$$

y entonces $\mathcal{C} \subseteq \{p(x) \in R_n \mid p(x)h(x) \equiv 0\}$.

Por otro lado, sea $p(x) \in R_n$ tal que $p(x)h(x) \equiv 0 \pmod{x^n - 1}$. Dividiendo $p(x)$ entre $g(x)$ obtenemos

$$p(x) = q(x)g(x) + r(x),$$

con $deg(r(x)) < deg(g(x)) = n - k$. Si multiplicamos ahora por $h(x)$, obtenemos

$$\begin{aligned} p(x)h(x) &= q(x)g(x)h(x) + r(x)h(x) \\ &= q(x)(x^n - 1) + r(x)h(x) \\ &\equiv r(x)h(x) \pmod{x^n - 1}, \end{aligned}$$

luego $r(x)h(x) \equiv 0 \pmod{x^n - 1}$. Como $deg(r(x)h(x)) < (n-k) + k = n$, debe ser $r(x)h(x) = 0$. Así, $r(x) = 0$ y entonces $p(x) = q(x)g(x) \in \mathcal{C}$ , obteniendo la inclusión restante.

(2). Dado $c(x) \in \mathcal{C}$, por pertenecer a $R_n$, tendrá una expresión de la forma $c(x) = c_0 + c_1x + \cdots + c_{n-1}x^{n-1}$. Como $deg(c(x)h(x)) < n + k$, dicho producto se podrá expresar como

$$\begin{aligned} c(x)h(x) = \sum_{t=0}^{n+k-1} \left( \sum_{i+j=t} c_i h_j \right) x^t \equiv \\ \equiv \sum_{t=0}^{k-1} \left( \left( \sum_{i+j=t} c_i h_j \right) + \left( \sum_{i+j=n+t} c_i h_j \right) \right) x^t + \sum_{t=k}^{n-1} \left( \sum_{i+j=t} c_i h_j \right) x^t, \end{aligned}$$

donde hemos reducido módulo $x^n - 1$. Por el apartado (1) tenemos que $c(x)h(x) \equiv 0$, luego, en particular, los coeficientes de los monomios $x^k$, $x^{k+1}$, ..., $x^{n-1}$ serán nulos. Por lo tanto,

$$\begin{aligned} c_0h_k + c_1h_{k-1} + \cdots + c_kh_0 &= 0 \\ c_1h_k + c_2h_{k-1} + \cdots + c_{k+1}h_0 &= 0 \\ &\cdots \\ c_{n-k-1}h_k + c_{n-k}h_{k-1} + \cdots + c_{n-1}h_0 &= 0 \end{aligned}$$

Esto significa que $(c_0, c_1, ..., c_{n-1})H^T = 0$, de modo que el código $\mathcal{C}'$ generado por $H$ es ortogonal a $\mathcal{C}$, luego $\mathcal{C}' \subseteq \mathcal{C}$.

Por otro lado, debe ser $h_k = 1 \neq 0$, por ser $g(x)h(x) = x^n - 1$ con $g(x)$ mónico de grado $n - k$. Entonces, el rango de $H$ es $n - k$, luego $dim(\mathcal{C}') = n - k = dim(\mathcal{C}^\perp)$ y, por lo tanto, $\mathcal{C}' = \mathcal{C}^\perp$. De este modo, $H$ es una matriz generadora de $\mathcal{C}^\perp$, esto es, una matriz de paridad de $\mathcal{C}$.

(3). La idea es probar que el polinomio mónico $h^\perp(x)$ es un divisor de $x^n - 1$ y entonces, por el Teorema 3.2, se tendrá que $h^\perp(x)$ es el polinomio generador del código cíclico $\langle h^\perp(x) \rangle$, que tiene como matriz generadora a $H$, luego obtendremos que $\langle h^\perp(x) \rangle = \mathcal{C}^\perp$.

Para ello, partiendo de que

$$h(x)g(x) = x^n - 1$$

deducimos que

$$h(x^{-1})g(x^{-1}) = x^{-n} - 1$$

y entonces

$$x^k h(x^{-1}) x^{n-k} g(x^{-1}) = 1 - x^n,$$

esto es,

$$-h_0 h^\perp(x) x^{n-k} g(x^{-1}) = x^n - 1,$$

obteniendo que $h^\perp(x) | x^n - 1$.

Cabe notar que como $g(x)h(x) = x^n - 1$, debe ser $g_0h_0 = -1$ y, en particular, $h_0 \neq 0$, luego la expresión de $h^\perp(x)$ tiene sentido. □

**Ejemplo 3.6.** Partamos de nuevo del código del Ejemplo 3.4 para obtener ahora su matriz de paridad.

```
sage> C.check_polynomial()
x^3 + 3*x^2 + 3*x + 2
sage> C.parity_check_matrix()
[1 3 3 2 0 0]
[0 1 3 3 2 0]
[0 0 1 3 3 2]
```

## 3.4. Codificación y decodificación

Por último, describamos las tareas de codificación y decodificacion para los códigos cíclicos. Viéndolos como subconjuntos de $\mathbb{F}_q^n$, podemos codificar con la matriz generadora y decodificar por síndrome de la forma ya explicada para códigos lineales.

Alternativamente, podemos trabajar directamente con los códigos cíclicos como conjuntos de polinomios y entonces los polinomios generador y de paridad tendrán un papel análogo al de las respectivas matrices.

Consideremos $\mathcal{C}$ un $[n,k]_q$-código cíclico con polinomio generador $g(x)$ y polinomio de paridad $h(x)$. Supongamos que queremos codificar la palabra $a(x) = a_0 + a_1x + \cdots + a_kx^k$. Entonces, su palabra código asociada será

$$c(x) = a(x)g(x) \in \mathcal{C}.$$

Por otro lado, podemos describir un procedimiento análogo a la decodificación por síndrome utilizando polinomios. Sea $u(x)$ el polinomio recibido al transmitir la palabra código $c(x)$. El polinomio de error se definirá como $err(x) = u(x) - c(x)$ y el peso de un polinomio como el número de coeficientes no nulos que contiene. Además, el síndrome de $u(x)$, notado por $s(u(x))$ será el resto de dividir $u(x)$ por $g(x)$, esto es,

$$u(x) = q(x)g(x) + s(u(x)),$$

con $deg(s(u(x))) < r$.

Con dichos conceptos así definidos, el procedimiento de la decodificación por síndrome para polinomios es equivalente al que ya se describió para elementos de $\mathbb{F}_q^n$ en la Sección 2.5.

# 4 Códigos Goppa

Los códigos Goppa constituyen una familia de códigos lineales introducida por V.D. Goppa en 1970 [Gop70] que posee interesantes propiedades. Formularemos su definición más general y calcularemos una matriz de paridad que nos dará información sobre la dimensión y distancia mínima de estos códigos. Nos centraremos después en el caso binario, para el que estudiaremos un algoritmo de decodificación. Nos basaremos en ([VL82], Capítulo 8), ([Rom92], Sección 8.3) y ([EOS07], Capítulo 2).

Consideremos $\mathbb{F}_{q^m}$ el cuerpo finito de $q^m$ elementos con $q$ primo y tomemos un polinomio $G(x) \in \mathbb{F}_{q^m}[x]$. Sea $S_m$ el anillo cociente

$$S_m = \frac{\mathbb{F}_{q^m}[x]}{\langle G(x) \rangle}$$

de los polinomios sobre $\mathbb{F}_{q^m}$ módulo $G(x)$. Observemos que $S_m$ será un cuerpo si, y solo si, el polinomio $G(x)$ es irreducible en $\mathbb{F}_{q^m}[x]$.

Dado $\alpha \in \mathbb{F}_{q^m}$ tal que $G(\alpha) \neq 0$, veamos que el polinomio $x - \alpha$ tiene inverso en $S_m$. Para ello, dividamos $G(x)$ entre $x - \alpha$, obteniendo

$$G(x) = q(x)(x - \alpha) + G(\alpha).$$

Deducimos que $q(x)(x - \alpha) \equiv -G(\alpha) \pmod{G(x)}$, luego

$$\left(-G(\alpha)^{-1} q(x)\right)(x - \alpha) \equiv 1 \pmod{G(x)}.$$

Como

$$q(x) = \frac{G(x) - G(\alpha)}{x - \alpha},$$

llegamos a que

$$\frac{1}{x - \alpha} = -\frac{G(x) - G(\alpha)}{x - \alpha} G(\alpha)^{-1} \ \ en\ S_m. \tag{4.1}$$

Estamos ya en condiciones de definir los códigos Goppa.

**Definición 4.1** (Código Goppa)**.** Sea $G(x) \in \mathbb{F}_{q^m}$ y $L = \{\alpha_1, \alpha_2, ..., \alpha_n\} \subseteq \mathbb{F}_{q^m}$ tal que $G(\alpha_i) \neq 0$ para todo $i \in \{1, ..., n\}$ y con $n > deg(G(x))$. Dada una palabra $a = a_1 a_2 \cdots a_n \in \mathbb{F}_q^n$ consideramos

$$R_a(x) = \sum_{i=1}^{n} \frac{a_i}{x - \alpha_i} \in S_m.$$

El *código Goppa q-ario* $\Gamma(L, G)$ se define como

$$\Gamma(L, G) = \{a \in \mathbb{F}_q^n | R_a(x) \equiv 0 \pmod{G(x)}\}.$$

Al polinomio $G(x)$ lo llamaremos *polinomio de Goppa* para $\Gamma(L, G)$. Si el polinomio de Goppa es irreducible, diremos que el código Goppa es *irreducible*. Por otro lado, al conjunto $L$ lo llamaremos conjunto de definición.

*Observación* 4.1. El código Goppa $\Gamma(L, G)$ es lineal puesto que se trata del núcleo de la aplicación lineal $a \mapsto R_a(x) \pmod{G(x)}$.

## 4.1. Matriz de paridad

Procedamos ahora a encontrar una matriz de paridad para el código Goppa $\Gamma(L, G)$. Por definición, si $a \in \Gamma(L, G)$, entonces

$$\sum_{i=1}^{n} \frac{a_i}{x - \alpha_i} \equiv 0 \pmod{G(x)},$$

luego por la ecuación (4.1) obtenemos

$$\sum_{i=1}^{n} \left( G(\alpha_i)^{-1} \frac{G(x) - G(\alpha_i)}{x - \alpha_i} \right) a_i \equiv 0 \pmod{G(x)}.$$

Si el polinomio de Goppa se expresa como $G(x) = \sum_{k=0}^{t} G_k x^k$ con $G_t \neq 0$, entonces la ecuación anterior es equivalente a

$$\sum_{i=1}^{n} \left( G(\alpha_i)^{-1} \sum_{k=1}^{t} G_k \frac{x^k - \alpha_i^k}{x - \alpha_i} \right) a_i \equiv 0 \pmod{G(x)}. \tag{4.2}$$

Trataremos ahora de expresar la suma interior de otra forma, para lo que usaremos que si $b \neq c$, entonces

$$\frac{b^p - c^p}{b - c} = \sum_{h=0}^{p-1} b^h c^{p-1-h} = \sum_{h+j=p-1} b^h c^j.$$

Por lo tanto, para cada $i \in \{1, ..., n\}$,

$$\sum_{k=1}^{t} G_k \frac{x^k - \alpha_i^k}{x - \alpha_i} = \sum_{k=1}^{t} \left( G_k \sum_{h+j=k-1} x^h \alpha_i^j \right) = \sum_{h+j \leq t-1} G_{h+j+1} x^h \alpha_i^j.$$

De este modo, la ecuación (4.2) es equivalente a

$$\sum_{i=1}^{n} \left( G(\alpha_i)^{-1} \sum_{h+j \leq t-1} G_{h+j+1} x^h \alpha^j \right) a_i \equiv 0 \pmod{G(x)}.$$

Como el primer miembro de la congruencia es un polinomio en $x$ de grado menor que $t = deg(G(x))$, todos sus coeficientes deben ser nulos, esto es, para todo $h \in \{0, ..., t-1\}$

$$\sum_{i=1}^{n} \left( G(\alpha_i)^{-1} \sum_{j=0}^{t-1-h} G_{h+j+1} \alpha^j \right) a_i = 0.$$

Expresándolo matricialmente, tenemos que $a \in \Gamma(L, G)$ si, y solo si,

$$\begin{pmatrix} G_t G(\alpha_1)^{-1} & \dots & G_t G(\alpha_n)^{-1} \\ (G_{t-1} + G_t \alpha_1) G(\alpha_1)^{-1} & \dots & (G_{t-1} + G_t \alpha_n) G(\alpha_n)^{-1} \\ \vdots & \ddots & \vdots \\ \left( \sum_{j=1}^{t} G_j \alpha_1^{j-1} \right) G(\alpha_1)^{-1} & \dots & \left( \sum_{j=1}^{t} G_j \alpha_n^{j-1} \right) G(\alpha_n)^{-1} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = 0. \tag{4.3}$$

Dado que $G_t \neq 0$, podemos aplicar operaciones elementales por filas a la matriz anterior para obtener

$$H = \begin{pmatrix} G(\alpha_1)^{-1} & G(\alpha_2)^{-1} & \dots & G(\alpha_n)^{-1} \\ \alpha_1 G(\alpha_1)^{-1} & \alpha_2 G(\alpha_2)^{-1} & \dots & \alpha_n G(\alpha_n)^{-1} \\ \alpha_1^2 G(\alpha_1)^{-1} & \alpha_2^2 G(\alpha_2)^{-1} & \dots & \alpha_n^2 G(\alpha_n)^{-1} \\ \vdots & \vdots & \dots & \vdots \\ \alpha_1^{t-1} G(\alpha_1)^{-1} & \alpha_2^{t-1} G(\alpha_2)^{-1} & \dots & \alpha_n^{t-1} G(\alpha_n)^{-1} \end{pmatrix},$$

verificando que $a \in \Gamma(L, G)$ si, y solo si,

$$Ha^T = 0.$$

Cabe notar que las entradas de la matriz $H$ están en $\mathbb{F}_{q^m}$. Podemos pensar en los elementos $c$ de $\mathbb{F}_{q^m}$ como vectores columna $[c]$ en $\mathbb{F}_q^m$. De este modo, a partir de $H$ obtenemos de forma natural la matriz

$$H' = \begin{pmatrix} [G(\alpha_1)^{-1}] & [G(\alpha_2)^{-1}] & \dots & [G(\alpha_n)^{-1}] \\ [\alpha_1 G(\alpha_1)^{-1}] & [\alpha_2 G(\alpha_2)^{-1}] & \dots & [\alpha_n G(\alpha_n)^{-1}] \\ [\alpha_1^2 G(\alpha_1)^{-1}] & [\alpha_2^2 G(\alpha_2)^{-1}] & \dots & [\alpha_n^2 G(\alpha_n)^{-1}] \\ \vdots & \vdots & \dots & \vdots \\ [\alpha_1^{t-1} G(\alpha_1)^{-1}] & [\alpha_2^{t-1} G(\alpha_2)^{-1}] & \dots & [\alpha_n^{t-1} G(\alpha_n)^{-1}] \end{pmatrix}.$$

En este sentido, $H$ es una matriz de dimensión $t \times n$ sobre $\mathbb{F}_{q^m}$, mientras que $H'$ es una matriz de dimensión $tm \times n$ sobre $\mathbb{F}_q$. Conviene notar que dichas matrices podrían tener filas linealmente dependientes.

En conclusión, la matriz $H'$ es una matriz de paridad para $\Gamma(L, G)$ salvo la eliminación de filas linealmente dependientes. En general, por comodidad, las matrices $H$ y $H'$ serán consideradas matrices de paridad para el código Goppa aunque no lo sean de forma estricta.

## 4.2. Dimensión y distancia mínima

A continuación, obtengamos cotas para la dimensión del código Goppa así como para su distancia mínima a partir de las matrices $H$ y $H'$ calculadas en la sección anterior. En particular, atenderemos al rango de la matriz $H'$ notado por rango$(H')$.

Por un lado, como $H'$ tiene dimensión $tm \times n$, es claro que rango$(H') \leq tm$.

Por otro lado, si tomamos $t$ columnas cualesquiera de la matriz $H$, la submatriz cuadrada resultante es similar a una matriz de Vandermonde y podemos deducir entonces que tendrá determinante no nulo. De este modo, cualesquiera $t$ columnas de $H$ son linealmente independientes, luego lo mismo ocurrirá en $H'$. Entonces, tenemos que rango$(H') \geq t$.

Teniendo en cuenta que la dimensión de una matriz generadora de $\Gamma(L, G)$ será $k \times n$ con $k = n - \text{rango}(H')$ y por el Teorema 2.1 obtenemos el siguiente resultado.

**Teorema 4.1.** *El código Goppa $\Gamma(L, G)$ con $L \subseteq \mathbb{F}_q^m$, $|L| = n$ y $deg(G(x)) = t < n$ es un $[n, k]_q$-código lineal con*

$$n - mt \leq k \leq n - t$$

*y con distancia mínima $d \geq t + 1$.*

## 4.3. Códigos Goppa Binarios

Los códigos Goppa binarios son un caso particular de los códigos Goppa tomando $q = 2$. Recordemos además que un código Goppa se dice irreducible si su polinomio de Goppa es irreducible. Para los códigos Goppa binarios irreducibles podemos mejorar la cota de la distancia mínima que nos propociona el teorema anterior.

Sea $\Gamma(L, G)$ un código Goppa binario irreducible y tomemos $a = a_1 \cdots a_n \in \Gamma(L, G)$. Observemos que $a_i \in \{0, 1\}$ para todo $i \in \{1, ..., n\}$. Nos interesa quedarnos solo con las posiciones no nulas. Consideremos entonces

$$J = \{j \in \{1, ..., n\} \mid a_j = 1\}.$$

Definamos a partir de este conjunto el polinomio

$$f_a(x) = \prod_{j \in J} (x - \alpha_j).$$

Si denotamos por $f_a'(x)$ a la derivada de $f_a(x)$, tenemos que

$$R_a(x) f_a(x) = f_a'(x). \tag{4.4}$$

Teniendo en cuenta la expresión de $f_a(x)$ y que $G(\alpha) \neq 0$ para todo $\alpha \in L$, es claro que $f_a(x)$ y $G(x)$ son primos relativos. Deducimos entonces que

$$G(x) | R_a(x) \ \ si, \ y \ solo \ si, \ \ G(x) | f_a'(x).$$

Como $f_a(x)$ está definido sobre un cuerpo de característica 2, el polinomio $f_a'(x)$ solo contendrá potencias pares de $x$, de modo que $f_a'(x) = h(x^2) = h(x)^2$ para cierto polinomio $h(x)$.

Por ser $G(x)$ irreducible, tendremos que $G(x)|h(x)^2$ si, y solo si, $G(x)|h(x)$, lo que a su vez será equivalente a que $G(x)^2|h(x)^2 = f_a'(x)$. El polinomio $G(x)^2$ es también primo relativo con $f_a(x)$, luego por la igualdad (4.4) dichas condiciones de divisibilidad equivaldrán a que $G(x)^2|R_a(x)$.

Uniendo todas las equivalencias descritas llegamos a que

$$R_a(x) \equiv 0 \pmod{G(x)} \quad \textit{si, y solo si,} \quad R_a(x) \equiv 0 \pmod{G(x)^2},$$

esto es,

$$\Gamma(L, G) = \Gamma(L, G^2).$$

De este modo, por el Teorema 4.1 y la Proposición 1.3 obtenemos el siguiente resultado.

**Teorema 4.2.** *Sea $\Gamma(L, G)$ un código Goppa binario irreducible con $deg(G) = t$. Entonces su distancia mínima es $d \geq 2t + 1$. En particular, el código $\Gamma(L, G)$ es capaz de corregir al menos $t$ errores.*

**Ejemplo 4.1.** Son los códigos Goppa binarios los que han sido implementados mediante la clase CodigoGoppaBinario en el archivo *codigo_goppa_binario.py*. De este modo, podemos emplear dicha clase para presentar un ejemplo de uno irreducible.

Dicha clase proporciona un constructor que crea el código a partir del polinomio generador y el conjunto de definición. En primer lugar, determinamos el cuerpo sobre el que trabajar.

```
sage> F= GF(2^7)
sage> R.<x> = F[]
```

A continuación, generamos aleatoriamente polinomios de grado 4 hasta obtener uno irreducible y lo hacemos mónico. Este será el polinomio generador.

```
sage> g = R(0)
sage> while not g.is_irreducible():
        g = R.random_element(4)
        g = g/g.list()[len(g.list())-1]
sage> g
x^4 + (z7^6 + z7^3 + z7)*x^3 + (z7^6 + z7^5 + z7^3 + z7^2)*x^2 + (z7^5 + z7^3 + z7 +
    1)*x + z7^5 + z7^2 + z7
```

Como se trata de un polinomio irreducible, podemos tomar $L = \mathbb{F}_{2^7}$ como conjunto de definición.

```
sage> L = F.list()
```

Podemos entonces instanciar ya un código con dichos parámetros.

```
sage> C = CodigoGoppaBinario(g, L)
sage> C
[128, 100] Codigo Goppa Binario
```

De este código podemos obtener una matriz de paridad de la forma dada en la expresión (4.3). Para la implementación de esta funcionalidad se ha tenido en cuenta que, por su estructura, dicha matriz se puede expresar como producto de otras tres.

```
def _calcular_matriz_paridad(g, L):
    """
    Calcula la matriz de paridad de un CGB sobre el cuerpo finito GF(2^m).

    Calcula la matriz de paridad de un CGB dado por el polinomio
    generador g y con conjunto de definicion L. Las entradas de la
    matriz seran elementos del cuerpo finito GF(2^m) en el que estan
    los elementos de L.

    La matriz tendra dimension (t x n) con t = g.degree() y n = |L|.
    """
    t = g.degree()
    n = len(L)

    X = matrix([[g.list()[j] for j in range(t-i,t+1)]
                + [0]*(t-1-i) for i in range(t)])
    Y = matrix([[L[j]**i for j in range(n)] for i in range(t)])
    Z = diagonal_matrix([g(a).inverse_of_unit() for a in L])
    return X*Y*Z
```

,Probándolo con el [128, 100]-código Goppa binario generado vemos como efectivamente obtenemos una matriz de la dimensión adecuada y sobre el cuerpo no primo $\mathbb{F}_{2^7}$.

```
sage> C.matriz_paridad()
4 x 128 dense matrix over Finite Field in z7 of size 2^7
```

También es de interés obtener la matriz con entradas en el cuerpo $\mathbb{F}_2$. Esta se calcula a partir de la anterior mediante un método implementado que permite extender una matriz convirtiendo sus elementos en vectores columna.

```
def _extender_matriz(H1):
    """
    Extiende matriz sobre GF(2^m) a matriz sobre GF(2).

    Si H1 es una matriz de dimension (t x n) con elementos sobre el
    cuerpo finito GF(2^m), la matriz H2 devuelta sera una matriz de
    dimension (tm x n) con elementos sobre el cuerpo finito GF(2),
    extendiendo en columna los elementos del cuerpo finito GF(2^m).
    """
    t = H1.nrows()
    n = H1.ncols()
    m = len(vector(H1[0][0]))

    H2 = matrix(GF(2), m*t, 1)

    for i in range(n):
        columna = vector(H1[0][i]).column()
        for j in range(1,t):
            v = vector(H1[j][i]).column()
            columna = columna.stack(v)
        H2 = H2.augment(columna)

    H2 = H2.delete_columns([0])

    return H2
```

Para obtener la mencionada matriz de paridad con entradas en $\mathbb{F}_2$ recurrimos al método *matriz_paridad_extendida*.

```
sage> H = C.matriz_paridad_extendida()
sage> H
28 x 128 dense matrix over Finite Field of size 2
```

A partir de una matriz de paridad, incluso si no es una de rango máximo, se puede calcular una matriz generadora para el código de forma directa.

```
def _calcular_matriz_generadora(H):
    """
    Calcula la matriz generadora de un CGB con matriz de paridad H.
    """
    return H.right_kernel().basis_matrix()
```

Podemos probarlo en el código Goppa con el que estamos trabajando. Es la dimensión de esta matriz generadora la que permite determinar que la dimensión del código es 100.

```
sage> G = C.matriz_generadora()
sage> G
100 x 128 dense matrix over Finite Field of size 2
```

Vemos como efectivamente se verifica con la matrices $H$ y $G$ la propiedad requerida para una matriz de paridad y una generadora en los códigos lineales.

```
sage> (G*H.transpose()).is_zero()
True
```

## 4.4. Algoritmo de Patterson

Nos centraremos en los códigos Goppa binarios irreducibles pues son los que emplearemos para el criptosistema de McEliece. En este sentido, abordaremos ahora una cuestión clave como es la decodificación de estos códigos entendida como la corrección de errores de una palabra recibida. El algoritmo de Patterson [Pat75] es uno de los procedimientos más extendidos para resolver esta tarea. Previo a su explicación introduciremos conceptos importantes para la decodificación de los códigos Goppa como son el de polinomio síndrome y polinomio localizador de errores.

Consideremos un código Goppa binario irreducible $\Gamma(L, G)$ con $L = \{\alpha_1, ..., \alpha_n\} \subseteq \mathbb{F}_{2^m}$ y $deg(G) = t$. Sea $r \in \mathbb{F}_q^n$ la palabra recibida al transmitir cierta palabra código $c \in \Gamma(L, G)$. En este sentido, la palabra $r$ se descompondrá como

$$r = c + e,$$

siendo $e \in \mathbb{F}_2^n$ el patrón de error. Sabemos que $\Gamma(L, G)$ puede corregir al menos $t$ errores, luego supondremos que el peso de $e$ es $w(e) \leq t$.

Recurriremos a la expresión (4.1) para definir el polinomio síndrome, que está estrechamente relacionado con el síndrome ya definido de forma general para códigos lineales. Presentemos la definición en el contexto del código binario $\Gamma(L, G)$ que estamos considerando.

**Definición 4.2** (Polinomio síndrome)**.** El *polinomio síndrome* asociado a una palabra $a \in \mathbb{F}_q^n$ se define como

$$S_a(x) = \sum_{i=1}^{n} \left( -\frac{G(x) - G(\alpha)}{x - \alpha} G(\alpha)^{-1} \right) a_i.$$

*Observación* 4.2. Los coeficientes del polinomio síndrome así definido son los que se obtienen al multiplicar la matriz de la expresión (4.3) por la palabra $r$ en vector columna. Recordamos que dicha matriz es de paridad salvo la conversión de sus entradas a $\mathbb{F}_q^m$ y la eliminación de las filas linealmente dependientes. Es aquí donde radica la relación entre el polinomio síndrome $S_r(x)$ y el concepto de síndrome ya presentado para códigos lineales con la Definición 2.7.

*Observación* 4.3. Por la definición del código Goppa tenemos que $S_c(x) \equiv 0 \pmod{G(x)}$. Por lo tanto, $S_r(x) \equiv S_e(x) \pmod{G(x)}$.

Presentamos también al otro polinomio clave en la decodificación y que ya utilizamos con otra notación para demostrar la cota de la distancia de esta familia de códigos.

**Definición 4.3** (Polinomio localizador de errores)**.** El *polinomio localizador de errores* asociado al error $e$ se define como

$$\sigma_e(x) = \prod_{j \in J} (x - \alpha_j),$$

donde

$$J = \{j \in \{1, ..., n\} \mid e_j = 1\}.$$

*Observación* 4.4. Las raíces de $\sigma_e(x)$ nos indicarán las posiciones no nulas del patrón de error $e$, esto es, las posiciones en que la palabra recibida $r$ tiene un valor erróneo. Dado que estamos trabajando con palabras sobre el alfabeto binario, conocer las posiciones erróneas de $r$ nos será suficiente para determinar cuál fue la palabra código $c$ enviada originalmente.

Del mismo modo que vimos en la expresión (4.4), tenemos que

$$\sigma_e(x) S_e(x) \equiv \sigma_e'(x) \pmod{G(x)}. \tag{4.5}$$

Por estar trabajando en un cuerpo de característica 2 podemos descomponer $\sigma_e(x)$ separando los términos con potencias pares y los de potencias impares como

$$\sigma_e(x) = \alpha^2(x) + x\beta^2(x), \tag{4.6}$$

resultando que $\sigma_e'(x) = \beta^2(x)$. Esto nos permite reescribir la ecuación (4.5) como

$$\beta^2(x)(xS_e(x) + 1) \equiv \alpha^2(x) S_e(x) \pmod{G(x)}.$$

Podemos suponer que el patrón de error $e$ no es una palabra código, lo que implica que $S_e(x) \neq 0 \pmod{G(x)}$. Entonces, existe el inverso de $S_e(x)$ módulo $G(x)$, que notaremos como $T(x) = S_e^{-1}(x)$, y multiplicando la expresión (4.6) por $T(x)$ obtenemos

$$\beta^2(x)(x + T(X)) \equiv \alpha^2(x) \pmod{G(x)}. \tag{4.7}$$

Cabe notar que, como estamos trabajando en un cuerpo de característica 2, la función $x \mapsto x^2$

es biyectiva, conocida como el automorfismo de Frobenius. En este sentido, podemos tomar $\tau(x) = \sqrt{T(x) + x}$. Así, tomando raíz cuadrada en la ecuación (4.7) llegamos a

$$\beta(x)\tau(x) \equiv \alpha(x) \pmod{G(x)}. \tag{4.8}$$

Conociendo $\tau(x)$ y $G(x)$, trataremos de calcular los polinomios $\alpha(x)$ y $\beta(x)$ de menor grado que satisfagan la ecuación (4.8). Por hipótesis, $deg(\sigma_e(x)) \leq t$, luego por la ecuación (4.6) llegamos a que $deg(\alpha(x)) \leq \lfloor t/2 \rfloor$ y $deg(\beta(x)) \leq \lfloor (t-1)/2 \rfloor$. De este modo, la ecuación (4.8) tendrá una solución única, que la podremos calcular mediante el algoritmo extendido de Euclides con la modificación apropiada para imponer dichas cotas sobre los grados de los polinomios.

Una vez conozcamos $\alpha(x)$ y $\beta(x)$ podremos obtener el valor de $\sigma_e(x)$ y a partir de sus raíces podremos corregir los errores presentes en $r$.

Estructuremos todo esto en los pasos que componen el algoritmo de Patterson.

1. Calculamos el polinomio síndrome $S_r(x)$, que coincidirá con $S_e(x)$ módulo $G(x)$.

   Recordamos que podemos obtener los coeficientes de este polinomio multiplicando por la matriz descrita en (4.3), que es la que se almacena en el atributo *_matriz_paridad* de la clase implementada. De este modo, podemos implementar el cálculo del polinomio síndrome de la siguiente forma.

```
def pol_sindrome(self, y):
    """
    Devuelve el polinomio sindrome asociado a una palabra y.
    """
    x = self._pol_generador.parent().gen()
    coefs = self._matriz_paridad*y

    return sum([coefs[i]*x**(len(coefs)-i-1) for i in range(len(coefs))])
```

2. Calculamos $T(x)$ el inverso de $S_e(x)$ módulo $G(x)$.

   La tarea de invertir un polinomio en el anillo cociente se puede resolver directamente tomando el coeficiente de Bézout apropiado que nos proporciona el algoritmo extendido de Euclides. Si $u(x)$ y $v(x)$ son polinomios tales que

   $$1 = \text{mcd}(S_e(x), G(x)) = u(x)S_e(x) + v(x)G(x),$$

   entonces

   $$u(x) = S_e^{-1}(x) \pmod{G(x)}.$$

```
def _invertir(p, g):
    """
    Invierte p en el anillo cociente sobre <g>.
    """
    return xgcd(p, g)[1].mod(g)
```

3. Si $T(x) = x$ entonces $\sigma_e(x) = x$ y ya hemos terminado. En caso contrario, calculamos $\tau(x)$ tal que $\tau(x)^2 \equiv T(x) + x \pmod{G(x)}$.

   Para calcular el polinomio $\tau(x)$ recurrimos al método propuesto por K. Huber [Hub96].

La idea es calcular primero el polinomio $w(x)$ que satisface

$$w^2(x) \equiv x \pmod{G(x)}.$$

Para ello, utilizamos que estamos en un cuerpo de característica 2 para descomponer $G(x)$ como

$$G(x) = G_1^2(x) + xG_2^2(x),$$

y entonces

$$w(x) = G_1(x)G_2^{-1}(x) \pmod{G(x)}.$$

Seguidamente, si descomponemos $T(x) + x$ como

$$T(x) + x = T_1^2(x) + xT_2^2(x),$$

entonces el polinomio $\tau(x)$ vendrá dado por

$$\tau(x) = T_1(x) + w(x)T_2(x).$$

Implementamos por un lado la función que descompone un polinomio.

```
def _descomponer(p):
    """
    Descompone un polinomio p sobre GF(2^m) como p(x) = p0(x)^2 + x*p1(x)^2.
    """
    PR = p.parent()
    raices = [sqrt(c) for c in p.list()]
    return PR(raices[0::2]), PR(raices[1::2])
```

A partir de ella, podemos implementar la función que calcula la raíz cuadrada.

```
def _sqrt(p, g):
    """
    Calcula la raiz cuadrada de p en el anillo cociente sobre <g>.
    """
    g1, g2 = _descomponer(g)
    w = (g1*_invertir(g2, g)).mod(g)
    p1, p2 = _descomponer(p)
    return (p1+w*p2).mod(g)
```

4. Calculamos $\alpha(x)$ y $\beta(x)$ de menor grado que satisfagan $\beta(x)\tau(x) \equiv \alpha(x) \pmod{G(x)}$, mediante el algoritmo extendido de Euclides modificado.

   La modificación consiste en detener las iteraciones del algoritmo cuando encontremos polinomios $\alpha(x)$ y $\beta(x)$ que satisfagan las condiciones sobre los grados, a saber, $deg(\alpha(x)) \leq \lfloor t/2 \rfloor$ y $deg(\beta(x)) \leq \lfloor (t-1)/2 \rfloor$.

   La función que implementa dicha funcionalidad es la siguiente.

```
def _xgcd_grado_acotado(p,q):
    """
    Calcula (d,u,v) tal que d = u*p + v*q acotando los grados de d y u.

    Aplica el algoritmo extendido de Euclides para obtener el maximo
    comun divisor y los coeficientes de Bezout para dos polinomios p y
    q acotando el grado del maximo comun divisor y de uno de los
    coeficientes de Bezout.

    Devuelve: (d,u,v) tal que:
        - d = u*p + v*q,
        - d.degree() <= q.degree()//2 y
        - u.degree() <= (q.degree()-1)//2.
    """
    t = q.degree()
    PR = p.parent()

    (d1, d2) = (p, q)
    (u1, u2) = (PR(1), PR(0))
    (v1, v2) = (PR(0), PR(1))

    while (d1.degree() > t//2) or (u1.degree() > (t-1)//2):
        quotient = d1 // d2
        (d1, d2) = (d2, d1 - quotient * d2)
        (u1, u2) = (u2, u1 - quotient * u2)
        (v1, v2) = (v2, v1 - quotient * v2)

    return (d1, u1, v1)
```

5. Obtenemos $\sigma_e(x) = \alpha^2(x) + x\beta^2(x)$.
6. Calculamos las raíces de $\sigma_e(x)$, que nos indican las posiciones erróneas de $r$.

   Cabe notar que estas raíces serán elementos del conjunto de definición $L$, de modo que basta con comprobar cuáles de los elementos de $L$ anulan el polinomio localizador de errores $\sigma_e(x)$.

Agrupando todos estos pasos y con las funciones así descritas, el algoritmo de Patterson se ha implementado de la siguiente forma.

```
def decodificar(self, y):
    """
    Devuelve la palabra codigo mas cercana a y.

    Aplica el algoritmo de Patterson.
    """
    g = self._pol_generador
    x = g.parent().gen()
    L = self._conjunto
    palabra = copy(y)

    # 1. Calculamos polinomio sindrome.
    pol_sind = self.pol_sindrome(y)
    if pol_sind == 0:
        return palabra

    # 2. Calculamos el inverso del polinomio sindrome.
    T = _invertir(pol_sind, g)

    # 3.a. Si T(x)=x, el polinomio localizador de errores es x.
    if T == x:
        sigma = x
    else:
        # 3.b. Si T(x)!=x, calculamos la raiz cuadrada de T(x)+x.
        tau = _sqrt(T+x, g)

        # 4. Calculamos alpha(x) y beta(x).
        (alpha, beta, v) = _xgcd_grado_acotado(tau, g)

        # 5. Obtenemos el polinomio localizador de errores sigma(x).
        sigma = alpha**2 + x*beta**2

    # 6. Las raices de sigma(x) indican las posiciones a corregir.
    for i in range(len(L)):
        if sigma(L[i]) == 0:
            palabra[i] +=1

    return palabra
```

**Ejemplo 4.2.** Podemos emplear el [128,100]-código Goppa binario irreducible del Ejemplo 4.1 para ilustrar la codificación y decodificación.

Por un lado, podemos generar una palabra aleatoria de $\mathbb{F}_2^{100}$ y codificarla.

```
sage> x = VectorSpace(GF(2), 100).random_element()
sage> x
(0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0,
0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1,
0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1,
1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0)
sage> y = C.codificar(x)
sage> y
(0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0,
0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1,
0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1,
1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1,
0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1)
```

Dado que el polinomio generador tiene grado 4, por el Teorema 4.2, sabemos que el código $\mathcal{C}$ es capaz de corregir al menos 4 errores. Generemos entonces un patrón de error de peso 4 y veamos como la decodificación es correcta.

```
sage> e = vector(GF(2), 128*[0])
sage> e[4] = 1
sage> e[5] = 1
sage> e[7] = 1
sage> e[100] = 1
sage> e
(0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
sage> y2 = y + e
sage> y2
(0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0,
0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1,
0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1,
1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1,
0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1)
sage> y3 = C.decodificar(y2)
sage> y3
(0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0,
0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1,
0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1,
1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1,
0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1)
sage> y3 == y
True
```

# 5 El criptosistema de McEliece

El criptosistema de McEliece [McE78] es un criptosistema de clave pública que se fundamenta en la teoría de códigos correctores de errores y, en particular, en los códigos Goppa. La teoría desarrollada hasta el momento nos permitirá presentar este criptosistema de forma sencilla. Describiremos su esquema de funcionamiento acompañándolo de su implementación. Además, estudiaremos la variante principal que se ha planteado, analizaremos su seguridad repasando los distintos ataques que se han ideado y se comentará la reciente propuesta en el NIST que busca convertirlo en un estándar.

Previamente, resulta conveniente contextualizar el problema que resuelve el criptosistema de McEliece con una breve descripción de la criptografía de clave pública.

## 5.1. Criptografía de clave pública

La criptografía resuelve la necesidad de que dos partes puedan intercambiar información de forma segura evitando que terceras personas puedan leer los mensajes intercambiados. La idea general es que el emisor cifra el mensaje antes de enviarlo y el receptor lo descifra una vez lo reciba para poder leer así su contenido original. La seguridad radica en que el receptor dispone de cierta clave que le permite descifrar el mensaje de manera sencilla, mientras que la tarea de descifrarlo sin la clave no es factible.

Hasta mediados de los años 70 solo se empleaban sistemas criptográficos simétricos, que se basan en que el emisor y receptor poseen un secreto o clave común conocida como clave simétrica. Esta clave simétrica es la que se utiliza para cifrar y descifrar los mensajes, y solo es conocida por aquellos que se están comunicando. En este sentido, dicha clave tiene que haber sido compartida de forma segura con anterioridad. Esto supone un gran inconveniente si dos partes quieren comunicarse pero no disponen de un medio seguro para compartir la clave simétrica en primer lugar.

En el año 1976, Diffie y Hellman [DH76] propusieron un esquema criptográfico que resolvía dicha problemática y que constituyó el inicio de la conocida como criptografía de clave pública o criptografía asimétrica. Esta se basa en que la clave para cifrar el mensaje es pública y solo la clave para descifrar es privada, de modo que no es necesario compartir ningún secreto previo al intercambio de información.

Describamos con más detalle el esquema básico de la criptografía asimétrica. Imaginemos un escenario en el que dos personas desean intercambiar información: Alice y Bob. En particular, supondremos que Alice quiere mandarle un mensaje a Bob de manera que solo él pueda leerlo. La idea es que Bob dispondrá de dos claves, una clave pública a la que todo el mundo tendrá acceso y una clave privada que solo él conocerá.

De este modo, Alice tomará el mensaje que quiere enviar y lo cifrará empleando la clave

pública de Bob. A continuación, enviará el mensaje cifrado a Bob. Cuando Bob lo reciba, utilizará su clave privada para descifrarlo y leer así el mensaje original. Si alguien distinto a Bob interceptara el mensaje, no sería capaz de entender su contenido, pues no dispondría de la clave privada requerida para el descifrado.

Uno de los criptosistemas de clave pública más extendidos es el propuesto en 1977 por Rivest, Shamir y Adleman [RSA78], el famoso criptosistema RSA. Este se basa en la complejidad computacional del problema de factorizar el producto de dos números primos grandes. Solo un año más tarde, McEliece propuso un criptosistema de clave pública basado en los códigos correctores de errores [McE78], criptosistema objeto de nuestro estudio.

Cabe resaltar que los procesos de cifrado y descifrado en la criptografía asimétrica suelen ser mucho más costosos computacionalmente y los tamaños de las claves son considerablemente más grandes comparando con la criptografía simétrica. En este sentido, la criptografía de clave pública suele usarse solo para el inicio de una comunicación segura, permitiendo intercambiar un secreto común entre las dos partes, en particular, para intercambiar una clave simétrica. A partir de entonces, la comunicación podrá continuar de manera más eficiente empleando un criptosistema simétrico.

De este modo, los mensajes que se intercambian con un criptosistema asimétrico serán, en general, un medio para construir claves simétricas. Por lo tanto, no tendremos que preocuparnos por darle un sentido al tipo de mensajes que estamos cifrando. Esta cuestión tendrá especial relevancia cuando estudiemos la variación al criptosistema de McEliece propuesta por Niederreiter.

## 5.2. Descripción del criptosistema

Sea $g(x)$ un polinomio irreducible de grado $t$ sobre $\mathbb{F}_{2^m}$. Podemos considerar entonces el código Goppa binario irreducible $\Gamma = \Gamma(g(x), L)$ con $L = \mathbb{F}_{2^m}$. Se trata de un código de longitud $n = |L| = 2^m$ y, por los Teoremas 4.1 y 4.2, sabemos que tendrá dimensión $k \geq n - mt$ y será capaz de corregir cualquier patrón de error de peso menor o igual que $t$.

Tomemos $G$ una matriz generadora $k \times n$ del código $\Gamma$. Además, tomemos aleatoriamente dos matrices: una matriz regular $S$ de dimensión $k \times k$ y una matriz de permutación $P$ de dimensión $n \times n$. Definimos entonces $G' = SGP$. Podemos ver la matriz $S$ como una matriz de cambio de base y la matriz $P$ como una permutación de las columnas de $G$. En este sentido, la matriz $G'$ genera un código lineal equivalente a $\Gamma$, luego tendrá su misma capacidad de corrección de errores.

De este modo, la clave privada es $(g(x), S, P)$ y la clave pública $(G', t)$.

Podemos definir entonces los procedimientos de cifrado y descifrado.

**Cifrado:** Sea $u \in \mathbb{F}_2^k$ la palabra de longitud $k$ que queremos cifrar. Tomemos aleatoriamente una palabra $e \in \mathbb{F}_2^n$ de longitud $n$ con peso $\mathrm{w}(e) = t$. Si notamos por $x \in \mathbb{F}_2^n$ la palabra resultante de cifrar $u$, entonces el cifrado viene dado por

$$x = uG' + e.$$

En definitiva, el cifrado consiste en codificar $u$ con la matriz generadora $G'$ y añadirle un patrón de error $e$ de peso $t$.

**Descifrado:** Sea ahora $x \in \mathbb{F}_2^n$ la palabra a descifrar. Observemos que $x = uSGP + e$, de modo que $xP^{-1} = (uS)G + eP^{-1}$. Calculamos entonces $x' = xP^{-1}$, revirtiendo así la permutación de columnas efectuada por la matriz $P$. Podemos buscar ahora la palabra código de $\Gamma$ más cercana a $x'$ aplicando el algoritmo de decodificación de Patterson, corrigiendo así los $t$ errores. Sea $x''$ la palabra resultante de aplicar dicho algoritmo. Resolviendo un sistema lineal obtenemos $u'$ tal que $x'' = u'G$. Solo queda revertir el cambio de base, luego podremos obtener la palabra original $u$ calculando $u = u'S^{-1}$.

**Ejemplo 5.1.** Utilicemos SageMath y la clase CodigoGoppaBinario implementada para mostrar un ejemplo de cifrado y descifrado con el criptosistema de McEliece.

Determinamos primero el grado $t$ del polinomio de Goppa y el cuerpo finito $\mathbb{F}_{2^m}$ sobre el que trabajar.

```
sage> t = 3
sage> m = 4
sage> n = 2^m
sage> F= GF(2^m)
sage> R.<x> = F[]
```

Generamos aleatoriamente un polinomio irreducible de grado $t$ sobre $\mathbb{F}_{2^m}$ que será el polinomio de Goppa $g$.

```
sage> g = R(0)
sage> while not g.is_irreducible():
        g = R.random_element(t)
        g = g/g.list()[len(g.list())-1]
sage> g
x^3 + (z4^3 + 1)*x^2 + (z4^3 + 1)*x + z4^3 + z4^2 + z4 + 1
```

Tomamos como conjunto de definición $L = \mathbb{F}_{2^m}$ y construimos el código Goppa binario irreducible.

```
sage> L = F.list()
sage> C = CodigoGoppaBinario(g, L)
sage> C
[16, 4] Codigo Goppa Binario
```

Tomamos $G$ una matriz generadora del código.

```
sage> G = C.matriz_generadora()
sage> G
[1 0 0 1 0 1 1 0 0 0 0 1 0 0 1 1]
[0 1 0 1 0 0 0 0 1 1 1 1 0 1 1 1]
[0 0 1 0 0 1 0 0 0 0 1 1 1 1 0 1]
[0 0 0 0 1 0 0 1 1 1 1 0 1 1 0 1]
```

Llamamos $k$ a la dimensión del código.

```
sage> k = C.dimension()
sage> k
4
```

Generamos $S$ una matriz regular aleatoria de dimensión $k \times k$.

```
sage> S = random_matrix(GF(2), k)
sage> while S.is_singular():
        S = random_matrix(GF(2), k)
sage> S
[1 0 1 1]
[1 1 1 1]
[0 0 1 0]
[0 1 1 1]
```

Generamos $P$ una matriz de permutaciones aleatoria de dimensión $n \times n$.

```
sage> P = Permutations(n).random_element().to_matrix()
sage> P = matrix(GF(2), P)
sage> P
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
```

Construimos la matriz generadora $G_2 = SGP$, que constituirá la clave pública junto a $t$.

```
sage> G2 = S*G*P
sage> G2
[1 0 0 1 1 0 1 0 0 1 1 1 1 1 1 0]
[1 0 0 0 1 1 0 1 1 0 1 0 1 0 1 1]
[0 1 1 0 1 1 0 0 1 0 0 1 0 0 0 1]
[0 1 0 0 1 1 0 1 0 1 1 1 1 1 0 1]
```

Generemos una palabra aleatoria $u \in \mathbb{F}_2^k$ para cifrarla.

```
sage> u = VectorSpace(GF(2),k).random_element()
sage> u
(0, 1, 1, 1)
```

Para cifrar, generamos un patrón de error $e$ aleatorio de peso $t$.

```
sage> errores = Combinations(range(n),t).random_element()
sage> errores
[2, 5, 14]
sage> e = vector(GF(2), n*[0])
sage> for pos in errores:
    e[pos]=1
sage> e
(0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0)
```

Ciframos $u$ multiplicando por la matriz generadora $G_2$ y sumándole el patrón de error.

```
sage> x = u*G2 + e
sage> x
(1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1)
```

Procedamos ahora a descifrar la palabra $x \in \mathbb{F}_2^n$.

Deshacemos la permutación de las columnas multiplicando por la inversa de $P$.

```
sage> x2 = x*P.inverse()
sage> x2
(1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0)
```

Aplicamos el algoritmo de Patterson para corregir los $t$ errores.

```
sage> x3 = C.decodificar(x2)
sage> x3
(1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0)
```

Resolvemos el sistema lineal para obtener la palabra $u_2 \in \mathbb{F}_2^k$ asociada.

```
sage> u2 = G.solve_left(x3)
sage> u2
(1, 0, 1, 0)
```

Finalmente, deshacemos el cambio de base producido por $S$.

```
sage> u3 = u2*S.inverse()
sage> u3
(0, 1, 1, 1)
```

Comprobamos que hemos descifrado correctamente.

```
sage> u3 == u
True
```

## 5.3. Versión de Niederreiter

En el año 1986, Harald Niederreiter [Nie86] propuso una variación del criptosistema de McEliece que lo convierte en un criptosistema más eficiente. Se trata de una versión dual del criptosistema de McEliece en tanto que se basa en el uso de una matriz de paridad en vez de una matriz generadora. La propuesta original de Niederreiter sugería el empleo de una familia de códigos lineales distinta, los llamados códigos Reed-Solomon generalizados, pero dicha elección lo convertía en un criptosistema inseguro (ver [SS92]). Sin embargo, la seguridad no se ve afectada si se emplea con los códigos Goppa binarios irreducibles. Estudiémoslo entonces usando esta familia de códigos.

En este sentido, tomemos de nuevo un polinomio $g(x)$ irreducible de grado $t$ sobre $\mathbb{F}_{2^m}$ y consideremos el código Goppa binario irreducible $\Gamma = \Gamma(g(x), L)$ donde $L = \mathbb{F}_{2^m}$. Se trata de un código con longitud $n = 2^m$, dimensión $k \geq n - mt$ y capaz de corregir al menos $t$ errores.

Tomemos ahora una matriz de paridad $H$ del código $\Gamma$. Por otro lado, tomemos de forma aleatoria dos matrices: una matriz regular $M$ de dimensión $(n-k) \times (n-k)$ y una matriz de

permutación $P$ de dimensión $n \times n$. Así, definimos $H' = MHP$. De forma análoga al esquema propuesto por McEliece, la matriz $H$ será matriz de paridad de un código equivalente a $\Gamma$.

En este caso, la clave privada será $(g(x), M, P)$ y la clave pública $(H', t)$.

Describamos ahora el procedimiento de cifrado y el de descifrado.

**Cifrado:** La palabra a cifrar será un elemento $u \in \mathbb{F}_2^n$ de peso $\mathrm{w}(u) = t$. Recordamos que no debemos preocuparnos por la aparente restricción que supone que la palabra a cifrar tenga dicha estructura, pues en general no utilizaremos este esquema para cifrar directamente el mensaje que queremos enviar sino como medio para establecer una clave simétrica. Si llamamos $x \in \mathbb{F}_2^{n-k}$ a la palabra resultante de cifrar $u$, podemos expresar el cifrado como

$$x^T = H'u^T,$$

esto es, el cifrado es el síndrome de la palabra $u$.

**Descifrado:** Consideremos ahora $x \in \mathbb{F}_2^{n-k}$ la palabra a descifrar. Tengamos en cuenta que $x^T = MHPu^T$, luego $M^{-1}x^T = H(Pu^T)$. De este modo, podemos calcular primero $x'^T = M^{-1}x^T$. Aplicando la decodificación por síndrome basada en el algoritmo de Patterson podemos encontrar la palabra $u' \in \mathbb{F}_2^n$ de peso $t$ que tiene como síndrome a $x'$. A partir de ella obtenemos $u$ como $u^T = P^{-1}u'^T$.

Dicha decodificación por síndrome es similar al procedimiento explicado del algoritmo de Patterson, con la salvedad de que el polinomio síndrome se calcula a partir del síndrome siendo sus coeficientes los elementos del cuerpo $\mathbb{F}_{2^m}$ obtenidos a partir de las entradas en $\mathbb{F}_2$ del síndrome. Además, los errores localizados por el algoritmo serán las posiciones no nulas de la palabra buscada.

De este modo, se puede implementar de la siguiente forma.

```
def decodificar_por_sindrome(self, sind):
    """
    Devuelve la palabra de menor peso con sindrome sind.

    Aplica el algoritmo de Patterson.
    """
    g = self._pol_generador
    x = g.parent().gen()
    L = self._conjunto
    palabra = vector([0]*(len(L)))
    F = g.base_ring()
    m = len(vector(L[0]))

    # 1. Calculamos polinomio sindrome a partir del sindrome.
    coefs = list(F(sind[i:i+m]) for i in range(0,len(sind),m))
    pol_sind = sum([coefs[i]*x**(len(coefs)-i-1) for i in range(len(coefs))])

    # 2. Calculamos el inverso del polinomio sindrome.
    T = _invertir(pol_sind, g)

    # 3.a. Si T(x)=x, el polinomio localizador de errores es x.
    if T == x:
        sigma = x
    else:
        # 3.b. Si T(x)!=x, calculamos la raiz cuadrada de T(x)+x.
        tau = _sqrt(T+x, g)
```

```
        # 4. Calculamos alpha(x) y beta(x).
        (alpha, beta, v) = _xgcd_grado_acotado(tau, g)

        # 5. Obtenemos el polinomio localizador de errores sigma(x).
        sigma = alpha**2 + x*beta**2

    # 6. Las raices de sigma(x) indican las posiciones no nulas.
    for i in range(len(L)):
        if sigma(L[i]) == 0:
            palabra[i] = 1

    return palabra
```

Es importante destacar que los criptosistemas de McEliece y Niederreiter son equivalentes en términos de seguridad, en el sentido de que un ataque que rompa uno de ellos también romperá el otro (ver [LDW94]). Estudiaremos más en detalle la cuestión de la seguridad en la siguiente sección.

**Ejemplo 5.2.** Partamos del mismo $[16, 4]$-código Goppa binario irreducible del ejemplo del criptosistema de McEliece para ejemplificar el cifrado y descifrado con el criptosistema de Niederreiter.

Tomamos entonces $H$ una matriz de paridad del código Goppa.

```
sage> H = C.matriz_paridad_extendida()
sage> H
[0 1 0 0 0 0 0 1 0 1 0 0 1 0 1 1]
[0 0 1 1 1 1 0 0 0 0 1 1 1 0 0 1]
[0 1 0 1 1 1 0 0 0 0 1 0 1 1 0 0]
[1 1 0 0 1 0 1 1 1 0 0 1 1 0 1 0]
[0 0 1 0 0 0 0 1 0 1 0 1 0 1 0 1]
[0 1 0 1 1 0 1 0 1 1 1 1 0 1 0 1]
[1 1 0 1 1 1 1 0 0 0 0 0 1 0 0 0]
[0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 1]
[0 1 1 0 1 0 0 0 0 0 1 0 1 0 1 1]
[0 1 1 1 0 0 0 1 0 0 0 0 1 1 0 1]
[1 0 0 1 1 0 0 1 0 1 1 1 1 1 1 0]
[0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0]
```

Generamos $M$ una matriz regular aleatoria de dimensión $(n-k) \times (n-k)$.

```
sage> M = random_matrix(GF(2), n-k)
sage> while M.is_singular():
        M = random_matrix(GF(2), n-k)
sage> M
[1 0 1 1 0 0 0 1 0 1 0 0]
[1 1 1 0 0 1 1 0 0 1 1 1]
[1 0 0 0 0 0 1 1 1 1 1 0]
[1 1 0 1 0 1 0 1 1 1 0 0]
[0 1 1 1 1 1 0 1 1 0 0 1]
[0 1 1 0 0 1 0 0 0 0 0 1]
[0 0 0 1 0 1 1 1 0 0 0 0]
[1 0 0 0 1 0 1 0 0 0 0 1]
[1 1 1 0 1 0 1 0 0 0 1 1]
[1 1 0 0 1 0 1 0 0 1 0 0]
[0 1 1 0 0 1 0 0 0 1 1 0]
[1 1 0 0 0 0 1 1 0 1 1 1]
```

Generamos $P$ una matriz de permutaciones aleatoria de dimensión $n \times n$.

```
sage> P = Permutations(n).random_element().to_matrix()
sage> P = matrix(GF(2), P)
sage> P
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
```

Construimos la matriz de paridad $H_2 = MHP$, que constituirá la clave pública junto con $t$.

```
sage> H2=M*H*P
sage> H2
[1 1 0 0 0 0 1 0 1 0 0 1 1 0 0 1]
[0 0 0 1 0 0 1 1 1 0 0 0 1 1 0 1]
[0 1 1 1 1 0 0 0 1 1 1 0 1 1 0 0]
[1 1 1 1 0 0 0 1 1 1 0 0 1 1 1 1]
[1 1 0 1 0 1 0 0 0 1 0 1 0 1 1 1]
[0 1 0 1 1 0 1 0 0 1 0 0 0 1 1 0]
[0 1 0 1 1 1 1 1 0 0 1 0 1 1 0 1]
[1 0 0 0 1 1 0 0 1 1 0 0 0 0 1 1]
[0 0 1 0 0 1 1 1 1 0 1 1 1 1 1 0]
[1 0 0 0 1 0 0 1 0 1 0 0 1 1 1 1]
[1 0 0 1 0 0 0 1 1 1 0 1 0 1 1 0]
[0 1 1 1 1 0 0 1 0 0 0 0 1 1 0 1]
```

Generemos una palabra aleatoria $u \in \mathbb{F}_2^n$ de peso t para cifrarla.

```
sage> posiciones = Combinations(range(n),t).random_element()
sage> posiciones
[0, 3, 6]
sage> u = vector(GF(2), n*[0])
sage> for pos in posiciones:
        u[pos]=1
sage> u
(1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

Ciframos $u$ multiplicando por la matriz de paridad $H_2$.

```
sage> x = H2*u
sage> x
(0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1)
```

Procedamos ahora a descifrar la palabra $x \in \mathbb{F}_2^{n-k}$.

Primero, deshacemos el cambio de base generado por $M$.

```
sage> x2 = M.inverse()*x
sage> x2
(1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0)
```

Aplicamos la decodificación por síndrome basada en el algoritmo de Patterson para corregir los $t$ errores.

```
sage> x3 = C.decodificar_por_sindrome(x2)
sage> x3
(1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0)
```

Finalmente, deshacemos la permutación de columnas.

```
sage> u2 = P.inverse()*x3
sage> u2
(1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

Comprobamos que hemos descifrado correctamente.

```
sage> u2 == u
True
```

## 5.4. Seguridad

La clave para que un criptosistema sea seguro es que no sea factible descifrar un mensaje sin el conocimiento de la clave privada. Se considera que la resolución de un problema es factible o no en función del esfuerzo computacional que conlleve resolverlo, que está directamente relacionado con la clase de complejidad en la que esté el problema.

Presentemos las dos clases de complejidad más distinguidas: la clase $\mathcal{P}$ y la clase $\mathcal{NP}$. La clase $\mathcal{P}$ se corresponde con el conjunto de problemas para los que existe un algoritmo que los resuelve y que termina en tiempo polinomial, esto es, en un número de pasos acotado polinomialmente respecto de la longitud de la entrada. Por otro lado, los problemas $\mathcal{NP}$ son aquellos para los que es posible verificar que una solución es correcta en tiempo polinomial.

En términos generales, podemos considerar que los problemas de la clase $\mathcal{P}$ son aquellos que son manejables y que los que no están en esta clase no son factibles de resolverse en un tiempo razonable. La inclusión $\mathcal{P} \subseteq \mathcal{NP}$ es evidente, pero la cuestión sobre si dicha inclusión es estricta o si se da la igualdad es uno de los problemas sin resolver más importantes de la informática. Los problemas $\mathcal{NP}$-completos constituyen una familia de problemas $\mathcal{NP}$ ampliamente estudiados para los que no se ha conseguido probar que estén en $\mathcal{P}$ y tal que si se probara que uno de ellos está en $\mathcal{P}$ entonces todos los problemas de $\mathcal{NP}$ lo estarían. En este sentido, no se espera que los problemas $\mathcal{NP}$-completos estén en $\mathcal{P}$, aunque aún no se ha conseguido probar.

En el caso de los criptosistemas de McEliece y Niederreiter, el problema subyacente al descifrado conociendo únicamente la clave pública es la tarea de decodificación por síndrome de un código lineal equivalente a un código Goppa binario irreducible. De este modo, el fundamento de la seguridad de dichos criptosistemas es el siguiente resultado.

**Teorema 5.1.** *Sean $H \in M_{r\times n}(\mathbb{F}_2)$, $y \in \mathbb{F}_2^r$, $t \in \mathbb{N}$. Entonces, el problema de determinar si existe $x \in \mathbb{F}_2^n$ con peso $\mathrm{w}(x) \leq t$ tal que $xH^T = y$ es $\mathcal{NP}$-completo.*

*Demostración.* Puede encontrarse en [BMvT78]. □

Este resultado conlleva que si se toman valores de $n$ y $t$ lo suficientemente grandes, la resolución de la decodificación por síndrome del código lineal no se espera que sea factible.

Pese a la importante trascendencia de este resultado, no es suficiente para garantizar la seguridad del criptosistema de McEliece. La cuestión es que se podría tratar de atacar el descifrado con otros enfoques que no conllevaran la resolución de la decodificación por síndrome. El estudio de las distintas formas de abordar un ataque a un criptosistema entran en la disciplina conocida como criptoanálisis.

De este modo, el criptoanálisis consiste en la búsqueda de vulnerabilidades de un criptosistema, en idear ataques que *rompan* su seguridad y comprobar si se pueden llevar a cabo en un tiempo razonable.

El criptosistema de McEliece ha sido objeto de un exhaustivo criptoanálisis desde su publicación y lo ha resistido, en el sentido de que se puede seleccionar un conjunto de valores razonables para sus parámetros que le permiten resistir a todos los ataques ideados.

Una de las técnicas más efectivas para atacar el criptosistema de McEliece es la conocida como decodificación por conjuntos de información o *information-set decoding*. Un enfoque básico es el que plantea el propio McEliece en su propuesta del criptosistema [McE78], consistente en encontrar tantas posiciones no erróneas como la dimensión del código y recuperar a partir de ellas la palabra codificada originalmente por álgebra lineal. Si la matriz generadora $G$ tiene dimensión $k \times n$, la idea es tomar $k$ columnas linealmente independientes de $G$, que constituyen una matriz regular $G_2$ de dimensión $k \times k$. Si las correspondientes $k$ posiciones de la palabra a decodificar no han sufrido ningún error, entonces la multiplicación del vector con dichas posiciones por $G_2^{-1}$ proporciona correctamente la palabra decodificada. Una sistematización de este ataque puede encontrarse en [LB88].

Variantes cada vez más sofisticadas se han ido desarrollando, destacando las ideadas por Leon [Leo88], Stern [Ste88], Canteaut y Chabaud [CC98] y Bernstein, Lange y Peters [BLP08]. Todas estas se basan en un problema cuya resolución permite atacar el criptosistema de McEliece: encontrar la palabra de menor peso en un código lineal. Detallemos este enfoque. Sea $\mathcal{C}$ un código lineal binario de longitud $n$ con distancia mínima $2t+1$. Sea $r \in \mathbb{F}_2^n$ la palabra recibida al enviar una palabra código $c \in \mathcal{C}$ habiéndose cometido $t$ errores, esto es, $r$ se descompone como $r = c + e$ con $\mathrm{w}(e) = t$. Sabemos que $c$ es la única palabra código a distancia menor o igual que $t$ de $r$. Por lo tanto, y teniendo en cuenta que estamos trabajando en binario, si desplazamos todas las palabras de $\mathcal{C}$ sumando $r$, obtenemos un código $\mathcal{C} + \{r\}$ en el que $e = c + r$ es la única palabra de peso menor o igual que $t$. Además, sabemos que, por su distancia mínima, en $\mathcal{C}$ no puede haber ninguna palabra no nula de peso menor o igual que $t$, luego $e$ es la única palabra no nula de peso menor o igual que $t$ en el código lineal $\mathcal{C} + \{0, r\}$. Podemos obtener una matriz generadora para este código ampliando una matriz generadora de $\mathcal{C}$ añadiéndole una última fila con el vector $r$. En este sentido, encontrando $e$ como la palabra no nula de menor peso en el código lineal $\mathcal{C} + \{0, r\}$ podemos corregir los errores de $r$ obteniendo $c$ como $c = r + e$. De este modo, es este el problema que han ido resolviendo con algoritmos cada vez más sofisticados en las propuestas antes listadas.

Siguiendo esta línea, el ataque más efectivo ideado contra el criptosistema de McEliece se basa en la combinación del algoritmo cuántico de Grover [Gro96] con la decodificación por conjuntos de información tal y como se decribe en [Ber10].

En cualquier caso, se puede asegurar al criptosistema de McEliece de todos los ataques propuestos hasta ahora mediante una adecuada elección de parámetros, permaneciendo como un criptosistema seguro tras tantos años de criptoanálisis.

Sin embargo, hay otro tipo de ataques para los que la propuesta inicial del criptosistema no estaba diseñada para resistir: ataques en los que el atacante puede ir modificando el mensaje cifrado y conocer si se ha decodificado de forma correcta. Este ataque se propuso en [VDvT02]. Suponemos que Alice está codificando mensajes con el criptosistema de McEliece y enviándoselos a Bob, pero hay un atacante situado entre ellos que intercepta los mensajes de Alice y los puede modificar antes de enviárselos a Bob. La idea es que si cuando el atacante modifica el mensaje, este acaba teniendo más de $t$ errores, entonces la decodificación por parte de Bob fallará y le solicitará a Alice el reenvío del mensaje, hecho que el atacante detectará. En este sentido, si el atacante modifica dos posiciones del mensaje y no se notifica ningún error en la decodificación, entonces significará que una de las posiciones modificadas contenía un error. Procediendo sistemáticamente, el atacante puede deducir todas las posiciones erróneas y descifrar así el mensaje original.

Este ataque pone en relieve el hecho de que más tipos de ataques se van ideando con el tiempo y es necesario ir mejorando los criptosistemas para que proporcionen la máxima protección posible.

## 5.5. Classic McEliece

Una de las propuestas que mejor recoge todas las mejoras desarrolladas a lo largo del tiempo sobre el criptosistema de McEliece es la llamada Classic McEliece [BCL+17]. Esta fue presentada en 2017 ante la petición del NIST (National Institute of Standards and Technology) de iniciar un proceso de selección de criptosistemas post-cuánticos a estandarizar para prevenir la posible llegada de los ordenadores cuánticos. Actualmente, esta propuesta se encuentra como finalista en el proceso de selección.

Classic McEliece aúna los avances en eficiencia y seguridad fruto del exhaustivo análisis desarrollado durante los años. Cabe destacar que sigue la variante propuesta por Niederreiter de trabajar con la matriz de paridad [Nie86], pero utilizando códigos Goppa binarios irreducibles tal y como propuso McEliece [McE78]. Una diferencia respecto de la versión del criptosistema estudiada en este trabajo es que en la propuesta de Classic McEliece no se toma como conjunto de definición del código Goppa los $2^m$ elementos de $\mathbb{F}_{2^m}$, sino que se toma un subconjunto de tamaño $n$ más reducido de forma aleatoria, lo que aumenta el número de claves privadas posibles.

Por otro lado, incluye un procedimiento basado en el uso de una función hash para protegerlo frente a ataques con mensaje cifrado adaptativo como el propuesto en [VDvT02].

Además, se optimiza el tamaño de las claves construyendo las matrices en forma sistemática. De este modo, no es necesario almacenar el bloque que se corresponde con la matriz identidad.

En definitiva, se trata de una propuesta basada en el criptosistema de McEliece que se nutre del exhaustivo criptoanálisis y mejoras desarrolladas para posicionarse como un criptosistema seguro que podría acabar siendo estandarizado.

# Conclusiones y trabajo futuro

La teoría de los códigos correctores de errores representa una potente herramienta para la construcción de sistemas criptográficos seguros. Su estudio me ha introducido en una interesante teoría con una rica estructura matemática y con aplicaciones en múltiples campos. Además, el trabajo con los códigos correctores de errores me ha permitido familiarizarme con un versátil software matemático de código abierto como es SageMath.

Por otro lado, he podido estudiar el criptosistema de McEliece, uno de los principales candidatos a estandarizarse para prevenir la amenaza que los ordenadores cuánticos plantean sobre la seguridad de las comunicaciones. La criptografía post-cuántica constituye un campo de estudio relativamente joven y que va a requerir una importante dedicación para refinar la seguridad de los criptosistemas, optimizar los tamaños de las claves y la eficiencia de los procesos de cifrado y descifrado, así como para su integración en los protocolos que usamos a diario para comunicarnos a través de internet.

Este trabajo se ha centrado en la versión del criptosistema de McEliece con códigos Goppa. Se trata de la familia de códigos que propuso McEliece originalmente y que ha resistido a todos los ataques que se han planteado hasta el momento. Otros autores han propuesto reemplazar los códigos Goppa por otras familias de códigos que permitieran una mayor eficiencia, pero en muchos casos han resultado ser inseguras, como es el caso de los códigos Reed-Solomon generalizados propuestos por Niederreiter. En cualquier caso, el estudio de otras familias de códigos y su criptoanálisis puede resultar en una interesante línea de trabajo.

Por otro lado, hay otros tipos de criptosistemas que también se cree que pueden resistir a los ordenadores cuánticos, como son los basados en hash, en retículos o en ecuaciones cuadráticas multivariantes.

En definitiva, la criptografía post-cuántica es un campo de estudio de vital importancia con muchas interesantes líneas de trabajo abiertas por abordar.

# Apéndices

## Código

Todo el código implementado, incluyendo los cuadernos de jupyter de los que se ha extraído el código que se encuentra en los distintos ejemplos a lo largo del documento, puede encontrarse en GitHub:

https://github.com/amerigal/criptosistema-de-mceliece.

## Redacción del documento

Este documento se ha redactado en LaTeX a partir de la plantilla ubicada en el siguiente repositorio de GitHub:

https://github.com/latex-mat-ugr/Plantilla-TFG.

## Agradecimientos

Me gustaría agradecer al catedrático Pedro A. García Sánchez sus exhaustivas revisiones, su disponibilidad y directrices que han hecho posible el desarrollo de este trabajo.

Del mismo modo, quiero agradecer el apoyo que me han brindado mi familia y amigos durante esta inolvidable etapa en Granada.

# Bibliografía


[BCL+17] Daniel J Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, et al. Classic McEliece: conservative code-based cryptography. *NIST submissions*, 2017.

[Ber10] Daniel J Bernstein. Grover vs. mceliece. In *International Workshop on Post-Quantum Cryptography*, pages 73–80. Springer, 2010.

[BLP08] Daniel J Bernstein, Tanja Lange, and Christiane Peters. Attacking and defending the McEliece cryptosystem. In *International Workshop on Post-Quantum Cryptography*, pages 31–46. Springer, 2008.

[BMvT78] E. Berlekamp, R. McEliece, and H. van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory*, 24(3):384–386, 1978.

[CC98] A. Canteaut and F. Chabaud. A new algorithm for finding minimum-weight words in a linear code: application to McEliece's cryptosystem and to narrow-sense bch codes of length 511. *IEEE Transactions on Information Theory*, 44(1):367–378, 1998.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[EOS07] D. Engelbert, R. Overbeck, and A. Schmidt. A Summary of McEliece-type Cryptosystems and their Security. *Journal of Mathematical Cryptology*, 1(2):151–199, 2007.

[Gop70] Valery Denisovich Goppa. A new class of linear error-correcting codes. *Probl. Inf. Transm.*, 6:300–304, 1970.

[Gro96] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[Hil86] Raymond Hill. *A first course in coding theory*. Oxford University Press, 1986.

[Hub96] K Huber. Note on decoding binary Goppa codes. *Electronics Letters*, 32(2):102–103, 1996.

[KRKP+16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter Notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

[LB88] Pil Joong Lee and Ernest F Brickell. An observation on the security of McEliece's public-key cryptosystem. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 275–280. Springer, 1988.

[LDW94] Yuan Xing Li, R.H. Deng, and Xin Mei Wang. On the equivalence of McEliece's and Niederreiter's public-key cryptosystems. *IEEE Transactions on Information Theory*, 40(1):271–273, 1994.

[Leo88] J.S. Leon. A probabilistic algorithm for computing minimum weights of large error-correcting codes. *IEEE Transactions on Information Theory*, 34(5):1354–1359, 1988.

[McE78] Robert J McEliece. A public-key cryptosystem based on algebraic coding. *Thv*, 4244:114–

116, 1978.

[Nie86] Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Prob. Contr. Inform. Theory*, 15(2):159–166, 1986.

[Pat75] N. Patterson. The algebraic decoding of Goppa codes. *IEEE Transactions on Information Theory*, 21(2):203–207, 1975.

[Rom92] Steven Roman. *Coding and information theory*, volume 134. Springer Science & Business Media, 1992.

[RSA78] RL Rivest, A Shamir, and L Adleman. A method for obtaining digital signatures and public key Cryptosystems. *Communications of the ACM*, 27:120–126, 1978.

[Sho94] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[SS92] Vladimir Michilovich Sidelnikov and Sergey O Shestakov. On insecurity of cryptosystems based on generalized Reed-Solomon codes. 1992.

[Ste88] Jacques Stern. A method for finding codewords of small weight. In *International Colloquium on Coding Theory and Applications*, pages 106–113. Springer, 1988.

[Tea21] The Sage Development Team. SageMath, version 9.4, 2021.

[VDvT02] Eric R Verheul, Jeroen M Doumen, and Henk CA van Tilborg. Sloppy Alice attacks! Adaptive chosen ciphertext attacks on the McEliece public-key cryptosystem. In *Information, coding and mathematics*, pages 99–119. Springer, 2002.

[VL82] J.H. Van Lint. Introduction to Coding Theory. 1982. *Grad. Texts in Math*, 1982.