UNIVERSIDAD
DE GRANADA

TRABAJO FIN DE GRADO

Doble Grado en Ingeniería Informática y Matemáticas

# Códigos correctores de errores y criptografía

## Criptosistemas post-cuánticos de tipo McEliece


**Autor**
Juan Antonio Velasco Gómez

**Tutores**
Pedro A. García-Sánchez
José Ignacio Farrán Martín





Facultad de Ciencias
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación


—

Granada, Junio de 2017

# Índice general

Escaneando este código QR se puede acceder al repositorio del trabajo.
También a través del siguiente enlace.

# Códigos correctores de errores y criptografía

**Estudio de códigos de Goppa. El criptosistema de McEliece.**

**Autor**
Juan Antonio Velasco Gómez

**Tutores**
Pedro A. García-Sánchez
José Ignacio Farrán Martín

## Códigos correctores de errores y criptografía

Juan Antonio Velasco Gómez (alumno)

**Palabras clave**: Códigos correctores, Goppa, Goppa binarios, Algoritmo, Patterson, McEliece, Post-Cuántico, Ataque, Seguridad, Vulnerable

### Resumen

En este Trabajo de Fin de Grado se aborda el estudio de la Teoría de Códigos Correctores basada en los códigos de Goppa. Para comenzar, plantearemos uno de los principales problemas de la criptografía actual frente a la posibilidad de sufrir un ataque cuántico. Incluyendo sus posibles consecuencias. De hecho, trataremos de responder a la pregunta de si la criptografía actual se encuentra en riesgo. Para responderla, buscaremos criptosistemas conocidos que pudiesen ser capaces de aguantar un posible ataque cuántico. Hasta el momento se piensa que algunos como RSA y los cifrados de curvas elípticas son vulnerables. Parece que el algoritmo de Shor es capaz de vulnerarlos. Debemos buscar otras posibilidades. Daniel J. Bernstein propuso el criptosistema de McEliece como respuesta a esta pregunta, puesto que supuestamente era capaz de resistir el algoritmo de Shor. A lo largo del trabajo se irá evolucionando desde los conceptos más básicos y que conforman la base de esta teoría de códigos correctores hasta los algoritmos de recuperación de información en los cuales residen las esperanzas de resistir un posible ataque cuántico. Para llegar hasta este punto introduciremos conceptos muy importante en los códigos correctores como el concepto de palabra, el concepto de distancia de Hamming que nos servirá para medir el número de elementos que difieren entre dos palabras cualesquiera de un determinado código. Esta distancia nos será muy importante cuando busquemos corregir los errores que tiene una palabra en concreto.

¿Pero cuándo diremos que un código está bien definido? En general hablaremos de códigos lineales, es decir, códigos que poseen una estructura algebraica. Esto nos llevará a introducir el concepto de códigos binarios, que

serán el tipo de códigos que usaremos en los algoritmos de Patterson y de McEliece.

Estos algoritmos usan procesos de codificación y decodificación, procesos que cuando estemos usando códigos lineales serán mucho más simples que en el caso general. Esto nos llevará a introducir una serie de conceptos muy importantes para entender los pasos de estos procesos. Se introducirá la matriz generatriz de un código, que será una aplicación lineal biyectiva que forma una base del código. Veremos también las matrices de control de un código y estableceremos una relación entre estas dos matrices.

¿Existe relación entre las matrices, en especial la matriz de control y la distancia mínima de un código? En efecto. Esta relación la veremos cuando introduzcamos los conceptos de soporte de un elemento y el peso de Hamming de un elemento del código.

Después de establecer la base de la teoría de códigos correctores estudiaremos el caso de detección de errores, introduciendo los patrones de errores o el síndrome de una palabra de un código, que nos ayudará a detectar esa existencia de errores en una palabra.

Además se presentarán las herramientas necesarias para poder realizar estos algoritmos, entre ellos los códigos de Goppa, más en concreto los códigos binarios de Goppa. Estos códigos presentan una serie de características que los dotan de una gran importancia ante el problema de encontrar algoritmos de cifrado capaces de aguantar ataques que otros algoritmos como RSA, ElGamal o la criptografía de curvas elípticas no han sido capaces (de forma teórica) de aguantar. El aprendizaje de los códigos de Goppa nos permitirá entender el algoritmo de Patterson, capaz de detectar y corregir los errores de una palabra recibida.

También veremos el cifrado de McEliece, un tipo de criptosistema de clave pública que usa códigos correctores para crear la clave pública y privada. Este criptosistema no es totalmente seguro, de hecho, son conocidos algunos de los ataques que podrían comprometer dicho algoritmo como por ejemplo el ataque Stern, que veremos más en profundidad en la última parte del trabajo y que consiste en la búsqueda de palabras codificadas con un peso de Hamming bajo. Sin embargo, su estudio es importante para el caso que nosotros estamos considerando, el posible ataque cuántico puesto que aún no se han encontrado algoritmos de corrección de errores eficientes sin el conocimiento de un polinomio generador y por tanto es complicado recomponer la estructura del código a partir de la información conocida por un posible atacante.

En relación con el ataque de Stern, analizaremos la seguridad del criptosistema de McEliece y trataremos de buscar soluciones ante estos ataques capaces de romperlo. El estudio de este tipo de criptosistemas capaces de resistir los posibles ataques de los sistemas cuánticos es de gran importancia. De hecho, el estudio cobra aún más importancia cuando la búsqueda de estos criptosistemas se realiza en la actualidad, es decir, buscamos aque-

llos algoritmos actuales que pudiesen resistir con éxito ataques tan potentes como los que se podrán realizar dentro de poco tiempo.

Finalmente, estableceremos una serie de conclusiones con respecto al trabajo realizado, analizaremos el cumplimiento de los objetivos establecidos y volveremos a reflexionar, al igual que en la introducción sobre el estado de la criptografía en la actualidad. Sus ventajas e inconvenientes frente a la posible llegada (inminente o no) de la computación y de la criptografía cuántica. Estas conclusiones vendrán acompañadas de una serie de comentarios sobre posible trabajo futuro en este campo, también relacionada con esa posible llegada de la criptografía cuántica, la búsqueda de nuevos algoritmos de cifrado post-cuánticos, las ventajas e inconvenientes frente a la criptografía actual y las posibilidades “reales"de que estos cifrados lleguen a nuestras manos razonando por qué debería de centrarse la comunidad en este nuevo tipo de criptografía en vez de seguir de lleno en la criptografía actual.

Todo el trabajo matemático viene acompañado de un desarrollo informático utilizando varios software matemáticos. El primero de ellos, basado en el lenguaje de programación Python es SageMath, un software de código libre que cubre muchos aspectos de las matemáticas como el álgebra, la combinatoria, el cálculo numérico y la teoría de números. Para ello dispone de múltiples paquetes matemáticos que dotan al sistema de una gran potencia. En él se han ido realizando los ejemplos visto en la teoría así como el algoritmo de Patterson y el cifrado de McEliece. Además de este software, también se han visto los mismos ejemplos implementados en GAP, un sistema para álgebra computacional y muy útil en la teoría de códigos que implementa su propio lenguaje de programación.

# Error-Correcting Code and Cryptography

Juan Antonio Velasco Gómez (student)

**Keywords**: Error-Correcting codes, Goppa codes, Binary Goppa codes, Patterson Algorithm, McEliece cryptosystem, Post-Quantum cryptography, Attack, Security, Vulnerability

## Abstract

In this manuscript we study error-correcting code theory and cryptography based on Goppa codes.

We start by describing the eventual issues that current cryptography systems might have against a possible quantum attack. So, in fact, we will be answering the following question: Is present cryptography at risk? To answer this question we will try to find known cryptosystems that could resist the attack of a quantum-computer. At the moment, we know that RSA and Elliptic Curve Digital Signature Algorithm might be broken under this kind of attacks. It seems like Shor algorithm is the right way to vulnerate these cryptosystems. Thus the need to find alternate cryptographic possibilities. A candidate came into scene when Daniel J. Bernstein proposed the McEliece Cryptosystem, because it was pottencially able to resist the above mentioned Shor algorithm.

In order to describe MacEkiece Cryptosystem, we need linear codes and some basic concepts of public key cryptography. Thus, we will go from the basic definitions of code correction theory to some information recovery algorithm that (so far) resist a possible quantum attack. We will introduce concepts like words, which are the elements of a code, distance between words, Hamming distance (a particular distance that will help us to know the number of different elements in two words from a particular code). This distance will be very important when we try to correct the errors in a word occurring during a transmission.

Linear codes are of particular interest because of their algebraic structure. Linear binary codes will be subspaces of $\mathbb{F}_q^n$, for a positive integer $n$ and

$\mathbb{F}_q$ a finite field. Thus its elements will be words of zeros and ones of length $n$. We will use these particular codes to study Patterson algorithm and the McEliece cryptosystem.

Linear codes use a codification and a de-codification process. For the case of linear binary codes these procedures are easier than in the general setting. The concept of generator matrix of a linear code is crucial to successfully understand these algorithms. The rows of this matrix are a basis of the code as a vector space. Encoding is just multiplying the word we want to transmit by the generator matrix of the code. Thus the codewords are simply the images of a linear map between vectors spaces over finite fields. A parity-check matrix of a linear code behaves like the dual of a generator matrix, and a code can be also understood precisely as the kernel of this matrix (seen as a linear map). Generator matrices are used to encode, whilst parity-check matrices are used to detect errors. From a parity-check matrix we will define error patterns and syndromes of a word in a code. Both of them will help us to detect the existence of errors in a word, and to correct them if possible.

One can construct equivalent codes to a given one by performing elementary row operations in the generator matrix and then some permutation on its columns. This is the idea behind the use of cryptography with error correcting codes. It is difficult to unwind the changes on the generator matrix to recover the original one, and so the information encoded with the new matrix will not be understood by a potential eavesdropper. The private key will be essentially the changes the owner of the key performs on the rows of the generator matrix and the permutation on the columns. In order to encode, the sender will generate a random error with given tolerance, and then it will be added to the word to be transmitted. The receiver can has the original generator matrix and thus decode the message even with this error added by the sender.

Cryptosystems based on Goppa codes, and in particular Binary Goppa codes, can resist quantum attacks. For this we will study what is an irreducible polynomial and a Goppa polynomial, used to define a Goppa Code. This resistance against quantum attacks is mainly due to their structure and properties, some of them related to the concept of distance that we will be seeing. This is why McEliece cryptosystem is gaining attention as an alternative to other systems like RSA, ElGamal or ECSDA (Elliptic Curve Digital Signature Algorithm) that could theoretically be broken with quantum computers. The Hamming distance of a Goppa code is harder to find than in other codes. Given this, the next question to answer is how to encode a word using Goppa codes. We will be answering this theorically and by given some examples of codewords. Another advantage of Goppa codes is that for them there exist several fast decoding alternatives to syndrome tables. We will describe (and implement) here one of these: Patterson's Algorithm. This procedure makes use of an error-positioning polynomial, which will help us finding and detecting the errors in a given codeword. In

Binary Goppa Codes we will see this is even easier because there is only two possible values for a position, one or zero. In case of finding the position of an error, switching zero for one will solve the problem.

Linear codes can be decoded by using syndrome tables. For Binary Goppa codes some extra (and faster) procedures can be used. One of them is Patterson algorithm. We will describe and implement this algorithm allowing us to obtain the original word and the positions where the errors were placed in a given codeword.

This does not mean that McEliece cryptosystem is totally secure nor that it has no vulnerabilities, in fact, there are some known attacks that could break it. Stern's attack, which we will analyze in the last part of this manuscript, tries to search for coded words that have small Hamming weight. We will also study the security of the McEliece cryptosystem and its possible vulnerabilities. We suggest some new features to improve the security.

The final part of this manuscript is devoted to conclusions and proposed future work. We will analyze the achievement of the initial objectives, and will think about the state of cryptography at the moment; the advantages and disadvantages in the case quantum-cryptography reaches our day life. We will suggest some future work in this area, like searching for new cypher post-quantum algorithms or the "real" possibilities of having these cryptosystems in our hands. Should we focus our work on improving present cryptography or should we switch to quantum-cryptography?

The theoretical part will be accompanied by a series of examples and implementation of the algorithms presented. This will be done in two different languages: GAP and SageMath. On the one hand, GAP has a library for coding theory, named GUAVA, though it does not have Patterson's Algorithm implemented. However its simplicity of use is helpful to propose different examples, and to deal with large ones that could be not affordable by hand. It also provides of some other large data libraries of algebraic objects and algebraic algorithms. GAP is used in research and teaching for studying groups and their representations, rings, vector spaces, algebras, combinatorial structures, and more. On the other hand, SageMath covers many aspects of mathematics, including algebra, combinatorics, numerical mathematics, number theory, and calculus. SageMath is a free and open source software and is based on Python syntax, a very popular programming language. Python is one of the most used programming languages at the moment, both in academic and working aspects. It acts like an wrapper for popular mathematical software like R, GAP, PARI/GP, Singular, 4ti2, etcetera. We will see how to use SageMath by providing several examples along this manuscript. Also we show how to implement Patterson algorithm to detect and correct errors in a codeword, and how to use McEliece's cryptosystem using Alice and Bob communication problem.

One of the motivations to accomplish this project was to study security

in communications. We had some introductory courses on Cryptography but I wanted to go a bit further now that I think I am prepared. In the last years I have been looking to study computer security unite with mathematics. This is why I proposed this subject for a degree thesis. Coding Theory was not in any of the courses I followed in this university, and so this was an excellent opportunity to cover this interesting topic. Besides, Algebra has been one of my favorite subjects this year, where I had the chance of studying in more detail Group Theory. I think this area will become one of the most interesting ones in the near future, by matching mathematical theory with computacional science. Privacy and communication security are constantly in risk and our security systems and cryptosystems suffer millions of cyber attacks each day. The good news here are that both Coding Theory and Cryptography amalgamete nicely to produce strong cryptographic systems that could resist these attacks. Security defense must evolve as fast as security offensive. Even faster. My plans for the near future are to continue my academic formation in these matters, and if possible work in the area, trying to focus my efforts on systems and communications security.

# Agradecimientos

Lo primero agradecer a los dos tutores del trabajo, Pedro A. García Sánchez y José Ignacio Farrán Martín por su tiempo, paciencia y por compartir su conocimiento y sus ganas de aprender aunque el tema elegido no sea su especialidad. Ha sido un placer trabajar con ustedes a lo largo de este año.

También me gustaría dar las gracias a aquellos que han sufrido y me han dado su apoyo desde la cercanía. En especial a los minions y a María, nuestra madre, que tiró de nosotros cuando llegaban los malos momentos. Esos cafés. Los recordaré siempre.

Y especial agradecimiento a mi familia, que han sufrido tanto o más que yo desde la distancia, lo que tiene aún más mérito. Cuando me preguntan a quién admiro, no tengo que irme muy lejos para encontraros.

A mi padre. Nunca seré capaz de agradecerte todo lo que me has dado. A lo largo de estos años hemos ido de la mano. Horas, días e incluso semanas trabajando juntos. Sufriendo los malos momentos y disfrutando de los buenos, por pequeños que fueran. Estoy seguro de que los dos hemos crecido y aprendido mucho de ellos. Pisar donde tu pisaste. Vivir lo que tu viviste. Dicen que todo hijo desde muy pequeño generalmente por instinto y por el ejemplo, llega a tener un sentimiento especial hacia su padre, con el correr de los años siendo hijos deseamos llegar a ser como él, y muchas veces imitamos lo que él hace. Ojalá algún día tenga la oportunidad de ser un ejemplo para otra persona como tu lo has sido para mi.

# Capítulo 1

# Introducción

## 1.1. ¿Está la criptografía en riesgo?

Tan pronto como los ordenadores cuánticos sean asequibles, el algoritmo cuántico de Shor para factorizar y discretizar logaritmos ("Factoring and discrete logarithm") es capaz de romper algunos de los cifrados más conocidos tales como RSA, ElGamal o la criptografía de curvas elípticas en tiempo polinómico, es decir, todos ellos códigos basados en algoritmos de clave pública. Por ejemplo, un ordenador cuántico será capaz de obtener una clave privada RSA prácticamente al mismo tiempo que el usuario del algoritmo introduzca dicha clave.

Nos planteamos entonces cómo podremos solucionar este problema. ¿Existe algún algoritmo que se pueda implementar con los medios que disponemos actualmente y que sea resistente a ataques con ordenadores cuánticos?

El estudio de esta cuestión ha sido la principal motivación de este Trabajo de Fin de Grado. En él, se han abordado una serie de conceptos y problemas en un campo nuevo para el alumno. A lo largo del grado no he tenido la oportunidad de aprender y conocer la teoría de códigos correctores así como los códigos más conocidos y que actualmente siguen sin haber sido vulnerados. Por ello, el trabajo se centra en el estudio y análisis en profundidad de los códigos de Goppa así como su uso en algoritmos de recuperación de información como el algoritmo de McEliece, uno de los más usados en la actualidad y en el cuál están situadas las esperanzas para aguantar un posible ataque cuántico.

Para acompañar este estudio teórico-matemático se irán desarrollando ejemplos prácticos a lo largo del trabajo que servirán para facilitar el entendimiento de esta teoría. Estos ejemplos, así como los principales algoritmos del trabajo serán también implementados en varios tipos de lenguajes de programación utilizando algunos software especializados.

Uno de los paquetes de cálculo elegidos ha sido SageMath [S+09], un software matemático de código abierto bajo licencia GPL cuya misión es ser

una alternativa a Magma, Maple, Mathematica y Matlab, que no cuentan con una licencia libre. SageMath se construye a partir de una serie de paquetes como Maxima, GAP, PARI/GP, R y muchos más. Además, se trata de un lenguaje de programación basado en Python, que actualmente está entre los lenguajes más populares tanto a nivel académico como a nivel laboral debido a su potencia y versatilidad (esto además permite que SageMath haga uso de SciPy, numpy y otras librerías de python). Este trabajo además ha servido para tomar contacto con este lenguaje prácticamente desde cero así como para conocer paquetes que lo acompañan.

Otro de los paquetes de manipulación algebraica que he querido aprender y he querido reflejar en este trabajo ha sido GAP [GAP], "Groups, Algorithms, Programming" (Grupos, algoritmos y programación), un sistema para la computación de álgebra discreta. GAP está formado por una serie de librerías de funciones implementadas y también de librerías de datos con objetos algebraicos. Es un lenguaje muy común en la investigación y enseñanza de grupos y sus representaciones, anillos, vectores, álgebras y estructuras combinatorias, y que sin embargo, no había visto a lo largo del grado.

Todo el trabajo desarrollado, así como los ejemplos y algoritmos programados están disponibles en:

https://github.com/juanvelascogomez/correction-code-cryptography.

## 1.2. Primeros pasos en criptografía post-cuántica

La idea es buscar criptosistemas que podamos implementar en nuestros ordenadores (clásicos) y que sean capaces de mantenerse seguros aunque se usasen ordenadores cuánticos para tratar de vulnerarlos.

Una lista de posibles candidatos es la siguiente:

1. Algoritmos basados en redes o retículos.
2. Algoritmos basados en teoría de códigos.
3. Algoritmos basados en hashes.

Algunos de estos algoritmos resistentes a la computación cuántica están de hecho, basados en sistemas de cifrado de clave pública.

Daniel J. Bernstein (2009) dijo que estos sistemas supuestamente serían capaces de resistir los ordenadores cuánticos puesto que nadie aún había conseguido aplicar el algoritmo de Shor a ninguno de ellos.

A lo largo de este trabajo veremos algunos de esos algoritmos post-cuánticos como por ejemplo el criptosistema de tipo McEliece, uno de los algoritmos más antiguos de teoría de códigos basados en sistemas de clave pública que parece ser inmune al algoritmo de Shor.

Existen evidencias que demuestran la fuerza que tienen los criptosistemas de tipo McEliece para aguantar los ataques cuánticos puesto que han sido capaces de aguantar los ataques cuánticos basados en la transformación de Fourier que rompen RSA y El Gamal y sin embargo, no son aplicables a McEliece o a los criptosistemas Niederreiter debido a las propiedades algebraicas que tienen.

Estos dos tipos de criptosistemas se basan en los códigos Goppa, en el caso de criptosistemas basados en McEliece, se trata de códigos Goppa racionales mientras que en el caso de los criptosistemas basados en Niederreiter, utilizan códigos Goppa clásicos sin perder seguridad.

No está probado que estos dos criptosistemas sean capaces de aguantar cualquier tipo de ataque cuántico, pero si son capaces de aguantar algunos de los más poderosos que se han encontrado y que han sido capaces de romper otros algoritmos de clave pública antes mencionados.

Además, conviene destacar que el peligro de la computación cuántica reside en que se puede hacer la transformada discreta de Fourier mediante el algoritmo de Shor, que rompe la factorización de enteros y el logaritmo discreto. Es decir, el peligro está esencialmente en la criptografía de clave pública, no en la de clave privada como hemos comentado previamente.

La pregunta que puede surgir en estos momentos es la siguiente. "Si parece que el criptosistema de tipo McEliece está aguantando tan bien estos ataques, ¿por qué no estamos usándolo ya en vez de RSA?". Bueno, la respuesta tiene que ver con la eficiencia y el tamaño de las claves. Las claves de RSA están formadas por miles de bits en comparación con las claves de McEliece que podrían llegar a alcanzar el millón de bits. De hecho, los sistemas basados en curvas elípticas usan tamaño de llaves incluso más pequeños que RSA.

El proceso por el que suelen pasar los criptosistemas actualmente es el siguiente.

1. Los criptógrafos diseñan nuevos sistemas para cifrar y descifrar información. Es decir, se encargan de buscar la manera de cifrar, descifrar, firmar, y verificar información. De sus manos se crean los cifrados conocidos en la actualidad como DES, AES, RSA, ECDSA (Criptografía de curvas elípticas) o el cifrado de McEliece.

2. Una vez diseñados dichos sistemas, los criptoanalistas estudian las posibles vulnerabilidades o fallos que puedan tener dichos sistemas. Por ejemplo se preguntan, "¿Qué podría llegar a hacer un atacante si hiciese $2^m$ operaciones con un ordenador clásico?". Pues bien, este paso es el que se encarga de "romper" dichos criptosistemas.

3. Por último, los diseñadores de algoritmos en conjunto con estos criptoanalistas buscan la manera más eficiente (menor coste y mayor veloci-

dad) de “romper” dichos criptosistemas. Es decir, una vez descubierta la vulnerabilidad, se busca la manera más óptima de aprovecharla.

# Capítulo 2

# Objetivos

Se definieron una serie de objetivos a alcanzar a lo largo del Trabajo de Fin de Grado.

- ✓ Entender cómo funciona la teoría de códigos correctores.
- ✓ Establecer una buena base en cuanto a los códigos de Goppa y su importancia en la teoría de códigos correctores.
- ✓ Aprender algoritmos de corrección de errores basados en los códigos de Goppa como el algoritmo de Patterson. Además, llevar a la práctica estos conocimientos.
- ✓ Conocer el algoritmo de McEliece, qué es, conocer el cifrado y descifrado de mensajes utilizando este algoritmo.
- ✓ Seguridad de los criptosistemas de tipo McEliece. Posibles vulnerabilidades y ataques más conocidos. Ataque de Stern.
- ✓ Defensa del criptosistema de tipo McEliece.
- ✓ Aprender en profundidad el lenguaje de programación Python, uno de los lenguajes más populares en la actualidad.
- ✓ Utilizando estos conocimientos de Python, comprender el funcionamiento del software matemático de código abierto SageMath así como los paquetes que contiene.
- ✓ Aprender el sistema de computación de álgebra discreta GAP (Grupos, Algoritmos y Programación), muy usado en investigación y enseñanza para estudio de grupos y sus representaciones.

# Capítulo 3

# Códigos correctores de errores

## 3.1. Introducción

### 3.1.1. Teorías de detección y corrección de errores

Cuando tratamos con un enorme volumen de datos no es de extrañar que uno de los problemas más importantes que se genera durante la manipulación y la transmisión de ésta información sea el de los errores. De hecho, una pequeña variación o cambio en el soporte que contiene o transmite dicha información basta para que una parte del mensaje se corrompa, es decir, para que algunos de los ceros sean leídos como unos y viceversa en caso de que por ejemplo, se trate de un código binario.

Por ejemplo, si hablamos sobre CD-ROMS (discos compactos de sólo lectura), los cuales contienen gran cantidad de información cuando en ellos disponemos de archivos de música o vídeo, un rayón sobre dicho disco o una onda parásita podría implicar un gran número de errores en los archivos y mensajes que se transmiten. En este caso, esa lectura de ceros y unos podría producir una alteración grave en los bits.

¿Cómo detectar cuando se han producido dichos errores?

La teoría de códigos correctores de errores comenzó a desarrollarse en 1947 gracias a Richard Wesley Hamming (11 Febrero 1915 - 7 Enero 1998), de origen americano y que desarrolló además la teoría de códigos Hamming. Esta teoría hizo su primera aparición en el libro de Claude Shannon 'Una teoría matemática de la comunicación' [20] y pronto fue generalizada por Marcel J.E. Golay en su trabajo 'Signal Corps Engineering Laboratories at Fort Monmouth'. [8] Esta teoría matemática nació para el estudio de la calidad del sonido que se transmitía en las comunicaciones telefónicas, puesto que ésta era baja.

Una de las primeras teorías de códigos que se utilizó fue la basada en

iniciales, es decir, dada una palabra cualquiera, por ejemplo "Teoría", se codificaba como "Teléfono-Elefante-Oruga-Rayo-Iglesia-Amapola". Según han avanzado los años, la teoría de códigos correctos ha sufrido grandes cambios que hayan provocado una gran evolución en este campo.

Pese a estos cambios, la base de la teoría de códigos sigue siendo la misma que en un inicio. Estas teorías se basan en la inclusión de información y datos redundantes en un mensaje, el cual se divide en varias partes. De manera que es posible detectar los errores que se han producido y en caso de que dichos errores no sean excesivamente grandes o no superen un máximo, corregirlos.

Esto nos ayuda a mantener la integridad del mensaje, lo que es una gran ventaja para las comunicaciones, pero también tiene algunos inconvenientes. Una de ellas es el incremento en el tamaño de los paquetes que se envían debido a esa información redundante que hemos introducido.

La teoría de códigos correctores de errores es uno de los campos más extensos y más importantes en esa intersección entre las matemáticas y la informática. Se trata por tanto de una solución tecnológica con una gran base matemática que veremos a lo largo de este trabajo y que permite dar soluciones elegantes a la vez que eficientes y seguras para la seguridad de las comunicaciones.

La estrategia a seguir para controlar los errores es precisamente la opuesta a la compresión de datos, es decir, añadir redundancia en vez de quitarla, ya que esta redundancia será la que permita detectar y corregir dichos errores.

### 3.1.2. La información digital

**Definición 3.1** (Información digital)**.** *Sea $\mathcal{A}$ un conjunto finito y sea $\mathcal{A}^*$ el conjunto de secuencias finitas de elementos de $\mathcal{A}$. Se define información digital como una secuencia $m = x_1x_2 \dots x_k \in \mathcal{A}^*$.*

Este conjunto $\mathcal{A}$ se suele identificar con un sistema numérico, por ejemplo, $\{0, 1\}$, conjunto que interpretamos como el cuerpo finito de números $\mathbb{F}_2$. Si $\mathcal{A}$ contiene $q$ elementos y $q$ es potencia de un número primo entonces, $\mathcal{A}$ se identifica con el cuerpo finito con $q$ elementos $\mathbb{F}_q$.

Se trata de una identificación muy importante puesto que permite aplicar a los problemas de codificación todo el álgebra y la geometría relacionada con los cuerpos finitos.

**Observación 3.1.** *A lo largo del proyecto trabajaremos sobre un cuerpo finito puesto que será necesario para poder definir más adelante los códigos lineales, con una estructura algebraica. Sin embargo, que sea un cuerpo finito no es tan importante desde el punto de vista de la información puesto que lo relevante es el número de elementos que contiene el alfabeto donde trabajamos.*

### Transmisión de la información digital

El siguiente paso, una vez que tenemos esa información de forma digital, es transmitirla o enviarla al receptor. El esquema de transmisión que se sigue es el siguiente.


Figura 3.1: Esquema de transmisión de la información

Pero este esquema de transmisión tiene algunos problemas. Uno de ellos, el más importante, es que el receptor no puede estar seguro de que alguna parte del mensaje haya sido corrompida durante la transmisión. Lo que si puede es conocer la frecuencia con que se producen los errores y por tanto, determinar el número de errores, en media, que espera hayan podido suceder. Por tanto, haciendo uso de la teoría de códigos correctores de errores, codificaríamos dicha información añadiéndole cierta redundancia con arreglo a una serie de reglas establecidas.

Ahora, el esquema de transmisión cambia puesto que estamos codificando dicha información antes del envío y descodificándola una vez recibida por parte del receptor. Los procesos que estamos llevando a cabo reciben el nombre de *codificación* y *descodificación*.

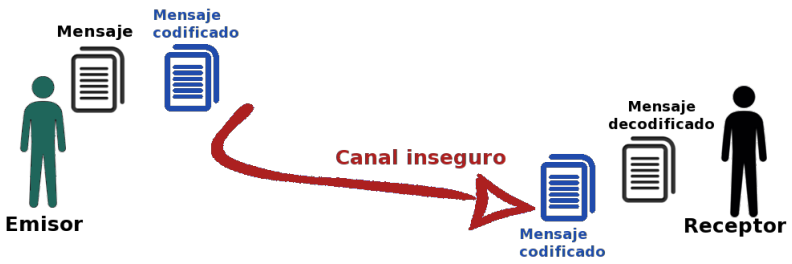


Figura 3.2: Esquema de transmisión de la información codificada

### 3.1.3. Códigos correctores

Hemos definido lo que se entiende por información digital, es decir, una secuencia $m \in \mathcal{A}^*$. Dada esa secuencia, podemos fijar dos números enteros $k$ y $n$ tales que cuando dividimos en $m$ bloques de longitud $k$ esa secuencia, tenemos:

$$m = (x_1 \dots x_k)(x_{k+1} \dots x_{2k}) \dots$$

Cada uno de estos bloques de información en los que estamos dividiendo el mensaje $m$ se codifica como su imagen mediante una aplicación inyectiva $c : \mathcal{A}^k \longrightarrow \mathcal{A}^n$.

Por tanto, una vez codificada cada una de las partes en las que hemos dividido el mensaje, la codificación del mensaje completo se obtendrá concatenando la codificación de cada una de las partes o bloques que lo forman, es decir:

$$c(m) = c(x_1 \dots x_k)c(x_{k+1} \dots x_{2k}) \dots \tag{3.1}$$

Donde $c(m)$ se refiere al mensaje codificado y $c(x_1 \dots x_k)$ al bloque codificado de longitud $n$. El *código* que utilizaremos será el conjunto $\mathcal{C} = \mathrm{Im}(c)$.

**Definición 3.2** (Código corrector de errores)**.** *Un código corrector de errores es un subconjunto $\mathcal{C} \subseteq \mathcal{A}^n$, siendo $\mathcal{A}$ un conjunto finito y $n$ un número entero positivo.*

**Definición 3.3** (Palabra de un código)**.** *Los elementos de $\mathcal{C}$ se llaman palabras y $n$ es su longitud.*

Cada palabra del conjunto $\mathcal{C}$ contiene $k$ símbolos de información y $n-k$ símbolos redundantes. El número $k/n$ recibe el nombre de *tasa de transmisión* del código $\mathcal{C}$ y representa el número de símbolos de información que tenía el mensaje sin codificar con respecto al total de símbolos que tiene el mensaje codificado. Ambos números están relacionados de una manera sencilla, se observa que a mayor tasa de transmisión del código la redundancia será menor.

Sea $c \in \mathcal{C}$, $x \in \mathcal{A}^n$ y $p$ la probabilidad de que un símbolo resulte alterado en la transmisión. Supongamos una media de $np$ símbolos erróneos en $x$.

La capacidad de corrección de errores de $\mathcal{C}$ debe superar al menos esa cota.

- Si $x \notin \mathcal{C}$, entonces se han encontrado errores.
- Si $x \in \mathcal{C}$, entonces nunca podremos estar realmente seguros de que hayan existido errores.

El código que hemos definido hará que resulte muy improbable que $x \in \mathcal{C}$ por los errores aleatorios que se sufren en el canal de la transmisión.

Para entender mejor esto, veamos un ejemplo.

*Ejemplo* 3.1. Supongamos el alfabeto $A = \mathbb{Z}_2 = 0, 1$ y consideramos el código $\mathcal{C}_3 = \{000, 111\}$, bien definido y formado por la repetición de los símbolos del alfabeto tres veces. Además, consideraremos que el canal por el que se transmiten los mensajes es seguro y fiable, entonces pueden ocurrir dos casos.

1. Mensaje recibido correctamente: se recibe el mensaje 000 ó 111, que se interpretará como 0 ó 1 respectivamente.

2. Mensaje con errores: se recibe el mensaje

   *a*) 001, 010 o 001, que se interpreta como el símbolo 0;

   *b*) 110, 101 o 011, que se interpreta como el símbolo 1.

Utilizando el código $\mathcal{C}_3 = \{000, 111\}$ no podemos determinar si hay dos errores puesto que no seremos capaces de distinguir cuál es el mensaje original y por tanto este código de error solamente será capaz de detectar un error.

Si recibimos el mensaje 110, puede provenir de dos errores en 000 o de un error en 111.

Por tanto, si solamente se ha producido un error, nuestro código será capaz de corregirlo y darnos la solución correcta al mensaje, pero en caso de que se produzcan más errores, se producen resultados no deseados.

Podemos asegurar un cierto grado de fiabilidad en la transmisión de un mensaje utilizando códigos que detecten y corrijan una cantidad de errores acorde a la tasa de error del canal. Es decir, debemos buscar un código que para pasar de una palabra a otra haya que modificar muchas posiciones puesto que será capaz de detectar y corregir una mayor cantidad de errores.

La distancia de Hamming, que definiremos a continuación, cuenta en cuántas posiciones se diferencian dos palabras para proporcionar información sobre el número de errores que un código puede detectar y corregir.

**Definición 3.4** (Distancia de Hamming)**.** *Dados dos elementos $x, y \in \mathcal{A}^n$, llamamos distancia de Hamming entre $x$ e $y$ al número de coordenadas distintas que poseen, es decir:*

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

*Ejemplo* 3.2. Si $A$ es el alfabeto español de letras y $A^{10}$ es el conjunto de todas las cadenas de diez letras, la distancia de Hamming entre 'matemático' y 'periodista' es máxima, puesto que no coincide ninguna de sus letras. Es decir,

$$\mathrm{d}(matematico, periodista) = 10.$$

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

**Propiedades 3.1.** *La distancia de Hamming d cumple las siguientes propiedades.*

1. *Para todo* $x, y \in A^n$, $\mathrm{d}(x, y) \geq 0$.
2. *Para todo* $x, y \in A^n$, $\mathrm{d}(x, y) = 0$ *si y solo si* $x = y$.
3. *Para todo* $x, y \in A^n$, $\mathrm{d}(x, y) = \mathrm{d}(y, x)$.
4. *Para todo* $x, y, z \in A^n$, $\mathrm{d}(x, z) \leq \mathrm{d}(x, y) + \mathrm{d}(y, z)$.

*Las tres primeras propiedades son la propia definición de distancia y la última corresponde a la propiedad triangular.*

*Demostración.* Las tres primeras propiedades se deducen directamente de la definición de distancia como hemos comentado. La desigualdad triangular se demuestra analizando posición a posición.

1. En las posiciones donde coinciden $x$ e $z$ no se aportan nada a las distancias entre ellas, que es el mínimo valor posible.
2. Supongamos que en la posición $i$ se cumple que $x_i$ e $z_i$ son distintos, entonces se añade una unidad a la distancia $\mathrm{d}(x, z)$. El elemento $y_i$ no puede ser ambos al mismo tiempo porque acabamos de ver que son distintos, por tanto tenemos dos opciones. La primera es suponer que pudiese ser uno de ellos en concreto, entonces aportaría una unidad a la suma $\mathrm{d}(x, y) + \mathrm{d}(y, z)$. La segunda opción es que no coincida con ninguno de ellos, en ese caso aporta dos unidades a la suma anterior.

Las $n$ posiciones de $\mathrm{d}(x, y) + \mathrm{d}(y, z)$ aportan más unidades que las de $\mathrm{d}(x, z)$ por tanto se cumple la desigualdad triangular. □

### 3.1.4. Distancia mínima. Detección y corrección de errores

**Definición 3.5** (Distancia mínima)**.** *Se define como distancia mínima a la capacidad de corrección de errores de* $\mathcal{C}$*, es decir:*

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

Vamos a definir la distancia de Hamming utilizando SageMath. Para ello definimos primero lo que significa distancia entre dos palabras o elementos

```
def dv(x,y):
    distancia = 0
    for i in range(len(x)):
        if x[i] != y[i]:
            distancia = distancia + 1
    return distancia
```

También podemos definir la distancia entre los elementos de un código $\mathcal{C}$.

```
def dc(C):
    distancias = set([dv(x,y) for x in C for y in C if x!=y])
    return min(distancias)
```

Sin embargo, esta versión no aprovecha la simetría de la distancia. Veamos una nueva versión que hace uso de ella para simplificar los cálculos.

```
def dc_simetrica(C):
    distancia_minima = oo
    for i in range(len(C)):
        for j in range(i+1,len(C)):
            distancia = dv(C[i],C[j])
            if dv(C[i],C[j])<distancia_minima:
                distancia_minima=distancia
    return(distancia_minima)
```

*Ejemplo* 3.3. Dado el código $\mathcal{C}_3 = \{000, 111\}$, su distancia mínima será $d(\mathcal{C}_3) = 3$, que es la menor distancia Hamming entre dos palabras distintas del código.

Siguiendo el ejemplo en Sage, definimos las dos palabras del código:

```
sage: a = vector([0,0,0])
sage: b = vector([1,1,1])
```

Definimos el código con esos vectores y calculamos su distancia mínima de Hamming.

```
sage: C3 = (a,b)
sage: dv(a,b)
3
```

Y obtenemos como salida 3. Que era el resultado esperado. De igual manera, veamos este ejemplo en GAP [GAP] con [GUAVA].

```
gap> c1:=Codeword(000);
[ 0 0 0 ]
gap> c2:=Codeword("111");
[ 1 1 1 ]
gap> DistanceCodeword(c1,c2);
3
```

Con estas dos palabras codificadas podemos definir un código y tratar de calcular la distancia utilizando los métodos que nos proporciona el paquete GUAVA de GAP.

```
gap> C:=ElementsCode([c1,c2],GF(2));
a (3,2,1..3)1 user defined unrestricted code over GF(2)
gap> IsLinearCode(C);
true
gap> MinimumDistance(C);
3
```

*Ejemplo* 3.4. Dado el código $\mathcal{C}_0 = \{000, 011, 101, 110\}$, su distancia mínima será $\mathrm{d}(\mathcal{C}_0) = 2$.

Definimos en SageMath el código $\mathcal{C}_0$ y calculamos la distancia mínima.

```
sage: C0 = (vector([0,0,0]),vector([0,1,1]),vector([1,0,1]),vector([1,1,0]))
sage: dc(C0)
2
```

Y en GAP.

```
gap> C:=ElementsCode(["000","011","101","110"],GF(2));
a (3,2,1..3)1 user defined unrestricted code over GF(2)
gap> MinimumDistance(C);
2
```

*Ejemplo* 3.5. Dado el código $\mathcal{C}_1 = \{000, 011, 101, 110, 111, 001\}$, su distancia mínima será $\mathrm{d}(\mathcal{C}_1) = 1$.

Definimos en SageMath el código $\mathcal{C}_1$ y calculamos la distancia mínima con la función `dc` descrita arriba.

```
sage: C1 = (vector([0,0,0]),vector([0,1,1]),vector([1,0,1]),vector([1,1,0]),vector
    ([1,1,1],vector([0,0,1]))
sage: dc(C1)
1
```

Y con GAP:

```
gap> C:=ElementsCode(["000","011","101","110","111", "001"],GF(2));
a (3,2,1..3)1 user defined unrestricted code over GF(2)
gap> MinimumDistance(C);
1
```

**Proposición 3.1.** *Sea $\mathcal{C}$ un código con distancia mínima $d(\mathcal{C})$, entonces $\mathcal{C}$ puede detectar hasta $k$ errores en cada palabra si y solo si $d(\mathcal{C}) \geq k + 1$.*

*Demostración. Necesidad.* Si $\mathcal{C}$ detecta cualquier configuración de hasta $k$ errores, si suponemos por reducción al absurdo que $k \geq d$, y se consideran dos palabras código $c$, $y$ a distancia exactamente $d$, si se emite $c$ y se recibe $y$ estamos suponiendo que esta configuración de $d \leq k$ errores se detecta, pero sin embargo $y \in \mathcal{C}$, con lo que la palabra $y$ la damos por buena y los errores no se detectan.

*Suficiencia.* Si el número de errores es $k \leq \mathrm{d} - 1$, emitimos $c$ y recibimos $y \neq c$ , entonces $\mathrm{d}(c, y) = k < \mathrm{d}(\mathcal{C})$ y por definición de distancia mínima tenemos $\mathrm{y} \notin \mathcal{C}$, con lo que los errores son detectados. □

**Proposición 3.2.** *Sea $\mathcal{C}$ un código con distancia mínima $d(\mathcal{C})$, entonces $\mathcal{C}$ puede corregir hasta $k$ errores en cada palabra si y solo si $d(\mathcal{C}) \geq 2k + 1$.*

*Demostración.* [17]

Sea $\mathcal{C}$ un código que puede corregir hasta $k$ errores en cada palabra $\mathrm{d}(\mathcal{C}) < 2k+1$. Tomemos $c$ y $c'$ elementos del código a distancia d, y supongamos que se emite $c$. Sea $x$ el resultado de cambiar al elemento $c$, $k$ de los $d$ símbolos en los que $c$ difiere de $c'$ por los correspondientes símbolos de $c'$. Entonces es fácil comprobar que $\mathrm{d}(c', x) < \mathrm{d}(c, x)$ con lo que la decodificación por mínima distancia devuelve $c'$ en vez de $c$.

Una vez vista esta implicación, vamos a ver la implicación de derecha a izquierda. Ahora estamos suponiendo que $d(\mathcal{C}) \leq 2k+1$ y vamos a demostrar que $\mathcal{C}$ puede corregir hasta $k$ errores en cada palabra.

Supongamos que se emite $c$ y se recibe $y$ con $\mathrm{d}(c, y) \leq k$ Se $c' \,!= \, c$ otra palabra de $\mathcal{C}$; como $\mathrm{d}(c, c') \geq \mathrm{d}(\mathcal{C}) \geq 2k+1$ y $\mathrm{d}(c, y) \leq k$, aplicando la desigualdad triangular a $c$, $y$, y $c'$ se obtiene que $\mathrm{d}(c', y) > k \geq \mathrm{d}(c, y)$, con lo que la decodificación por la palabra más próxima solo puede devolver la palabra correcta c. □

Vamos a ver esta proposición aplicada a una serie de ejemplos que estamos manejando a lo largo de la teoría.

*Ejemplo* 3.6. Sea el código $\mathcal{C} = \{0, 1\}$, puesto que su distancia mínima es $\mathrm{d}(\mathcal{C}) = 1$, aplicando la proposición anterior podemos deducir que $k = 0$, por tanto el código no corrige ningún error.

Sea el código $\mathcal{C}_3 = \{000, 111\}$, puesto que su distancia mínima es $\mathrm{d}(\mathcal{C}_3) = 3 = 2 \times 1 + 1$, aplicando la proposición anterior podemos deducir que $k = 1$, por tanto el código corregirá un error como máximo.

Sea el código $\mathcal{C}_0 = \{000, 011, 101, 110\}$, puesto que su distancia mínima es $\mathrm{d}(\mathcal{C}_0) = 2 = 1 + 1$, aplicando la proposición anterior podemos deducir que $k = 0$, por tanto el código no corrige ningún error.

## 3.2. Códigos lineales

### 3.2.1. Introducción a códigos lineales

Consideramos que el conjunto que estamos utilizando tiene como cardinal $q$, la potencia de un número primo e identificamos ese conjunto con $\mathbb{F}_q$, el cuerpo finito con $q$ elementos.

*Ejemplo* 3.7 (Código lineal). Vamos a ver un primer ejemplo de código binario lineal $(7, 4)$ que el ingeniero Richard Hamming diseño en 1947 para reducir el número de errores que se producían en los ordenadores de los laboratorios Bell. Este código binario lineal se denota por $H_7$ y sirve para detectar y corregir un error en una cadena de 7 bits de los que 4 de ellos son de información y los otros 3 restantes son bits redundantes. Puesto que se trata de un código binario, la codificación, detección de errores, corrección y decodificación son procesos más simples. Queremos saber si se trata de un buen código, para ello estudiaremos si ese código o conjunto de Hamming tiene alguna estructura algebraica.

En el caso de la codificación, por ejemplo, el código de Hamming $H_7$ protege la información convirtiendo una cadena de 4 bits $v = v_1v_2v_3v_4$ en una cadena de 7 bits $u = u_1u_2u_3u_4u_5u_6u_7$ añadiendo 3 bits de redundancia y colocando los bits iniciales en posiciones predeterminadas según las siguientes reglas:

- Los bits $v_1, v_2, v_3 y v_4$ serán los bits $u_3, u_5, u_6 y u_7$ de la nueva cadena respectivamente.
- Los bits $u_1, u_2 y u_4$ se calculan sumando en binario los bits ya conocidos de la siguiente manera:

$$u_1 = v_1 + v_2 + v_4 u_2 = v_1 + v_3 + v_4 u_4 = v_2 + v_3 + v_4$$

Es decir, podemos definir una aplicación $g : \mathbb{Z}_2^4 \longrightarrow \mathbb{Z}_2^7$ tal que:

$$g(v_1, v_2, v_3, v_4) = (v_1 + v_2 + v_4, v_1 + v_3 + v_4, a_1, v_2 + v_3 + v_4, v_2, v_3, v_4)$$

Que se trata de una aplicación lineal entre dos espacios vectoriales sobre el cuerpo $\mathbb{Z}_2$.

Como el procedimiento de codificación que acabamos de ver se aplican a cualquier entrada de 4 bits, las palabras del código de Hamming $H_7$,

000000,1101001,0101010,1000011,1001100,0100101,
1100110,0001111,1110000,0011001,1011010,0110011,
0111100,1010101,0010110,1111111,

interpretadas como vectores son los elementos de la imagen de $g$, es decir: $H_7 = \mathrm{Im}(g)$.

La dimensión de la imagen de $g$ es 4 porque la dimensión del espacio de partida es 4 y la dimensión del núcleo es 0, esto se comprueba viendo que la aplicación lineal $g$ es inyectiva ya que si $(a_1, a_2, a_3, a_4) \neq (a_1', a_2', a_3', a_4')$ entonces $g(a_1, a_2, a_3, a_4) \neq g(a_1', a_2', a_3', a_4')$, luego $\mathrm{Ker}(g) = \{(0, 0, 0, 0)\}$.

Por tanto:

$$\dim(\mathbb{Z}_2^4) = \dim(\mathrm{Ker}(g)) + \dim(\mathrm{Im}(g)),$$

por lo que el cardinal de la imagen de $g$ es $2^4 = 16$ que coincide con el número de palabras del código de Hamming. Por tanto podemos escribir el subespacio imagen de $g$ de la siguiente forma:

$$\mathrm{Im}(g) = \{(u_1, u_2, u_3, u_4, u_5, u_6, u_7) \in \mathbb{Z}_2^7 : u_4 + u_5 + u_6 + u_7 = 0,$$
$$u_2 + u_3 + u_6 + u_7 = 0, u_1 + u_3 + u_5 + u_7 = 0\}.$$

**Definición 3.6** (Código lineal)**.** *Llamamos código lineal al subespacio vectorial $\mathcal{C} \subseteq \mathbb{F}_q^n$. con $q$ una potencia de un primo. Además, $n$ y $k$ reciben el nombre de longitud y dimensión del código. En ese caso, el código tendrá $q^k$ palabras de longitud $n$. Además, un código lineal es binario si $q = 2$.*

**Notación 3.1.** *Notaremos por* $(n, k)$ *al tipo de código lineal de longitud* $n$ *y dimensión* $k$*.*

**Notación 3.2.** *Cuando denotamos un código con la pareja* $(n, k)$*, nos referiremos a él como el código de longitud* $n$ *y de dimensión* $k$*. En caso de denotarlo por* $(n, k, \mathrm{d})$*, llamaremos distancia mínima del código a* $\mathrm{d}$*.*

*Ejemplo* 3.8. El código $\mathcal{C} = \{000, 011, 101, 110\}$ es un código lineal $(3, 2)$, es decir, un código lineal de longitud 3 y de dimensión 2. Esto se debe a que $C = \mathrm{Im}(f)$ siendo $f : \mathbb{Z}_2^2 \to \mathbb{Z}_2^3$ la aplicación lineal tal que $f(b_1, b_2) = (b_1, b_2, b_1 + b_2)$.

```
gap> C:=ElementsCode([[0,0,0],[0,1,1],[1,0,1],[1,1,0]],GF(2));
a (3,4,1..3)1 user defined unrestricted code over GF(2)
gap> IsLinearCode(C);
true
```

*Ejemplo* 3.9. El código binario de repetición $R_5^2 = \{00000, 11111\}$ es un código lineal $(5, 1)$ porque $R_5^2$ es un subespacio vectorial de $\mathbb{Z}_2^5$, ya que la suma de dos palabras de $R_5^2$ es otra palabra de $R_5^2$.

*Ejemplo* 3.10. Vamos a obtener una base del código lineal de Hamming $(7, 4)$, definido por $\mathcal{C} = \{x \in \mathbb{Z}_2^7 \mid Hx = 0\}$, siendo

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

y calcular cuántas palabras distintas tiene este código.

Dado que $\mathrm{rango}(H) = 3$, entonces la dimensión del subespacio vectorial $\mathcal{C}$ es 4. Para obtener una base del espacio solución se hacen operaciones elementales en las filas de la matriz teniendo en cuenta que todas las operaciones se realizan módulo 2.

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \in C \Leftrightarrow \begin{cases} x_1 = \alpha + \beta + \delta \\ x_2 = \alpha + \gamma + \delta \\ x_3 = \alpha \\ x_4 = \alpha + \gamma + \delta \\ x_5 = \beta \\ x_6 = \gamma \\ x_7 = \delta \end{cases}$$

$$\Leftrightarrow x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} = \alpha \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} + \gamma \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \delta \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Por tanto, una base de $\mathcal{C}$ es:

$$B_C = \left\{ \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \right\}.$$

Las combinaciones lineales de estos cuatro elementos con los coeficientes 0 y 1 son $2^4 = 16$, que es el número total de palabras código.

Lo siguiente que veremos será cómo los procesos de codificación y decodificación, así como el cálculo de la distancia mínima, son mucho más simples para los códigos lineales que para aquellos que no lo son.

### 3.2.2. Matriz generatriz

**Definición 3.7** (Matriz generatriz de un código)**.** *Llamamos matriz generatriz de un código $\mathcal{C}$ a cualquier matriz de una aplicación lineal biyectiva,*

$$C : \mathbb{F}_q^k \to \mathcal{C} \subset \mathbb{F}_q^n,$$

*cuyas filas son una base del código $\mathcal{C}$.*

*Es decir, dada una base $\{c_1, \ldots, c_n\}$ de un código $\mathcal{C}$, una matriz generatriz será:*

$$G = \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,k-1} & c_{1,k} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,k-1} & c_{2,k} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{n-1,1} & c_{n-1,2} & \cdots & c_{n-1,k-1} & c_{n-1,k} \\ c_{n,1} & c_{n,2} & \cdots & c_{n,k-1} & c_{n,k} \end{pmatrix}.$$

*Ejemplo* 3.11. El conjunto $B = \{101, 011\}$ es una base del código lineal $\mathcal{C} = \{000, 011, 101, 110\}$ que hemos utilizado anteriormente en el Ejemplo 3.8. Por tanto,

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

es una matriz generatriz del código $\mathcal{C}$. Podemos observar que la matriz $G$ es la matriz de la aplicación lineal de $f(b_1, b_2) = (b_1, b_2, b_1 + b_2)$. Cualquier palabra puede obtenerse por tanto multiplicando la matriz $G$ por el par adecuado,

$$(b_1, b_2) \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} = (b_1 \;\; b_2 \;\; b_1 + b_2).$$

Por tanto, conociendo una matriz generatriz y un bloque del mensaje entrante es inmediato calcular la palabra código asociada a ese bloque. Basta pues almacenar la matriz generatriz en lugar de todas las palabras del código junto con su bloque relacionado, evitando así almacenar tanta información. Este hecho nos vendrá especialmente bien cuando estemos tratando con códigos cuyos valores de $n$ y $k$ sean realmente altos.

Al igual que vimos en la definición de código lineal, vamos a ver este mismo ejemplo en GAP para tratar de calcular la matriz generatriz.

```
gap> C:=ElementsCode([[0,0,0],[0,1,1],[1,0,1],[1,1,0]],GF(2));
a (3,4,1..3)1 user defined unrestricted code over GF(2)
gap> G:=GeneratorMat(C);
[ [ 0*Z(2), Z(2)^0, Z(2)^0 ], [ Z(2)^0, 0*Z(2), Z(2)^0 ] ]
gap> List(GeneratorMat(C),Codeword);
[ [ 0 1 1 ], [ 1 0 1 ] ]
```

GAP utiliza la notación $Z(p^k)$, con $p$ un primo, para referirse a un generador del grupo cíclico $\mathbb{F}_{p^k} \setminus \{0\}$.

Por tanto, para codificar una palabra tendremos que multiplicar el vector que queramos por la matriz generatriz. Es decir:

```
gap> Codeword([0,1]*G);
[ 1 0 1 ]
```

*Ejemplo* 3.12. El código de Hamming definido por $\mathcal{C} = \{x \in \mathbb{Z}_2^7 \mid Hx = 0\}$ siendo

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

es un código lineal $(7, 4)$ ya que es un subespacio vectorial de dimensión 4 de $\mathbb{Z}_2^7$, por ello, el código de Hamming tiene $2^4$ palabras de longitud 7. Su matriz generatriz es

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

cuyas filas se corresponden con los 4 elementos de la base del código de Hamming que hemos calculado en el ejemplo 3.10.

### 3.2.3. Códigos lineales equivalentes

**Definición 3.8** (Códigos equivalentes)**.** *Sean $\mathcal{C}_1$ y $\mathcal{C}_2$ dos códigos de misma longitud $n$ sobre $\mathbb{F}_q$, son equivalentes si:*

1. *O bien existe una permutación $\sigma$ del conjunto $\{1, \ldots, n\}$ tal que $C_2 = \{(c_{\sigma(1)}, \ldots, c_{\sigma(n)}) \mid (c_1, \ldots, c_n) \in C\}$.*

2. *O bien existe una secuencia de transformaciones multiplicativas entre los símbolos de una posición corregida por un escalar distinto de cero en* $\mathbb{F}_q$.

*En cualquiera de los dos casos se aplican las transformaciones en todos elementos del código para obtener un nuevo código.*

Nótese que si el código es binario, entonces sólo la primera de las condiciones en la definición anterior es relevante.

**Teorema 3.1.** *Dos matrices* $k \times n$ *generan códigos equivalentes si se puede obtener una a partir de la otra a partir de las siguientes operaciones.*

1. *Permutación de las filas.*
2. *Multiplicación de una fila por un escalar distinto de cero.*
3. *Añadiendo un múltiplo escalar de una de las columnas en otra.*
4. *Permutación de las columnas.*
5. *Multiplicación de cualquier columna arreglada por un escalar no nulo.*

*Demostración.* Los tres primeros tipos de operación consisten en un cambio de base del código $\mathcal{C}$, y el código $\mathcal{C}$ queda invariante ante estos cambios. De hecho, la primera propiedad reordena la base. Los dos últimos tipos de operaciones son las operaciones matriciales de la propia definición de códigos equivalentes. □

**Observación 3.2.** *Dados dos códigos* $\mathcal{C}_1$ *y* $\mathcal{C}_2$*, si los códigos son equivalentes entonces sus valores de* $n, k, d$ *coinciden.*

*Ejemplo* 3.13. Sea el código Hamming $H_7$. La matriz

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

es una matriz generatriz de $H_7$.

Si permutamos la matriz intercambiando la columna tres por la columna cuatro, se obtiene la matriz:

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

que genera un código lineal $(n, k) = (7, 4)$ con las mismas propiedades que el código Hamming $H_7$ pero con distintas palabras:

$$0000000, 1110001, 0110010, 1000011, 1010100, 0100101, 1100110, 0010111,$$

$$1101000, 0011001, 1011010, 0101011, 0111100, 1001101, 0001110, 1111111.$$

Podemos ver que la palabra 1110001 no pertenece a $H_7$.

Sin embargo, el código generado por la matriz $G'$ y el código $H_7$ son equivalentes puesto que el número de palabras, longitud de cada una de ellas, la distancia mínima y demás características de este nuevo código coinciden con las de $H_7$.

Veamos este mismo ejemplo desarrollado en GAP.

```
gap> C:=HammingCode(3);
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> G:=GeneratorMat(C);
gap> List(G,Codeword);
[ [ 1 1 1 0 0 0 0 ], [ 1 0 0 1 1 0 0 ], [ 0 1 0 1 0 1 0 ], [ 1 1 0 1 0 0 1 ] ]
gap> Dimension(C);
4
gap> MinimumDistance(C);
3
gap> WordLength(C);
7
gap> Redundancy(C);
3
gap> AsSortedList(C);
[ [ 0 0 0 0 0 0 0 ], [ 0 0 0 1 1 1 1 ], [ 0 0 1 0 1 1 0 ], [ 0 0 1 1 0 0 1 ],
  [ 0 1 0 0 1 0 1 ], [ 0 1 0 1 0 1 0 ], [ 0 1 1 0 0 1 1 ], [ 0 1 1 1 1 0 0 ],
  [ 1 0 0 0 0 1 1 ], [ 1 0 0 1 1 0 0 ], [ 1 0 1 0 1 0 1 ], [ 1 0 1 1 0 1 0 ],
  [ 1 1 0 0 1 1 0 ], [ 1 1 0 1 0 0 1 ], [ 1 1 1 0 0 0 0 ], [ 1 1 1 1 1 1 1 ] ]
```

Dado un código $\mathcal{C}$, puesto que la codificación no tiene por qué ser única, tampoco lo será la matriz generatriz, es decir, dado un código de la forma $\mathcal{C} = \{aG \mid a \in \mathbb{F}_q^k\}$ donde $G$ es una matriz generatriz cualquiera, entonces un mensaje $a \in \mathbb{F}_q^k$ se codificará como $aG \in \mathbb{F}_q^n$.

Una vez codificada la palabra $a \in \mathbb{F}_q^k$ mediante un código lineal, en algunos casos interesa tener la propia palabra $a$ al principio como una subpalabra puesto que así, los $k$ primeros símbolos de la palabra contienen la información y el resto se consideran símbolos de control. Este tipo de codificación recibe el nombre de codificación sistemática.

### 3.2.4. Códigos lineales sistemáticos

**Definición 3.9** (Códigos lineales sistemáticos)**.** *Un código lineal $\mathcal{C}$ es sistemático si tiene una matriz generatriz de la forma $(R|I)$, con $I$ la matriz identidad. En estos códigos, al codificar un bloque $a$, obtenemos una palabra código que contiene una copia de $a$ en los últimos bits.*

**Observación 3.3.** *Conviene resaltar que los símbolos de información pueden estar al final o al principio de la palabra. En general, se podría definir codificación sistemática en k posiciones prefijadas.*

*Ejemplo* 3.14. El código equivalente al código $H_7$ del ejemplo anterior es sistemático mientras que el código Hamming $H_7$ no lo es.

**Proposición 3.3.** *Todo código binario lineal es equivalente a un código sistemático.*

*Demostración.* Sea $\mathcal{C}$ un código lineal $(n, k)$. Basta calcular la forma normal de Hermite por filas de una matriz generatriz de $\mathcal{C}$. Mediante permutaciones por columnas podemos obtener una matriz de la forma $(R|I)$. Por el Teorema 3.1, las matrices son equivalentes y por tanto, por la Definición 3.8 el código asociado a esta matriz es equivalente a $\mathcal{C}$. □

*Ejemplo* 3.15. Sea la matriz

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

la matriz generatriz de un código lineal $(6, 3)$ y que no está expresada de forma sistemática.

Veamos que es equivalente a una sistemática. En efecto, aplicamos operaciones sobre las filas de la matriz,

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix} \overset{f_2+f_3}{\equiv} \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$
$$\overset{f_1+f_2}{\equiv} \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$
$$\overset{c_3/c_4}{\equiv} \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix},$$

y se obtiene una matriz sistemática que genera el código $(6, 3)$ sistemático que además es equivalente como hemos visto mediante transformaciones elementales entre filas al código lineal generado por la matriz inicial $G$.

### 3.2.5. Matriz de control

**Definición 3.10** (Matriz de control)**.** *Diremos que una matriz H de rango $n - k$ es una matriz de control del código $\mathcal{C}$ si:*

*para todo* $x \in \mathbb{F}_q^n$ *se verifica que* $x \in \mathcal{C}$ *si y solo si,* $Hx^t = 0$.

*En ese caso, si* $\mathcal{C}$ *es de tipo* $(n, k)$*, entonces* $H$ *es de tamaño* $(n-k) \times n$ *y rango* $n - k$.

*Ejemplo* 3.16. Calcular la matriz de control del código $\mathcal{C} = \{1110000, 1001100, 0101010, 1101001\}$.

```
gap> m:=[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]];
gap> C:=GeneratorMatCode(m,GF(2));
a linear [7,4,1..3]1 code defined by generator matrix over GF(2)
gap> Elements(C);
[ [ 0 0 0 0 0 0 0 ], [ 0 0 0 1 1 1 1 ], [ 0 0 1 0 1 1 0 ], [ 0 0 1 1 0 0 1 ], [ 0 1 0 0
    1 0 1 ], [ 0 1 0 1 0 1 0 ], [ 0 1 1 0 0 1 1 ], [ 0 1 1 1 1 0 0 ], [ 1 0 0 0 0 1 1
    ], [ 1 0 0 1 1 0 0 ], [ 1 0 1 0 1 0 1 ], [ 1 0 1 1 0 1 0 ], [ 1 1 0 0 1 1 0 ], [ 1 1
     0 1 0 0 1 ], [ 1 1 1 0 0 0 0 ], [ 1 1 1 1 1 1 1 ] ]
gap> Dimension(C);
4
gap> H:=CheckMat(C);
[ <an immutable GF2 vector of length 7>, <an immutable GF2 vector of
    length 7>, <an immutable GF2 vector of length 7> ]
gap> List(H, Codeword);
[ [ 0 1 1 1 1 0 0 ], [ 1 0 1 1 0 1 0 ], [ 1 1 0 1 0 0 1 ] ]
gap> c:=Elements(C)[2];
[ 0 0 0 1 1 1 1 ]
gap>H*c;
[ 0 0 0 ]
```

La siguiente proposición nos da la relación que tienen dos matrices generatriz y de control de un código $\mathcal{C}$.

**Proposición 3.4.** *Dadas dos matrices generatriz y de control de un código* $\mathcal{C}$*,* $G$ *y* $H$*, entonces*

$$GH^t = 0.$$

Una vez vista esta relación entre dos matrices generatriz y de control de un código $\mathcal{C}$, vamos a pasar a ver la relación que existe entre esta matriz de control y la distancia mínima de un código.

Para ello, lo que vamos a hacer es definir el concepto de soporte de un elemento de $\mathbb{F}_q^n$, así como el concepto de peso de Hamming del elemento.

**Definición 3.11** (Soporte de un elemento)**.** *Sea* $x = (x_1, \ldots, x_n) \in \mathbb{F}_q^n$ *un elemento cualquiera. Llamamos soporte de* $x$ *al conjunto*

$$\operatorname{sop}(x) = \{i \in \{1, \ldots, n\} \mid x_i \neq 0\}.$$

**Definición 3.12** (Peso de Hamming)**.** *Sea* $x = (x_1, \ldots, x_n) \in \mathbb{F}_q^n$ *un elemento cualquiera. Llamamos peso de Hamming de* $x$ *a:*

$$\omega(x) = \#\operatorname{sop}(x) = \mathrm{d}(x, 0),$$

.

De esta manera, una vez definido el peso de Hamming de un elemento cualquiera $x = (x_1, \ldots, x_n) \in \mathbb{F}_q^n$ podemos plantearnos cuál será el peso mínimo del código $\mathcal{C}$:

$$\omega(\mathcal{C}) = \min\{\omega(\mathbf{c}) \mid \mathbf{c} \in \mathcal{C}, \mathbf{c} \neq \mathbf{0}\}. \tag{3.2}$$

Veamos un ejemplo del peso de Hamming de algunos elementos, para ello utilizaremos el software GAP junto con el paquete GUAVA.

```
gap> c1:=Codeword("000");
[ 0 0 0 ]
gap> c2:=Codeword("111");
[ 1 1 1 ]
gap> Weight(c1);
0
gap> Weight(c2);
3
```

Vista la definición de distancia mínima y de peso mínimo de Hamming, podemos llegar al siguiente lema, que relaciona ambos conceptos.

**Lema 3.1.** *Dado un código lineal $\mathcal{C}$ y dados dos elementos del código $x$ e $y$, entonces:*

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

*Es decir, la distancia mínima entre dos elementos cualesquiera del código equivale al cálculo del peso mínimo del elemento formado por la diferencia entre ambos.*

*Además, se puede concluir que:*

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

*Demostración.* Sea un código lineal $\mathcal{C}$ y dados dos elementos del código $x$ e $y$, entonces

$$\mathrm{d}(x, y) = \mathrm{d}(x - y, \mathbf{0}) = \omega(x, y),$$

consecuencia directa de la Definición 3.12. □

Podemos caracterizar la distancia mínima de un código lineal a partir de las columnas de su matriz de control, la siguiente proposición nos relacionará ambas cosas.

**Corolario 3.1.** *Dado un código lineal $\mathcal{C}$, $H$ una matriz de control del código y* d *la distancia mínima de $\mathcal{C}$. Entonces dado $r$ un entero positivo,*

$$\mathrm{d} > r \textit{ si, y solo si, cualesquiera } r \textit{ columnas de } H \textit{ son linealmente independientes.}$$

*Demostración.* Si $c \in \mathcal{C}$, sabemos que se cumple que $Hc^T = 0$. Si $c$ tiene peso $k$, esto quiere decir que una combinación de $k$ columnas de $H$ da 0. □

Como consecuencia de esta proposición, d es el menor $r$ tal que existen $r$ columnas linealmente dependientes. Otra consecuencia de este lema es que podemos hablar de la conocida como Cota de Singleton.

**Proposición 3.5.** *La distancia mínima de un código lineal de* $(n, k)$ *verifica* $\mathrm{d} \leq n - k + 1$.

*Demostración.* La distancia mínima se preserva por equivalencia de códigos. Si tomamos un código sistemático equivalente al dado y una de sus matrices generatrices, el peso mínimo de las filas está acotado superiormente por $1 + n - k$. Por tanto, el peso mínimo del código estará por debajo de esa cota. □

*Ejemplo* 3.17. Comprobemos que el código $\mathcal{C} = \{1110000, 1001100, 0101010, 1101001\}$ verifica la proposición anterior.

```
gap> m:=[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]];
gap> C:=GeneratorMatCode(m,GF(2));
a linear [7,4,1..3]1 code defined by generator matrix over GF(2)
gap> Elements(C);
[ [ 0 0 0 0 0 0 0 ], [ 0 0 0 1 1 1 1 ], [ 0 0 1 0 1 1 0 ], [ 0 0 1 1 0 0 1 ], [ 0 1 0 0
    1 0 1 ], [ 0 1 0 1 0 1 0 ], [ 0 1 1 0 0 1 1 ], [ 0 1 1 1 1 0 0 ], [ 1 0 0 0 0 1 1
    ], [ 1 0 0 1 1 0 0 ], [ 1 0 1 0 1 0 1 ], [ 1 0 1 1 0 1 0 ], [ 1 1 0 0 1 1 0 ], [ 1 1
     0 1 0 0 1 ], [ 1 1 1 1 0 0 0 ], [ 1 1 1 1 1 1 1 ] ]
gap> MinimumDistance(C)<=1+WordLength(C)-Dimension(C);
true
```

Los códigos lineales que alcanzan la Cota de Singleton son llamados de *máxima distancia de separación* y son muy importantes tanto en resultados teóricos como prácticos.

### 3.2.6. Dualidad

La matriz de control, $H$, de un código lineal $\mathcal{C}$, puede ser interpretada como matriz generatriz de otro código sobre $\mathbb{F}_q$, llamado dual de $\mathcal{C}$ y que denotamos por $\mathcal{C}^\perp$

**Proposición 3.6.** *El código dual* $\mathcal{C}^\perp$ *de un código lineal* $\mathcal{C}$ *es el ortogonal de* $\mathcal{C}$ *con respecto a la forma bilineal:*

$$\langle x, y \rangle = \sum_{i=1}^{n} x_i y_i \in \mathbb{F}_q.$$

Una de las propiedades del dual es que el dual del dual de un código es el propio código. También puede darse la situación de que $\mathcal{C}$ y $\mathcal{C}^\perp$ sean iguales.

**Definición 3.13.** *Un código lineal* $\mathcal{C}$ *se dice autodual cuando coincide con su código dual, es decir:*

$$\mathcal{C} = \mathcal{C}^\perp.$$

Para que un código sea autodual es condición necesaria (pero no suficiente) que la dimensión sea par.

A diferencia de lo que ocurre con la dimensión, no es posible, en general, determinar la distancia mínima del dual únicamente en términos de la distancia mínima del código lineal.

*Ejemplo* 3.18. En este ejemplo vamos a calcular el código dual del código binario lineal de repetición $R_5^2$ que vimos en el Ejemplo 3.9.

Sabemos que la matriz generatriz de $R_5^2$ es

$$G_5 = \left( \begin{array}{cccc|c} 1 & 1 & 1 & 1 & 1 \end{array} \right) = \left( \begin{array}{c|c} R & I_1 \end{array} \right),$$

porque $R_5^2$ es un subespacio vectorial de dimensión 1 y sólo hay un vector no nulo. Como el último elemento de $G_5$ es la matriz identidad de dimensión 1, el código viene expresado en forma sistemática, luego la matriz de comprobación de paridad del código de repetición $R_5^2$ es:

$$H_5 = \left( \begin{array}{c|c} I_4 & R^T \end{array} \right) = \left( \begin{array}{cccc|c} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{array} \right).$$

El código dual de $R_5^2$ estará generado por la matriz $H_5$ y será

$$\begin{aligned} R_5^2 &= \{(b_1, b_2, b_3, b_4, b_1 + b_2 + b_3 + b_4) \mid b_i \in \mathbb{Z}_2\} \\ &= \{(u_1, u_2, u_3, u_4, u_5) \in \mathbb{Z}_2 \mid u_1 + u_2 + u_3 + u_4 + u_5 = 0\} \\ &= \{00000, 00011, 00101, 00110, 01001, 01010, 01100, 01111, 10001, \\ &\quad 10010, 10100, 10111, 11000, 11011, 11101, 11110\} \\ &= P_5^2. \end{aligned}$$

El código dual del código de repetición $R_5^2$ es el código detector de paridad $P_5^2$.

### 3.2.7. Síndrome y detección de errores

Sea $\mathcal{C}$ un código lineal binario $(n, k)$ del que se conoce su matriz de detección de paridad, $H$, y sea $u$ una palabra del código que se transmite por un canal con ruido. Sea $v$ la palabra recibida al final del canal. Vamos a estudiar qué pasa cuando, por causas del ruido del canal, las palabras $u$ y $v$ no coinciden.

Sea $w$ la palabra $w = u + v$, no nula. En caso de que exista ruido en el canal, el vector $w$ tiene un uno en las posiciones modificadas y un cero en las posiciones que han permanecidos inalteradas, representando así los errores de transmisión que se han producido.

**Definición 3.14** (Patrón de error)**.** *Bajo estas condiciones, $w$ sería patrón de error para el código $\mathcal{C}$.*

En la posición del receptor, se conoce la palabra $v = u + w$, pero no se conoce ni la palabra del emisor $u$ ni $w$, sin embargo, se puede determinar si $v$ pertenece al código $\mathcal{C}$ calculando $Hv^t$.

**Definición 3.15** (Síndrome de una palabra)**.** *Se llama síndrome de $v$ al palabra $s = Hv^t \in \mathbb{Z}_2^{n-k}$. En caso de que la palabra síndrome sea la palabra nula, entonces la palabra $v \in C$ y en caso contrario la palabra no pertenece al código y se detecta la existencia de errores.*

Cuando el patrón de error también pertenece al código, el error es indetectable porque la palabra recibida es una palabra código al ser suma de dos palabras código. En total hay $2k - 1$ patrones de error indetectables, tantos como palabras no nulas hay en el código $\mathcal{C}$.

**Proposición 3.7.** *El síndrome de una palabra recibida $v$ no depende de la palabra en sí, sino que sólo depende del patrón de error producido.*

*Demostración.* Sea $v$ la palabra recibida, sabemos que se puede descomponer como suma de la palabra enviada y del patrón de error producido, es decir, $v = u + w$, entonces su síndrome será

$$s = Hv^t = (u + w)H^T = uH^T + wH^T = 0 + wH^T = wH^T,$$

ya que el síndrome de una palabra código es el vector nulo. □

**Observación 3.4.** *Puede suceder que varios patrones de error tengan el mismo síndrome, lo que no facilita la detección exacta de las posiciones alteradas en la transmisión.*

*Ejemplo* 3.19. Sea

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

la matriz de comprobación de paridad para un código binario lineal $\mathcal{C}$ $(3, 1)$. En ese caso, las palabras $a = 010$ y $b = 111$ son patrones de error que tienen el mismo síndrome puesto que:

$$s1 = aH^T = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \end{pmatrix},$$

$$s2 = bH^T = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \end{pmatrix}.$$

Desarrollemos este ejemplo con SageMath.
Definimos las dos palabras codificadas.

```
sage: a = vector([0,1,0])
sage: b = vector([1,1,1])
```

Construimos nuestra matriz de comprobacion de paridad $H$.

```
sage: H = matrix([[1,0,1],[0,1,0]])
sage: show(H)
[[1,0,1],[0,1,0]]
```

Calculamos el sindrome de cada una de las palabras codificadas dadas.

```
sage: s1 = a * H.transpose() % 2
sage: show(s1)
(0,1)
sage: s2 = (b * H.transpose()) %2
sage: show(s2)
(0,1)
```

Podemos ver también el mismo ejemplo pero desarrollado utilizando el software GAP, que nos servirá para comprobar el síndrome del ejemplo.

```
gap> H:=[[1,0,1],[0,1,0]];
[ [ 1, 0, 1 ], [ 0, 1, 0 ] ]
gap> C:=CheckMatCode(H,GF(2));
a linear [3,1,2]1..2 code defined by check matrix over GF(2)
gap> Syndrome(C,Codeword("010"));
[ 0 1 ]
gap> Syndrome(C,Codeword("111"));
[ 0 1 ]
```

En el ejemplo de código de Hamming, también podemos calcular el síndrome de dos maneras diferentes:

```
gap> C:=HammingCode(3);
gap> v:=Codeword("0000111");
[ 0 0 0 0 1 1 1 ]
gap> m:=Decode(C,v);
[ 0 1 1 1 ]
gap> m*G;
[ 0 0 0 1 1 1 1 ]
```

y como

```
gap> Syndrome(C,v);
[ 1 0 0 ]
gap> H:=CheckMat(C);;
gap> H*c;
[ 1 0 0 ]
gap> st:=SyndromeTable(C);
[ [ [ 0 0 0 0 0 0 0 ], [ 0 0 0 ] ], [ [ 1 0 0 0 0 0 0 ], [ 0 0 1 ] ],
  [ [ 0 1 0 0 0 0 0 ], [ 0 1 0 ] ], [ [ 0 0 1 0 0 0 0 ], [ 0 1 1 ] ],
```

```
  [ [ 0 0 0 1 0 0 0 ], [ 1 0 0 ] ], [ [ 0 0 0 0 1 0 0 ], [ 1 0 1 ] ],
  [ [ 0 0 0 0 0 1 0 ], [ 1 1 0 ] ], [ [ 0 0 0 0 0 0 1 ], [ 1 1 1 ] ] ]
gap> First(st, x->x[2]=Syndrome(C,v))[1];
[ 0 0 0 1 0 0 0 ]
gap> v+last;
[ 0 0 0 1 1 1 1 ]
```

# Capítulo 4

# Códigos binarios de Goppa

## 4.1. Introducción

### 4.1.1. ¿Qué son los códigos de Goppa?

Un código de Goppa es un código corrector de errores lineal que puede ser utilizado para cifrar y descifrar mensajes. A continuación, veremos una definición más formal. [2]

**Definición 4.1** (Polinomio de Goppa)**.** *Definiremos un polinomio de Goppa como un polinomio $g(x)$ sobre un cuerpo finito* $\mathrm{GF}(p^m)$*, esto es:*

$$g(x) = g_0 + g_1 x \cdots + g_t x^t = \sum_{i=0}^{t} g_i x^i, \tag{4.1}$$

*donde cada* $g_i \in \mathrm{GF}(p^m)$*.*

A continuación vamos a definir el concepto de polinomio irreducible.

**Definición 4.2** (Polinomio irreducible)**.** *Un polinomio $g(x)$ sobre* $\mathrm{GF}(p^m)$ *se dice irreducible cuando sus coeficientes se encuentran en* $\mathrm{GF}(p^m)$ *y el polinomio no se puede factorizar como producto de dos polinomios no constantes con coeficientes en* $\mathrm{GF}(p^m)$*.*

Sea $L$ el subconjunto finito del cuerpo en el que estamos trabajando $\mathrm{GF}(p^m)$ donde $p$ es un número primo, es decir:

$$L = \{\alpha_1, \ldots, \alpha_n\} \subseteq \mathrm{GF}(p^m), \tag{4.2}$$

tal que $g(\alpha_i) \neq 0 \ \forall \alpha_i \in L$.

Tomamos ahora una palabra $c = (c_1, \ldots, c_n)$ sobre un cuerpo finito $\mathrm{GF}(p^q)$ y tomamos la función:

$$R_c(x) = \sum_{i=1}^{n} \frac{c_i}{x - \alpha_i}, \tag{4.3}$$

donde $\frac{1}{x-\alpha_i}$ es el único polinomio que cumple $(x-\alpha_i)\frac{1}{x-\alpha_i} \equiv 1 \pmod{g(x)}$ y que tiene grado menor o igual a $t$.

En estas condiciones, definimos un código de Goppa de la siguiente manera.

**Definición 4.3** (Código de Goppa)**.** *Un código de Goppa sobre $\mathbb{F}_q$, que denotaremos por $\Gamma(L, g(x))$, es un código corrector que está formado por todos los vectores o palabras $c = (c_1, \ldots, c_n)$ tales que:*

$$R_c(z) \equiv 0 \pmod{g(x)}. \tag{4.4}$$

*Es decir, aquellas palabras tales que el polinomio $g(x)$ divide a $R_c(x)$.*

### 4.1.2. Propiedades códigos de Goppa

Los códigos de Goppa son códigos lineales (puesto que son el núcleo de una aplicación lineal, $R_c(x)$) , por tanto, podemos usar la misma notación que usábamos $(n, k, d)$ para describir un código de Goppa cuya longitud sea $n$, su dimensión sea $k$ y su distancia mínima de Hamming sea $d$. Recordemos también que la longitud $n$ dependía exclusivamente del subconjunto elegido $L$.

**Teorema 4.1.** *La dimensión $k$ de un código de Goppa $\Gamma(L, g(x))$ de longitud $n$ es mayor o igual a $n - mt$, es decir, $k \geq n - mt$.*

**Teorema 4.2.** *La distancia mínima* d *de un código de Goppa $\Gamma(L, g(x))$ de longitud $n$ es mayor o igual a $t + 1$, es decir,* $\mathrm{d} \geq t + 1$.

### 4.1.3. ¿Por qué son interesantes para la criptografía?

Estas son algunas de las razones por las que los códigos de Goppa y especialmente los códigos binarios Goppa irreducibles que veremos a continuación son importantes en la criptografía.

- La cota inferior de su distancia mínima Hamming es fácil de calcular.
- Conocer el polinomio generador permite una corrección de errores eficiente.
- Aún no se han encontrado algoritmos de corrección de errores eficientes sin el conocimiento de ese polinomio generador.
- Como consecuencia del punto anterior, los códigos de Goppa son los únicos códigos eficientes que siguen siendo invulnerables al criptoanálisis del sistema criptográfico de McEliece.

## 4.2. Códigos binarios Goppa

**Definición 4.4** (Código binario Goppa)**.** *Un código binario Goppa es un código de Goppa* $\Gamma(L, g(x))$ *sobre el cuerpo finito* $\mathrm{GF}(2^m)$ *de grado* $t$*.*

Se trata por tanto de un caso particular de código de Goppa sobre el cuerpo finito $\mathrm{GF}(2^m)$, es decir, con $p = 2$ número primo. En este trabajo nos centraremos especialmente en los códigos binarios Goppa irreducibles, dada su importancia.

**Observación 4.1.** *Veamos una serie de observaciones sobre los códigos binarios Goppa.*

- *Cuando hablamos de códigos binarios Goppa nos estamos refiriendo a aquellos que tienen característica dos, es decir, aquellos definidos sobre el cuerpo binario* $\mathbb{F}_2$*.*
- *Un código binario Goppa irreducible es aquel cuyo* $g(x)$ *que tomamos es un polinomio irreducible. Esto es así puesto que el polinomio de un código de este estilo nos permite generar algoritmos de códigos correctores eficientes. De esta manera, además, podemos aproximar de mejor manera la cota inferior de la distancia Hamming.*

**Teorema 4.3.** *Un código binario Goppa irreducible,* $\Gamma(L, g(x))$*, tiene una distancia mínima de Hamming,* d*, mayor o igual a* $2t+1$*, es decir,* $\mathrm{d} \geq 2t+1$*.*

Es por esta razón por la que un código binario Goppa irreducible puede corregir un máximo de $t = \frac{(2t+1)-1}{2}$ errores en una palabra de tamaño $2t+1$ usando palabras codificadas de tamaño $n$.

## 4.3. Matriz de comprobación de paridad

La matriz de comprobación de paridad de un código binario Goppa es útil a la hora de decodificar un mensaje $m$.

Recordamos la definición de matriz de control o de comprobación de paridad que vimos en la definición 3.10.

**Observación 4.2.** *Para poder realizar correctamente la multiplicación de matrices tenemos que considerar el vector o palabra* $\mathcal{C}$ *como* $c^T$*, es decir, el traspuesto.*

**Proposición 4.1.** *Si definimos la matriz* $H = XYZ$ *donde:*

$$X = \begin{pmatrix} g_t & 0 & 0 & \dots & 0 \\ g_{t-1} & g_t & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ g_2 & g_3 & g_4 & \ddots & 0 \\ g_1 & g_2 & g_3 & \dots & g_t \end{pmatrix}$$

$$Y = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ \alpha_1 & \alpha_2 & \alpha_3 & \dots & \alpha_n \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \alpha_1^{t-2} & \alpha_2^{t-2} & \alpha_3^{t-2} & \ddots & 0 \\ \alpha_1^{t-1} & \alpha_2^{t-1} & \alpha_3^{t-1} & \dots & \alpha_n^{t-1} \end{pmatrix}$$

$$Z = \begin{pmatrix} \frac{1}{g(\alpha_1)} & 0 & & \dots & 0 \\ 0 & \frac{1}{g(\alpha_2)} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \dots & \frac{1}{g(\alpha_n)} \end{pmatrix}$$

*entonces, $H$ es la matriz de comprobación de paridad para un código binario Goppa $\Gamma(L, g(x))$, que será una matriz sobreyectiva.*

*Demostración.* La demostración de esta proposición se puede encontrar en [1] y en [7]. Puesto que sabemos $g(x)$ es irreducible, entonces existe un elemento primitivo $\alpha \in \mathrm{GF}(2^m)$ tal que $g(\alpha) \neq 0$. Por otra parte, tenemos que:

$$\frac{g(x) - g(\alpha_i)}{x - \alpha_i} = \sum_{j=0}^{t} g_j \cdot \frac{x^i - \alpha_i^j}{x - \alpha_i} = \sum_{w=0}^{t-1} x^w \sum_{j=w+1}^{t} g_j \alpha_i^{j-1-w},$$

para todo $i \in \{1, \dots, n+1\}$.

Para ver que se $H$ es una matriz de comprobación de paridad tenemos que ver que para todo $c \in \mathrm{GF}(2^m)$ se cumple que $Hc^T = 0$. Sea entonces un vector $c$ de $\mathrm{GF}(2^m)$. Tenemos que

$$\sum_{i=1}^{n} \left(\frac{1}{g(\alpha_i)} \sum_{j=w+1}^{t} g_j \alpha_i^{j-1-w}\right) \cdot c_i = 0,$$

para todo $w \in \{0, 1, \dots, t-1\}$.

Además, $H$ se puede escribir como $H = XYZ$ y por tanto tenemos que toda palabra codificada $c \in \Gamma(L, g(z))$ si y solo si $Hc^T = 0$. □

A continuación vamos a ver dos ejemplos de códigos binarios de Goppa desarrollados en GAP y que ponen de manifiesto la teoría vista hasta ahora. Después, pasaremos a ver el proceso de codificación de estos códigos.

*Ejemplo* 4.1. Estudiar el código binario de Goppa dado por el polinomio de Goppa $g(x) = x^2 + x + 1$ en el cuerpo $\mathrm{GF}(8)$.

```
1 gap> x:=X(GF(8),"x");
2 x
3 gap> g:=x^2+x+1;
```

```
x^2+x+Z(2)^0
gap> L:=Elements(GF(8));
[ 0*Z(2), Z(2)^0, Z(2^3), Z(2^3)^2, Z(2^3)^3, Z(2^3)^4, Z(2^3)^5, Z(2^3)^6 ]
gap> C:=GoppaCode(g,L);
a linear [8,2,5]3 classical Goppa code over GF(2)
gap> Elements(C);
[ [ 0 0 0 0 0 0 0 0 ], [ 0 0 1 1 1 1 1 1 ], [ 1 1 0 0 1 0 1 1 ],
  [ 1 1 1 1 0 1 0 0 ] ]
gap> G:=GeneratorMat(C);
[ <an immutable GF2 vector of length 8>, <an immutable GF2 vector of
    length
    8> ]
gap> List(G,Codeword);
[ [ 1 1 1 1 0 1 0 0 ], [ 1 1 0 0 1 0 1 1 ] ]
```

Podemos calcular $\frac{1}{x-a}$ mód $g(x)$ con $a \in L$

```
gap> inv:=List(L, a->GcdRepresentation(x-a,g)[1]);
[ x+Z(2)^0, x, Z(2^3)^2*x+Z(2^3)^5, Z(2^3)^4*x+Z(2^3)^3, Z(2^3)^2*x+Z
    (2^3)^3,
  Z(2^3)*x+Z(2^3)^6, Z(2^3)*x+Z(2^3)^5, Z(2^3)^4*x+Z(2^3)^6 ]
```

*Ejemplo* 4.2. Estudiamos el código binario de Goppa dado por el polinomio de Goppa $g(x) = x^2 + x + Z(16)^3$ en el cuerpo GF(16) ($Z(16)$ es la notación que usa GAP para denotar un elemento primitivo del cuerpo finito de tamaño 16).

```
gap> x:=Indeterminate(GF(16),"x");
x
gap> g:=x^2+x+Z(16)^3;
x^2+x+Z(2^4)^3
gap> L := List([2..13], i->Z(16)^i);
[ Z(2^4)^2, Z(2^4)^3, Z(2^4)^4, Z(2^2), Z(2^4)^6, Z(2^4)^7, Z(2^4)^8,
  Z(2^4)^9, Z(2^2)^2, Z(2^4)^11, Z(2^4)^12, Z(2^4)^13 ]
gap> C:=GoppaCode(g,L);
a linear [12,4,5]4..5 classical Goppa code over GF(2)
gap> H:=CheckMat(C);;
gap> List(H,Codeword);
[ [ 0 0 1 0 1 0 0 0 0 0 0 1 ], [ 0 1 0 1 1 0 0 0 1 0 0 1 ],
  [ 0 0 0 0 1 1 0 1 0 1 1 1 ], [ 1 0 0 1 0 0 1 0 1 1 1 0 ],
  [ 0 0 0 1 0 0 0 0 1 0 0 1 ], [ 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 0 0 0 0 0 1 0 0 1 0 0 ] ]
gap> G:=GeneratorMat(C);;
gap> List(G,Codeword);
[ [ 0 1 1 1 1 0 0 1 1 0 0 0 ], [ 0 1 1 0 1 0 1 0 0 1 0 0 ],
  [ 1 1 1 0 1 1 0 1 0 0 1 0 ], [ 1 1 0 1 1 0 0 0 0 0 0 1 ] ]
```

## 4.4. Proceso de codificación

La codificación de un código binario Goppa implica multiplicar el mensaje por la matriz generatriz del código.

En este punto conviene recordar la definición de matriz generatriz de un código que vimos en la definición 3.7.

**Proposición 4.2.** *Cualquier matriz $G$ de tamaño $k \times n$ y de rango máximo tal que $GH^T = 0$, es una matriz generatriz de un código binario Goppa.*

*Demostración.* La demostración de esta proposición es directa utilizando la proposición 4.1 puesto que, como $G$ es una matriz que cumple $GH^T = 0$, entonces aplicando la definición de matriz generatriz de un código binario Goppa se ve que $G$ lo es. □

Para poder enviar mensajes usando códigos binarios Goppa primero hay que escribir el mensaje en bloques de $k$ símbolos. Después, cada uno de esos bloques será multiplicado por la matriz generatriz $G$ del código dando resultado a un conjunto de vectores o palabras codificadas.

Un ejemplo de esta operación sería la siguiente:

*Ejemplo* 4.3. Dado un mensaje $m = (m_1, \ldots, m_k)$ cualquiera y una matriz generatriz, $G$, de dimensión $k \times n$ de nuestro código binario Goppa, entonces el mensaje codificado $c = (c_1, \ldots, c_n)$ se calcula de la siguiente manera:

$$c = m \times G,$$

es decir, $(c_1, \ldots, c_n) = (m_1, \ldots, m_k) \times G$.

### 4.4.1. Ejemplo de código binario Goppa irreducible

Nótese que $\mathrm{GF}(2^4) \cong \frac{\mathrm{GF}(2)[x]}{K(x)}$, para todo $K(x)$ polinomio irreducible de grado 4. Lo primero es buscar un elemento primitivo $\alpha$, para ello podemos factorizar $x^{15} - 1 \in \mathbb{Z}_2[X]$ en factores irreducibles. Es decir:

$$x^{15} - 1 = (x+1)(x^2+x+1)(x^4+x+1)(x^4+x^3+1)(x^4+x^3+x^2+x+1).$$

Tomamos, por ejemplo, el polinomio irreducible de grado 4, $K(x) = x^4 + x + 1$ y sea $\alpha$ una raíz del polinomio $K(x)$, entonces sabemos que

$$\alpha \text{ es un elemento primitivo} \Leftrightarrow \mathrm{orden}(\alpha) = 15.$$

Como $\alpha^1 \neq 1$ y el orden de un elemento tiene que dividir al orden del grupo, entonces sólo podrá ser 3, 5 o 15. Veamos primero que $\mathrm{orden}(\alpha) \neq 3$ y que $\mathrm{orden}(\alpha) \neq 5$, es decir, que $\alpha^3 \neq 1$ y que $\alpha^5 \neq 1$ respectivamente.

Como $\alpha^4 = \alpha + 1$, entonces es claro que $\alpha^3 \neq 1$ y además $\alpha^5 = \alpha\alpha^4 = \alpha(1+\alpha) = \alpha^2 + \alpha \neq 1$. Luego entonces sólo puede ocurrir que $\mathrm{orden}(\alpha) = 15$, por tanto $\alpha$ es un elemento primitivo.

El grupo multiplicativo de elementos no nulos de $\mathrm{GF}(2^4)$, que notamos por $\mathrm{GF}(2^4)^*$, es un subgrupo cíclico generado por $\alpha$, $\langle\alpha\rangle$, es decir,

$$\mathrm{GF}(2^4)^* = \mathrm{GF}(2^4) \cup \{0\} = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{14}\}.$$

Podemos escribir los elementos de $\mathrm{GF}(2^4)$ como potencias de $\alpha$, utilizando que $\alpha^4 = \alpha + 1$.

$$\begin{aligned}
0 &= 0\cdot 1 + 0\cdot\alpha + 0\cdot\alpha^2 + 0\cdot\alpha^3 = (0,0,0,0)^T,\\
1 &= 1\cdot 1 + 0\cdot\alpha + 0\cdot\alpha^2 + 0\cdot\alpha^3 = (1,0,0,0)^T,\\
\alpha &= 0\cdot 1 + 1\cdot\alpha + 0\cdot\alpha^2 + 0\cdot\alpha^3 = (0,1,0,0)^T,\\
\alpha^2 &= 0\cdot 1 + 0\cdot\alpha + 1\cdot\alpha^2 + 0\cdot\alpha^3 = (0,0,1,0)^T,\\
\alpha^3 &= 0\cdot 1 + 0\cdot\alpha + 0\cdot\alpha^2 + 1\cdot\alpha^3 = (0,0,0,1)^T,\\
\alpha^4 &= 1\cdot 1 + 1\cdot\alpha + 0\cdot\alpha^2 + 0\cdot\alpha^3 = (1,1,0,0)^T,\\
\alpha^5 &= 0\cdot 1 + 1\cdot\alpha + 1\cdot\alpha^2 + 0\cdot\alpha^3 = (0,1,1,0)^T,\\
\alpha^6 &= 0\cdot 1 + 0\cdot\alpha + 1\cdot\alpha^2 + 1\cdot\alpha^3 = (0,0,1,1)^T,\\
\alpha^7 &= 1\cdot 1 + 1\cdot\alpha + 0\cdot\alpha^2 + 1\cdot\alpha^3 = (1,1,0,1)^T,\\
\alpha^8 &= 1\cdot 1 + 0\cdot\alpha + 1\cdot\alpha^2 + 0\cdot\alpha^3 = (1,0,1,0)^T,\\
\alpha^9 &= 0\cdot 1 + 1\cdot\alpha + 0\cdot\alpha^2 + 1\cdot\alpha^3 = (0,1,0,1)^T,\\
\alpha^{10} &= 1\cdot 1 + 1\cdot\alpha + 1\cdot\alpha^2 + 0\cdot\alpha^3 = (1,1,1,0)^T,\\
\alpha^{11} &= 0\cdot 1 + 1\cdot\alpha + 1\cdot\alpha^2 + 1\cdot\alpha^3 = (0,1,1,1)^T,\\
\alpha^{12} &= 1\cdot 1 + 1\cdot\alpha + 1\cdot\alpha^2 + 1\cdot\alpha^3 = (1,1,1,1)^T,\\
\alpha^{13} &= 1\cdot 1 + 0\cdot\alpha + 1\cdot\alpha^2 + 1\cdot\alpha^3 = (1,0,1,1)^T,\\
\alpha^{14} &= 1\cdot 1 + 0\cdot\alpha + 1\cdot\alpha^2 + 1\cdot\alpha^3 = (1,0,0,1)^T.
\end{aligned}$$

Consideramos el código de Goppa sobre $L = \{\alpha^i \mid 2 \leq i \leq 13\}$ con $g(x) = x^2+x+\alpha^3$. El código es irreducible sobre $\mathrm{GF}(2^4)$ y sus parámetros son $p = 2$, $m = 4$, $n = 12$, $t = 2$. Por el teorema 4.1 sabemos que $k \geq n - mt = 12 - 4\cdot 2 = 4$ y por el teorema 4.3 sabemos que $d \geq 2t + 1 = 2\cdot 2 + 1 = 5$. Por tanto tenemos un código de Goppa $(12, \geq 4, \geq 5)$.

Buscamos una matriz de paridad $H$, para ello utilizamos la proposición 4.1 asignando los valores de $g_1 = \alpha_7$, $g_2 \equiv 1$, $\alpha_1 = \alpha^2$, $\alpha_2 = \alpha^3$,..., $\alpha_{12} = \alpha^{13}$. Podemos ahora calcular los factores $\frac{1}{g(\alpha_i)}$ para todo $i \in \{1, \ldots, 12\}$.

$$\begin{aligned}
\frac{1}{g(\alpha_1)} &= \frac{1}{(\alpha^2)^2 + x + \alpha^3} = \frac{1}{\alpha^4 + \alpha^2 + \alpha^3}\\
&= [(1,1,0,0)^T + (0,0,1,0)^T + (0,0,0,1)^T]^{-1} = [(1,1,1,1)^T]^{-1}\\
&= (\alpha^{12})^{-1} = \alpha^3,
\end{aligned}$$

$$\begin{aligned}
\frac{1}{g(\alpha_2)} &= \frac{1}{(\alpha^3)^2 + x + \alpha^3} = \frac{1}{\alpha^6 + \alpha^3 + \alpha^3}\\
&= [(0,0,1,1)^T + (0,0,0,1)^T + (0,0,0,1)^T]^{-1} = [(0,0,1,1)^T]^{-1}\\
&= (\alpha^6)^{-1} = \alpha^9,
\end{aligned}$$

$$\frac{1}{g(\alpha_3)} = \frac{1}{(\alpha^4)^2 + x + \alpha^3} = \frac{1}{\alpha^8 + \alpha^4 + \alpha^3}$$
$$= [(1,0,1,0)^T + (1,1,0,0)^T + (0,0,0,1)^T]^{-1} = [(0,1,1,1)^T]^{-1}$$
$$= (\alpha^{11})^{-1} = \alpha^4,$$

$$\frac{1}{g(\alpha_4)} = \frac{1}{(\alpha^5)^2 + x + \alpha^3} = \frac{1}{\alpha^{10} + \alpha^5 + \alpha^3}$$
$$= [(1,1,1,0)^T + (0,1,1,0)^T + (0,0,0,1)^T]^{-1} = [(1,0,0,1)^T]^{-1}$$
$$= (\alpha^{14})^{-1} = \alpha,$$

$$\frac{1}{g(\alpha_5)} = \frac{1}{(\alpha^6)^2 + x + \alpha^3} = \frac{1}{\alpha^{12} + \alpha^6 + \alpha^3}$$
$$= [(1,1,1,1)^T + (0,0,1,1)^T + (0,0,0,1)^T]^{-1} = [(1,1,0,1)^T]^{-1}$$
$$= (\alpha^7)^{-1} = \alpha^8,$$

$$\frac{1}{g(\alpha_6)} = \frac{1}{(\alpha^7)^2 + x + \alpha^3} = \frac{1}{\alpha^{14} + \alpha^7 + \alpha^3}$$
$$= [(1,0,0,1)^T + (1,1,0,1)^T + (0,0,0,1)^T]^{-1} = [(1,1,0,1)^T]^{-1}$$
$$= (\alpha^9)^{-1} = \alpha^6,$$

$$\frac{1}{g(\alpha_7)} = \frac{1}{(\alpha^8)^2 + x + \alpha^3} = \frac{1}{\alpha^1 + \alpha^8 + \alpha^3}$$
$$= [(0,1,0,0)^T + (1,0,1,0)^T + (0,0,0,1)^T]^{-1} = [(1,1,1,1)^T]^{-1}$$
$$= (\alpha^{12})^{-1} = \alpha^3,$$

$$\frac{1}{g(\alpha_8)} = \frac{1}{(\alpha^9)^2 + x + \alpha^3} = \frac{1}{\alpha^3 + \alpha^9 + \alpha^3}$$
$$= [(0,0,0,1)^T + (0,1,0,1)^T + (0,0,0,1)^T]^{-1} = [(0,1,0,1)^T]^{-1}$$
$$= (\alpha^9)^{-1} = \alpha^6,$$

$$\frac{1}{g(\alpha_9)} = \frac{1}{(\alpha^{10})^2 + x + \alpha^3} = \frac{1}{\alpha^5 + \alpha^{10} + \alpha^3}$$
$$= [(0,1,1,0)^T + (1,1,1,0)^T + (0,0,0,1)^T]^{-1} = [(1,0,0,1)^T]^{-1}$$
$$= (\alpha^{14})^{-1} = \alpha,$$

$$\frac{1}{g(\alpha_{10})} = \frac{1}{(\alpha^{11})^2 + x + \alpha^3} = \frac{1}{\alpha^7 + \alpha^{11} + \alpha^3}$$
$$= [(1,1,0,1)^T + (0,1,1,1)^T + (0,0,0,1)^T]^{-1} = [(1,0,1,1)^T]^{-1}$$
$$= (\alpha^{13})^{-1} = \alpha^2,$$

$$\begin{aligned}\frac{1}{g(\alpha_{11})} &= \frac{1}{(\alpha^{12})^2 + x + \alpha^3} = \frac{1}{\alpha^9 + \alpha^{12} + \alpha^3} \\ &= [(0,1,0,1)^T + (1,1,1,1)^T + (0,0,0,1)^T]^{-1} = [(1,0,1,1)^T]^{-1} \\ &= (\alpha^{13})^{-1} = \alpha^2,\end{aligned}$$

$$\begin{aligned}\frac{1}{g(\alpha_{12})} &= \frac{1}{(\alpha^{13})^2 + x + \alpha^3} = \frac{1}{\alpha^{11} + \alpha^{13} + \alpha^3} \\ &= [(0,1,1,1)^T + (1,0,1,1)^T + (0,0,0,1)^T]^{-1} = [(1,1,0,1)^T]^{-1} \\ &= (\alpha^7)^{-1} = \alpha^8\end{aligned}$$

Podemos ahora calcular $H = XYZ$ de la siguiente manera. Dadas $X$, $Y$, $Z$ de la forma

$$X = \begin{pmatrix} g_t & 0 & 0 & \dots & 0 \\ g_{t-1} & g_t & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ g_2 & g_3 & g_4 & \ddots & 0 \\ g_1 & g_2 & g_3 & \dots & g_t \end{pmatrix},$$

$$Y = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ \alpha_1 & \alpha_2 & \alpha_3 & \dots & \alpha_n \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \alpha_1^{t-2} & \alpha_2^{t-2} & \alpha_4^{t-2} & \ddots & 0 \\ \alpha_1^{t-1} & \alpha_2^{t-1} & \alpha_4^{t-1} & \dots & \alpha_n^{t-1} \end{pmatrix},$$

$$Z = \begin{pmatrix} \frac{1}{g(\alpha_1)} & 0 & & \dots & 0 \\ 0 & \frac{1}{g(\alpha_3)} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \dots & \frac{1}{g(\alpha_n)} \end{pmatrix}.$$

Entonces

$$H = \begin{pmatrix} g_2 g(\alpha_1)^{-1} & g_2 g(\alpha_2)^{-1} & \dots & g_2 g(\alpha_{12})^{-1} \\ (g_1 + g_2 \cdot \alpha_1) \cdot g(\alpha_1)^{-1} & (g_1 + g_2 \cdot \alpha_2) \cdot g(\alpha_2)^{-1} & \dots & (g_1 + g_2 \cdot \alpha_{12}) \cdot g(\alpha_{12})^{-1} \end{pmatrix}$$

$$= \begin{pmatrix} \alpha^3 & \alpha^9 & \alpha^4 & \alpha & \alpha^8 & \alpha^6 & \alpha^3 & \alpha^6 & \alpha & \alpha^2 & \alpha^2 & \alpha^8 \\ 1 & \alpha^{13} & \alpha^7 & \alpha^{14} & \alpha^3 & 0 & \alpha^{14} & \alpha^6 & \alpha^7 & \alpha^{10} & \alpha^4 & \alpha^{13} \end{pmatrix}$$

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

Además, podemos utilizar $GH^T = 0$ para calcular la matriz $G$, que será una matriz de dimensión $4 \times 12$:

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

Por tanto, la dimensión de $\Gamma(L, g(x))$ será 4 y el código de Goppa tendrá parámetros $(12, 4, \geq 5)$.

```
gap> x:=Indeterminate(\mathrm{GF}(16),"x");
x
gap> g:=x^2+x+Z(16)^3;
x^2+x+Z(2^4)^3
gap> L := List([2..13], i->Z(16)^i);
[ Z(2^4)^2, Z(2^4)^3, Z(2^4)^4, Z(2^2), Z(2^4)^6, Z(2^4)^7, Z(2^4)^8,
  Z(2^4)^9, Z(2^2)^2, Z(2^4)^11, Z(2^4)^12, Z(2^4)^13 ]
gap> C:=GoppaCode(g,L);
a linear [12,4,5]4..5 classical Goppa code over GF(2)
gap> H:=CheckMat(C);;
gap> List(H,Codeword);
[ [ 0 0 1 0 1 0 0 0 0 0 0 1 ], [ 0 1 0 1 1 0 0 0 1 0 0 1 ],
  [ 0 0 0 0 1 1 0 1 0 1 1 1 ], [ 1 0 0 1 0 0 1 0 1 1 1 0 ],
  [ 0 0 0 1 0 0 0 0 1 0 0 1 ], [ 0 0 0 0 0 0 1 0 1 1 0 0 0 ],
  [ 0 0 0 0 0 0 0 0 1 1 0 1 0 ], [ 0 0 0 0 0 0 0 1 0 0 1 0 0 ] ]
gap> G:=GeneratorMat(C);;
gap> List(G,Codeword);
[ [ 0 1 1 1 1 0 0 1 1 0 0 0 ], [ 0 1 1 0 1 0 1 0 0 1 0 0 ],
  [ 1 1 1 0 1 1 0 1 0 0 1 0 ], [ 1 1 0 1 1 0 0 0 0 0 0 1 ] ]
```

*Ejemplo* 4.4. Dado un código $\mathcal{C}$ sobre $\mathbb{F}_q = \mathrm{GF}(2^4)$ con $q$ elemento primitivo de GF(16). Calculemos una matriz generatriz del código.

Consideramos los 15 elementos no nulos que forman este anillo. Es decir:

$$\begin{aligned} p_1 = 1, p_2 = a, p_3 = a^2, p_4 = a^3, \\ p_5 = a+1, p_6 = a^2+a, p_7 = a^3+a^2, \\ p_8 = a^3+a+1, p_9 = a^2+1, p_{10} = a^3+a, \\ p_{11} = a^2+a+1, p_{12} = a^3+a^2+a, p_{13} = a^3+a^2+a+1, \\ p_{14} = a^3+a^2+1, p_{15} = a^3+1. \end{aligned}$$

Definimos el cuerpo donde vamos a trabajar

```
sage: m = 4
sage: K = GF(2)
sage: F.<a> = GF(2^m)
```

Y creamos el anillo de polinomios PR utilizando `PolynomialRing` de Sage.

```
sage: PR = PolynomialRing(F,'X')
sage: X = PR.gen()
sage: N = 15
sage: L = [a^i for i in range(N)]
sage: g = X^3 + X + 1
```

De entre ellos, consideramos el polinomio de Goppa siguiente:

$$g(x) = x^3 + x + 1.$$

Buscamos calcular la matriz generatriz del código como el producto de tres matrices como hemos comentado en la teoría, es decir:

$$H = XYZ,$$

donde la matriz $X$ viene dada por el polinomio de Goppa como:

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

```
rango = 3
X = matrix(F,rango,rango)
for i in range(rango):
    count = rango - i + 1 - 1
    for j in range(rango):
        if i > j:
            X[i,j]=g.list()[count]
            count = count + 1
        if i < j:
            X[i,j] = 0
        if i == j:
            X[i,j] = 1
```

nuestra matriz $Y$ vendrá dada por los polinomios que hemos formado:

$$Y = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & a & a^2 & a^3 & a+1 \\ 1 & a^2 & a+1 & a^3+a^2 & a^2+1 \end{pmatrix}$$

$$\begin{matrix} 1 & 1 & 1 & 1 & 1 \\ a^2+a & a^3+a^2 & a^3+a+1 & a^2+1 & a^3+a \\ a^2+a+1 & a^3+a^2+a+1 & a^3+1 & a & a^3 \end{matrix}$$

$$\begin{matrix} 1 & 1 & 1 & 1 & 1 \\ a^2+a+1 & a^3+a^2+a & a^3+a^2+a+1 & a^3+a^2+1 & a^3+1 \\ a^2+a & a^3+a+1 & a^3+a & a^3+a^2+a & a^3+a^2+1 \end{matrix}\Bigg)$$

La matriz $Y$ es de tamaño $3 \times n$, es decir, 3 filas y $n$ columnas.

```
Y = matrix([[L[j]^i for j in range(N)] for i in range(rango)])
```

Tomamos nuestra matriz diagonal $Z$ cuyos elementos en la diagonal son los siguientes:

$$1,\ a^2+1,\ a,\ a^3+a^2+1,\ a^2,\ a^2+a+1,\ a^3+a+1,\ a+1,\ a^3+a^2+1,$$
$$a^2+a,\ a^2+a,\ a^3+1,\ a^2+a{+}1,\ a^2+a.$$

```
Z = diagonal_matrix([1/g(L[i]) for i in range(N)])
```

La matriz Z sabemos es diagonal y de tamaño $n \times n$. Finalmente, calculamos la matriz generatriz $H$:

```
H = T*Y*Z
```

$$H = \begin{pmatrix} 1 & a^2+1 & a & a^3+a^2+a & a^2 \\ 1 & a^3+a & a^3 & a^3+1 & a^3+a^2 \\ 0 & a & a^2 & a^3+a & a+1 \end{pmatrix}$$

$$\begin{matrix} a^2+a+1 & a^3+a+1 & a^2+a+1 & a+1 & a^3+a^2+1 \\ 1 & a^3+a^2+1 & a^2 & a^3+a^2+a+1 & a^3+a+1 \\ 1 & a^3 & a^3+a^2+1 & a^2+1 & a^3+a^2+a+1 \end{matrix}$$

$$\begin{matrix} a^2+a & a^2+a & a^3+1 & a^2+a+1 & a^2+a \\ 1 & a & a^3+a^2+a & a^2+1 & a+1 \\ 1 & a^3+1 & a^3+a^2 & a^3+a+1 & a^3+a^2+a \end{matrix}\Bigg)$$

## 4.5. Corrección de errores

### 4.5.1. Proceso de corrección de errores

Vamos ahora a estudiar el proceso de corrección de errores de una palabra codificada dada. Sea $y$ una palabra codificada dada cualquiera con $r \leq t$

errores. Entonces

$$y = (y_1, \ldots, y_n) = (c_1, \ldots, c_n) + (e_1, \ldots, e_n),$$

donde hay $r$-posiciones donde se verifica $e_i \neq 0$.

Buscamos corregir la palabra codificada de manera que volvamos a tener la palabra tenemos que primero encontrar ese vector de error. Es más, tenemos que buscar el conjunto de posiciones donde se dan los errores,

$$E = \{i \mid \ e_i \neq 0\}$$

y los correspondientes valores $e_i$ para todo $i \in E$.

**Definición 4.5** (Polinomio de localización de errores)**.** *Definimos el polinomio de localización de errores $\sigma(x)$ como*

$$\sigma(x) = \prod_{i \in E} (x - \sigma_i).$$

Puesto que estamos trabajando con códigos binarios de Goppa basta con calcular las posiciones de los errores puesto que como en binario sólo hay dos valores, 0/1, entonces halladas las posiciones no hay más que cambiar 0 por 1. En caso de estar trabajando con códigos de Goppa normales, necesitaríamos calcular un polinomio de corrección de errores localizador y otro diferente para evaluar los errores, complicando el uso del algoritmo original.

## 4.6. Proceso de decodificación

Una vez hemos corregido todos los posibles errores de una palabra codificada, entonces el receptor puede fácilmente recuperar el mensaje original. Recordemos que $(c_1, \ldots, c_n) = (m_1, \ldots, m_k)G$. Pensemos en esta función como una función,

$$\begin{aligned} F \colon \mathrm{F_q} &\to F_n, \\ m &\mapsto mG. \end{aligned}$$

Puesto que $m_k$ tiene rango $k$, $G$ tiene rango $k$ y $F_n$ tiene rango $n$, entonces la función $F$ será inyectiva. Podremos por tanto modificar la ecuación,

$$(c_1, \ldots, c_n) = (m_1, \ldots, m_k)G.$$

como sigue,

$$G^T \begin{pmatrix} m_1 \\ m_2 \\ \vdots \\ m_k \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}.$$

Se trata de un sistema de $n$ ecuaciones con $k$ incógnitas, que utilizando reducción por filas se resuelve,

$$\left(\begin{array}{c|c} G^T & \begin{matrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{matrix} \end{array}\right) \cong \begin{pmatrix} 1 & 0 & \dots & 0 & m_1 \\ 0 & 1 & \dots & 0 & m_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & m_k \end{pmatrix},$$

donde la matriz de la derecha, que llamaremos $X$, es una matriz de dimensión $(n-k) \times (k+1)$.

### 4.6.1. Algoritmo de Patterson

De cara a corregir los errores de una palabra recibida tenemos que aplicar el algoritmo de Patterson, [1]. Dados $r \leq t$ errores para un polinomio irreducible $g(x)$ sobre $\mathrm{GF}(2^m)$ (concepto visto en la Definición 4.2) este algoritmo consiste en aplicar los siguientes pasos.

1. Sea $y = y_1, \dots, y_n$ una palabra codificada dada. Calculamos la función $s(x)$ definida así,

$$s(x) = \sum_{i=1}^{n} \frac{y_i}{x - \alpha_i} \pmod{g(x)}.$$

2. Calculamos el polinomio de localización de errores $\sigma(x)$ en cuatro pasos.

   *a*) Encontrar $h(x)$ tal que $s(x)h(x) \equiv 1 \pmod{g(x)}$. Si $h(x) = x$ entonces, hemos terminado y la solución es $\sigma(x) = x$.

   *b*) Calcular $d(x)$ tal que $d^2(x) \equiv h(x) + x \pmod{g(x)}$.

   *c*) Encontrar $a(x)$ y $b(x)$ de grado menor tales que $d(x)b(x) \equiv a(x) \pmod{g(x)}$.

   *d*) Definir el polinomio de localización de errores como $\sigma(x) = a^2(x) + b^2(x)x$.

3. Una vez que tenemos el polinomio de localización de errores definido, lo usamos para determinar el conjunto de posiciones de error $E = \{i \mid e_i \neq 0\}$.

4. Definimos el vector de errores $e = (e_1, \dots, e_n)$ como $e_i = 1$ si $i \in E$ y $e_i = 0$ en caso contrario.

5. Definimos la palabra codificada $c = y - e$.

### 4.6.2. Ejemplo de codificación y decodificación de Patterson

Continuamos el ejemplo 4.4 que vimos anteriormente, ahora vamos a ver como codificar y decodificar un mensaje utilizando el algoritmo de Patterson.

Primero definiremos una serie de funciones auxiliares que serán útiles a la hora de desarrollar el algoritmo.

La primera de ellas descompone o separa un polinomio dado en dos partes que nos permitirán definir el polinomio de localización de errores como $\sigma(x) = a^2(x) + b^2(x)x$ para el paso $2(d)$ del algoritmo.

```
def descomponer_polinomio(p):
    Phi1 = p.parent()
    p0 = Phi1([sqrt(c) for c in p.list()[0::2]])
    p1 = Phi1([sqrt(c) for c in p.list()[1::2]])
    return (p0,p1)
```

También usaremos el algoritmo de Euclídes extendido que hemos programado en Sage. Recordar que el algoritmo de Euclídes extendido además de encontrar el máximo común divisor de dos números enteros $a$ y $b$, expresarlo como la mínima combinación lineal de esos números, es decir, encontrar números enteros $s$ y $t$ tales que $\text{mcd}(a, b) = as + bt$.

```
def algoritmo_euclides_extendido(self, other):
    delta = self.degree() #grado de polinomio 1
    if other.is_zero(): # si el polinomio introducido es
        ring = self.parent() #comprobamos el cuerpo en el que trabajamos
        return self, R.one(), R.zero() #mcd = mismo polinomio y devuelve
            un uno (s) y un cero (t) en el cuerpo que trabajamos.

    # mcd (a,b) = as+bt

    ring = self.parent() #comprobamos el cuerpo en el que trabajamos
    a = self # guardamos una copia del primer polinomio 1 (self)
    b = other # guardamos una copia del segundo polinomio (other)

    s = ring.one() # guardamos en s el uno del anillo
    t = ring.zero() # guardamos en t el cero del anillo

    resto0 = a
    resto1 = b

    while true:
        cociente,resto_auxiliar = resto0.quo_rem(resto1) # La funcion
            quo_rem de Sage devuelve el cociente y el resto. Que guardamos
            en Q y ring.
        resto0 = resto1
        resto1 = resto_auxiliar

        s = t
        t = s - t*cociente
```

```
26
27          if resto1.degree() <= floor((delta-1)/2) and resto0.degree() <= floor((
                delta)/2):
28              break
29
30      V = (resto0-a*s)//b
31      coeficiente_lider = resto0.leading_coefficient() # guardamos el
            coeficiente lider del resto 0
32
33  return resto0/coeficiente_lider, s/coeficiente_lider, V/coeficiente_lider
```

Por último, definiremos una función que utiliza el algoritmo de Euclídes (no extendido) y ya programado en Sage que necesitaremos para el paso 2(c) del algoritmo de Patterson, en la búsqueda de los $a(x)$ y $b(x)$ de grado menor.

```
1  def inversa_g(p,g):
2      (d,u,v) = xgcd(p,g)
3      return u.mod(g)
```

Una vez vistas estas funciones auxiliares, podemos pasar a ver la función del algoritmo de Patterson implementada, que, recibirá un vector codificado y lo devolverá decodificado. Además, hemos añadido a la función que informe sobre las posiciones donde se han encontrado los errores.

```
1  def decodePatterson(y):
2      alpha = vector(H*y)
3      polinomioS = PR(0)
4      for i in range(len(alpha)):
5          polinomioS = polinomioS + alpha[i]*(X^(len(alpha)-i-1))
6
7      vector_g = descomponer_polinomio(g)
8      w = ((vector_g[0])*inversa_g(vector_g[1],g)).mod(g)
9      vector_t = descomponer_polinomio(inversa_g(polinomioS,g) + X)
10
11     R = (vector_t[0]+(w)*(vector_t[1])).mod(g)
12
13     (a11,b11,c11) = algoritmo_euclides_extendido(g,R)
14
15     sigma = a11^2+X*(c11^2)
16
17     for i in range(N):
18         if (sigma(a^i)==0):
19             print ("Error encontrado en la posicion: " + str(i))
20             y[i] = y[i] + 1
21     return y
```

Una vez desarrolladas las funciones principales del algoritmo podemos continuar con el ejemplo. Lo primero será calcular la matriz de control de paridad y la matriz generatriz del código.

Definimos nuestra matriz de control de paridad H de tamaño correcto (nula en estos momentos).

```
sage: H_Goppa_K = matrix(K, m*H.nrows(), H.ncols())
```

Y la rellenamos correctamente.

```
sage: for i in range (H.nrows()):
        for j in range(H.ncols()):
            be = bin(eval(H[i,j].__int__repr()))[2:]
            be = '0'*(m-len(be))+be; be = list(be)
            H_Goppa_K[m*i:m*(i+1),j]=vector(map(int,be))
```

Tenemos la matriz $H$ del código Goppa, vamos a calcular la matriz $G$ que será de dimensión $(k \times n)$. Para calcularla usamos $GH^T = 0$

```
sage: inversa_H_Goppa_K = H_Goppa_K.right_kernel()
```

Y usamos la funcion $basis_matrix()$ que nos permite quedarnos con la matriz G.

```
sage: G_Goppa = inversa_H_Goppa_K.basis_matrix()
```

Generamos un vector sin codificar aleatorio de tamaño correcto.

```
sage: u = vector(K,[randint(0,1) for n in range(G_Goppa.nrows())])
```

Y lo codificamos utilizando la matriz $G$ de Goppa.

```
sage: c = u*G_Goppa
```

Definimos un vector de errores en principio nulo, añadimos algunos unos en el vector $e$ y lo sumamos con el vector sin codificar que habíamos definido. De esta manera tenemos en $y$ el vector codificado y con los errores añadidos.

```
sage: e = vector(K,N)
sage: e[2] = 1
sage: e[3] = 1
sage: y = c + e
```

Por último, le pasaremos al algoritmo de Patterson definido el vector codificado para así detectar y corregir los errores, obteniendo el vector original.

```
sage: decodePatterson(y)
```

# Capítulo 5

# Algoritmo de McEliece

## 5.1. Introducción

### 5.1.1. ¿Qué es el criptosistema McEliece?

El criptosistema McEliece es un tipo de criptosistema de clave pública que usa códigos correctores lineales para crear la clave pública y la clave privada. El código corrector que se propone en este criptosistema es el código binario Goppa. La clave pública es, como se puede entender, pública y por tanto cualquiera puede encontrarla. Esta clave pública está basada en la clave privada, sin embargo, no es posible recuperar esta clave privada a partir de la pública conocida.

Por tradición, se propone un ejemplo basado en dos amigos, Alice y Bob, que buscan intercambiarse mensajes para explicar conceptos criptográficos. El problema es el siguiente: Alice y Bob tratan de intercambiar mensajes a través de un canal, que puede ser inseguro o seguro. Suponemos también una tercera persona, que actuará como atacante y tratará de conseguir capturar los mensajes entre los amigos Alice y Bob. En esta primera imagen, vemos cómo sería una comunicación entre Alice y Bob a través de un canal inseguro.


Figura 5.1: Comunicación entre Alice y Bob insegura.

Se aprecia en la imagen como Eve, el atacante que está escuchando la conversación, es capaz de obtener los mensajes que se intercambian Alice y

Bob puesto que están siendo transmitidos por un canal inseguro. Éste es el problema que tratamos de resolver, establecer una comunicación segura entre los amigos de manera que ninguna persona pueda capturar los mensajes de ésta.

En el caso del criptosistema McEliece, hemos comentado previamente que se trataba de un criptosistema basado en clave pública - clave privada. Veamos que significa ésto.

Supongamos que Alice quiere mandar un mensaje privado a Bob, para ello, Bob lo primero que tiene que hacer es hacer pública (para Alice) su clave pública, que se basa, como hemos dicho, en su clave privada. Una vez que Alice tiene la clave pública de Bob, puede cifrar el mensaje que le quiere mandar con esa clave pública.

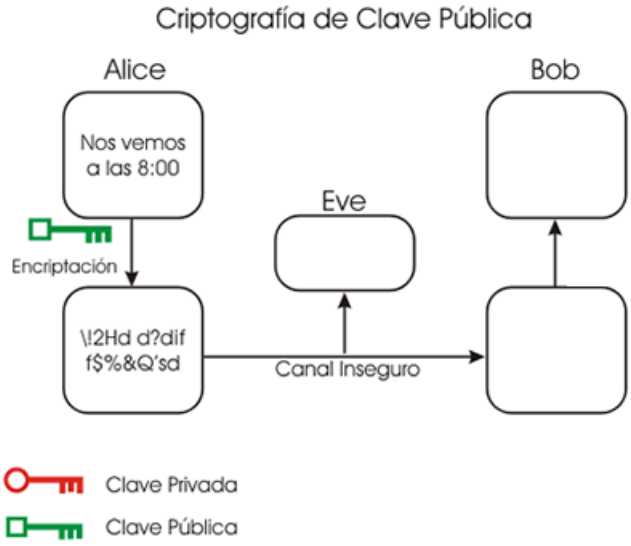


Figura 5.2: Bob envía su clave pública a Alice, quien cifra el mensaje con esa clave.

Después, Alice manda el mensaje cifrado con la clave pública a través del canal inseguro, de manera que ahora, si hubiese una tercera persona en la conversación, Eve, podría obtener el mensaje cifrado pero no la clave privada de Bob para descifrarlo.

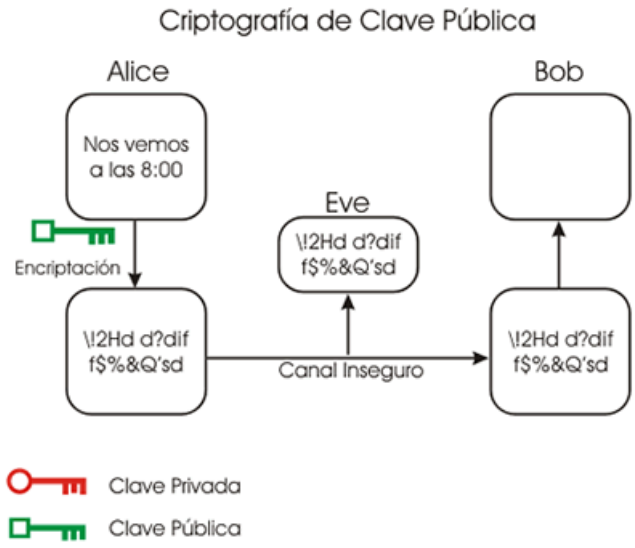


Figura 5.3: Alice envía el mensaje cifrado a través del canal inseguro.

En este momento, tanto Eve como Bob tendrían el mensaje cifrado. La diferencia, como hemos dicho, es que únicamente Bob tiene la llave privada (relacionada con la clave pública que cifró el mensaje) y que permite descifrar el mensaje.

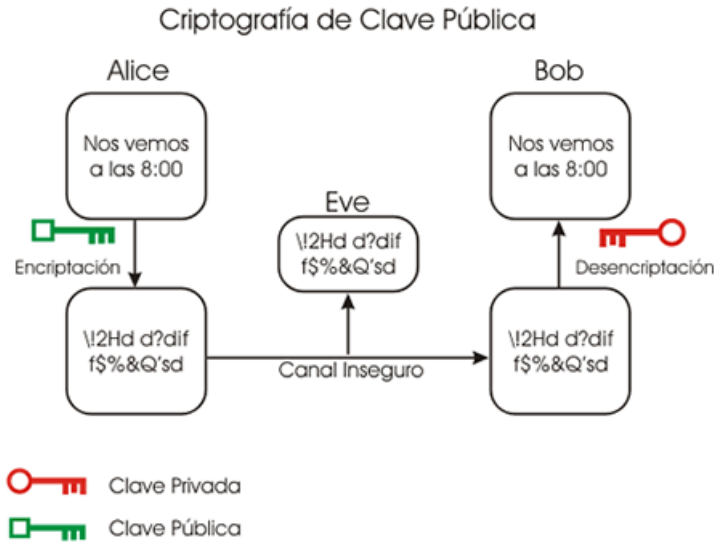


Figura 5.4: Únicamente Bob dispone de la clave privada para descifrar el mensaje.

Por tanto, estamos tratando de asegurar esa conversación entre dos amigos a través de un canal inseguro. Veamos como construir esta clave pública

y privada utilizando los códigos binarios Goppa.

### 5.1.2. Generando clave pública y privada

Para que Bob pueda construir la clave pública y la clave privada, tiene que elegir primero un polinomio irreducible arbitrario $g(z)$ de grado $t$ sobre $GF(2^m)$. El código binario Goppa quedará por tanto determinado, como ya dijimos, por éste polinomio y por un suconjunto $L = \{\alpha_1, \ldots, \alpha_n\} \subseteq GF(p^m)$ previamente establecido. Por tanto, tenemos el código binario Goppa $\Gamma(L, g(x))$.

Haciendo uso de esto, Bob puede calcular la matriz generadora $G$ del código Goppa, que será una matriz de tamaño $k \times n$. Después, Bob elegirá de forma aleatoria una matriz invertible, $S$, de dimensión $k \times k$ y una matriz de permutaciones $P$ de rango $n \times n$, lo que significa que la matriz $P$ tendrá un uno por fila y columna, siendo el resto de números ceros. Con estas matrices, se calcula $G' = SGP$, donde $G'$ será la matriz de codificación. De esta manera, hemos definido la clave pública con $G'$ y $t$.

Para el cálculo de la clave privada haremos uso del polinomio Goppa $g(x)$, de la matriz original $G$ y de las matrices $S$ y $P$ tales que $G' = SGP$.

Una vez que Bob manda su clave pública a Alice para comenzar la comunicación entre ellos, Alice genera un vector binario aleatorio $e$ de longitud $k$. De esta manera, Alice podrá codificar el mensaje $m = \{m_1, \ldots, m_k\}$ utilizando la clave pública de Bob de la siguiente forma,

$$y = mG' + e. \tag{5.1}$$

Veamos este primer paso con un gráfico de la comunicación.

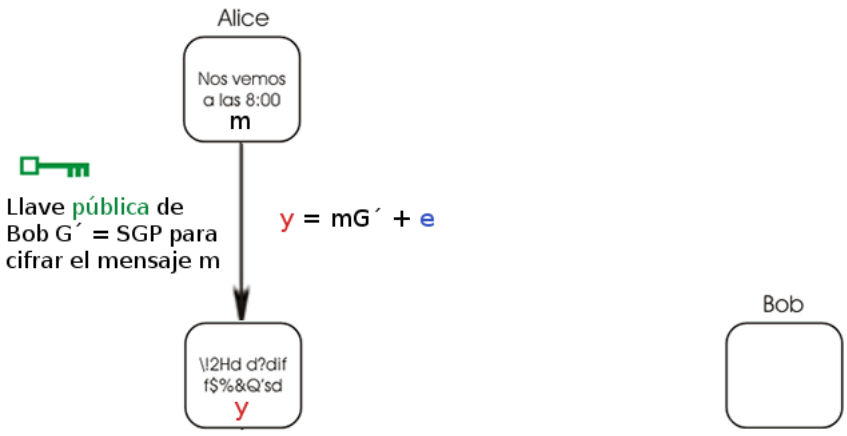


Figura 5.5: Alice cifra el mensaje m con la clave pública de Bob.

En ese momento, como ya hemos comentado antes, Alice envía el mensaje

cifrado o codificado $y$ a través el canal inseguro hasta que es Bob quien recibe ese mensaje cifrado.

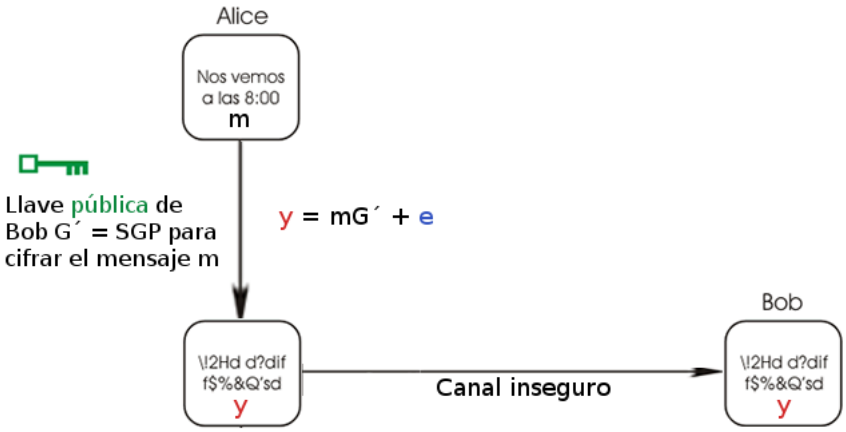


Figura 5.6: Alice envía el mensaje cifrado y a Bob a través el canal inseguro.

Bob utiliza su matriz de permutación $P$ para calcular,

$$y' = yP^{-1} = mG'P^{-1} + eP^{-1} = mSGPP^{-1} + e' = (mS)G + e'.$$

Y ahora Bob puede decodificar $y'$ encontrando $e'$, ésto se realiza aplicando el algoritmo de Patterson, visto en el punto 4.6.1. Una vez calculado, basta con que Bob resuelva $y - e' = mSG$ utilizando $S^{-1}$, que es una matriz conocida para poder recuperar el mensaje original resolviendo la ecuación,

$$m = m'S^{-1}. \tag{5.2}$$

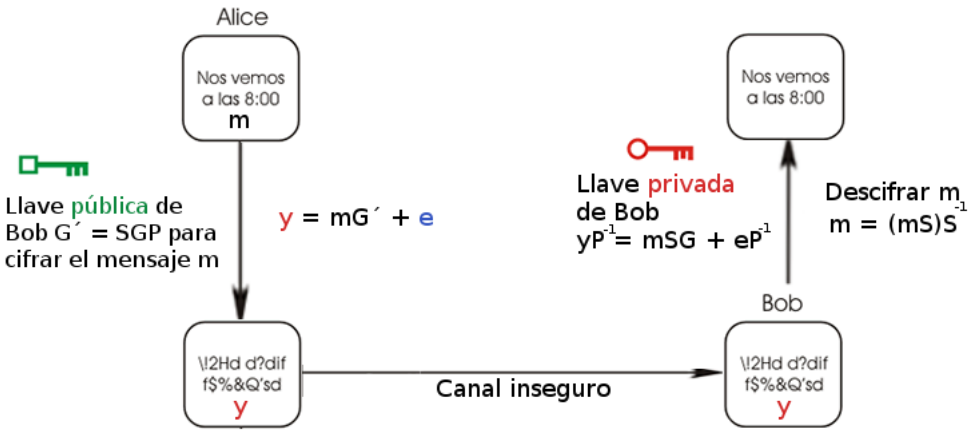


Figura 5.7: Bob descifra el mensaje con su clave privada.

### 5.1.3. Ejemplo del criptosistema de McEliece

Para ver un ejemplo del criptosistema de McEliece usaremos el mismo código de Goppa que vimos en la Sección 4.4.1. Para ello recordemos que tenemos nuestra matriz generadora:

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

Consideramos el problema de Alice y Bob antes explicado. Alice trata de mandar un mensaje de forma segura a Bob. Siguiendo los pasos del algoritmo que veíamos en la Subsección 5.1.2 Bob tendría que elegir de forma aleatoria una matriz invertible, $S$, de dimensión $k \times k$ y una matriz de permutaciones $P$ de rango $n \times n$.

En el ejemplo que estamos siguiendo, sabemos que $G$ es una matriz de rango $4 \times 12$ así que $S$ será una matriz $4 \times 4$ y la matriz de permutaciones $P$ será de rango $12 \times 12$. Elegimos las matrices:

$$S = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix},$$

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

Dadas estas matrices aleatorias, podemos calcular la matriz de codificación $G' = SGP$, de esta manera tendremos definida la clave pública.

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

Por tanto Bob ya dispone de su clave pública, que mandará a Alice para que pueda cifrar los mensajes con esa clave pública. Supongamos que el mensaje que quiere enviar Alice es:

$$m = (1, 0, 1, 0).$$

Lo primero será cifrar el mensaje utilizando la clave pública de Bob, es decir:

$$mG' = (1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0).$$

Y ahora Alice le añade un error aleatorio que llamaremos $e$ y que en nuestro ejemplo será:

$$e = (1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0).$$

es decir que finalmente, tenemos que el mensaje codificado será:

$$y = mG' + e = (0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0).$$

Ahora Alice manda el mensaje a Bob a través de un canal inseguro. El mensaje es recibido por Bob, que tratará de descifrarlo para obtener el mensaje original. Para ello, tendrá que calcular su matriz de permutación o clave secreta $P$ para poder calcular $yP^{-1}$.

Recordemos que teníamos la siguiente igualdad:

$$y' = yP^{-1} = mG'P^{-1} + eP^{-1} = mSGPP^{-1} + e' = (mS)G + e'.$$

En nuestro caso, calculamos directamente $yP^{-1}$:

$$yP^{-1} = (0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0) \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$= (0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0).$$

Observamos ahora el nuevo mensaje, nos damos cuenta de que los errores se han movido a la primera y sexta columna (contando con que comenzamos a contar desde la posición cero), entonces Bob utiliza su algoritmo de corrección de errores para corregirlos y por tanto nos queda:

$$mSG = (1,1,1,1,1,1,1,0,1,1,1,0).$$

Nuestro objetivo es recuperar el mensaje $m$, pero para ello utilizaremos $m = m'S^{-1}$ donde $m' = mS$. Sabemos por la sección 4.6 que podemos recuperar $mS$ a partir de la reducción por filas $[G^T|(mSG)^T]$. Resolviendo esta reducción obtenemos:

$$m' = mS = (1,1,0,1).$$

Basta con resolver por tanto $m = m'S^{-1}$ para recuperar finalmente el mensaje $m$ original.

$$m = m'S^{-1} = (1,1,0,1)\begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} = (1,0,1,0).$$

Veamos como utilizar el cifrado de McEliece con SageMath continuando el ejemplo que se planteó en la sección anterior 4.6.2. Para ello, una vez definidas las matrices $H$ (de control de paridad) y $G$ (generatriz del código) entonces vamos a definir el resto de matrices que utilizaremos $S$, $P$ y la matriz $G'$ que se corresponde con la clave pública de Bob.

```
sage: S = random_matrix(GF(2), N-m*rango)

sage: while (S.determinant()==0):
        S = random_matrix(GF(2), N-m*rango)

sage: rng = range(N)
sage: P = matrix(GF(2),N);

sage: for i in range(N):
        p = floor(len(rng)*random());
        P[i,rng[p]]=1; rng=rng[:p]+rng[p+1:];

sage: G_prima = S*G*P
```

Ahora definimos al igual que hicimos un vector sin codificar aleatorio del tamaño correcto.

```
sage: u = vector(K_,[randint(0,1) for _ in range(G_prima.nrows())])
```

Lo codificamos utilizando la clave pública de Bob, $G'$.

```
sage: c = u*G_prima
```

Y definimos y añadimos un vector de errores a la palabra codificada.

```
sage: e = vector(K_,N)
sage: e[8] = 1
sage: e[9] = 1
sage: y = c + e
```

Una vez codificado el mensaje, vamos a utilizar el algoritmo de descifrado de McEliece para recuperar el mensaje original detectando y corrigiendo los errores que hemos introducido.

```
sage: yP = y*(P.inverse())
sage: yd = decodePatterson(yP)
sage: corregido = (G.transpose()\yd)*S.inverse()
```

## 5.2. Ataques al criptosistema de McEliece

### 5.2.1. Algunos tipos de ataque

En criptografía existen multitud de ataques diferentes. Por ejemplo, un ataque por fuerza bruta es un tipo de ataque en el cual un enemigo o atacante intenta todas las posibles claves hasta dar con la correcta, la que descifraría el mensaje. Estos intentos son aleatorios y por tanto, conllevan una gran cantidad de computación para poder realizarse, pero ningún esfuerzo. En el caso de darse este ataque, cuanto mayor sea la longitud de nuestra clave, mayor será el tiempo que necesite este atacante hasta dar con ella, dado que tiene que ir probando con todas las claves posibles.

Otro posible ataque es un ataque menos aleatorio, sino más dirigido. Un ataque directo o un ataque a un mensaje en concreto es un ataque que intenta decodificar un mensaje sin necesidad de resolver el criptosistema por completo. Este tipo de ataque suele llamarse también ataque restructural puesto que el objetivo de un atacante, en nuestro caso, Eve, sería recomponer la estructura del mensaje (o al menos una parte) del mensaje original utilizando la clave pública, que para Eve, también es conocida.

Sin embargo, el ataque más efectivo contra el criptosistema McEliece es el ataque conocido como "decodificación por conjuntos de información". Muchos criptólogos han publicado algunas variaciones de este tipo de ataque pero la mayoría se basan en el ataque de tipo "Stern".

### 5.2.2. El ataque Stern

Uno de los ataques más efectivos contra los criptosistemas de McEliece es el conocido como "ataque de Stern" [4]. Este tipo de ataque se considera un ataque de "decodificación de un conjunto de información". Existen múltiples variantes para estos ataques.

Una de las primeras versiones se corresponde con el propio McEliece en su [19], aunque poco después surgieron las versiones de Leon [16] y de Stern [21].

Estos dos últimos se refieren al método o algoritmo de búsqueda de palabras codificadas con un peso de Hamming bajo que publicó Jacques Stern en 1989.

Supongamos que tenemos un código $\mathcal{C}$ sobre un cuerpo binario. Supongamos que $y$ tiene una distancia $w$ a una palabra codificada $c \in C$, entonces $y - c$ es un elemento con un peso $w$ del código $C + \{0, y\}$. Si por el contrario contamos con un código $\mathcal{C}$ sobre un cuerpo binario cuya distancia mínima es más grande que $w$ entonces un elemento de peso $w$, $e \in C + \{0, y\}$ no podría estar en $\mathcal{C}$ por lo que tendría que estar en $C + \{y\}$, en otras palabras, $y - e \in C$ con distancia $w$ a $y$.

Recordemos que en los criptosistemas de tipo McEliece tenemos que un texto cifrado $y$ tiene una distancia $t$ a la palabra codificada más cercana $c \in C$ con distancia mínima del código $\mathcal{C}$ de al menos $2t + 1$. Sabemos que un posible atacante posee la clave pública de McEliece, $G'$ y $t$. Además, Eve puede añadir $y$ a la lista de generadores de forma que pueda formar una matriz generadora de $C + \{0, y\}$. Es importante darse cuenta de que la única palabra codificada con peso $t$ es $y - c$, que es precisamente la palabra codificada que hemos encontrado con el ataque Stern.

Por tanto, Eve puede utilizar esta palabra codificada para encontrar $\mathcal{C}$ y resolver el problema para obtener así el mensaje original.

En un artículo publicado por Daniel J. Bernstein, Tanja Lange y Christiane Peters [3] se presentó una mejora del ataque Stern, siendo más efectivo que el anterior. En la documentación se mostró un ejemplo del ataque sobre un código binario Goppa $(1024, 524)$ demostrando que se puede romper el código en aproximadamente 1400 días haciendo uso de un ordenador con una capacidad de 2,4 GHz Intel Core 2 Quad Q6600. Además, como observación de este hecho, si en vez de utilizar un solo ordenador utilizásemos 200 ordenadores con estas características sólo nos llevaría una semana.

### 5.2.3. Búsqueda de las palabras de menor peso

Como hemos dicho, el ataque de Stern se basa en la búsqueda de palabras codificadas con un peso de Hamming bajo. En este caso, el ataque de Stern recibe dos argumentos, el primero de ellos sería un entero $w$ mayor o igual que cero y el segundo sería una matriz de control de paridad $H$ de tamaño $(n - k) \times n$ para un código $\mathcal{C}$ de tipo $(n, k)$.

En ese momento se seleccionan de forma aleatoria $n - k$ columnas de las $n$ columnas que contiene la matriz $H$ elegida. Se selecciona también un subconjunto $Z$ de tamaño $l$ de esas $n - k$ columnas. Se particionan las $k$ columnas restantes en dos subconjuntos $X$ e $Y$ haciendo que cada columna decida de forma independiente y uniforme si estar en $X$ o en $Y$.

El siguiente paso del ataque de Stern es buscar las palabras codificadas que tienen exactamente $p$ bits en el subconjunto $X$, $p$ bits en el subconjunto $Y$, 0 bits cero en $Z$ y $w - 2p$ bits en las columnas restantes (todos ellos distintos de cero). En este caso, $p$ será un parámetro del algoritmo que se podrá optimizar. En el caso de que no hubiese palabras codificadas, Stern comenzaría de nuevo tomando otra selección aleatoria de columnas.

La búsqueda de palabras codificadas se divide también en varios pasos. Primero se aplican operaciones elementales sobre $H$ para que las $n-k$ columnas se conviertan en la matriz identidad. Cuando la matriz que hemos tomado no es invertible, entonces este paso fallaría y tendríamos que volver a elegir como en el paso anterior. Sin embargo, Stern asegura que existe una submatriz invertible, evitando así el reinicio del algoritmo. Una vez que tenemos ya la submatriz $H$ de tamaño $(n-k) \times (n-k)$ como la matriz identidad, el conjunto $Z$ de $l$ columnas se corresponde con las $l$ filas. Para cada subconjunto $A \subset X$ de tamaño $p$, Stern calculará la suma de las columnas de $A$ para cada una de las $l$-filas, obteniendo así un vector $\pi(A)$ compuesto por $l$-bits. De manera análoga, se obtendría un vector $\pi(B)$ para un subconjunto $B \subset Y$ de tamaño $p$.

Por último, para cada colisión $\pi(A) = \pi(B)$, Stern calcula la suman de las $2p$ columnas en $A \cup B$. Esta suma será un vector compuesto de $(n-k)$ bits. Si esta suma tiene un peso $w-2p$ entonces el algoritmo obtiene un cero añadiendo las $w-2p$ columnas en la submatriz de tamaño $(n-k) \times (n-k)$. Estas $w-2p$ columnas, junto a $A$ y $B$, forman una palabra codificada de peso $w$.

### 5.2.4. Primer ataque con éxito

En Junio de 2008 se realizó el primer ataque con éxito del ataque de Stern sobre un criptosistema de tipo McEliece [15]. En dicho ataque se utilizaron 200 ordenadores con un total de 300 núcleos. Los cálculos llevaron unos 90 días en completarse (finalizaron a principios de Octubre).

En este primer ataque se fueron afinando el valor de los parámetros y ya en los siguientes ataques se partió de los más óptimos. Estos parámetros permitían disminuir el coste de cálculo de una manera considerable.

## 5.3. Seguridad del criptosistema de McEliece

Supongamos que Eve, el atacante que se sitúa en medio de la conversación entre los amigos Alice y Bob, conoce el tipo de criptosistema que se está utilizando. Esta suposición se propone para medir correctamente la seguridad de un criptosistema y proviene del "Principio de Kerckhoff" que dice lo siguiente: "Para ser capaces de determinar la seguridad de un criptosistema uno debería asumir siempre que el enemigo conoce el método que se está utilizando." [9]

La seguridad del criptosistema de McEliece se basa en la propia dificultad que tiene decodificar $y'$ para conseguir $m'$. Eve necesitaría mucho tiempo para lograr separar la matriz $G$ de $G'$ puesto que tendría que conocer la inversa de la matriz $S$, la cual no es pública. Además, Eve tampoco conoce la matriz de permutaciones $P$, por tanto la búsqueda de $y'$ se vuelve incluso más compleja.

Algunas familias de códigos, como los códigos Reed-Solomon, muy utilizados para la recuperación de información en CD-Roms, han sido "rotos" puesto que se puede recomponer la estructura del código a partir de la matriz $G'$. Los códigos de Goppa, hasta el momento, han sido capaces de resistir este tipo de ataques.

De hecho, es importante recalcar que aunque el polinomio de Goppa está definido sobre $\mathbb{F}_q^n$, el código de Goppa está definido sobre $\mathbb{F}_q$. De hecho, se piensa que esta "extensión de cuerpos oculta" seguramente es la que hace que estos códigos no se hayan roto aún.

Puesto que buscamos conseguir una matriz que parezca aleatoria a primera vista y sea difícil de obtener un método de decodificación a partir del método de codificación, multiplicamos las matrices $SG$ y después, a esa matriz, la multiplicamos por la matriz de permutaciones $P$. De esta manera, se complica el descifrado de la clave privada a partir de la clave pública (que sabemos Eve conoce puesto que es pública).

Todo esto nos lleva a pensar que necesitamos un código lo más largo posible para que sea capaz de esconder el código binario Goppa con la matriz generadora que hemos formado, $G'$. Aquí es donde más problemas podemos tener puesto que, si hacemos el código de un tamaño muy grande, complicaríamos el propio uso del sistema. Según vaya avanzando la tecnología y exista una mayor potencia de cálculo, este tipo de criptosistemas será más y más útil.

Una vulnerabilidad importante en este criptosistema se produce cuando se manda repetidamente el mismo mensaje. El problema está en que al mandar el mismo mensaje se generan dos o más mensajes cifrados y puesto que la localización de los errores no tiene por qué ser la misma, Eve, el atacante, podría comparar esos mensajes hasta potencialmente, encontrar el mensaje original.

En el paper original publicado sobre el criptosistema de McEliece [18], el propio McEliece propuso un código binario Goppa $[1024, 524]$, que podía corregir hasta 50 errores. Desde entonces, se han propuesto diferentes variaciones del criptosistema, la mayoría utilizando códigos diferentes. Sin embargo y pese a los intentos, la mayoría de ellos ha sido más propenso a fallos y menos seguro que el original.

### 5.3.1. Defensa del criptosistema de McEliece

Una vez visto los ataques más comunes al criptosistema de tipo McEliece y la seguridad de la que dispone, se propone en esta sección algunos cambios en el algoritmo para reforzarlo [4] y convertirlo en un sistema más seguro y capaz de defenderse de una manera más eficiente de este tipo de ataques antes comentados.

1. **Aumentando la longitud del código:** Una de las opciones que se

pueden implementar y que parece la más obvia es aumentar la longitud del código que se utiliza en el criptosistema, es decir, aumentar el valor de $n$. Una de las posibilidades y puesto que $n$ no tiene por qué ser una potencia de 2 es permitirle valores de $n$ entre dos potencias de 2, de esta manera se optimiza mejor el tamaño de la clave pública del criptosistema de McEliece.

2. **Optimización de los parámetros del algoritmo:** Otra de las propuestas es la optimización de los parámetros del algoritmo se propone utilizar códigos de Goppa de longitud 2048. La clave pública en este caso estaría formada por 520047 bits.

3. **Aumentando el número de errores que se pueden corregir:** Una buena opción para securizar el criptosistema de tipo McEliece podría ser también aumentar la cantidad de errores que pueden corregir, mejorando los algoritmos de decodificación de los códigos de Goppa. De esta manera, en vez de corregir $t$ errores, se podrían llegar a corregir un número mayor. Con esto en mente, un posible atacante tendría más problemas para enfrentarse a ello y ser capaz de decodificar ese "extra" de errores que hemos introducido.

# Capítulo 6

# Conclusiones y trabajo futuro

## 6.1. Conclusiones finales

Una vez finalizado el Trabajo de Fin de Grado podemos comprobar que hemos sido capaces de cumplir los objetivos que se establecieron al inicio de este proyecto. El estudio y compresión de la Teoría de códigos correctores suponía un reto complejo pero muy apetecible tras estos años de carrera donde he buscado enlazar la parte matemática con la parte informática enfocada a la seguridad de los sistemas. La criptografía y en especial la criptografía cuántica eran una "asignatura pendiente" y que estaba deseando poder investigar por mi cuenta. Una de las partes más importantes del trabajo ha sido lograr entender el álgebra que tiene detrás la teoría de códigos correctores y la criptografía de clave pública, es esa base matemática la que desde un principio traté de entender para poder avanzar de una manera más eficiente. Al igual que suele ocurrir en este tipo de investigaciones, es complicado tratar la parte informática sin tan siquiera entender correctamente la parte matemática que lleva debajo, por ello no fue hasta más adelante cuando comencé a trabajar en el desarrollo de los algoritmos y de los ejemplos usando los software SageMath y GAP. Para poder hacer esto también tuve que aprender Python, lenguaje base de SageMath y que actualmente está considerado como uno de los lenguajes de programación con más proyección dado su gran uso en la educación y en el mercado laboral.

Una vez establecida la base matemática e informática era el momento de entrar más en materia presentando los códigos de Goppa, un concepto totalmente nuevo y sobre el cual nunca había podido indagar. Ponencias como las de Joachim Rosenthal [11], Doctor en la Universidad de Zürich, en la Universidad de Granada me ayudaron a entender algunos detalles que no había logrado captar tras el estudio y ver una visión mucho más amplia de las posibilidades que ofrecía esta teoría de códigos correctores.

El estudio de los criptosistemas de tipo McEliece, así como los algoritmos de decodificación y de corrección de errores como el algoritmo de Patterson fue una continuación de esta base matemática, pero sin embargo sí considero que fue un paso adelante el desarrollo de estos algoritmos en el lenguaje de programación Python, donde finalmente fui consciente de la importancia de los criptosistemas de tipo McEliece ante los posibles ataques cuánticos, es decir, ver dónde reside la fuerza que les permitiría aguantar.

Una vez estudiado este tipo de criptosistema el siguiente paso era obvio. Había que ser capaz de entender los diferentes ataques que podrían "romperlo". Por ello se estudió el ataque de Stern, uno de los ataques más conocidos y más eficientes contra el sistema de McEliece. Este ataque ha ido evolucionando con el paso del tiempo, al mismo tiempo que surgían nuevas variantes al propio criptosistema. Éste análisis ha permitido ser consciente de la "raíz"de las vulnerabilidades que tiene McEliece dando paso a la parte más complicada cuando se trata de seguridad de sistemas, la mejora del sistema para que la defensa sea más eficiente. Este paso no suele ser fácil, de hecho, es común que los cambios o mejoras en los criptosistemas vengan una vez que un atacante ha detectado un fallo de seguridad. Por ello, en la última parte del trabajo se han propuesto algunas posibles mejoras en el criptosistema de tipo McEliece, que, aunque no convertirán el sistema en invulnerable, complicarán el trabajo de los atacantes.

Desde mi punto de vista, el presente Trabajo de Fin de Grado no sólo ha servido para aprender una teoría tan importante como la de códigos correctores, presente en muchas situaciones cotidianas en las cuales no somos conscientes de este tipo de algoritmos, sino también como un punto de inicio hacia el estudio y profundización en este campo. Conviene recordar que aún es pronto para hablar de computación y criptografía cuántica pero el avance de la tecnología es tal que no tardará demasiado en invadir nuestro día a día y, como se suele decir en la seguridad informática, será de vital importancia prevenir este tipo de ataques antes de que pudiesen ocurrir para poder minimizar las consecuencias.

## 6.2. Trabajo futuro

Como hemos comentado, el camino de la criptografía y la computación cuántica no ha hecho más que comenzar. Se trata de un campo en pleno desarrollo y que sin duda alguna dará que hablar en un futuro no muy lejano.

Es importante que los investigadores en este campo sean capaces de encontrar soluciones, al menos momentáneas ante la posibilidad de que surjan los ataques cuánticos, dado que la potencia de éstos sería capaz de romper muchos algoritmos de cifrado que existen y cuyo uso está muy extendido en nuestra vida como el algoritmo RSA.

En este sentido, hemos estudiado que el criptosistema de tipo McEliece es un fuerte candidato para solucionar este problema. Pero la pregunta que surge en estos momentos es la siguiente, ¿cuánto tiempo será capaz de resistir? Con los recursos que existen en la actualidad parece que aguantaría, pero es cuestión de tiempo que los atacantes tengan los medios necesarios para encontrar el fallo de seguridad necesario.

Otra de las opciones que habría que barajar una vez entendido el problema que existe y las posibles soluciones con las que contamos a día de hoy sería entrar de lleno en el estudio de algoritmos de cifrado cuánticos, es decir, una nueva serie de algoritmos de cifrado que utilizan la computación cuántica a su favor.

Algunos estudios predicen que la computación y la criptografía cuántica serán más comunes en unos 15 años, pero no existe garantía alguna de que tarden tanto tiempo. ¿Por qué no aprovechar ese tiempo para centrarse en RSA y ECDSA? ¿Por qué no simplemente cambiar a McEliece una vez se anuncie la llegada de los ordenadores cuánticos? Estas son algunas de las razones por las que la comunidad está cada vez más centrada en la criptografía cuántica. [5]

- Se necesita tiempo para mejorar la eficiencia de la criptografía post-cuántica.
- Se necesita tiempo para comprobar la eficacia y construir una confianza en estos cifrados post-cuánticos.
- Se necesita tiempo para mejorar la estabilidad de la criptografía post-cuántica.

Es decir, aún no estamos preparados para dar el salto a esta nueva tecnología. Tal vez nunca llegue esta criptografía cuántica, tal vez no sea necesaria esta preparación o tal vez nunca se anuncie la construcción de un gran ordenador cuántico. Pero desde luego, si no hacemos nada y no nos preparamos para la posible llegada de esta nueva tecnología, traerá consigo consecuencias graves. Algunas de las situaciones en las que podría aplicarse sería en la lectura y manipulación de firmas electrónicas o de ficheros confidenciales cifrados. Una vez "rotos" sus cifrados, no habría manera de detener a un posible atacante de su lectura y uso.

La criptografía cuántica expande la idea de clave pública con un transmisión infinita de información entre las dos personas que tratan de comunicarse. Para ello sería necesario que estas dos personas tuviesen claves secretas de un número de bits impredecible. Una de las claves de esta nueva criptografía es que ya no sólo usaría funciones matemáticas para las claves sino que también se utilizarían técnicas y leyes físicas para ello. Sin embargo, el coste de implementación de la criptografía cuántica es altísimo en comparación con los sistemas y los cifrados que tenemos actualmente puesto que además la

llegada de este tipo de tecnología tendría que venir acompañada de nuevo hardware que fuese capaz de soportarlo, aumentando también los costes en este aspecto.

Por tanto, la criptografía cuántica tendrá de momento que enfrentarse a los diseños de dichos sistemas cuánticos, al criptoanálisis de los mismos y a la búsqueda de parámetros que encajen en los diferentes modelos que se planteen.

# Bibliografía


[1] Ashley Valentijn. *"Goppa Codes and Their Use in the McEliece Cryptosystems"*, Syracuse Universitym, SURFACE Honors Program Project. Spring 5-1-2005

[2] Berlekamp.E, *Goppa Codes*, IEEE Transactions on Information Theory 1975.

[3] Berlekamp.E, Tanja Lange and Christiane Peters. *"Attacking and Defending the McEliece Cryptosystem"*, Post-quantum Cryptography: Second International Workshop, PQCrypto 2008 Cincinnati, OH, USA, October 17-19,2008.

[4] Bernstein, Daniel J. Lange, Tanja. Peters, Christiane *"Attacking and defending the McEliece cryptosystem"*Department of Mathematics, Statistics, and Computer Science (M/C 249) University of Illinois at Chicago, Chicago, IL 60607–7045, USA

[5] Bernstein, Daniel J. Buchmann, Johannes. Dahmen, Erik. *"Post-Quantum Cryptography"*Editorial Springer

[6] Canteaut, Anne, and Chabaud, Florent. *"Cryptoanalysis of the original McEliece Cryptosystem"*, Advances in Cryptology, ASIACRYPT '98 volume 1514 of Lecture Notes in Computer Science, pages 187-199. Spriinger, Berlin, 1998.

[7] Engelbert, Daniela, Raphael Overbeck and Arthur Schmidt. *"A Summary of McEliece-Type Cryptosystems and their Security"*, Journal of Mathematical Cryptology 1,199.

[GAP] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.8.7*; 2017, `http://www.gap-system.org`.

[GUAVA] Baart, R., Boothby, T., Cramwinckel, J., Fields, J., Joyner, D., Miller, R., Minkes, E., Roijackers, E., Ruscio, L. and Tjhai, C., GUAVA, a GAP package for computing with error-correcting codes, Version 3.12 (2012), (Refereed GAP package), `http://www.southernct.edu/~fields/Guava/`.

[8] Golay, Marcel J. E. *"Signal Corps Engineering Laboratories at Fort Monmouth."*

[9] IEEE Digital Library, *Kerckhoffs' principle for intrusion detection*, `http://ieeexplore.ieee.org/document/6231360/?reload=true`

[10] José Ignacio Farrán Martín *"Numerical Semigroups and Codes"*, International Meeting on Numerical Semigroups, Universidad de Valladolid. Trabajo conjunto con M.Delgado, P.A. García-Sánchez y D.Llena

[11] Joachim Rosenthal, University of Zürich. *"Code Based System for Post-Quantum Cryptography"*, University of Granada speech, February 16,2017

[12] Joachim Rosenthal, University of Zürich. *"Bases de Gröbner. Aplicaciones a la codificación algebraica"*, Cap.Códigos Correctores de Errores. Mérida, Venezuela. 2 al 7 de Septiembre de 2007.

[13] J.H.Van Lint *"Introduction to Coding Theory. Third Edition"*, Chapter 9. Goppa Codes. Edit. Springer. Graduate texts in mathematics.

[14] J.H.Van Lint *"Introduction to Coding Theory and Algebraic Geometry."*, Chapter 1. Coding Theory. Edit Birkauser DMV Seminar Band12.

[15] Lange, Tanja. *"Post-Quantum Cryptography"*Technische Universiteit Eindhoven, 17 December 2008

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

[17] Martín Garcia, Lorenzo J. *"Apuntes de la asignatura de Criptografía y Teoría de Códigos Correctores de la Universidad de Valladolid, Campus de Segovia."*

[18] McEliece, R.J. *"A Public-Key Cryptsystem Based on Algebraic Coding Theory"*, Jet Propulsion Laboratory DSM Progress Report 42-44 N.p. Web 12 Jan.2015

[19] McEliece, Robert J. *"A public-key cryptosystem based on algebraic coding theory."*Technical report, CA, 1978.

[S+09] *SageMath, the Sage Mathematics Software System (Version x.y.z)*, The Sage Developers, YYYY, `http://www.sagemath.org`.

[20] Shannon, Claude E. (July 1948) *"A Mathematical Theory of Communication."*

[21] Stern, Jacques. *"A method for finding codewords of small weight."* In Gerard D. Cohen and Jacques Wolfmann, editors, Coding Theory and Applications, volume 388 of Lecture Notes in Computer Science, pages 106–113. Springer, 1988.