Universidad de Valladolid

Escuela de Ingeniería Informática

Trabajo Fin de Grado

Grado en Ingeniería Informática de Servicios y Aplicaciones

Criptografía Post-cuántica: Implementación de McEliece y una nueva versión

Autor: David Moreno Centeno

Tutor: José Ignacio Farrán Martín

Agradecimientos

A los dos tutores de ambos trabajos fin de grado, Diego Ruano y Jose Ignacio Farrán por todo el tiempo empleado a lo largo de este año para guiarme y ayudarme a realizar el proyecto, y a mis padres, que me han dado el apoyo y ánimos necesarios a lo largo de la realización de todo el documento y también durante toda la carrera.

Resumen

La seguridad empleada en prácticamente todas comunicaciones realizadas actualmente utiliza una criptografía de clave pública o híbrida mediante el uso de sistemas criptográficos como el RSA, Gamal o de curva elíptica entre otros. Dichos sistemas aunque son actualmente seguros, en cuanto seamos capaces de construir un ordenador cuántico, se conoce un algoritmo que permite romperlos en tiempo polinómico. Por ello, actualmente se están estudiando distintos criptosistemas que sean resistentes a ataques realizados por un ordenador cuántico.

El presente documento se centra en el criptosistema de McEliece, el cual es uno de los criptosistemas resistente frente a estos ataques. En adicción se muestran los códigos Reed-Solomon, Goppa y de producto de matrices, que se pueden emplear, entre otros, para construir el criptosistema. Después vemos un posible ataque contra el criptosistema construido a partir de un código Reed-Solomon.

Por último se muestra un método innovador que nos permite reducir notablemente el tamaño de las claves del criptosistema mediante el empleo de códigos de producto de matrices.

Abstract

The security used in almost all communications currently performed a public key cryptography or hybrid through the use of cryptographic systems such as RSA, Gamal or elliptical curve among others. These systems although they are currently safe, as soon as we are able to build a quantum computer, it is known an algorithm that can break them in polynomial time. For this reason, different cryptosystems that are resistant to attacks carried out by a quantum computer are currently being studied.

This paper focuses on the McEliece cryptosystem, which is one of the cryptosystems resistant to these attacks. In addition, Reed-Solomon, Goppa and matrix product codes are shown, which can be used to build the cryptosystem. Afterwards, a possible attack against the cryptosystem built from a Reed-Solomon code is shown.

Finally, it shows an innovative method that allows us to significantly reduce the size of the cryptosystem keys by using matrix product codes.

Índice general

Capítulo 1

Introducción

Cuando una persona desea transmitir un mensaje a una o varias personas a través de un canal de comunicación, debe tener en cuenta que existe la posibilidad de que una persona ajena a la comunicación intente obtener dicho mensaje.

Para evitar este problema surge la criptografía, la cual se centra en el estudio de métodos que nos permiten convertir un mensaje de texto plano en una secuencia aleatoria de caracteres de un alfabeto para que dicho mensaje resulte ilegible para cualquier persona ajena a la comunicación que capte el mensaje.

Para ello, se puede emplear tanto la codificación como el cifrado, los cuales se centran en que la información sea transmitida de forma confidencial. Se pueden distinguir dos tipos de codificaciones en función de la longitud que tienen sus palabras; si todas las palabras de un código tienen la misma longitud se denomina código en bloque y si no lo son se denomina código de longitud variable.

Por tanto, aunque realizar la codificación o cifrado de un mensaje tiene como principal objetivo garantizar la seguridad de la comunicación establecida, también nos permite detectar y corregir cierta cantidad de errores que pueden surgir durante la transmisión del mensaje, si empleamos cierto tipo de códigos en bloque o comprimir la información si en su lugar empleamos códigos de longitud variable, donde para ello tenemos que codificar los caracteres que se emplean de forma mas frecuente con las palabras del código de menor longitud.

Los primeros sistemas criptográficos, conocidos como clásicos o como cifrados de clave privada emplean una clave que debe conocer tanto el emisor como el receptor antes de comenzar la comunicación, ya que emplean dicha clave tanto para cifrar como para descifrar los distintos mensajes. Por ello, no eran suficientes para el intercambio de mensajes de forma segura mediante un canal inseguro ya que, dicha clave debe cambiarse cada cierto tiempo y para realizar esto de forma segura, es necesario reali-

zar una comunicación directa, para evitar que una persona externa a la comunicación obtenga dicha clave privada.

En consecuencia, surgen los sistemas criptográficos de clave pública en 1976 gracias a Whitfield Diffie y Martin Hellman, los cuales, se basan en la teoría de la complejidad computacional. Estos sistemas criptográficos permiten que el descifrado sea imposible de forma práctica en un tiempo relativamente corto a no ser que el receptor conozca información que simplifique dicho proceso de descifrado, para ello se emplean funciones matemáticas cuyo cálculo es sencillo pero el cálculo de la función inversa es muy complejo. Dichas funciones son conocidas como funciones trampa o funciones de una vía. El principal ejemplo de función trampa es la función que se encarga de multiplicar dos números primos. Dicha operación se realiza rápidamente, sin embargo realizar la función inversa, es decir, obtener los dos números primos que multiplicados nos proporcionan un número dado tiene una complejidad exponencial.

En los sistemas de clave pública cada persona de la comunicación tiene un par de claves:

- La clave pública y por tanto, conocida por cualquier persona que lo desee, la cual deberán emplear para cifrar las personas que deseen enviar un mensaje a la persona a la que pertenece dicha clave pública.
- La clave privada, solo conocida por la persona a quien pertenece, la cual permite descifrar los mensajes que hayan sido cifrados con la clave pública que le corresponde.

Estos sistemas de clave pública deben cumplir las llamadas condiciones de Diffie-Hellman, las cuales se corresponden con que el cálculo de las claves tanto pública como privada, así como el proceso de cifrado tiene que ser computacionalmente sencillo, mientras que el proceso de descifrado deberá ser computacionalmente imposible si no se conoce la clave privada. Ademas, debe ser computacionalmente imposible obtener la clave privada a partir de la pública y también dichos sistemas deben ser resistentes frente ataques a texto claro elegido, es decir, ataques en los que se puede obtener texto cifrado correspondiente a un texto plano deseado y mediante un análisis de dicho texto poder descifrar cualquier otro mensaje cifrado, ya que la clave pública para cifrar es conocida por cualquier persona.

Una desventaja de los cifrados asimétricos en comparación con los cifrados simétricos es que para tener una seguridad semejante es necesario emplear claves mas extensas. Debido a esto, los cifrados asimétricos son mucho mas costosos computacionalmente y por ello, se suelen emplear los cifrados asimétricos dentro de una comunicación únicamente para el intercambio de una clave simétrica, que se genera y emplea

solo en esa comunicación, la cual permite una mayor rapidez en la transmisión de información manteniendo el nivel de seguridad. Este método se conoce como criptografía híbrida, debido al empleo de ambos tipos de cifrados.

Actualmente, se emplea la criptografía híbrida, para la cual se utilizan principalmente como sistemas criptográficos de clave pública el RSA (Rivest, Shamir y Adleman), el Gamal, el cual se basa en el problema matemático del logaritmo discreto, y el de curva elíptica. Anteriormente también se empleaba el criptosistema de Merkle-Hellman basado en el problema de la mochila aunque se dejó de emplear rápidamente debido a que fue criptoanalizado exitosamente por Shamir en 1982 [2].

Por tanto, todas las comunicaciones que realizamos actualmente dependen de la seguridad de dichos sistemas. Debido a esto, debemos hacernos la pregunta, ¿Son actualmente seguros estos sistemas criptográficos y lo seguirán siendo en un futuro próximo?

La respuesta es que actualmente son seguros. Sin embargo, el algoritmo cuántico de Shor, desarrollado por Peter Shor en el año 1998, es capaz de criptoanalizar exitosamente prácticamente todos los sistemas criptográficos de clave pública que se están empleando ya que dicho algoritmo permite descomponer un número en sus factores primos en tiempo polinomial. Por tanto, si en un futuro próximo somos capaces de crear un ordenador cuántico suficientemente potente, dichos sistemas de clave pública quedarían obsoletos. Por otro lado, los sistemas de clave privada son seguros frente a ataques que empleen el algoritmo de Shor.

Debido a esto, surge la necesidad de buscar otros sistemas de clave pública que sean capaces de resistir ataques realizados por un ordenador cuántico.

De momento se cree que los sistemas basados en Hash como por ejemplo el hash-tree de Merkle [3], los basados en teoría de códigos como por ejemplo el McEliece, o los basados en ecuaciones cuadráticas multivariantes como por ejemplo el de Patarin [4] son resistentes frente ataques realizados tanto por ordenadores clásicos como por ordenadores cuánticos, esto se debe a que para dichos sistemas no se ha encontrado de momento ninguna forma de aplicar el algoritmo de Shor.

Por tanto, deberíamos pensar porque no se están empleando ya estos criptosistemas si parecen ser resistentes frente a ataques realizados por ordenadores cuánticos. La respuesta se encuentra en que cada uno de ellos presenta algún tipo de desventaja que no presentan los sistemas de clave pública que se están empleando actualmente.

Por ejemplo, para el sistema de Mceliece, que es el sistema basado en la teoría de códigos en que centraremos nuestro estudio en este trabajo fin de grado, su principal inconveniente es que presenta un tamaño de claves mucho mas elevado que las del

sistema RSA, lo cual incrementaría notablemente el tamaño que necesitamos emplear para guardar las distintas claves que vamos a usar.

1.1. Motivación

Como ya hemos mencionado, en cuanto seamos capaces de construir ordenadores cuánticos, todos los métodos implementados actualmente que permiten realizar cualquier comunicación de forma segura quedarán obsoletos.

Debido a esto, se realiza en este trabajo fin de grado un estudio del criptosistema de McEliece, al ser un método alternativo a los empleados actualmente, resistente frente al algoritmo de Shor, el cual permite paliar la situación crítica en la que nos encontraríamos si no fuéramos capaces de realizar ninguna comunicación segura.

El principal aspecto que ha proporcionado el interés para la realización de este documento es la gran relevancia que presenta la seguridad en cualquier tema que se desee realizar, junto con la oportunidad de poder ampliar mis conocimientos sobre dicho aspecto, realizando un estudio de la teoría de códigos, los cuales hacen referencia a una serie de conceptos completamente nuevos para el alumno.

1.2. Objetivos

Ahora mostramos una serie de objetivos que se pretenden alcanzar tras la realización del presente trabajo fin de grado.

- Estudio de la teoría básica de códigos lineales.
- Aprender los conocimientos básicos del lenguaje de programación Python necesarios para entender el funcionamiento detallado del lenguaje de programación *SageMath*, el cual nos permite realizar distintas operaciones matemáticas, que resultarían costosas de calcular de forma escrita.
- Conocimiento de una serie de códigos lineales como los códigos Reed-Solomon y Reed-Solomon generalizados, códigos cíclicos, códigos de Goppa y códigos de producto de matrices, entre otros que emplearemos en menor medida como por ejemplo los códigos punteados y los códigos recortados. Además, para todos ellos se realiza un estudio en el lenguaje *Sagemath* para saber que códigos se encuentran ya implementados y cuales es necesario implementar completamente:

  - Los códigos lineales, cíclicos, Reed-Solomon y Reed-Solomon generalizados ya se encuentran implementados completamente en *Sage*, por lo que únicamente se realiza un estudio de todas las funciones implementadas para estos códigos.
  - Los códigos de Goppa no están implementados en *Sage* actualmente, por tanto realizamos una implementación que nos permita construir los códigos de Goppa, así como un conjunto de funciones básicas para el empleo de estos códigos, como por ejemplo una función que nos permita codificar usando estos códigos, otra que nos permita incorporar cierta cantidad de errores a un mensaje, así como otra que nos permita tanto corregir cierta cantidad de errores como descodificar los mensajes codificados. También se implementan un conjunto de funciones sencillas que permitan recuperar propiedades básicas del código construido, como su longitud, dimensión, mínima distancia, matriz generatriz y de control así como el cuerpo que emplea el código.
  - Los códigos de producto de matrices tampoco cuentan con ninguna implementación en *Sage*, por ello también los generamos completamente, de forma que seamos capaces tanto de construirlos como de utilizarlos para codificar, descodificar y para mostrar una versión innovadora que nos permita reducir el tamaño de las claves empleadas por el criptosistema de McEliece. De forma semejante a los códigos de Goppa, también se implementaran un conjunto de funciones para recuperar las propiedades básicas de estos códigos.

- Entender el criptosistema de McEliece, sus ventajas, desventajas así como realizar tanto el cifrado como el descifrado. Dicho criptosistema también es implementado completamente mediante el empleo de *Sage*, donde construiremos funciones que nos permitan tanto cifrar como descifrar cierto tipo de mensajes. De igual, forma que para los códigos de Goppa y de producto de matrices, dicho criptosistema también cuenta con un conjunto de funciones para recuperar sus principales propiedades.

- Analizar la seguridad del criptosistema de McEliece dependiendo del tipo de códigos con el que se implementan. Mostramos un criptoanalisis exitoso del McEliece con códigos Reed-Solomon generalizados, conocido como ataque por filtración, el cual también implementamos mediante el empleo de *Sage*.

- Explicar un método innovador que permite implementar el criptosistema de McEliece reduciendo de notablemente el tamaño de las claves que es necesario emplear. Este método también se implementa en *Sage* mediante la construcción de dos funciones.

1.3. Organización de la memoria

El presente documento se estructura al igual que el trabajo fin de grado de matemáticas [1], mediante el empleo de un conjunto de capítulos. En primer lugar, se muestra en el capítulo 2 la metodología de trabajo empleada a lo largo de la realización de todo el documento, viendo las distintas etapas por las que el proyecto a ido evolucionando, mostrando información de las herramientas utilizadas y realizando un presupuesto en función de la estimación de horas empleadas.

En el capítulo 3 se introduce el lenguaje de programación que empleamos, llamado *Sage* y se realiza un estudio de los distintos códigos que emplearemos a lo largo del documento que ya se encuentran implementados en *Sage*, mostrando para cada uno de ellos, una serie de funciones que emplearemos, así como la teoría básica para comprender su uso.

El capítulo 4 introduce los códigos de Goppa, cuyo empleo es esencial en prácticamente todos los capítulos siguientes. En dicho capítulo mostramos su teoría mas elemental y al menos un método de codificación y descodificación eficiente para dichos códigos, para los cuales nos restringimos a los códigos construidos sobre cuerpos binarios. También iremos mostrando junto con la teoría, el código implementado en *Sage*, el cual explicaremos detalladamente.

Ahora procedemos a introducir el criptosistema de McEliece en el capítulo 5, para el cual también vamos intercalando la teoría básica junto con el código desarrollado.

En el capítulo 6, mostramos un ataque eficiente en tiempo polinomial contra los códigos Reed-Solomon generalizados, el cual, adaptamos para poder atacar exitosamente un criptosistema construido mediante el empleo de los códigos Reed-Solomon generalizados. Tanto el ataque estándar como el del criptosistema de McEliece son también implementados en *Sage*. El principal objetivo de este capítulo trata de mostrar que aunque el criptosistema de McEliece sea resistente de momento frente a ataques que empleen el algoritmo de Shor, es necesario elegir cuidadosamente unos códigos que sean seguros para realizar la construcción del criptosistema.

A continuación, en el capítulo 7, introducimos los últimos códigos que mostramos en el documento, llamados códigos de producto de matrices, para los cuales mostramos en primer lugar tanto la teoría de estos códigos como todo el código desarrollado en *Sage*. Para estos códigos destacamos que es necesario restringirse a un subconjunto de todos los posibles, sobre los cuales es posible emplear un algoritmo de descodificación eficiente. En segundo lugar, se muestra una propuesta innovadora que permite reducir el tamaño de las claves del criptosistema de McEliece, eliminando de esta forma la principal desventaja que presenta el criptosistema.

En el capítulo 8 se realizan comprobaciones de todas las implementaciones que hemos desarrollado por completo. Para ello se construyen todos los códigos, el criptosistema de McEliece y se realizan varios ataques por filtración, donde empleamos parámetros elevados para los que validamos que la ejecución obtenida es la esperada y además, medimos el tiempo de ejecución que necesitan emplear.

Por último, se muestra en el capítulo 9 las conclusiones del proyecto junto con el trabajo futuro a realizar.

Además, en cada capítulo se muestra al menos un ejemplo, en el que se desarrolla de forma práctica toda la teoría explicada junto con el empleo de las correspondientes funciones de *Sage* ya implementadas o que hemos desarrollado. De esta forma, se facilita la comprensión de las distintas partes teóricas mostradas. En los archivos *.ipynb* que contienen el código desarrollado, se muestra una mayor cantidad de ejemplos que los que explicamos a lo largo del documento, los cuales también son accesibles.

Capítulo 2

Metodología de trabajo

En cualquier proyecto es necesario realizar una estructuración de todo el trabajo que se va a realizar. Existen distintas formas de realizar este trabajo, sin embargo nosotros hemos decidido emplear un desarrollo iterativo que nos permita ir validando el código que vamos desarrollando. Por tanto, comenzamos implementando las funciones mas importantes del proyecto que se emplearán en adicción por el resto de funciones que vayamos desarrollando en etapas posteriores.

Además, es evidente que la metodología empleada debe ser secuencial ya que, por ejemplo no podemos realizar la construcción completa de los códigos de producto de matrices sin haber realizado la implementación de los códigos de Goppa y mucho menos sin haber comprendido ambos códigos de forma teórica.

Hay que tener en cuenta que el empleo del método iterativo nos permite ademas identificar los aspectos mas importantes del proyecto a desarrollar, sin los cuales no podríamos comenzar a realizar otros aspectos de menor relevancia. También, nos permite identificar un conjunto de funciones que quizás en un principio no considerábamos de gran importancia y después vemos que presentan una mayor relevancia.

2.1. Ciclo de vida del proyecto

Ahora mostramos las distintas etapas que hemos ido realizando a lo largo de todo el trabajo realizado:

1) En una primera etapa de prácticamente cualquier proyecto que se vaya a desarrollar es necesario realizar un estudio del estado del arte. Por tanto, se comienza por adquirir una serie de conocimientos teóricos acerca de los tipos de códigos que vamos a utilizar a lo largo del proyecto. Obviamente dicha adquisición debe ser secuencial ya que será necesario por ejemplo realizar un estudio en primer

lugar de los códigos y códigos lineales de forma general antes de comenzar a estudiar un tipo particular de código lineal. Una vez realizado dicho estudio, el cual se irá ampliando también a lo largo del resto de etapas en partes mas concretas, se procede a realizar una planificación de la estructura que a priori deseamos que tenga el presente documento. En adicción a todo este estudio teórico de los códigos, también se realiza un estudio básico del lenguaje de programación Python junto con un estudio mas detallado del lenguaje de programación *SageMath*, debido a que ninguno de los dos lenguajes se ha estudiado a lo largo de los cursos académicos de ambas carreras. Para finalizar la primera etapa, también es necesario adquirir algunos conocimientos que nos permitan redactar el documento empleando Latex, el cual nos facilitará la introducción de distintas expresiones matemáticas a lo largo del documento.

2) En una segunda etapa, tras adquirir un cierto manejo del lenguaje *Sage*, así como un conocimiento teórico elemental de todos los aspectos a tratar, vamos a centrarnos en realizar un estudio acerca de todos los códigos que ya se encuentran implementados en *Sage*, que además necesitaremos emplear a lo largo del proyecto que vamos a desarrollar. Esto nos permitirá obtener información acerca del tiempo necesario que tendremos que emplear para implementar las distintas funciones que usaremos en el proyecto y no se encuentran desarrolladas en *Sage* actualmente. También provocará una reducción del tiempo necesario para realizar las implementaciones al adquirir un mayor conocimiento de distintas clases y métodos ya desarrollados, evitando en ocasiones que generemos cierta cantidad de código o sentencias que ya se encuentran desarrolladas en *Sage*.

   Para finalizar la segunda etapa, se realiza la construcción en *Sage* de los códigos de Goppa binarios, para el cual se dispone de funciones para codificar, introducir errores y descodificar vectores entre otras. También se realiza la implementación del criptosistema de McEliece en *Sage*, de forma que dicha construcción solo se puede realizar empleando los códigos de Goppa binarios implementados. Una vez realizadas ambas construcciones realizamos una serie de ejemplos para validar su correcto funcionamiento.

3) Ahora en la tercera etapa profundizamos en el estudio teórico de los códigos de producto de matrices para poder realizar la implementación de dichos códigos en *Sage*. En esta etapa únicamente se construyen dichos códigos mediante el empleo de códigos ya desarrollados en *Sage*. La razón principal de esto, es que a diferencia de los códigos de Goppa que hemos implementado, el algoritmo de descodificación de los códigos ya implementados nos avisa mediante el empleo de un error en caso de que se intenten corregir una cantidad de errores superior a la capacidad correctora del código, lo cual facilita la implementación del algoritmo de descodificación de los códigos de producto de matrices. En adicción, solo

se permite la construcción de estos códigos mediante el empleo de una matriz A que debe ser cuadrada. Para finalizar, se incrementan las implementaciones de los códigos de Goppa y del criptosistema de McEliece; de forma que ahora es posible aplicar la funciones de codificación, introducción de errores y descodificación de los códigos de Goppa en vectores sobre los que no se realiza una restricción del tamaño y también es posible realizar la codificación y descodificación en listas y cadenas de caracteres. En cuanto al criptosistema de McEliece, ahora también es posible realizar su construcción empleando cualquiera de los códigos que estaban implementados desde el principio en *Sage*.

4) En cuanto a la cuarta etapa continuamos aumentando el estudio teórico de los códigos de producto de matrices y también la implementación en *Sage* de estos códigos, permitiendo ahora realizar su construcción mediante el empleo de una matriz no necesariamente cuadrada y también utilizando códigos de Goppa para su generación. Por último, añadimos los códigos de producto de matrices como otra construcción alternativa que se puede realizar sobre el criptosistema de McEliece.

5) En la quinta etapa nos centramos en el estudio del ataque de filtración contra los códigos Reed-Solomon generalizados donde, en un comienzo realizamos la implementación en *Sage* del ataque estandar contra estos códigos y después realizamos las modificaciones pertinentes que nos permiten implementar dicho ataque contra criptosistemas de McEliece que empleen los códigos Reed-Solomon generalizados para su construcción. Para finalizar se realiza en esta última etapa una pequeña fase de investigación mediante la cual obtenemos como resultado un método innovador que permite modificar el criptosistema de McEliece paliando la principal desventaja que presenta este criptosistema, que es el gran tamaño que presentan las claves que emplea. Para ello, es necesario la construcción del criptosistema empleando códigos de producto de matrices. Por tanto, incrementamos la implementación de estos códigos añadiendo dos funciones donde una de ellas nos permite obtener las claves con tamaño reducido y la otra nos permite obtener las claves totales, a partir de las reducidas, necesarias para la construcción del criptosistema. También se incrementa el código generado del criptosistema de McEliece, de forma que permita generar el criptosistema empleando las matrices obtenidas mediante el empleo de las otras funciones.

6) Para finalizar, en esta etapa realizamos una serie de ejemplos mas complejos que nos permitan validar el correcto funcionamiento de todas las funciones y clases implementadas. También analizamos el tiempo que emplean todas las funciones generadas, modificando en algunos casos distintos algoritmos, para que su complejidad sea menor y por tanto la eficiencia de las funciones que los emplean mejore.

Durante la realización de todo el proyecto se han ido realizando reuniones con los dos tutores para mostrar los avances realizados, solventar dudas y problemas que iban surgiendo e ir mostrando los ejemplos que se iban realizando. En adicción, al finalizar cada una de las etapas que acabamos de explicar, se realizaba una reunión para dar el visto bueno a los avances teóricos y de implementación, junto con los resultados obtenidos hasta el momento. De esta forma, íbamos validando el trabajo realizado y por tanto, en la mayoría de ocasiones los problemas iban surgiendo en relación a temas que se iban tratando en las etapas siguientes.

2.2. Herramientas empleadas

Para la realización de todo el proyecto, vamos a emplear únicamente dos tipos de herramientas:

- **Jupyter Notebook** el cual es un entorno de trabajo *open source* que permite la implementación de código en *Python* u otros lenguajes, entre los cuales se incluye *Sage* el cual es el lenguaje de programación que emplearemos para la construcción de todo el código que realicemos. En el capítulo 3 mostramos mas información acerca de *Sage*. *Jupyter* surge en 2014 como resultado de la evolución del proyecto *IPython* y aunque en un principio fue creado por un profesor de la Universidad de California, es un software completamente libre y por tanto presenta un gran soporte de mantenimiento. Todo el código que generamos, lo organizamos en hojas de trabajo que se almacenan como archivos en formato *.ipynb* dentro de una carpeta que elegimos previamente. Dicho entorno se ejecuta mediante el empleo de un navegador web y por tanto se puede considerar como un entorno multiplataforma. A partir de estos archivos, podemos ir modificando e incrementando el código fácilmente, además de poder enviar dicho código a cualquier persona por correo electrónico o cualquier otra forma de comunicación alternativa en el formato estándar *.ipynb* u en otros de los formatos mas empleados actualmente como por ejemplo *Latex*, *HTML* o *PDF* entre otros.

- **El entorno TexMaker** es un editor de texto que nos permite manejar distintas variantes del lenguaje *Tex*, entre ellas la variante *Latex* que emplearemos para la redacción del presente documento. Este editor es gratuito y fue desarrollado en 2003 por Pascal Brachet empleando para ello el lenguaje de programación C++. Actualmente se encuentra bajo la licencia pública general (GPL) e incluye soporte para mas de 18 idiomas, corrección ortográfica, visor incorporado en pdf interconectado con las líneas de código generadas entre otros aspectos. Para finalizar, destacamos que es necesario haber instalado previamente una distribución de Tex, como por ejemplo MikTex para su correcto funcionamiento, aunque di-

cha instalación se puede realizar dentro del propio instalador de TexMaker, por tanto esto no dificulta nada su instalación.

2.3. Planificación y estimación de costes

2.3.1. Empleo de recursos de personal

Para la realización de todo el proyecto se dispone de un único trabajador que irá ejerciendo los distintos roles por los que debe pasar el proyecto. Vamos a distinguir tres posibles roles. En primer lugar se ejercerá como jefe de proyecto para realizar una planificación acerca de los distintos temas a tratar a lo largo de todo el proyecto, para lo cual será necesario un primer contacto con toda la teoría que se podría estudiar en el proyecto y después realizar una selección y descarte de distintas partes que vayamos considerando mas o menos importantes para el proyecto. Por último, para cada parte se asigna una estimación de las horas que se creen necesarias emplear para su desarrollo, aunque al tratarse de un modelo iterativo, dichas horas pueden variar según los distintos incrementos que se vayan realizando durante todo el proyecto.

En segundo lugar se toma el rol de analista, el cual se centra en realizar un estudio detallado de las distintas partes elegidas por el jefe de proyecto para comenzar su desarrollo teórico, así como la construcción de toda la documentación necesaria para el proyecto, entre las que se encuentran explicar detalladamente todas las partes teóricas y también documentar todo el código que implementará el proyecto. Por tanto, es claro que a lo largo del proyecto se destinará una mayor cantidad de horas en el rol de analista frente a los otros dos roles.

Por último, también se ejercerá el rol de programador, el cual se centra en realizar la implementación en *Sage* de los distintos programas del proyecto. Para este proyecto hay que destacar que el número de horas destinadas al rol de programador se ha reducido respecto si el proyecto hubiera sido desarrollado realmente por distintas personas ejerciendo un rol distinto, ya que los programas que hay que desarrollar necesitan de al menos un pequeño estudio previo de la teoría, lo cual sería realizado en principio por el analista.

Por otro lado, el tiempo estimado empleado por el trabajador en el desarrollo del proyecto ha cambiado en función de los distintos meses. Para los meses de octubre y noviembre del año 2018 el tiempo estimado de empleo se encuentra en unas $5 - 7$ horas semanales. De igual forma, para los meses de enero, febrero, marzo y abril del año 2019 el tiempo también se encuentra en torno a unas $5 - 7$ horas semanales.

Después la mitad del mes de junio y los meses de julio y agosto se incrementaron notablemente la cantidad de horas empleadas ascendiendo hasta unas $40 - 45$ horas

semanales.

Por último, en el mes de septiembre se han empleado unas $60 - 65$ horas semanales hasta realizar la entrega del proyecto, donde de nuevo se han incrementado el número de horas empleadas para poder realizar todas las tareas previstas dentro del plazo esperado, así como la revisión completa de todo el proyecto desarrollado.

En resumen, haciendo un recuento por meses de la cantidad de horas empleadas, podemos ver en la tabla 2.1 que la estimación de horas utilizadas para la realización de todo el proyecto se encuentra entre 690 y 807 horas.

Meses	Horas/Semana	Nº de semanas	Horas
Octubre-Noviembre	$5 - 7$	10	$50 - 70$
Enero-Abril	$5 - 7$	16	$80 - 112$
Junio-Agosto	$40 - 45$	11	$440 - 495$
Septiembre	$60 - 65$	2	$120 - 130$
Total		39	$690 - 807$

Tabla 2.1: Estimación de horas.

2.3.2. Estimación de costes del proyecto

Ahora vamos a realizar un presupuesto del proyecto desarrollado en función principalmente de la estimación de horas realizada, ya que en dicho presupuesto no tendremos en cuenta el hardware empleado para su realización ni tampoco otros costes relacionados con la ubicación y recursos energéticos empleados durante su desarrollo. En adicción, analizando el software empleado, no tendremos en cuenta el coste de la distribución de Microsoft Windows del portátil empleado y como únicamente se emplean los programas TexMaker y Jupyter para el empleo de *Latex* y *Sage* respectivamente, como vimos en la sección 5.2, consideramos que el coste del software es nulo al ser ambos programas gratuitos.

Por tanto, únicamente tendremos en cuenta el trabajo realizado por el único trabajador para realizar el presupuesto. Para ello, diferenciaremos entre los distintos roles que ejerce y para cada uno de ellos tenemos en cuenta el salario medio de cada rol. Según [31] el salario medio neto de un programador se encuentra en 1200€, el de un analista en 1400€ y el de un jefe de proyecto en 1900€, luego el sueldo neto por hora es aproximadamente $7,5$€, 9€ y 12€ respectivamente.

Rol	Horas	Precio/Hora	Precio
Programador	$130 - 150$	$7,5$	$975 - 1125$
Analista	$460 - 527$	9	$4140 - 4743$
Jefe proyecto	$100 - 130$	12	$1200 - 1560$
	$690 - 807$		$6315 - 7428$

Tabla 2.2: Presupuesto del proyecto.

Teniendo en cuenta esto, obtenemos un presupuesto del proyecto cuyo precio total supone entre 6315€ y 7428€ según una estimación de las horas empleadas en cada rol que podemos observar en la tabla 2.2.

Capítulo 3

SageMath

3.1. Introducción

Como hemos dicho en el capítulo 2, el trabajo que se desarrolla en el documento se apoya en el empleo de un lenguaje de programación basado en Python llamado *Sagemath* o *Sage*. Dicho lenguaje de programación nos permite realizar fácilmente los distintos cálculos que necesitaremos utilizar para realizar los distintos ejemplos que se van presentando a lo largo del documento, para que la comprensión de las distintas partes teóricas sea mas sencilla. También, emplearemos *Sage* para realizar la construcción automática de las claves públicas y privadas del criptosistema de McEliece, donde podremos escoger entre una serie de códigos para realizar su construcción, los cuales diferenciamos entre los códigos que ya están implementados total o parcialmente, que son como veremos los códigos Reed-Solomon, Reed-Solomon generalizados y los códigos cíclicos, y los códigos que necesitamos construir desde cero, que son los códigos de Goppa y los códigos de producto de matrices. Además de esto, también emplearemos *Sage* para realizar un ataque al criptosistema de McEliece que emplea códigos Reed-Solomon o Reed-Solomon generalizados para su construcción. Para ello, recurrimos a una vulnerabilidad que presentan estos códigos, la cual nos permite recuperar los parámetros ocultos de los códigos Reed-Solomon o Reed-Solomon generalizados que se emplean para realizar la construcción del criptosistema de McEliece. Para realizar la implementación de dicho ataque será necesario emplear los códigos punteados y los códigos recortados, los cuales están ya implementados en Sage.

La realización de todo lo mostrado es posible como ya hemos dicho, realizando una cantidad mínima de cálculos a mano, gracias al empleo de *Sage* al ser un software dotado de gran cantidad de paquetes matemáticos, que nos permiten abordar distintos aspectos matemáticos como son el álgebra o cálculo numérico teniendo en cuenta el cuerpo sobre el que se está trabajando, pudiendo emplear cuerpos finitos, los cuales son de gran utilidad en el desarrollo de las distintas partes que veremos de la teoría

de códigos.

Además, *Sage* emplea en ocasiones otros lenguajes de programación para realizar distintos tipos de tareas, entre ellos destacamos:

- *Singular*, que es un sistema de álgebra computacional empleado principalmente en cálculos relacionados con álgebra conmutativa, geometría algebraica y teoría de singularidades.
- *Pari*, el cual es también un sistema de álgebra computacional empleado principalmente para realizar cualquier cálculo relacionado con la teoría de números.
- *GAP* es un sistema mas completo también de álgebra computacional dedicado en especial a la teoría de grupos computacionales, por lo que también permite realizar una gran diversidad de operaciones entre polinomios y cuerpos finitos entre otros. Este sistema además cuenta con una librería llamada GUAVA, que se centra en realizar una gran diversidad de cálculos y construcción de códigos. Por tanto, *Sage* es uno de los sistemas a los que mas acudirá durante la ejecución del código de este proyecto.
- *R* que es un lenguaje y entorno de programación orientado prácticamente en su totalidad a realizar cálculos y análisis estadísticos. Destaca por ser orientado a objetos, capacidad de integración en distintos tipos de bases de datos y tener una interfaz propia de documentación basada en *Latex*.
- *Maxima* el cual también es un sistema de cálculo computacional centrado en realizar cálculos simbólicos, por lo que es capaz de realizar una gran diversidad de operaciones con polinomios entre las que destacamos integrales y derivaciones.

Como hemos dicho, *Sage* esta basado en el lenguaje de programación Python, el cual es un lenguaje de programación antiguo, ya que se creó en los años 80, que cuenta con una serie de buenas propiedades, al ser por ejemplo un lenguaje de programación multiparadigma debido a que permite realizar programación estructurada y orientada a objetos. También es un lenguaje dinámicamente tipado, interpretado y es multiplataforma, ya que es un lenguaje soportado por prácticamente todos los sistemas operativos. Además, dicho lenguaje presenta una gran cantidad de documentación y de funciones ya implementadas, las cuales también se pueden emplear en *Sage*, enriqueciendo así notablemente el lenguaje *Sage*. Otra de las características de Python es que en dicho lenguaje de programación no existen ni las variables ni las funciones privadas, es decir, no existen ni variables ni funciones que solo puedan emplearse desde dentro de un objeto. A pesar de ello, prácticamente todos los programadores de Python han convenido emplear un nombre prefijado con un guión bajo para hacer

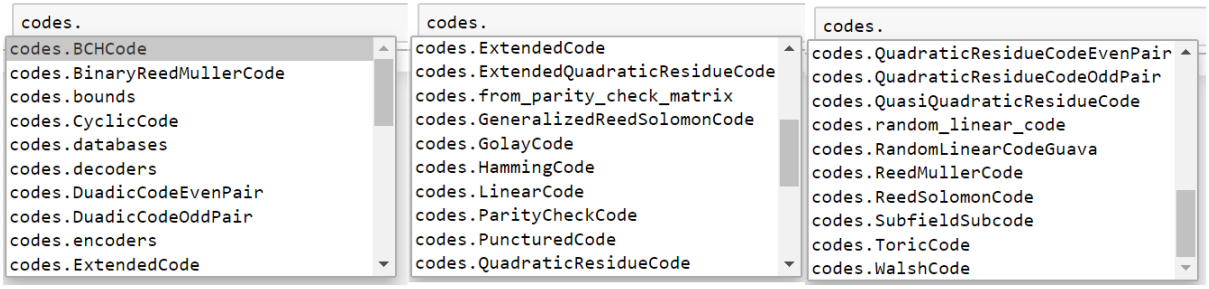

Imagen 3.1: Códigos implementados

referencia a una parte privada del código construido. Por ello, nosotros también emplearemos esta estructura cuando queramos construir una función o variable privada.

Por tanto, como hemos visto en el capítulo 2, *Sage* es el lenguaje de programación que vamos a emplear junto con el entorno de trabajo *Jupyter Notebook* para realizar toda la implementación de código necesaria del proyecto, donde dicha construcción se realizará empleando una programación orientada a objetos. En primer lugar, vamos a realizar un estudio acerca de los códigos y métodos que ya se encuentran implementados en *Sage* y que necesitamos emplear.

Dentro de una hoja de trabajo para código de *Sage*, para acceder al módulo que contiene el catálogo con todos los códigos ya implementados en *Sage* hay que escribir la sentencia "*codes.*", y presionando el tabulador obtenemos el listado de todos ellos como se muestra en la imagen 3.1.

A lo largo del trabajo fin de grado emplearemos, como ya hemos mencionado los códigos lineales, códigos de Reed-Solomon y Reed-Solomon generalizados, códigos cíclicos, códigos punteados, códigos recortados, códigos de Goppa y códigos de producto de matrices. Por tanto, a partir de la imagen 3.1 vemos que no están implementados en *Sage* solamente los códigos de Goppa y los códigos de producto de matrices, entonces realizaremos la programación de estos códigos desde cero. Sin embargo, vemos que para el resto de códigos *Sage* si tiene una referencia a ellos, entonces vamos a comprobar los distintos métodos que el lenguaje de programación ya tiene implementados y necesitaremos emplear.

3.2. Códigos lineales

Los códigos lineales son códigos en bloque que poseen una estructura algebraica, la cual permite que la codificación y descodificación realizada sea menos costosa computacionalmente. Por ello, todos los códigos mencionados que vamos a emplear

en adelante serán lineales.

Definición 3.1. Un código lineal C de dimensión k, es un k-subespacio vectorial de un espacio vectorial $\mathbb{F}_q^n$. Normalmente, los códigos lineales se suelen denotar con los llamados parámetros fundamentales del código, $[n, k]$, donde, n indica la longitud de los elementos del subespacio vectorial, k indica la dimensión del subespacio vectorial, es decir, del código lineal.

Definición 3.2. Una palabra del código es un elemento del código lineal C, y todas las palabras del código tienen la misma longitud.

En *Sage* existen tres formas distintas de construir un código lineal:

- A partir de la matriz generatriz (definicion 3.3) que deseamos que tenga el código lineal empleando el código:

  ```
  In: codes.LinearCode(self, generator, d=None)
  ```

  Donde el parámetro *generator* hace referencia a la matriz generatriz definida sobre un cuerpo finito y el parámetro d es opcional, en el cual debemos indicar, si se conoce la distancia mínima del código (nota 3.14), aunque normalmente no se emplea.

- Indicando el cuerpo sobre el que queremos construir el código, junto con su dimensión y longitud. Para ello utilizamos el código:

  ```
  In: codes.random_linear_code(F, length, dimension)
  ```

  En dicha sentencia, F hace referencia al cuerpo que queremos emplear, *length* a la longitud y *dimension* a la dimensión deseada para el código lineal.

  Para la construcción del código, dicha función simplemente va generando matrices de tamaño $dimension \times length$ con elementos sobre el cuerpo F hasta la obtención de una cuyo rango sea máximo, y por tanto se pueda emplear como matriz generatriz del código lineal a construir.

- A partir de la matriz de control (definicion 3.10) que deseamos que tenga el código lineal empleando el código:

  ```
  In: codes.from_parity_check_matrix(H)
  ```

  siendo H la matriz de control del código lineal que deseamos.

Definición 3.3. Una matriz generatriz G de un código lineal $[n, k]$, es una matriz $k \times n$ cuyas filas son vectores linealmente independientes del código lineal que además lo generan, es decir, dichas filas forman una base del código lineal como espacio vectorial.

Una vez definido en *Sage* un código lineal C, podemos obtener su matriz generatriz:

```
In: C.generator_matrix()
```

Nota 3.4. Podemos observar de la definición 3.3 que un código lineal C, puede tener distintas matrices generatrices, aunque todas ellas son semejantes. Aun así, hay que destacar que distintas matrices generatrices de un código generan distintas codificaciones, ya que $C = \{aG \mid a \in \mathbb{F}_q^k\}$.

Nota 3.5. También podemos relacionar a cada matriz generatriz G de un código C con una aplicación lineal inyectiva, f, que llamaremos aplicación de codificación:

$$f : \mathbb{F}_q^k \longrightarrow C \subset F_q^n$$

Definición 3.6. Dados dos códigos lineales C_1 y C_2 de igual longitud, n, sobre $\mathbb{F}_q$, decimos que son códigos equivalentes, si existe una permutación π del conjunto de indices $\{1, \cdots, n\}$ de forma que $C_2 = \{\pi(c) \mid c \in C_1\}$

El código de Sage que nos permite comprobar si dos códigos son equivalentes es:

```
In: C1.is_permutation_equivalent(other, algorithm=None)
```

donde el parámetro *other* hace referencia al código C_2 denotado en la definición 3.6 y el parámetro opcional *algorithm*, si se emplea, se suele utilizar el algoritmo "*verbose*", el cual nos proporciona la permutación que debemos realizar para obtener el código C_2 a partir del código C_1 en el caso en que los códigos sean equivalentes.

Definición 3.7. Dado un código lineal C, diremos que es un código sistemático si y solo si su matriz generatriz G es de la forma $G = (I_k|A)$, siendo I_k la matriz identidad de tamaño $k \times k$.

Nota 3.8. La matriz generatriz de un código sistemático se llama matriz generatriz en forma estándar.

Cuando el código es sistemático, al realizar la codificación de un vector se obtiene como resultado una palabra del código formada por el vector sin codificar seguido, hasta completar la longitud de la palabra del código, de coordenadas llamadas de control.

Las palabras de un código lineal $[n, k]$ en forma sistemática tienen k símbolos de información y $n - k$ símbolos redundantes, los cuales permiten validar que los k símbolos de información obtenidos son correctos y por tanto, no tienen errores.

Esto facilita la obtención de la información de forma correcta a pesar de que dicha información, bien durante su transmisión a través del canal de comunicación o cualquier otra circunstancia, haya sufrido una cierta cantidad de errores.

La matriz generatriz en forma estándar de un código C también se puede obtener fácilmente en Sage:

```
In: C.systematic_generator_matrix()
```

También podemos obtener, si se prefiere, la matriz formada únicamente por las columnas que no pertenecen a la matriz identidad, es decir, obtener la matriz A de la definición 3.7:

```
In: C.redundancy_matrix()
```

Definición 3.9. Dado un código lineal, C, sobre el cuerpo $\mathbb{F}_q$ de parámetros $[n, k]$, llamaremos codificación de $m \in \mathbb{F}_q^k$ a la obtención de la palabra del código $c \in C$ mediante su multiplicación por la matriz generatriz de C, G, es decir, $c = mG$.

Sage también nos proporciona una serie de funciones relacionadas con la codificación:

- Aunque se puede realizar la codificación de un vector fácilmente según la definición 3.9, existe una función para realizarla:

  ```
  In: C.encode(word, encoder_name=None, *args)
  ```

  donde el parámetro *word* hace referencia al vector que deseamos codificar. Además, tenemos los parámetros opcionales *encoder_name* y $*args$ donde el primero indica el nombre del codificador que queremos usar de los disponibles para C y el segundo son el conjunto de parámetros que se desean pasar a dicho codificador.

- También, podemos ver el conjunto de codificadores disponibles para el código lineal C:

  ```
  In: C.encoders_available()
  ```

- Una función que devuelve un codificador de los posibles para el código lineal C:

  ```
  In: C.encoder(encoder_name, *args)
  ```

  la cual tiene dos parametros opcionales; *encoder_name* donde debemos introducir el nombre de un codificador de entre todos los posibles para obtenerlo, y $*args$ donde introducimos los argumentos que deseamos pasar al codificador indicado.

- Por último, podemos incorporar otro codificador para C que hayamos construido:

  ```
  In: C.add_encoder(name, decoder)
  ```

  donde el parámetro *name* indica el nombre del nuevo codificador y en el parámetro *decoder* tenemos que introducir el nombre de la función donde hemos implementado el nuevo codificador.

Definición 3.10. Una matriz de control, H, de un código lineal $[n,k]$, C, es una matriz $(n-k)\times n$ para la cual se cumple que $c \in C \Leftrightarrow Hc^t = 0\ \forall c \in \mathbb{F}_q^n$

Debido a ello, es claro que si G es una matriz generatriz y H es una matriz de control de un código lineal C entonces tenemos que $GH^t = HG^t = 0$

Para su obtención usando *Sage* empleamos:

```
In: C.parity_check_matrix()
```

Los códigos lineales son capaces de corregir cierta cantidad de errores, pero para saber la cantidad de errores que es capaz un código lineal de corregir es necesario introducir los conceptos de peso y distancia mínima del código.

Definición 3.11. El peso de Hamming de un vector $c = (c_1, ..., c_n) \in \mathbb{F}_q^n$, denotado por $\omega(c)$, es el numero de coordenadas no nulas del vector, es decir, $\omega(c) = \#\{i \mid c_i \neq 0,\ 1 \leq i \leq n\}$.

Nota 3.12. Una vez definido el peso de Hamming de un vector, podemos deducir que el peso mínimo de un código lineal C sera el mínimo de los pesos de Hamming de las palabras del código, es decir,

$$\omega(C) = min\{\omega(c) \mid c \in C \setminus \{0\}\}$$

Definición 3.13. La distancia de Hamming entre dos vectores $a, b \in \mathbb{F}_q^n$, denotado por $d(a,b)$, es el numero coordenadas donde ambos vectores no coinciden, es decir, $d(a,b) = \#\{i \mid a_i \neq b_i,\ 1 \leq i \leq n\}$.

Nota 3.14. De igual forma que ocurría con el peso, la distancia mínima de un código lineal C sera la mínima distancia de Hamming entre cualquier par de palabras de código distintas.

Además, para los códigos lineales, existe una relación entre la distancia mínima y el peso mínimo de un código:

Proposición 3.15. *Dado un código lineal $[n, k]$, C, la distancia mínima de C es igual a el peso mínimo de C.*

Sage también nos proporciona una función capaz de obtener la distancia mínima de un código C dado, aunque dicha función se encuentra limitada por un algoritmo de GAP, de forma que solo es posible emplearla en códigos construidos sobre cuerpos finitos de menos de 256 elementos:

```
In: C.minimum_distance()
```

También, nos proporciona otra función que nos devuelve una lista de tamaño n, $a = [a_1, ..., a_n]$, siendo n la longitud del código C, la cual indica mediante el entero que se encuentra en la posición a_i de la lista a, el número de palabras de peso i que tiene el código C:

```
In: C.weight_distribution()
```

Obviamente, el costo computacional de ambas funciones es muy elevado al tener que ir realizando operaciones sobre cada palabra que pertenece al código C, por ello, para códigos con parámetros fundamentales elevados, dichas funciones necesitarán emplear una gran cantidad de tiempo, aunque se empleen otros métodos que permitan obtener la distancia de una forma mas óptima, como vimos en [1, Corolario 2.18] del trabajo fin de grado de matemáticas. Por tanto no serán funciones útiles para códigos con grandes parámetros.

Proposición 3.16. ***(cota de Singleton)*** *Dado un código lineal $[n, k]$, C, cuya mínima distancia es d, se cumple que $d \leq n - k + 1$.*

Los códigos que satisfacen la igualdad en la cota de Singleton, es decir, que cumplen $d = n - k + 1$, son conocidos como códigos de máxima distancia separable o MDS.

A partir de los conceptos de distancia y peso mínimo de un código lineal definidos, podemos observar que estos son los que nos permiten detectar cierta cantidad de errores. Denotamos por e un vector, llamado vector de errores, con longitud igual a la longitud de las palabras del código que se esta empleando y con peso $\omega(e) < d$, siendo d la distancia mínima de dicho código. Ya que si el error tiene peso $\omega(e) \geq d$, entonces una palabra del código, c, con dicho error, puede dar lugar a otra palabra del código, $c' = c + e$, lo que provoca en ocasiones que no nos percatemos del error producido.

Por otro lado, una vez detectada una palabra con cierta cantidad de errores, debemos preguntarnos cuantos errores somos capaces de corregir. Para ello veamos ahora la capacidad correctora de dichos códigos:

Definición 3.17. Dado un código lineal, C, diremos que es corrector de t errores si para cualesquiera dos palabras del código, $a, b \in C$ y para cualesquiera dos vectores de errores, e y e' con $\omega(e), \omega(e') \leq t$, tenemos que $a + e \neq b + e'$.

El siguiente teorema nos proporciona una forma mas sencilla de obtener la capacidad correctora del código:

Teorema 3.18. *Un código $[n, k]$ es corrector de t errores $\Leftrightarrow t < \frac{d}{2}$, siendo d la distancia mínima de dicho código.*

Definición 3.19. Dada una palabra c del código C sobre el cuerpo $\mathbb{F}_q$ de parámetros $[n, k]$, llamaremos descodificación de c a la obtención del vector $m \in \mathbb{F}_q^k$ tal que su codificación según la definición 3.9 sea c.

En *Sage* existen una serie de funciones relacionadas con la corrección de errores y la descodificación:

- Para realizar la corrección de errores de una palabra del código C tenemos la función:

  ```
  In: C.decode_to_code(word, decoder_name=None, *args)
  ```

  donde el parámetro *word* hace referencia a la palabra sobre la que deseamos corregir los errores que contiene. Los parámetros *decoder_name* y $*args$ son ambos opcionales, para los cuales, el primero indica el nombre del descodificador que queremos usar y el segundo son el conjunto de parámetros que se desean pasar a dicho descodificador.

- Para realizar la corrección de errores de la palabra y ademas la descodificación se emplea:

  ```
  In: C.decode_to_message(word, decoder_name=None, *args)
  ```

  cuyos parámetros son idénticos a los de la función *decode_to_code*(). Hay que destacar que si en la descodificación realizada por dicha función tenemos que obtener un vector cuyas últimas coordenadas se corresponden con elementos nulos, la función nos devuelve el vector obtenido a partir de la eliminación de dichas últimas coordenadas nulas.

- Ademas tenemos una función que permite ver los distintos descodificadores disponibles para el código C:

  ```
  In: C.decoders_available()
  ```

- Una función que nos proporciona un descodificador de los posibles para C:

  ```
  In: C.decoder(decoder_name, *args)
  ```

  donde debemos introducir dos parámetros, *decoder_name* donde indicamos el nombre de un descodificador de los todos los posibles para C, que queremos obtener y $*args$ que hace referencia a los argumentos que queremos pasar al descodificador seleccionado.

- Por último, podemos añadir un nuevo descodificador para el código C que hayamos implementado:

  ```
  In: C.add_decoder(name, decoder)
  ```

  donde el parámetro *name* indica el nombre con el que se reconocerá el descodificador y en el parámetro *decoder* debemos introducir el nombre de la función en la que hemos implementado nuestro descodificador.

Si partimos ahora de un código lineal C de parámetros $[n, k]$, hemos visto que su matriz de control, H, tiene rango máximo, por tanto, puede coincidir con la matriz generatriz de otro código lineal C', donde ambos códigos están definidos sobre $\mathbb{F}_q$. Podemos observar que al tener C dimension k, necesariamente C' tiene que tener dimension $n - k$ y una matriz de control de C' coincidirá con una matriz generatriz de C, G, ya que por la definición 3.10 sabemos que $GH^t = 0$.

Definición 3.20. Llamaremos código dual de un código lineal, C, y lo denotaremos por $C^{\perp}$, al código C'.

La función de *Sage* que nos permite obtener el código dual es:

```
In: C.dual_code()
```

Además de todas las funciones de *Sage* descritas, existen una gran variedad de ellas que no hemos comentado como por ejemplo $C.base\_field()$ que nos permite saber el cuerpo sobre el que se encuentra el código C, $C.list()$ para conseguir un listado con todos los elementos que constituyen el código C, $C.cardinality()$ para obtener la cantidad de elementos que componen C, $C.dimension()$ para conocer la dimensión de C, $C.is\_subcode(C_1)$ para saber si C es un subcodigo de C_1, $C.length()$ para saber la longitud de C o $C.syndrome(m)$ para obtener el síndrome del vector m para el código C (ver [1, Definición 2.23] en el trabajo fin de grado de matemáticas), entre otros. Todas estas funciones, se pueden emplear en el resto de códigos que se encuentran ya implementados en *Sage* y que vamos a emplear, al ser todos ellos códigos lineales.

Ejemplo 3.1. Veamos un ejemplo sencillo en el que empleemos la mayoría de las funciones que hemos descrito. Para ello, vamos ha construir un código lineal, C, sobre el cuerpo $\mathbb{F}_2$ con parámetros fundamentales $n = 6$, $k = 3$ e imponemos que la distancia mínima del código que obtengamos sea $d = 3$ mediante un bucle while:

```
In: seguir=True;
    while seguir:
        C=codes.random_linear_code(GF(2),6,3)
        if C.minimum_distance()==3:
            seguir=False
    C

Out: [6, 3] linear code over GF(2)
```

Obviamente si validamos la distancia del código, obtenemos que es 3:

```
In: C.minimum_distance()

Out: 3
```

De igual forma, podemos comprobar la dimensión, longitud o cuerpo que esta empleando el código C:

```
In: C.dimension()

Out: 3

In: C.length()

Out: 6

In: C.base_field()

Out: Finite Field of size 2
```

Como la dimensión es $k = 3$ y el cuerpo empleado es $\mathbb{F}_2$ sabemos que la cantidad de elementos del código es $2^3 = 8$, lo cual también podemos obtener mediante la función de *Sage* descrita anteriormente:

```
In: C.cardinality()

Out: 8
```

Ahora obtenemos todos los elementos del código C:

```
In: C.list()

Out: [(0, 0, 0, 0, 0, 0),
      (1, 0, 0, 1, 1, 0),
      (1, 1, 1, 0, 0, 0),
      (0, 1, 1, 1, 1, 0),
      (1, 1, 0, 1, 0, 1),
      (0, 1, 0, 0, 1, 1),
      (0, 0, 1, 1, 0, 1),
      (1, 0, 1, 0, 1, 1)]
```

A partir de una base de esos elementos podemos construir una matriz generatriz del código C, siendo las filas de la matriz generatriz los correspondientes elementos de la base. En lugar de realizar esto, empleamos la función de *Sage* que nos proporciona directamente una matriz generatriz:

```
In: C.generator_matrix()

Out: [1 0 0 1 1 0]
     [1 1 1 0 0 0]
     [1 1 0 1 0 1]
```

Ahora obtenemos la matriz generatriz en forma estándar y la matriz A de la definición 3.7:

```
In: C.systematic_generator_matrix()

Out: [1 0 0 1 1 0]
     [0 1 0 0 1 1]
     [0 0 1 1 0 1]

In: C.redundancy_matrix()

Out: [1 1 0]
     [0 1 1]
     [1 0 1]
```

En cuanto a una matriz de control del código obtenemos:

```
In: C.parity_check_matrix()

Out: [1 0 1 0 1 1]
     [0 1 1 0 0 1]
     [0 0 0 1 1 1]
```

Por último, vamos a realizar la codificación de un vector m y su posterior descodificación con y sin errores. Para ello en primer lugar, vamos a ver los codificadores disponibles para un código lineal:

```
In: C.encoders_available()

Out: ['GeneratorMatrix', 'Systematic']
```

Seleccionamos el codificador *GeneratorMatrix* y lo guardamos en una variable llamada *codificador*:

```
In: codificador=C.encoder('GeneratorMatrix')
    codificador

Out: Generator matrix-based encoder for [6, 3] linear code over GF(2)
```

Seleccionamos un vector m de $\mathbb{F}_2$, $m = (1,0,1)$:

```
In: m=vector(GF(2),[1,0,1])
    m

Out: (1, 0, 1)
```

y empleamos la variable *codificador* para realizar su codificación:

```
In: c=codificador(m)
    c

Out: (0, 1, 0, 0, 1, 1)
```

Podemos comprobar fácilmente que obtenemos el mismo resultado si multiplicamos el mensaje m por la matriz generatriz de C:

```
In: c1=m*C.generator_matrix()
    c1

Out: (0, 1, 0, 0, 1, 1)
```

Una vez realizada la codificación, vamos a realizar su descodificación, para ello primero vamos a ver los descodificadores disponibles para un código lineal:

```
In: C.decoders_available()

Out: ['InformationSet', 'NearestNeighbor', 'Syndrome']
```

De los tres disponibles, vamos a emplear el descodificador del síndrome, el cual fue explicado en [1, Sección 2.2] del trabajo fin de grado de matemáticas:

```
In: mm=C.decode_to_message(c, 'Syndrome', 1)
    mm

Out: (1, 0, 1)
```

Ahora vamos a introducir un vector de error e de peso de Hamming 1 que podamos corregir, el cual añadimos al mensaje codificado, c:

```
In: e=vector(GF(2),[0,0,0,0,0,1])
    e

Out: (0, 0, 0, 0, 0, 1)

In: cc=c+e
    cc

Out: (0, 1, 0, 0, 1, 0)
```

y realizamos su corrección de errores para obtener la palabra del código y también para obtener el mensaje descodificado:

```
In: mm=C.decode_to_code(cc, 'Syndrome', 1)
    mm

Out: (0, 1, 0, 0, 1, 1)

In: mm=C.decode_to_message(cc, 'Syndrome', 1)
    mm

Out: (1, 0, 1)
```

Para finalizar calculamos el código dual de C:

```
In: C.dual_code()

Out: [6, 3] linear code over GF(2)
```

3.3. Códigos Reed-Solomon y Reed-Solomon generalizados

Los códigos Reed-Solomon y Reed-Solomon generalizados, también conocidos de forma abreviada como códigos RS y códigos GRS respectivamente, son un tipo de códigos creados por Irving S. Reed y Gustave Solomon en el año 1960. Dichos códigos pertenecen a la categoría Forward Error Correction (FEC).

Estos códigos resultan ser muy útiles para la corrección de errores a ráfagas que afectan a bloques contiguos. Por ello se emplean en una gran diversidad de casos, por ejemplo, en la tecnología DSL y variantes como la ADSL y VDSL, en distintas unidades de almacenamiento como discos duros, CD, DVD, BlueRay y también en códigos de barras y radiodifusión digital actual como DVB y sus variantes.

Definición 3.21. Un código Reed-Solomon es un código sobre un cuerpo finito $\mathbb{F}_q$, con un vector $a = (a_1, ..., a_n) \in \mathbb{F}_q^n$, donde $a_i \neq a_j \, \forall i \neq j$, y n es un número entero con $0 < n \leq q - 1$. Normalmente, se considera $n = q - 1$ para tener la longitud máxima que puede alcanzar el código, y de esta forma sería un código cíclico.

Sea $L_k \subset \mathbb{F}_q[x]$ el espacio vectorial formado por todos los polinomios de grado menor que k, donde k es un entero con $0 < k \leq n$. Es evidente que la dimensión de L_k es k debido a que cualquier polinomio de grado menor que k se puede generar como combinación lineal de los elementos $\{1, x, ..., x^{k-1}\}$ sobre el cuerpo $\mathbb{F}_q$, y al ser dichos elementos linealmente independientes, forman una base de L_k. Entonces el código Reed-Solomon asociado al vector a, se denota por $RS_k(a)$, y esta formado por los vectores obtenidos al evaluar los polinomios de L_k en las coordenadas del vector a, es decir, $RS_k(a) = \{(f(a_1), ..., f(a_n)) \mid f \in L_k\}$.

En *Sage* podemos realizar la construcción de los códigos Reed-Solomon fácilmente al estar ya implementados:

```
In: codes.ReedSolomonCode(base_field, length, dimension,
                                                    primitive_root=None)
```

donde el parámetro *base_field* hace referencia al cuerpo finito sobre el que se quiere construir el código, *length* la longitud que queremos que posea el código y *dimension* la dimensión que deseamos que tenga el código. En cuanto al parámetro *primitive_root*, es opcional y en el debemos indicar un elemento primitivo del cuerpo sobre el que nos encontramos que queremos para el código que vamos a construir. Lo habitual es no emplear dicho parámetro y que Sage seleccione automáticamente un elemento primitivo.

Las principales características de los códigos Reed-Solomon son que son códigos lineales y además siempre son códigos MDS (máxima distancia separable).

Veamos ahora la definición de los códigos Reed-Solomon generalizados, la cual es muy parecida a la de los Reed-Solomon:

Definición 3.22. Un código Reed-Solomon generalizado es un código sobre un cuerpo finito $\mathbb{F}_q$, con un vector $b = (b_1, ..., b_n) \in \mathbb{F}_q^n$, donde $b_i \neq 0 \, \forall i$ y con un vector $a = (a_1, ..., a_n) \in \mathbb{F}_q^n$, donde $a_i \neq a_j \, \forall i \neq j$, y n es un número entero con $0 < n \leq q - 1$.

Considerando igual que para los códigos Reed-Solomon $L_k \subset \mathbb{F}_q$, el espacio vectorial de dimensión k formado por todos los polinomios de grado menor que k, tenemos que el código Reed-Solomon asociado al vector a y b, denotado por $GRS_k(a, b)$ es:

$$GRS_k(a, b) = \{(b_1 f(a_1), ..., b_n f(a_n)) \mid f \in L_k\}$$

Para la construcción de los códigos Reed-Solomon generalizados mediante *Sage*, también disponemos de una clase que nos facilita su construcción:

```
In: codes.GeneralizedReedSolomonCode(self, evaluation_points, dimension,
                                     column_multipliers=None)
```

donde tenemos dos parámetros obligatorios y uno opcional: para el parámetro *evaluation_points* debemos introducir una lista de elementos distintos del cuerpo que vamos a emplear que se corresponde con el vector a de la definición 3.22. El otro parámetro obligatorio es *dimension* donde debemos indicar la dimensión que deseamos que tenga el código que vamos a generar. Por último, tenemos el parámetro opcional *column_multipliers*, donde debemos introducir una lista de elementos no nulos del cuerpo que vamos a emplear, la cual se corresponde con el vector b de la definición 3.22. Si no se introduce dicho parámetro, se considera automáticamente que el vector b que vamos a emplear para la construcción del código tiene todas sus entradas con valor 1.

En el caso en que no introduzcamos el parámetro opcional, podemos observar que el código de Reed-Solomon generalizado que vamos a construir coincide con el código Reed-Solomon. Por tanto, lo habitual es emplear dicho parámetro opcional para construir realmente códigos Reed-Solomon generalizados.

Proposición 3.23. *El código dual de $GRS_k(a, b)$ es $GRS_{n-k}(a, b')$, donde $b' = (b'_1, ..., b'_n)$ es un vector de $\mathbb{F}_q^n$ con $b'_i \neq 0 \, \forall i = 1, ..., n$. Además, $b' \in GRS_{n-1}(a, b)^{\perp}$ y se cumple que*

$$\sum_{i=1}^{n} b'_i b_i f(a_i) = 0 \ \forall f \in L_{k-1} \tag{3.1}$$

Corolario 3.24. *El código dual de $GRS_k(a,b)$ es MDS.*

Proposición 3.25. *Una matriz generatriz, $G \in \mathbb{F}_q^{k\times n}$, de un código Reed-Solomon generalizado es:*

$$G = \begin{pmatrix} b_1 & b_2 & \cdots & b_n \\ b_1a_1 & b_2a_2 & \cdots & b_na_n \\ \vdots & \vdots & & \vdots \\ b_1a_1^{k-1} & b_2a_2^{k-1} & \cdots & b_na_n^{k-1} \end{pmatrix}$$

Corolario 3.26. *Si el elemento $b \in \mathbb{F}_q^n$ es $b = (1,...,1)$ entonces tenemos que $GRS_k(a,b) = RS_k(a)$ como hemos mencionado anteriormente. Por tanto una matriz generatriz de un Reed-Solomon será igual que la matriz generatriz del Reed-Solomon generalizado, tomando en dicha matriz el valor de 1 todas las coordenadas del vector b, es decir, $b_i = 1\ \forall i = 1,...,n$.*

En cuanto a la matriz de control de los códigos Reed-Solomon y Reed-Solomon generalizados, al ser dichos códigos lineales, se obtiene según lo visto en la definición 3.10.

Para la obtención tanto de la matriz generatriz como de la matriz de control de un código Reed-Solomon generalizado o Reed-Solomon mediante *Sage*, tenemos que emplear las mismas funciones que empleamos para su obtención en los códigos lineales en la sección 3.2:

```
In: C.generator_matrix()
```

```
In: C.parity_check_matrix()
```

Para realizar la codificación de un mensaje $m = (m_1,...,m_k)$, de tamaño k, empleando un código Reed-Solomon generalizado $GRS_k(a,b)$, tenemos que obtener su polinomio: $f_m(x) = \sum_{i=1}^{k} m_i x^{i-1}$. Ahora, evaluando dicho polinomio sobre las coordenadas del vector a y multiplicando el resultado obtenido de cada evaluación por la coordenada correspondiente del vector b obtenemos su codificación: $c = (b_1f_m(a_1),...,b_nf_m(a_n))$. Obviamente, podemos observar que esto coincide con realizar la codificación explicada en 3.3, para cualquier código lineal, realizando simplemente la multiplicación del mensaje por la matriz generatriz, es decir, $c = mG$.

Si observamos en *Sage* los codificadores disponibles para un código Reed-Solomon o Reed-Solomon generalizado mediante la función *encoders_available*(), obtenemos como resultado los codificadores *EvaluationPolynomial* y *EvaluationVector* que se corresponden con las dos formas de codificar explicadas en el párrafo anterior. Ademas, seguimos teniendo el codificador *Systematic* que actúa de forma semejante al de los códigos lineales.

En cuanto al proceso de descodificación, suponemos que se envía el bloque de mensaje $c \in GRS_k(a,b)$ y recibimos el mensaje $y = c + e$ donde el peso del vector de error es menor que la mitad de la distancia del código $GRS_k(a,b)$, d, es decir, $t = \omega(e) \leq \frac{d-1}{2}$. Mirando los descodificadores disponibles en *Sage* mediante la función *decoders_available*() para un código Reed-Solomon o Reed-Solomon generalizado obtenemos:

```
In: C.decoders_available()

Out: ['BerlekampWelch',
      'ErrorErasure',
      'Gao',
      'GuruswamiSudan',
      'InformationSet',
      'KeyEquationSyndrome',
      'NearestNeighbor',
      'Syndrome']
```

Por tanto, vemos que tenemos los mismos descodificadores de los códigos lineales junto con otros cinco disponibles exclusivamente para los Reed-Solomon generalizados. Nosotros cuando necesitemos utilizar alguno de ellos en los ejemplos que mostremos mas adelante, nos centraremos en el empleo únicamente del descodificador *BerlekampWelch*, al ser el descodificador general explicado en [1, Sección 3.3] del trabajo fin de grado de matemáticas.

Ejemplo 3.2. Tomamos el cuerpo finito $\mathbb{F}_7$ para la construcción del código C que vamos a realizar. Ademas, tomamos como parámetros fundamentales para C, $n = 6$ y $k = 3$. Al ser los códigos Reed-Solomon generalizados MDS, tenemos que la distancia mínima del código es $d = 4$ y por tanto la capacidad correctora del código C es $t = 1$. Para su construcción emplearemos los vectores $a = (1,2,3,4,5,6)$ y $b = (2,4,6,1,3,5)$ obteniendo como resultado $C = GRS_3(a,b)$:

```
In: a=vector(GF(7),[i for i in GF(7).list()[1:7]])
    b=vector(GF(7),[2,4,6,1,3,5])
    C=codes.GeneralizedReedSolomonCode(a,3,b)
    C

Out: [6, 3, 4] Generalized Reed-Solomon Code over GF(7)
```

Obtenemos también su matriz generatriz:

```
In: C.generator_matrix()
```

```
Out: [2 4 6 1 3 5]
     [2 1 4 4 1 2]
     [2 2 5 2 5 5]
```

Ahora, calculamos su código dual y comprobamos que se verifica la proposición 3.23:

```
In: C.dual_code()

Out: [6, 3, 4] Generalized Reed-Solomon Code over GF(7)

In: C.dual_code().evaluation_points()

Out: (1, 2, 3, 4, 5, 6)

In: C.dual_code().column_multipliers()

Out: (3, 3, 3, 3, 3, 3)
```

Podemos observar que el vector a sigue siendo el mismo y el vector $b' = (3,3,3,3,3,3)$ nos permite verificar la ecuación 3.1 para cualquier polinomio de grado menor que 3:

```
In: r=random_vector(GF(7),3)
    r

Out: (4, 4, 2)

In: C.encode(r)*C.dual_code().column_multipliers()

Out: 0
```

Por último realizamos la codificación del vector $m = (3,5,2)$ mediante la función *EvaluationVector* y empleando también la función *EvaluationPolynomial*, para la cual necesitamos expresar el vector m como polinomio, es decir, $m(x) = 3 + 5x + 2x^2$:

```
In: m=vector(GF(7),[3,5,2])
    m

Out: (3, 5, 2)

In: R.<x> = PolynomialRing(GF(7))
    mx=R(3+5*x+2*x**2)
    mx
```

```
Out: 2*x^2 + 5*x + 3

In: c=C.encode(m,'EvaluationVector')
    c

Out: (6, 0, 6, 6, 3, 0)

In: c=C.encode(mx,'EvaluationPolynomial')
    c

Out: (6, 0, 6, 6, 3, 0)
```

Introducimos un vector error e de peso de Hamming 1 y lo incorporamos al mensaje codificado c obteniendo como resultado el vector y:

```
In: e=vector(GF(7),[0,5,0,0,0,0])
    e

Out: (0, 5, 0, 0, 0, 0)

In: y=c+e
    y

Out: (6, 5, 6, 6, 3, 0)
```

Ahora realizamos la corrección de errores aplicando el algoritmo de Berlekamp-Welch, recuperando de esta forma el vector codificado sin errores c:

```
In: cc=C.decode_to_code(y,'BerlekampWelch')
    cc

Out: (6, 0, 6, 6, 3, 0)
```

Por último realizamos la descodificación de la palabra del código sin errores para recuperar el mensaje de partida:

```
In: mm=C.decode_to_message(cc)
    mm

Out: (3, 5, 2)
```

3.4. Códigos punteados y recortados

En esta sección explicamos los códigos punteados y recortados ya que necesitaremos emplear ambos cuando implementemos el ataque por filtración a los códigos Reed-Solomon generalizados en el capítulo 6. Tanto los códigos punteados como los recortados necesitan partir de un código lineal para su construcción.

Definición 3.27. Dado un código lineal C de parámetros $[n,k]$, y sea (J,J') una partición de $\{1,...,n\}$ la cual verifica que $J \cup J' = \{1,...,n\}$ y que $J \cap J' = \emptyset$. Diremos que el código punteado de C en J, denotado por $P_J(C)$, es el código, también lineal, cuyas palabras se obtienen a partir de las palabras del código de C restringidas a las posiciones de J':

$$P_J(C) = \{(c_i)_{i\in J'} \mid c \in C\}$$

Para realizar la construcción en *Sage* de los códigos punteados, existen dos formas:

- Mediante la clase *sage.coding.punctured_code.PuncturedCode*$(C, position)$, donde para poder emplearla necesitamos construir el código de igual forma que se realiza la construcción de cualquiera de los códigos que ya están implementados en *Sage*:

  ```
  In: codes.PuncturedCode(C,position)
  ```

  donde en el parámetro C debemos indicar el código lineal a partir del cual queremos construir nuestro código punteado y en el parámetro *position* debemos introducir las posiciones en las que deseamos puntear el código.

- Empleando la función $C.punctured(L)$ del código lineal C a partir del cual deseamos construir nuestro código punteado:

  ```
  In: C.punctured(L)
  ```

  siendo el parámetro L una lista con el conjunto de posiciones en las que queremos puntear el código.

Nota 3.28. Si G es una matriz generatriz para C, una matriz generatriz para $P_J(C)$ se obtiene eliminando de la matriz G el conjunto de J columnas y omitiendo, en caso de que ocurra, las filas con todos los elementos nulos, filas duplicadas y dependientes.

Una vez construido el código punteado, la matriz generatriz se puede obtener en *Sage* como siempre mediante la función *generator_matrix*$()$ donde hay que destacar que *Sage* considera siempre la matriz en forma estándar como matriz generatriz del código.

Definición 3.29. Dado un código lineal C de parámetros $[n,k]$, y sea (J,J') una partición de $\{1,...,n\}$ la cual verifica que $J \cup J' = \{1,...,n\}$ y que $J \cap J' = \emptyset$. Diremos que el código recortado de C en J, denotado por $S_J(C)$, es el código cuyas palabras se obtienen a partir de las palabras del código de C que verifican que sus coordenadas son nulas en las posiciones de J, restringidas a las posiciones de J':

$$S_J(C) = \{(c_i)_{i\in J'} \mid c \in C\,,\, c_j = 0\, \forall j \in J\}$$

Para los códigos recortados, solo se puede realizar su construcción en *Sage* mediante el empleo de la función $C.shortened(L)$ del código lineal C a partir del cual deseamos construir nuestro código recortado:

```
In: C.shortened(L)
```

siendo el parámetro L una lista con el conjunto de posiciones que coinciden con el conjunto J de la definición 3.29.

Nota 3.30. Los códigos recortados son un subconjunto de los códigos punteados.

Nota 3.31. Si una matriz generatriz de C es G, una matriz generatriz para $S_J(C)$ se obtiene moviendo las columnas de las posiciones de J a las $|J|$ primeras columnas de la matriz y aplicando después eliminación Gaussiana sobre estas para obtener la matriz identidad en las $|J|$ primeras filas y ceros en la submatriz $k-|J| \times |J|$, es decir, obteniendo

$$G = \begin{pmatrix} I_{|J|} & T \\ O & G' \end{pmatrix}$$

donde $I_{|J|}$ es la matriz identidad cuadrada de tamaño $|J|$, T es una matriz cualquiera, O es una matriz con todas sus coordenadas nulas de tamaño $k-|J| \times |J|$ y G' es la matriz generatriz del código $S_J(C)$.

De igual forma que para los códigos punteados, la matriz generatriz se obtiene en *Sage* mediante la función $generator\_matrix()$ donde también se considera siempre la matriz en forma estándar como matriz generatriz del código.

Nota 3.32. Si el conjunto J se reduce únicamente a un valor, es decir, $J = \{j\}$, escribiremos $P_j(C)$ y $S_j(C)$ en lugar de escribir $P_{\{j\}}(C)$ y $S_{\{j\}}(C)$ respectivamente.

Ahora introducimos una proposición que es elemental para poder realizar el ataque por filtración que explicamos en el capítulo 6.

Proposición 3.33. *Dado C un código $GRS_k(a,b)$, se cumple que el código recortado de C es también un código GRS y se tiene que:*

$$S_J(C) = GRS_{k-|J|}(P_J(a), b')$$

donde las coordenadas de b' vienen dadas por:

$$b'_i = b_i \prod_{j \in J} (a_i - a_j)$$

Ejemplo 3.3. Para este ejemplo, construimos un código lineal, C, sobre el cuerpo $\mathbb{F}_2$ y parámetros $[7,3]$:

```
In: C=codes.random_linear_code(GF(2),7,3)
    C

Out: [7, 3] linear code over GF(2)
```

cuya matriz generatriz es:

```
In: C.generator_matrix()

Out: [0 0 1 1 0 1 1]
     [1 0 0 0 0 1 1]
     [0 1 1 1 1 0 0]
```

Ahora calculamos el código punteado en la primera posición de las dos formas posibles:

```
In: Cpunt=codes.PuncturedCode(C,[0])
    Cpunt

Out: Puncturing of [7, 3] linear code over GF(2) on position(s) [0]

In: Cpunt=C.punctured([0])
    Cpunt

Out: Puncturing of [7, 3] linear code over GF(2) on position(s) [0]
```

y calculamos la matriz generatriz del código punteado en la primera posición:

```
In: Cpunt.generator_matrix()

Out: [1 0 0 1 0 0]
     [0 1 1 0 0 0]
     [0 0 0 0 1 1]
```

Si comparamos la matriz generatriz obtenida para el código lineal C con la matriz generatriz obtenida para el código punteado, vemos que son bastante diferentes en relación a lo que debemos obtener según la nota 3.28. Esto es debido a que esta en forma sistemática la matriz generatriz del código punteado en la primera posición como ya comentamos. Por ello, vamos a comprobar que se genera el mismo código con la matriz generatriz obtenida para el código punteado y la que tendríamos que obtener según la nota 3.28:

```
In: G=matrix(GF(2),[[C.generator_matrix()[i,j] for j in range(1,7)]
                                                  for i in range(3)])
    G

Out: [0 1 1 0 1 1]
     [0 0 0 0 1 1]
     [1 1 1 1 0 0]
```

Para el cálculo de la matriz G simplemente quitamos la primera columna de la matriz generatriz del código lineal C. Ahora a partir de esta matriz construimos otro código lineal y obtenemos su matriz en forma estándar:

```
In: Clineal=codes.LinearCode(G)
    Clineal.systematic_generator_matrix()

Out: [1 0 0 1 0 0]
     [0 1 1 0 0 0]
     [0 0 0 0 1 1]
```

Donde vemos que se corresponde con la matriz generatriz que tenemos del código punteado en la primera posición.

Por último, vamos a calcular el código recortado en la quinta posición a partir del mismo código lineal C:

```
In: Cshort=C.shortened([4])
    Cshort

Out: [6, 2] linear code over GF(2)
```

donde su matriz generatriz es:

```
In: C.systematic_generator_matrix()

Out: [1 0 0 0 0 1 1]
     [0 1 0 0 1 1 1]
     [0 0 1 1 0 1 1]
```

3.5. Códigos cíclicos

Para finalizar el capítulo vamos a introducir el último de los códigos implementados en *Sage* que vamos a emplear en los siguientes capítulos, que son los códigos cíclicos.

Definición 3.34. Sea C un código lineal, diremos que C es un código cíclico si y solo si para cualquier palabra del código $c = (c_1, ..., c_n) \in C$ se tiene que $c' \in C$ donde $c' = (c_n, c_1, ..., c_{n-1})$.

La siguiente proposición puede considerarse como una definición alternativa de un código cíclico, donde se representan las palabras del código como polinomios en lugar de vectores, pero antes de mostrarla, vamos a introducir la definición de un concepto necesario para dicha proposición.

Definición 3.35. Llamamos ideal de un anillo A a un conjunto $I \subseteq A$ no vacio que verifica:

- Es un subgrupo aditivo de A, es decir, que I es cerrado respecto a la operación suma, el elemento neutro pertenece a A, luego $0 \in I$ y es cerrado respecto a los inversos.
- $\forall x \in I$ y $\forall a \in A$ se cumple que $ax \in I$.

Proposición 3.36. *Dado $\mathbb{F}_q[x]/\langle x^n - 1\rangle$, consideramos el siguiente isomorfismo*

$$\varphi : \mathbb{F}_q^n \longrightarrow \mathbb{F}_q[x]/\langle x^n - 1\rangle$$

$$(c_1, ..., c_n) \longmapsto c_1 + \cdots + c_n x^{n-1}$$

Se tiene que $C \subset \mathbb{F}_q[x]/\langle x^n - 1\rangle \cong \mathbb{F}_q^n$ es un código cíclico si y solo si C es un ideal de $\mathbb{F}_q[x]/\langle x^n - 1\rangle$.

Para la versión actual de *Sage*, los códigos cíclicos están limitados, de forma que solo los códigos cíclicos que verifican que su longitud, n, y tamaño u orden del cuerpo finito empleado, q, son coprimos son los que se pueden implementar, es decir, los códigos cíclicos deben cumplir que $mcd(q, n) = 1$.

A pesar de ello, muchos libros como por ejemplo [35] realizan dicha restricción para poder emplear algunas técnicas algebraicas. Dicha restricción también obliga a que el polinomio $x^n - 1$ tenga todos sus factores irreducibles distintos.

Corolario 3.37. *Dado un código cíclico C de longitud n, existe un polinomio mónico único, $g(x) \in F_q[x]$, llamado polinomio generador de C, divisor de $x^n - 1$ y tal que $C = \langle g(x)\rangle$.*

Corolario 3.38. *Los elementos de un código cíclico C de longitud n pueden identificarse con los polinomios de grado menor que n y que sean múltiplos de $g(x)$.*

Actualmente *Sage* proporciona tres formas para realizar la construcción de un código cíclico, todas ellas empleando siempre la clase:

```
In: codes.CyclicCode(self, generator_pol=None, length=None, code=None,
                     check=True, D=None, field=None, primitive_root=None)
```

cuyos parámetros se emplean parcialmente para cada una de las formas empleadas para realizar su construcción:

- Proporcionando el polinomio generador y la longitud deseada para el código cíclico en los parámetros *generator_pol* y *length* respectivamente.
- Mediante el empleo de un código lineal dado en el parámetro *code*, para el cual se calcula el polinomio generador para dicho código. Se puede emplear también el parámetro *check* si se desea comprobar si el código lineal dado es o no cíclico.
- Indicando la longitud deseada para el código cíclico, el cuerpo sobre el que queremos construirlo y un subconjunto de un conjunto para el código mediante los parámetros *length*, $field$ y D respectivamente.

Por ello, necesitamos especificar el nombre de los argumentos, además del valor que les deseamos dar. Veamos los posibles parámetros que podemos introducir:

- El parámetro *generator_pol* es donde debemos indicar el polinomio generador que hemos definido en el corolario 3.37.
- *length* indica la longitud que deseamos que tenga el código cíclico.
- *code* introducimos un código lineal a partir del cual realizar la construcción del código cíclico.
- *check* indica mediante un valor booleano si queremos que compruebe si el código introducido para generar el código es cíclico o no, dicho parámetro no se suele emplear ya que en caso de que el código no sea cíclico obtenemos directamente un error.
- $field$, indicamos el cuerpo sobre el que queremos construir el código.

Además tiene los parámetros D y *primitive_root* que permite realizar la construcción de los códigos de la tercera forma. Sin embargo, como no mostramos la teoría necesaria para realizar la construcción de los códigos cíclicos de esta forma, no explicamos dichos parámetros.

Una vez generado el código cíclico de alguna de las formas posibles, podemos obtener el polinomio generador en *Sage* mediante el empleo de la función *generator_polynomial*().

Ahora veamos como obtener una matriz generatriz y de control de forma asequible para códigos cíclicos.

Teorema 3.39. *Dado un código cíclico C con parámetros fundamentales* $[n, k]$ *y polinomio generador* $g(x) = g_0 + g_1x + \cdots + g_{n-k}x^{n-k}$*, tenemos que una matriz generatriz de C es:*

$$G = \begin{pmatrix} g_0 & g_1 & g_2 & g_3 & \cdots & g_{n-k} & 0 & 0 & \cdots & 0 \\ 0 & g_0 & g_1 & g_2 & \cdots & g_{n-k-1} & g_{n-k} & 0 & \cdots & 0 \\ 0 & 0 & g_0 & g_1 & \cdots & g_{n-k-2} & g_{n-k-1} & g_{n-k} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 & g_0 & g_1 & \cdots & g_{n-k} \end{pmatrix}$$

En *Sage* podemos obtener dicha matriz generatriz, de igual forma que para el resto de códigos, mediante el empleo de la función *generator_matrix*().

Antes de mostrar una matriz de control para los códigos cíclicos, vamos a definir el polinomio de control de un código cíclico, el cual nos facilitará la obtención de una matriz de control.

Definición 3.40. Dado un código cíclico C con parámetros fundamentales $[n, k]$ y polinomio generador $g(x)$ de grado $n - k$, llamamos polinomio de control de C al polinomio

$$h(x) = \frac{x^n - 1}{g(x)} = h_0 + h_1x + \cdots + h_kx^k$$

Sage también cuenta con la función *check_polynomial*(), que nos proporciona el polinomio de control del código cíclico.

Teorema 3.41. *Dado un código cíclico C con parámetros fundamentales* $[n, k]$ *y polinomio de control de grado k,* $h(x) = h_0 + h_1x + \cdots + h_kx^k$*, tenemos que una matriz de control de C es:*

$$H = \begin{pmatrix} 0 & 0 & \cdots & 0 & h_k & h_{k-1} & \cdots & h_1 & h_0 \\ 0 & 0 & \cdots & h_k & h_{k-1} & h_{k-2} & \cdots & h_0 & 0 \\ 0 & 0 & \cdots & h_{k-1} & h_{k-2} & h_{k-3} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ h_k & h_{k-1} & \cdots & h_0 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix}$$

También, mediante la función *parity_check_matrix*() podemos obtener una matriz de control del código cíclico.

Por último, veamos los procesos de codificación y descodificación de un código cíclico.

En cuanto al proceso de codificación de un código cíclico C con parámetros fundamentales $[n, k]$ y polinomio generador $g(x)$ existen dos posibles formas de realizarla:

- Al ser C un código lineal, se puede realizar la codificación explicada en la sección 3.2, es decir, multiplicar el vector m que deseamos codificar por una matriz generatriz de C, G.
- Para los códigos cíclicos, también se puede realizar una codificación conocida como polinómica. Para ello, tenemos que identificar el mensaje m que queremos codificar con el polinomio $m(x)$ de grado menor que k que le corresponde. Entonces para realizar su codificación solo tenemos que multiplicarlo por el polinomio generador $g(x)$:
$$g(x)a(x) = c(x) \mid c \in C$$
donde c hace referencia a la codificación de m.

Veamos los codificadores disponibles en *Sage* para estos códigos:

```
In: C.encoders_available()

Out: ['Polynomial', 'Systematic', 'Vector']
```

Por tanto, vemos que tenemos tanto el codificador *Vector* como el *Polynomial*, los cuales se corresponden con los dos métodos que hemos explicado. En adicción también tenemos el codificador *Systematic*, que realiza la codificación de igual forma que el codificador *Vector* pero empleando la matriz generatriz en forma estándar.

Una vez escogido el codificador, se emplea para realizar la codificación la función *encode*() que explicamos para los códigos lineales en la sección 3.2.

En cuanto a la descodificación de los códigos cíclicos el proceso que se emplea habitualmente es el de *síndrome-líder*, explicado en [1, Sección 2.2] del trabajo fin de grado de matemáticas, el cual es por tanto válido para cualquier código lineal. A pesar de ello, también vimos que dicho método es poco eficiente en cuanto trabajamos con códigos lineales de elevada longitud y dimensión. Sin embargo, dicho inconveniente se ve reducido notablemente debido a la estructura cíclica, ya que si C es un código cíclico, basta con realizar la corrección de los errores en una posición fija, la cual se suele tomar la última.

Esto permite reducir el tamaño de la tabla que debemos almacenar con los lideres y síndromes en n, la causa es que tenemos que guardar únicamente los síndromes y lideres cuyos lideres tengan la ultima coordenada no nula.

Por ejemplo, para [1, Ejemplo 2.4] del trabajo fin de grado de matemáticas únicamente tendríamos que almacenar el síndrome $(1,1,1)$ junto con el líder $(0,0,0,0,0,0,1)$.

De igual forma que para los codificadores, obtenemos todos los posibles descodificadores ya implementados en *Sage*:

```
In: C.decoders_available()

Out: ['InformationSet','NearestNeighbor','SurroundingBCH',
        'Syndrome','UnderlyingGRS']
```

donde de los cinco posibles emplearemos el descodificador *Syndrome*, al ser el que hemos explicado.

Entonces realizamos la descodificación empleando las funciones *decode_to_code*() y *decode_to_message*() que también explicamos para los códigos lineales en la sección 3.2.

Para terminar vamos a realizar un ejemplo para mostrar el empleo de todos los conceptos que hemos visto:

Ejemplo 3.4. Vamos a realizar la construcción de un código cíclico C sobre $\mathbb{F}_2$, cuyos parámetros fundamentales sean $[7,4]$. Para ello, emplearemos la longitud deseada junto con el polinomio generador, $g(x)$.

Según vimos en el corolario 3.37, $g(x)$ debe ser un divisor de x^7-1. Por tanto, vamos a obtener los factores en que se descompone:

```
In: F.<x> = PolynomialRing(GF(2))
    p=F(x^7-1)

In: p.factor()

Out: (x + 1) * (x^3 + x + 1) * (x^3 + x^2 + 1)
```

Como queremos que la dimensión del código sea $k=4$, tenemos que escoger un polinomio generador de grado 3, por lo que son válidos tanto x^3+x+1 como x^3+x^2+1. Elegimos $g(x)=x^3+x^2+1$ y realizamos la construcción del código cíclico C:

```
In: g=F(x**3+x**2+1)
    C=codes.CyclicCode(generator_pol = g, length = 7)
    C

Out: [7, 4] Cyclic Code over GF(2)
```

Obtenemos la matriz generatriz:

```
In: C.generator_matrix()

Out: [1 0 1 1 0 0 0]
     [0 1 0 1 1 0 0]
     [0 0 1 0 1 1 0]
     [0 0 0 1 0 1 1]
```

donde vemos que coincide con lo visto en el teorema 3.39.

Ahora obtenemos el polinomio de control del código C:

```
In: C.check_polynomial()

Out: x^4 + x^3 + x^2 + 1
```

el cual podemos observar que coincide con la multiplicación del resto de factores de $x^7 - 1$ distintos al del polinomio $g(x)$:

```
In: F(x+1)*F(x^3+x+1)

Out: x^4 + x^3 + x^2 + 1
```

Calculamos la matriz de control del código C:

```
In: C.parity_check_matrix()

Out: [1 1 1 0 1 0 0]
     [0 1 1 1 0 1 0]
     [0 0 1 1 1 0 1]
```

donde vemos que dicha matriz no coincide con la que obteníamos en el teorema 3.41, aunque es claro que también es una matriz de control ya que podemos observar que ambas están constituidas por las mismas filas pero ordenadas de distinta forma.

Ahora vamos a realizar, la codificación del vector $m = (1,0,0,1)$ empleando el codificador *Polynomial*, por tanto, primero identificamos el vector m con el polinomio $m(x) = x^3 + 1$:

```
In: m=vector(GF(2),[1,0,0,1])
    m

Out: (1, 0, 0, 1)
```

```
In: c=C.encode(F(x^3+1),'Polynomial')
    c

Out: (1, 0, 1, 0, 0, 1, 1)
```

Si identificamos c con el polinomio $c(x) = 1 + x^2 + x^5 + x^6$ podemos comprobar que la codificación coincide con realizar el producto de $m(x)$ y $g(x)$:

```
In: c=F(x^3+1)*g
    c

Out: x^6 + x^5 + x^2 + 1
```

Calculamos la distancia mínima del código C, d:

```
In: C.minimum_distance()

Out: 3
```

Al ser $d = 3$, obtenemos que el código C es capaz de corregir $t = 1$ errores. Por tanto, introducimos un vector e con peso de Hamming 1 y lo añadimos a la codificación, c, realizada obteniendo el vector y:

```
In: e=vector(GF(2),[0,0,0,1,0,0,0])
    e

Out: (0, 0, 0, 1, 0, 0, 0)

In: y=c+e
    y

Out: (1, 0, 1, 1, 0, 1, 1)
```

Por último, realizamos la descodificación del vector y. Para ello, podemos corregir el error introducido y después realizar la descodificación sobre la palabra del código obtenida, sin errores, recuperando el mensaje m:

```
In: cc=C.decode_to_code(y,'Syndrome')
    cc

Out: (1, 0, 1, 0, 0, 1, 1)

In: mm=C.decode_to_message(cc,'Syndrome')
    mm
```

```
Out: (1, 0, 0, 1)
```

También podemos calcular simultáneamente tanto la corrección de errores como su descodificación para obtener directamente el mensaje m de partida:

```
In: mm=C.decode_to_message(y,'Syndrome')
    mm

Out: (1, 0, 0, 1)
```

Capítulo 4

Códigos de Goppa

4.1. Introducción

En 1970, V.D. Goppa creó unos códigos que actualmente tienen bastante interés en la criptografía, los cuales eran conocidos por muchos autores como códigos casi aleatorios aunque son normalmente llamados códigos de Goppa en su honor.

Algunas de sus propiedades son que hay una gran cantidad de códigos de Goppa con parámetros fundamentales muy parecidos, que son difíciles de distinguir de otros códigos lineales aleatorios, se conocen algoritmos tanto de codificación como de descodificación para dichos códigos y son relativamente sencillos de construir.

Es por ello que McEliece eligió los códigos de Goppa para introducir su criptosistema en 1978, el cual mostraremos en el capítulo 5. En este tema definiremos dichos códigos y un algoritmo de descodificación para ellos entre otras propiedades.

Definición 4.1. Sea $L = (\alpha_0, ..., \alpha_{n-1}) \in \mathbb{F}_{q^m}^n$ con $\alpha_i \neq \alpha_j \ \forall i \neq j$ y $g(x)$ un polinomio de grado t con coeficientes en $\mathbb{F}_{q^m}$, de forma que $g(\alpha_i) \neq 0 \ \forall \ \alpha_i \in L$. Llamaremos *código de Goppa* correspondiente a L y $g(x)$, denotado por $\Gamma(L, g)$, al código corrector de errores formado por todos los vectores $c = (c_0, ..., c_{n-1}) \in \mathbb{F}_q$ que cumplen que:

$$R_c(x) = \sum_{i=1}^{n} \frac{c_{i-1}}{x - \alpha_{i-1}} \equiv 0 \mod g(x)$$

Nota 4.2. Si el polinomio de Goppa, $g(x)$, es irreducible entonces el código de Goppa $\Gamma(L, g)$ se dice que es irreducible y se verifica de forma trivial que $g(\alpha_i) \neq 0$ ya que si α_i fuera una raíz de $g(x)$, entonces $g(x)$ sería reducible.

Nota 4.3. Si $g(x)$ y $f(x)$ son polinomios de Goppa de cierto grado con coeficientes en $\mathbb{F}_{q^m}$ de los códigos $\Gamma(L, g)$ y $\Gamma(L, f)$ respectivamente. Se verifica que si $f(x)|g(x)$

entonces $\Gamma(L,g) \subseteq \Gamma(L,f)$. Esto es evidente ya que si $c \in \Gamma(L,g)$ se tiene que $R_c(x) \equiv 0 \mod g(x)$ y como $f(x)|g(x)$ entonces $R_c(x) \equiv 0 \mod f(x)$.

Proposición 4.4. *Dado un código de Goppa $\Gamma(L,g)$, se tiene que una matriz de control para dicho código es:*

$$H = \begin{pmatrix} g(\alpha_0)^{-1} & \cdots & g(\alpha_{n-1})^{-1} \\ g(\alpha_0)^{-1}\alpha_0 & \cdots & g(\alpha_{n-1})^{-1}\alpha_{n-1} \\ \vdots & & \vdots \\ g(\alpha_0)^{-1}\alpha_0^{t-1} & \cdots & g(\alpha_{n-1})^{-1}\alpha_{n-1}^{t-1} \end{pmatrix}$$

donde $\alpha_i \in L \; \forall i \in \{0,...,n-1\}$ y $g(\alpha_i)^{-1}$ hace referencia al elemento inverso del polinomio de Goppa g evaluado en α_i.

En la siguiente proposición mostramos algunas propiedades básicas de estos códigos.

Proposición 4.5. *Sea el código de Goppa $\Gamma(L,g)$. Tenemos que:*

a) $\Gamma(L,g)$ es un código lineal.

b) La longitud de las palabras del código $c \in \Gamma(L,g)$ es $n = |L|$.

c) La dimensión del código $\Gamma(L,g)$ sobre $\mathbb{F}_q$ es $k \geq n - mt$.

d) La distancia mínima de $\Gamma(L,g)$ es $d \geq t+1$.

Por tanto, para la obtención de la matriz generatriz del código, recurrimos a la forma vista en la definición 3.10 de los códigos lineales.

Además, vemos que la distancia mínima de los códigos de Goppa es mayor que $t+1$. Sin embargo, si nos restringimos sobre ciertos polinomios y el cuerpo $\mathbb{F}_2$ podemos garantizar una distancia mínima de al menos $2t+1$.

Proposición 4.6. *Si el código de Goppa $\Gamma(L,g)$ se define sobre $\mathbb{F}_2$, con $L = (\alpha_0,...,\alpha_{n-1}) \in \mathbb{F}_{2^m}^n$ y el polinomio de Goppa $g(x) \in \mathbb{F}_{2^m}[x]$ de grado t es separable, es decir, el polinomio tiene todas sus raíces simples, entonces la distancia mínima del código $d \geq 2t+1$.*

Por tanto, para estos casos doblamos la capacidad correctora de los códigos de Goppa. Esta es la principal causa por la que realizaremos únicamente la implementación de los códigos de Goppa sobre el cuerpo $\mathbb{F}_2$.

Para realizar dicha implementación, vamos a construir los códigos de Goppa completamente, ya que *Sage* no tiene actualmente implementado ninguna función acerca de estos códigos. Para su construcción, creamos una clase, que llamamos *Goppa*,

la cual estará dotada de un constructor, denotado en *Sage* como la función llamada $\_\_init\_\_(n, m, t, LL = None)$. Dicho constructor realiza la construcción del código a partir de la obtención de la matriz de control, H, según lo visto en la proposición 4.4. Para ello, es necesario introducir tres parámetros al constructor:

- n indica la longitud deseada del código de Goppa que vamos a construir.
- m la extensión del cuerpo $\mathbb{F}_2$ que queremos emplear en ese código.
- t es un parámetro que admite dos tipos distintos de valores. Por un lado, podemos introducir un entero positivo que hace referencia a la capacidad correctora del código y también al grado del polinomio de Goppa $g(x)$ que debemos emplear. Por otro lado, podemos introducir un polinomio que debe tener como variable, x, estar construido sobre la misma extensión del cuerpo que deseamos emplear y ser irreducible o al menos separable.

En adicción, presenta el parámetro opcional LL por si queremos indicar la primera potencia del elemento generador del cuerpo F_{2^m} en que se encuentra el código de Goppa, el cual se emplea para realizar la construcción del vector L, cuando la longitud n que emplea el código no es máxima.

Como hemos visto, el parámetro t puede admitir dos tipos distintos de datos. Por ello, después de comprobar que los parámetros n y m sean positivos, distinguimos el tipo de información que contiene t:

```
In:  if t.parent()==PolRing and t.is_squarefree():
         g=t;
         t=g.degree();
```

donde primero comprobamos si t es un polinomio recurriendo a la función $parent()$, la cual proporciona el tipo de datos que almacena la variable, que comparamos con una variable creada anteriormente con el nombre de $PolRing$, que hace referencia al anillo de polinomios sobre el que trabajamos. Después vemos si es irreducible o separable mediante la función $is\_squarefree()$.

Si el parámetro t es un entero positivo, lo validamos y entonces generamos un polinomio irreducible cuyo grado es el valor de t.

```
In:  elif boolInt and t>0:
         g=PolRing.irreducible_element(t);
```

Una vez que tenemos identificados los distintos parámetros a emplear en el código, procedemos a construir una matriz de control del código de Goppa, H. Para ello, como tenemos ya calculado el polinomio de Goppa $g(x)$, únicamente necesitamos obtener el vector L de tamaño n. Si el tamaño de dicho vector es igual al tamaño del cuerpo $\mathbb{F}_{2^m}$ simplemente construimos L a partir de todos los elementos del cuerpo:

```
In: if n==2**m:
        for i in range(n):
            if i==(n-1):
                L[i]=F(0);
            else:
                L[i]=F.list()[i+1];
```

En caso de que el valor de n sea menor que el tamaño del cuerpo, elegimos un entero aleatorio menor que n, el cual nos permite obtener aleatoriamente el primer elemento del cuerpo a emplear, obteniéndolo como potencia del elemento primitivo del cuerpo:

```
In: elif n<2**m:
        aleatorio=randint(0,n-1)
        for i in range(n):
            L[i]=(F.gen())**(aleatorio+ i+1);
```

Una vez que tenemos calculado g y L, para construir la matriz H multiplicamos una matriz V, construida únicamente por las potencias de los elementos del vector L, por una matriz diagonal formada por los elementos $g(\alpha_i)^{-1}$.

```
In: for i in range(t):
        for j in range(n):
            try:
                V[i,j]= L[j]^i;
            except ArithmeticError: #Evitamos la posible indeterminacion 0^0
                V[i,j]= 1;
    D=diagonal_matrix([1/g(L[i]) for i in range(n)]);
    H=V*D;
```

Entonces a partir de la matriz de control, H, calculamos la matriz generatriz del código, G, según lo visto en la definición 3.10.

```
In: Htrans=transpose(H);
    G=Htrans.left_kernel().basis_matrix();
```

Sin embargo, para realizar la codificación y descodificación de los códigos de Goppa mas fácilmente, trabajaremos únicamente en el cuerpo $\mathbb{F}_2$. Para ello, es necesario realizar una técnica conocida como descenso del cuerpo $\mathbb{F}_{2^m}$ a $\mathbb{F}_2$, que consiste en

identificar cada elemento de $\mathbb{F}_{2^m}$ con un vector de $\mathbb{F}_2^m$. Para ello, se fija una base formada por las potencias de un elemento primitivo de $\mathbb{F}_{2^m}$ y después se identifican los elementos como vectores en esa base, por ejemplo, si α representa dicho elemento, la base que se considera habitualmente es $\{1, \alpha, ..., \alpha^{m-1}\}$. Si ahora tomamos el elemento $\alpha + 1$, este se identificará con el vector $(1, 1, 0, ..., 0)$ de longitud m.

Para realizar este proceso, *Sage* cuenta con una clase que pertenece a otra clase experimental actualmente la cual se llama $codes.SubfieldSubcode()$, en la que muchas propiedades aún no están disponibles aunque si la que nosotros necesitamos, la cual se encuentra en la clase llamada $embedding()$. En dicha clase, hay implementado un método llamado $relative\_field\_representation()$ que realiza la identificación entre un elemento de $\mathbb{F}_{q^m}$ y su correspondiente vector de $\mathbb{F}_2^m$:

```
In: E=Csubfield.embedding()
    for i in range(H.nrows()):
        for j in range(H.ncols()):
            elemento=E.relative_field_representation(H[i,j])
            Hbin[m*i:m*(i+1),j] = elemento;
```

Una vez obtenida la matriz de control formada únicamente por elementos de $\mathbb{F}_2$, denotada *Hbin*, obtenemos la matriz generatriz binaria de igual forma que antes, la cual tendrá $k \geq n - mt$ filas según lo visto en la proposición 4.5.

```
In: Gbin=transpose(Hbin).left_kernel().basis_matrix();
```

Por último, antes de finalizar la ejecución del constructor de la clase implementada, *Sage* tiene una variable preasignada a la clase llamada $self$, que permite emplear todas las variables que hayamos generado durante la ejecución del constructor en otras funciones de la clase que las necesitan. Para ello, simplemente hay que asignarle un nombre dentro de la variable $self$ e igualarlo a la variable que deseamos almacenar, por ejemplo, para guardar la variable n empleamos:

```
In: self._n = n;
```

4.2. Codificación

En cuanto al proceso de codificación, este se realiza de forma semejante a la vista en la definición 3.9 de los códigos lineales, es decir, multiplicando el vector que deseamos codificar por la matriz generatriz del código de Goppa que vamos a emplear.

En *Sage* implementamos dos funciones para realizar la codificación. La primera se llama $\_codificar\_vector(self, vectorbin)$, la cual denotamos según la convención establecida como una función privada ya que queremos que únicamente sea utilizada por la segunda función llamada $codificar(self, mensaje)$.

La función $codificar(self, mensaje)$ tiene como parámetro a introducir por el usuario solamente *mensaje*, el cual puede ser una cadena de caracteres, una lista de caracteres o un vector construido sobre el cuerpo $\mathbb{F}_2$. Si es una cadena o lista de caracteres importamos un método de una librería de *Sage* llamado *ascii_to_bin* para transformar el parámetro *mensaje* en un vector sobre $\mathbb{F}_2$, el cual introducimos como parámetro en la función $\_codificar\_vector()$, que llamamos para realizar la codificación, cuyo resultado es el devuelto. Si el parámetro *mensaje* es directamente un vector sobre $\mathbb{F}_2$, llamamos directamente a la función $\_codificar\_vector()$, cuyo resultado es también el devuelto.

```
In: def codificar(self,mensaje):
        if type(mensaje)==str or type(mensaje)==list:
            mensajebin=ascii_to_bin(mensaje);
            vectorbin=vector(self._K, [self._K(str(list(mensajebin)[i]))
                                       for i in range(len(list(mensajebin)))]);
            return self._codificar_vector(vectorbin);
        elif mensaje.is_vector() and mensaje.base_ring()== GF(2):
            return self._codificar_vector(mensaje);
```

En cuanto a la función $\_codificar\_vector(self, vectorbin)$ tiene como parámetro a introducir el llamado *vectorbin* el cual, a diferencia de las funciones implementadas ya en *Sage* para codificar en otros de los códigos vistos en el capítulo 3, permite realizar la codificación de vectores binarios cuyo tamaño no necesariamente se corresponde con la dimensión del código. El motivo principal para realizarlo de esta forma surge al querer implementar la transmisión de caracteres ASCII, donde cada uno de ellos esta compuesto por 8 bits. Por tanto, no es posible en la mayoría de los casos tener fijo el tamaño del mensaje a transmitir de forma que este sea un múltiplo de 8 y también un múltiplo de la dimensión, k. En el caso particular en que si estemos en la situación en que sea múltiplo de ambos, empleamos:

```
In: iterar=RR(len(vectorbin))/RR(self._k);
    if iterar.is_integer():
        lista=[];
        for i in range(iterar):
            bb=vectorbin[i*self._k:(i+1)*self._k]*self._Gbin;
            lista+=[bb[i] for i in range(len(bb))]
```

Si esto no ocurre, sobre el mensaje ya convertido en vector binario, se incrementa su longitud añadiendo cierta cantidad de elementos nulos al final hasta obtener un vector que sea múltiplo de k y de esta forma poder dividir dicho vector en bloques de tamaño k que podemos codificar mediante la matriz generatriz binaria del código.

```
In: else:
    lista=[];
    for i in range(iterar.ceil()):
       if i<>(iterar.ceil()-1):
           bb=vectorbin[i*self._k:(i+1)*self._k]*self._Gbin;
           lista+=[bb[i] for i in range(len(bb))];
       else:
           tamrest=len(vectorbin)-i*self._k;
           numeroceros= self._k-tamrest;
           listrest=list(vectorbin[len(vectorbin)-tamrest:len(vectorbin)]);
           listrest+=[0 for i in range(numeroceros)];
           vectrest= vector(self._K, [self._K(listrest[i]) for i in
                                                  range(len(listrest))]);
           bb=vectrest*self._Gbin;
           lista+=[bb[i] for i in range(len(bb))];
```

El principal problema que surge al añadir elementos nulos al vector se encuentra en la hora de realizar la descodificación como vemos a continuación en la sección 4.3. Una vez que se realiza todo el proceso del método $\_codificar\_vector()$, se devuelve un vector binario que contiene la correspondiente codificación del mensaje.

4.3. Descodificación

Si consideramos una palabra del código $c = (c_0, ..., c_{n-1}) \in \Gamma(L, g)$ la cual durante la transmisión se convierte en $y = (y_0, ..., y_{n-1}) = c + e$ donde $e = (e_0, ..., e_{n-1})$ es el vector de error que surge respecto la palabra c. Como vamos a construir únicamente códigos binarios, suponemos que $w(e) \leq t$, ya que t es su capacidad correctora al ser $2t + 1$ su distancia según la proposición 4.6.

Para simular estos errores que pueden ocurrir durante la transmisión o que necesitamos añadir cuando estemos realizando el procedimiento de cifrado del criptosistema de McEliece que veremos en el capítulo 5, hemos construido una función llamada $introducir\_errores(self, mensajecodificado, cantidad)$ que permite incorporarlos.

```
In: if mensajecodificado.is_vector() and mensajecodificado.base_ring()
                            == GF(2) and len(mensajecodificado)>= self._n:
```

```
iterar=RR(len(mensajecodificado))/RR(self._n);
if iterar.is_integer():
     for i in range(iterar):
          indices =range(self._n);
          for j in range(cantidad):
               nn=indices.pop(randint(0,len(indices)-1));
               mensajecodificado[i*self._n+nn]+= 1;
          return mensajecodificado;
```

Podemos observar, que los dos parámetros que debe introducir el usuario se corresponden con *mensajecodificado*, que es el vector binario sobre el que queremos introducir los errores y *cantidad* donde indicamos la cantidad de errores que deseamos introducir por cada bloque de tamaño n sobre el vector binario.

Para la corrección de los errores producidos, vamos a introducir los conceptos de síndrome y polinomio localizador y evaluador de errores para los códigos de Goppa.

Definición 4.7. Llamamos síndrome del vector $y = (y_0, ..., y_{n-1})$ para el código de Goppa $\Gamma(L, g)$ y lo denotamos $s(x)$ al valor:

$$s(x) = \sum_{i=0}^{n-1} \frac{y_i}{x - \alpha_i} (\text{mod } g(x)) = \sum_{i=0}^{n-1} \frac{e_i}{x - \alpha_i} (\text{mod } g(x))$$

donde $e = (e_0, ..., e_{n-1})$ hace referencia al vector de errores.

En *Sage*, al final de la ejecución del constructor hemos guardado en una variable de tipo matriz llamada *SyndromeC* el conjunto de valores $\frac{1}{x - \alpha_i} (\text{mod } g(x))$, para evitar realizar su cálculo cada vez que vayamos a obtener un síndrome distinto. Ademas, dicha variable es de tipo matriz en lugar de ser un vector para que cuando lo multipliquemos por un vector se realice automáticamente la suma de todos ellos.

```
In: SyndromeC = matrix(PolRing,1, len(L));
    for i in range(len(L)):
        SyndromeC[0,i] = (PolRing.gen()-L[i]).inverse_mod(g);
```

Definición 4.8. Llamamos polinomios localizador y evaluador de errores, denotados respectivamente por $L(x)$ y $E(x)$ a los polinomios dados de la forma:

$$L(x) = \prod_{i|e_i \neq 0} (x - \alpha_i) \ , \ E(x) = \sum_{i|e_i \neq 0} e_i \prod_{j|e_j \neq 0, j \neq i} (x - \alpha_j)$$

Nota 4.9. Como vamos a emplear solo los códigos de Goppa binarios, basta con calcular el polinomio localizador de errores, $L(x)$, ya que una vez calculadas las posiciones en que se encuentra el error, como los únicos valores posibles son 0 y 1, simplemente habría que cambiar en dichas posiciones el valor que tenemos.

Para la obtención del polinomio localizador de errores que nos permite corregir los errores producidos, vamos a explicar el algoritmo de Patterson, cuyo uso es exclusivo para códigos de Goppa binarios.

4.3.1. Algoritmo de Patterson

Dado un código de Goppa binario, $\Gamma(L, g)$, tenemos que $g(x)$ es un polinomio con coeficientes en $\mathbb{F}_{2^m}$ y $L = (\alpha_0, ..., \alpha_{n-1}) \in \mathbb{F}_{2^m}^n$.

El algoritmo consiste primero en calcular el síndrome del vector y visto en la definición 4.7 y a partir del síndrome, calculamos el polinomio localizador de errores $L(x)$, para ello:

1) Tenemos que buscar el polinomio $f(x)$ que cumpla que $s(x)f(x) \equiv 1 \bmod g(x)$. Si dicho polinomio es la identidad, es decir, $f(x) = x$ tendríamos que $L(x) = x$ y hemos terminado. Si esto no ocurre, seguimos realizando los pasos siguientes.

2) Calculamos el polinomio $h(x)$ tal que $h^2(x) \equiv f(x) + x \bmod g(x)$.

3) Buscamos los polinomios $a(x)$ y $b(x)$ de grado mínimo que cumplan que $h(x)b(x) \equiv a(x) \bmod g(x)$.

4) El polinomio localizador de errores será $L(x) = a^2(x) + xb^2(x)$.

Ahora, con el polinomio localizador de errores que hemos calculado procedemos a obtener las posiciones en que hay errores $(i \mid e_i \neq 0)$. Para ello, calculamos las raíces del polinomio $L(x)$, las cuales, deben ser elementos del vector L. Entonces, las posiciones de errores se corresponden con las posiciones que ocupan en el vector L los elementos que son raices del polinomio localizador de errores, $L(x)$.

Por tanto, una vez que conocemos el vector de errores $e = (e_0, ..., e_{n-1})$, ya podemos obtener la palabra codificada sin errores: $c = y - e = y + e$.

4.3.2. Funciones auxiliares

Para poder realizar todos los pasos del algoritmo en *Sage*, es necesario implementar un conjunto de funciones adicionales que permitan realizar cada paso mas fácilmente.

En el paso 2), es necesario calcular la raíz cuadrada de un polinomio con coeficientes en $\mathbb{F}_{2^m}[x]$, lo cual es difícil de obtener en la mayoría de casos. Sin embargo, podemos observar que todas las raíces que deseamos hallar, son elementos de la forma $f(x)+x$. Para este tipo de elementos, en el artículo [28] se muestra un método, el cual denotamos *_dividir_polinomio*$(self, p)$, que permite obtener dichas raíces fácilmente.

```
In: def _dividir_polinomio(self,p):
        Phi = p.parent()
        p0 = Phi([sqrt(c) for c in p.list()[0::2]]);
        p1 = Phi([sqrt(c) for c in p.list()[1::2]]);
        return (p0,p1);
```

Como podemos ver, dicho método consiste en dividir el polinomio, $f(x)+x$ del cual queremos calcular su raíz cuadrada, en los polinomios cuyos coeficientes son las raíces cuadradas de los coeficientes de los términos con exponente impar, $f_0(x)$ y par, $f_1(x)$. Al realizar esto, se verifica que $f(x)+x = f_0(x)^2 + xf_1(x)^2$.

Debido a esto tenemos que $\sqrt{f(x)+x} = f_0(x) + \sqrt{x}f_1(x)$ ya que en cuerpos binarios sabemos que se verifica que $a(x)^2 + b(x)^2 = (a(x)+b(x))^2$ con $a(x), b(x) \in \mathbb{F}_{2^m}[x]$.

Por tanto, si conseguimos calcular la raíz cuadrada de x, seremos capaces de calcular cualquier raíz cuadrada del tipo que necesitamos. Para su obtención, vamos a mostrar una técnica semejante a la explicada antes, ya que tenemos que aplicar también la función *_dividir_polinomio*$(self, p)$ ahora sobre el polinomio de Goppa $g(x)$ obteniendo de igual forma $g(x) = g_0(x)^2 + xg_1(x)^2$. A partir de ello, tenemos que $\sqrt{x} = g_0(x)g_1^{-1}(x)$ ya que podemos ver que $g(x) - g_0(x)^2 = xg_1(x)^2$ y aplicando ahora modulo $g(x)$ y obteniendo el polinomio inverso de $g_1(x)^2$ tenemos que $x \equiv g_0(x)^2 g_1(x)^{-2}$.

Para el paso 3) en lugar de implementar una función del algoritmo de Euclides extendido como se indica en el artículo [28], vamos a implementar un método mas sutil explicado por Bernstein en el artículo [29], al permitir este método incorporar el elemento nulo de $\mathbb{F}_{2^m}$, con lo que podemos alcanzar la máxima longitud que puede tener un código sobre un determinado cuerpo finito.

Para la introducción del método, vamos a introducir dos definiciones dadas por Bernstein en el artículo [29] de dos conceptos necesarios.

Definición 4.10. Llamamos norma de un polinomio $f \in \mathbb{F}_{2^m}[x]$, que denotamos $|f|$, al valor $|f| = 2^{\deg f}$ si $f \neq 0$ y $|f| = 0$ si $f = 0$.

Definición 4.11. Llamamos longitud de un vector $(a, b) \in \mathbb{F}_{2^m}[x]^2$, que denotamos $|(a, b)|$, a la norma del polinomio $a^2 + xb^2$, es decir, $|(a, b)| = |a^2 + xb^2|$.

Por tanto, definimos en *Sage* una función que calcule la longitud de dos polinomios dados a partir de la norma explicada en la definición 4.10.

```
In: def _longitud_norma(self,a,b):
        X = self._g.parent().gen();
        return 2**((a**2+X*b**2).degree());
```

El principal problema del paso 3) se encuentra en que los valores que tenemos que calcular los cuales cumplen la congruencia, deben tener grado mínimo. Sin embargo, este problema desaparece rápidamente mediante este método ya que únicamente existen un par de polinomios a, b que verifican la congruencia y su longitud es menor que 2^t, siendo t la capacidad correctora del correspondiente código de Goppa binario. Esta propiedad también es probada por Bernstein en [29].

Por último, la técnica básica empleada por el método implementado es semejante a la del algoritmo de Euclides extendido, donde vamos reduciendo los polinomios mediante el empleo de la función *quo_rem*() ya implementada en *Sage*, la cual te devuelve tanto el cociente como el resto de una división dada. Para ello, tenemos que partir de los vectores $(h(x), 1)$ y $(g(x), 0)$, donde vemos que únicamente varía el polinomio $h(x)$, por ello es el único parámetro que se debe introducir en la función *_reduccion_base*() implementada para realizar el método explicado.

```
In: def _reduccion_base(self, s):
        g = self._g;
        t = g.degree();
        a = [];
        b = [];
        a.append(0);
        b.append(0);
        (q,r) = g.quo_rem(s);
        (a[0],b[0]) = simplify((g - q*s, 0 - q))
        if self._longitud_norma(a[0],b[0]) > 2**t:
            a.append(0); b.append(0);
            (q,r) = s.quo_rem(a[0]);
            (a[1],b[1]) = (r, 1 - q*b[0]);
        else:
            return (a[0], b[0]);
        i = 1;
        while self._longitud_norma(a[i],b[i]) > 2**t:
            a.append(0); b.append(0);
            if a[i]==0:
                return (a[i-1],b[i-1]);
```

```
            else:
                (q,r) = a[i-1].quo_rem(a[i]);
                 (a[i+1],b[i+1]) = (r, b[i-1] - q*b[i]);
            i+=1;
        return (a[i],b[i]);
```

Para que este método funcione correctamente es necesario que la cantidad de errores introducida en el vector sea mayor que 2 y por tanto que la capacidad correctora del código empleado sea de al menos 3. Esto no supone una limitación excesiva ya que los códigos considerados actualmente con buenos parámetros fundamentales tienen una capacidad correctora mucho mayor a 3. Además, cuando empleemos estos códigos en el criptosistema de McEliece que veremos en el capítulo 5, siempre se suele introducir la máxima cantidad de errores posibles para realizar el proceso de cifrado, por lo que tampoco supone ningún problema.

4.3.3. Función de descodificación

Una vez introducidas las funciones necesarias, vamos a mostrar la parte esencial de la función implementada llamada $descodificar(self, mensaje, ASCII, decodificarV)$. En dicha función tenemos 3 parámetros, donde *mensaje* hace referencia al vector binario codificado que tiene o no errores, y los parámetros $ASCII$ y $decodificarV$ son ambos de tipo booleano, en los cuales debemos indicar respectivamente si queremos que se transforme el vector obtenido en caracteres o no, y si queremos que solo se realice la corrección de errores o también que se descodifique el mensaje.

Tras indicar los parámetros de la función, en primer lugar se divide el vector en bloques de tamaño n, al ser este el tamaño correspondiente de las palabras del código que estamos empleando. Hacemos hincapié en que esta vez la división en bloques de tamaño n debe ser exacta, ya que el vector debe estar formado por un conjunto de palabras del código, con o sin errores. Una vez realizamos dicha división, calculamos el síndrome de cada bloque, donde tenemos que diferenciar dos posibles casos.

El primero es que el síndrome obtenido sea el elemento nulo, lo cual significa que dicho bloque es una palabra del código y por tanto no se han producido errores. Por otro lado, si este no es nulo entonces dicho bloque presenta errores, por lo que aplicamos los pasos del algoritmo de Patterson para corregirlos:

```
In: if(sindrome<>0):
        pol_sindrome = sindrome[0,0];
        (g0,g1)= self._dividir_polinomio(self._g);
        w= g0*g1.inverse_mod(self._g);
        T=pol_sindrome.inverse_mod(self._g)
```

```
        if T== self._g.parent().gen():
            sigma= self._g.parent().gen();
        else:
            (T0,T1)= self._dividir_polinomio(T+ self._g.parent().gen());
            R= (T0 + w*T1).mod(self._g);
            (a,b)= self._reduccion_base(R);
            sigma= a**2+self._g.parent().gen()*b**2;
        j=Integer(0);
        for a in self._L:
            if sigma(a)==0:
                trozomensaje[j]+= 1;
            j+=1;
```

Para encontrar el polinomio f del paso 1), tenemos que calcular el inverso del polinomio del síndrome, $s(x)$, modulo $g(x)$, para ello *Sage* cuenta con una función ya implementada para elementos de distintos tipos, entre ellos polinomios con coeficientes en un cuerpo finito, llamada *inverse_mod*(), empleada como podemos ver en la línea 5 del código mostrado.

Para realizar el paso 2), hacemos uso de la función auxiliar privada *_dividir_polinomio*() vista en la sección 4.3.2, la cual empleamos para obtener la raíz cuadrada según el procedimiento explicado. Después llamamos a la función privada *_reduccion_base*() para obtener los polinomios a y b que nos permiten calcular el polinomio localizador de errores.

Por último, las 5 filas del final del código permiten obtener las raíces del polinomio localizador de errores, de forma que cuando obtengamos una raíz, cambiamos el valor de esa posición del bloque de mensaje, recuperando el bloque sin errores.

Una vez que recuperamos todo el mensaje sin errores, procedemos a descodificarlo si el usuario lo ha indicado mediante el parámetro booleano $decodificarV$:

```
In [ ]: if decodificarV==True:
            Gcuadrada=matrix(GF(2),self._k);
            while Gcuadrada.rank()<> self._k or Gcuadrada.ncols()<>self._k:
                columnas=[];
                totalcolumnas=range(self._n);
                Gcuadrada=self._Gbin;
                for i in range(self._k):
                    aleatorio=randint(0,len(totalcolumnas)-1);
                    columnas.append(totalcolumnas.pop(aleatorio));
                columnas.sort();
```

```
            Gcuadrada=transpose(matrix(GF(2),[self._Gbin.column(a)
                                              for a in columnas]));
        partemensaje=vector(GF(2),[trozomensaje[j] for j in columnas]);
        mensajedescifrado=partemensaje*Gcuadrada.inverse();
        lista=lista+[a for a in mensajedescifrado];
```

Como podemos observar, para realizar la descodificación calculamos una matriz cuadrada, M, que sea no singular formada por k columnas de la matriz generatriz del código. Entonces, mediante la multiplicación del vector formado por las coordenadas del bloque de mensaje correspondientes con las columnas que forman la matriz M, y la inversa de la matriz M nos permiten recuperar el mensaje descodificado.

Para finalizar la ejecución de la función, si el usuario desea su conversión a caracteres ASCII, en caso de que haya indicado en adicción que se realizara la descodificación, se emplean las siguientes sentencias:

```
In [ ]: if ASCII==True and decodificarV==True:
            h=RR(len(lista))/RR(8);
            i=0;
            continuar=True;
            ascii='';
            while i<h.ceil() and continuar:
                if i<>(h.ceil()-1):
                    caracterbin=lista[i*8:(i+1)*8];
                    if vector(GF(2),caracterbin)==0:
                        continuar=False;
                    else:
                        ascii+=bin_to_ascii(caracterbin);
                else:
                    if h.is_integer():
                        caracterbin=lista[i*8:(i+1)*8];
                        ascii+=bin_to_ascii(caracterbin);
                i+=1;
            return ascii;
        else:
            return vector(GF(2),lista);
```

Tenemos que destacar que al realizar el proceso de descodificación, recuperamos el mensaje junto con los elementos nulos que hemos tenido que añadir en caso de que la longitud del mensaje no fuera un múltiplo de k, como vimos en la sección 4.2. Si hemos partido de un conjunto de caracteres ASCII, podemos identificar fácilmente la cantidad de elementos nulos que se han incorporado al mensaje, ya que la longitud del

mensaje debe ser un múltiplo de 8 y además no existe ningún carácter ASCII que se identifique con el bloque formado por 8 elementos nulos. Por tanto, en cuanto tengamos un bloque de longitud 8 formado únicamente por elementos nulos o si llegamos al bloque final del mensaje y su longitud no es 8, paramos la conversión y por tanto, eliminamos exitosamente todos los elementos nulos introducidos.

Sin embargo, en caso de que se realice la descodificación del mensaje y no su conversión a caracteres ASCII, no seremos capaces de identificar los elementos nulos introducidos, al no conocer el tamaño original del mensaje. Esto en realidad no supone ninguna limitación extra respecto a cualquiera de los códigos implementados en *Sage* que ya hemos visto en el capítulo 3, ya que para realizar la codificación en dichos códigos era necesario introducir un mensaje cuya longitud fuera la dimensión del código.

4.4. Otras funciones

Ademas de las funciones que hemos explicado hasta el momento, también hemos implementado un conjunto de funciones que nos permiten obtener propiedades fundamentales del código creado:

- *minimum_distance*() es una función que devuelve la distancia mínima del código de Goppa.
- *generator_matrix_nobin*() es una función que nos proporciona la matriz generatriz formada por elementos de $\mathbb{F}_{2^m}$ del código de Goppa binario creado.
- *parity_check_matrix_nobin*() es una función que nos proporciona la matriz de control formada por elementos de $\mathbb{F}_{2^m}$ del código de Goppa binario creado.
- *generator_matrix*() es una función que nos proporciona la matriz generatriz formada por elementos de $\mathbb{F}_2$ del código de Goppa binario creado.
- *parity_check_matrix*() es una función que nos proporciona la matriz de control formada por elementos de $\mathbb{F}_2$ del código de Goppa binario creado.
- *cuerpo_finito*() es una función que nos proporciona el cuerpo $\mathbb{F}_2$.
- *polinomio_g*() es una función que nos permite saber cual es el polinomio de Goppa empleado para la generación del código.
- *dimension*() es una función que nos proporciona la dimensión del código de Goppa generado a partir de la matriz generatriz formada por elementos de $\mathbb{F}_2$.
- *length*() es una función a partir de la cual obtenemos la longitud del código de Goppa.

- *base_field*() es una función que nos devuelve el cuerpo $\mathbb{F}_{2^m}$ sobre el que se ha construido el código de Goppa.
- *is_Goppa*() es una función que nos permite diferenciar un código de Goppa de otro código cualquiera, devolviendo en caso de que sea de Goppa el valor booleano *True*.

Podemos observar que muchas de estas funciones se llaman de igual forma y devuelven el mismo resultado que funciones de otros códigos ya implementados en *Sage* vistos en el capítulo 3. Esto permite que se puedan emplear las mismas sentencias de código independientemente del código que estemos empleando, por ejemplo, cuando realicemos la implementación del criptosistema de McEliece con diferentes códigos en el capítulo 5.

Para finalizar el capítulo, vamos a mostrar la construcción de un código de Goppa, así como la codificación y correspondiente descodificación, tras incorporar una cantidad de errores, de un mensaje formado por caracteres ASCII.

Ejemplo 4.1. En este ejemplo, vamos a construir un código de Goppa sobre el cuerpo $\mathbb{F}_{2^4}$, con máxima longitud, es decir, $n = 16$ y que sea capaz de corregir $t = 3$ errores. Para ello, fijamos de antemano el polinomio de Goppa a $g(x) = x^3 + a^2x + a$, siendo a un elemento primitivo de $\mathbb{F}_{2^4}$, el cual podemos ver también que es irreducible:

```
In: K.<a>=GF(2);
    F.<a>=GF(2**4);
    PolRing=PolynomialRing(F,'x');
    x=PolRing.gen();
    polinomiogoppa=x^3+F.gen()^2*x+F.gen();
    polinomiogoppa

Out: x^3 + a^2*x + a

In: polinomiogoppa.is_irreducible()

Out: True
```

Llamamos a la clase *Goppa* e introducimos los parámetros para realizar la construcción del código Goppa, que almacenamos en la variable *codigo_goppa*:

```
In: codigo_goppa=Goppa(2**4,4,polinomiogoppa);
```

Obtenemos directamente las matrices generatriz y de control del código de Goppa formadas por elementos de $\mathbb{F}_2$:

```
In: G=codigo_goppa.generator_matrix()
    G

Out: [1 0 0 0 1 0 1 0 1 1 0 0 0 0 1 1]
     [0 1 0 0 1 0 0 1 0 0 1 0 1 1 1 1]
     [0 0 1 0 0 0 0 1 0 1 0 1 0 1 1 1]
     [0 0 0 1 1 1 1 0 0 0 0 1 1 0 1 1]

In: H=codigo_goppa.parity_check_matrix()
    H

Out: [1 0 1 1 1 0 1 0 0 0 0 0 0 0 0 1]
     [0 0 1 0 1 0 0 0 1 1 1 1 1 0 1 0]
     [0 1 0 0 1 0 1 0 1 0 1 1 0 0 1 0]
     [1 0 0 0 1 1 1 1 1 1 0 0 0 1 0 1]
     [1 1 1 1 0 0 0 0 0 1 0 0 1 0 0 0]
     [0 1 1 1 0 1 1 1 1 1 1 0 0 0 1 0]
     [0 0 0 0 1 0 1 1 1 0 0 1 0 1 1 0]
     [0 0 1 0 0 1 0 1 0 0 0 0 1 0 0 0]
     [0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0]
     [1 0 1 0 1 0 1 1 0 0 1 0 1 1 1 0]
     [0 1 1 1 0 0 1 0 0 0 1 0 1 0 1 0]
     [0 1 0 0 1 0 1 0 1 1 1 1 1 0 0 0]
```

Como la matriz generatriz G tiene cuatro filas es claro que la dimensión del código es $k = 4$. A pesar de ello, podemos llamar a la función *dimension*() para comprobarlo:

```
In: codigo_goppa.dimension()

Out: 4
```

Ahora vamos a realizar la codificación de un mensaje formado por caracteres ASCII. Para que no sea muy extenso, codificaremos simplemente una palabra:

```
In: codificado=codigo_goppa.codificar('Hola')
    codificado

Out: (0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1,
0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0,
1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0,
1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1,
1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0,
1, 1, 0, 1, 1)
```

Podemos observar que al ser la dimension del código $k = 4$ y los bloques tener longitud 8, no es necesario añadir ningún elemento nulo para realizar la codificación completa.

Ahora llamamos a la función *introducir_errores*() para que añada tres errores por cada bloque del mensaje codificado:

```
In: cifrado= codigo_goppa.introducir_errores(codificado,3)
    cifrado
```

```
Out: (0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1,
1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0,
1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0,
1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1,
1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0,
1, 1, 0, 0, 1)
```

Por último, llamamos a la función *decodificar*() y realizamos en primer lugar únicamente la corrección de errores, dando a los parámetros $ASCII$ y *decodificarV*() el valor *False*:

```
In: codigo_goppa.decodificar(codificado, False, False)
```

```
Out: (0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1,
0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0,
1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0,
1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1,
1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0,
1, 1, 0, 1, 1)
```

En segundo lugar, realizamos su descodificación pero no transformamos el vector binario en caracteres ASCII:

```
In: codigo_goppa.decodificar(cifrado,False , True)
```

```
Out: (0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0,
0, 0, 1, 1, 0, 0, 0, 0, 1)
```

Para finalizar, damos el valor *True* a ambos parámetros booleanos para realizar tanto la descodificación como su correspondiente transformación a caracteres ASCII:

```
In: codigo_goppa.decodificar(cifrado,True, True)
```

```
Out: 'Hola'
```

Capítulo 5

Criptosistemas de McEliece

5.1. Introducción

El criptosistema McEliece es un criptosistema de clave pública el cual fue desarrollado por Robert McEliece en 1978. Aunque dicho criptosistema se olvido durante años al tener poca aceptación debido principalmente a los tamaños de las claves que necesita, actualmente ha ganado importancia al ser uno de los criptosistemas resistente a ataques de ordenadores cuánticos.

El criptosistema se basa en la teoría de codificación y su seguridad se encuentra principalmente en que la matriz generatriz del código parece aleatoria y en la dificultad de descodificación de un código lineal del que no conocemos su estructura, al tratarse este de un problema computacionalmente difícil (NP-Hard) cuando el tamaño del código es grande (si es pequeño, encontrar la palabra del código con distancia menor o igual que la enviada no supondría ningún problema).

Para algunos tipos de códigos existen algoritmos de descodificación, los cuales tienen complejidad polinómica y por ello, nos permiten eliminar el error de una palabra del código de forma eficiente. Por ejemplo, los códigos de Reed-Solomon y los códigos de Goppa que hemos visto en la sección 3.3 y capítulo 4.

Para el criptosistema de McEliece emplearemos los códigos de Goppa clásicos binarios, debido a que son seguros, sencillos de construir y emplear y presentan mejores propiedades que los códigos de Goppa construidos sobre otros cuerpos finitos según lo visto en la proposición 4.6.

También emplearemos los códigos Reed-Solomon y Reed-Solomon generalizados para la construcción del criptosistema de McEliece aunque, como veremos mas adelante, estos se pueden criptoanalizar de forma exitosa.

Por último, también se podrá realizar la construcción del criptosistema de McEliece con cualquiera de los otros códigos lineales que hemos visto en el capítulo 3 y en adicción con los códigos de producto de matrices que veremos en el capítulo 7.

5.2. Descripción del criptosistema

Para realizar la construcción del criptosistema de McEliece primero tenemos que escoger uno de los códigos indicados anteriormente, el cual debe poseer un algoritmo de descodificación eficiente para la capacidad correctora del código, t. Para dicho código, obtenemos una matriz generatriz, G, de tamaño $k \times n$.

A partir de la matriz generatriz G, calculamos la matriz G', que tiene el mismo tamaño que G y se obtiene mediante el producto de tres matrices: $G' = SGP$ donde S es una matriz no singular cuadrada de tamaño k, es decir, una matriz que es invertible y P es una matriz cuadrada de tamaño n permutacional, es decir, una matriz con todos sus elementos nulos salvo un elemento por cada columna y fila que vale 1.

En *Sage* hemos implementado una clase llamada *McEliece* que nos permite realizar la construcción de criptosistemas de McEliece rápidamente tras indicar una serie de datos. Dichos datos se deben introducir por tanto como parámetros para el constructor de la clase:

- En *code* debemos indicar el código sobre el que queremos construir el criptosistema de McEliece.
- En t debemos introducir un numero entero no negativo que no sobrepase la capacidad correctora del código que vamos a emplear. Dicho parámetro no se fija automáticamente con el valor de la capacidad correctora del código para que podamos indicar, si lo deseamos, un valor menor y de esta forma que se realice el proceso de cifrado que veremos en la sección 5.3.1 introduciendo una cantidad de errores menor.
- *reduction_MPC* es un parámetro opcional que por defecto tiene el valor asignado *None*. En caso de querer emplearlo, se debe introducir una lista formada por una matriz invertible, S, y una matriz de permutación, P. Mediante su empleo, se fijan por tanto las matrices P y S en lugar de seleccionarlas aleatoriamente, lo cual nos permitirá construir el criptosistema de McEliece aplicando la nueva versión que veremos en la sección 7.4.

Una vez introducidos los parámetros necesarios en el constructor de la clase, dicha función calcula las matrices de permutación e invertible si el parámetro

reduction_MPC no ha sido introducido y también calcula y almacena una serie de variables que necesitamos emplear a lo largo del resto de funciones creadas en la clase, como por ejemplo la matriz pública G':

```
In: S=random_matrix(base,k);
    while rank(S)<k:
        S=random_matrix(base,k);
    columnas= range(n);
    P= matrix(base,n);
    for i in range(n):
        posicion=randint(0,len(columnas)-1);
        P[i,columnas.pop(posicion)]=1;
```

En caso de que el parámetro *reduction_MPC* se haya introducido, tras realizar las validaciones pertinentes se asignan las matrices de permutación e invertibles para su posterior almacenamiento en variables de la clase mediante las sentencias $P = reduction\_MPC[1]$ y $S = reduction\_MPC[0]$ respectivamente. Después, se realiza el mismo cálculo del resto de variables que emplearemos en otras funciones.

Analizando la multiplicación de matrices realizada, podemos ver que la matriz SG es también una matriz generatriz del código que teníamos, y al multiplicarla por la matriz P tenemos que la matriz G' es la matriz generatriz de un código equivalente al que tenemos y por ello, ambas matrices tienen la misma capacidad correctora. Dicha matriz G' debe caracterizarse por ser una matriz que no revele la estructura del código del que partimos, es decir, debe ser una matriz que oculte el código inicial.

Entonces las claves pública, κ_p, y privada, κ_s, del criptosistema de McEliece son respectivamente:

$$\kappa_p = (G', t) \text{ y } \kappa_s = (G, S, P, \vartheta)$$

donde ϑ denota el algoritmo descodificador de hasta t errores del código de partida.

En cuanto al proceso de comunicación entre dos personas mediante el empleo del criptosistema de McEliece, procedemos a explicar tanto el cifrado de un mensaje como su correspondiente descifrado:

5.3. Cifrado

5.3.1. Proceso de cifrado

Para realizar el cifrado de un mensaje $m = (m_1, ..., m_k)$, de tamaño k, dirigido a un usuario con clave pública $\kappa_p = (G', t)$, tenemos que multiplicar el mensaje m por la matriz G' y obtenemos el vector c de tamaño n: $c = mG' = (c_1, ..., c_n)$.

Ahora añadimos una cantidad de errores, de tamaño menor o igual a t, al vector c, es decir, tomamos $e = (e_1, ..., e_n)$ con $\omega(e) \leq t$ y realizamos la suma de ambos vectores, $c' = c + e$. Normalmente se escoge un vector de errores cuyo peso sea el máximo posible.

5.3.2. Implementación en Sage

En *Sage* hemos implementado una función llamada $introducir\_errores(self, mensajecodificado, cantidad)$ en la clase *McEliece*, cuyo funcionamiento es muy semejante a la función $introducir\_errores()$ de la clase *Goppa* que vimos en la sección 4.3. En este caso, simplemente diferenciamos entre si los códigos que se están empleando son Goppa, en cuyo caso se retorna el resultado proporcionado por la función $introducir\_errores()$ de la clase *Goppa*, si son códigos de producto de matrices donde también se devuelve el valor obtenido de una función llamada $introducir\_errores()$ creada en la clase *MatrixProductCodes* que veremos en la sección 7.3, y por último si es cualquier otro código lineal, donde se emplea el mismo procedimiento para introducir la cantidad de errores deseada que realizábamos en la clase *Goppa*, con la diferencia que ahora el cuerpo no necesariamente es $\mathbb{F}_{2^m}$ y por tanto para incorporar el error tenemos que buscar un elemento aleatorio, mediante la función $random\_element()$, que sea distinto del elemento nulo.

También hemos construido una función llamada $cifrado(self, mensaje, fichero = True)$, que nos permite realizar el proceso de cifrado que acabamos de ver en la sección 5.3.1. Dicha función admite en el parámetro *mensaje* tanto un vector de tamaño k sobre el cuerpo en que hemos construido el criptosistema como una cadena o lista de caracteres ASCII.

En adicción, es posible leer y cifrar completamente un fichero de texto con formato "*.txt*", para el cual guardamos todos sus caracteres en una variable de tipo *str*. Después sobre dicha variable realizamos su correspondiente cifrado, empleando el mismo proceso que el que mostramos a continuación para una cadena o lista de caracteres ASCII. Para ello, tenemos que dar el valor *True* al parámetro booleano "$fichero$" e indicar en el parámetro "*mensaje*" el nombre del fichero de texto que queremos cifrar.

Por tanto, si el parámetro *mensaje* es un vector de longitud k, simplemente realizamos el cifrado que hemos explicado en la sección 5.3.1:

```
In: vectorCC=mensaje;
    bb=vector(self.matriz_publica.base_ring(),
         [a for a in vectorCC[i*self.k:(i+1)*self.k]*self.matriz_publica]);
    bb=vector(self.matriz_publica.base_ring(),
              [a for a in self.introducir_errores(bb,self.t)]);
```

```
    lista+=[bb[i] for i in range(len(bb))]
    mensajecifrado=vector(self.K, [lista[i] for i in range(len(lista))]);
    return mensajecifrado;
```

donde el iterador i siempre vale 0 en este caso al tener la longitud del mensaje fija en k.

Por otro lado, si introducimos una cadena de caracteres ASCII o leemos un fichero de texto, el proceso se complica notablemente ya que, en primer lugar, tenemos que transformar cada carácter ASCII en su correspondiente bloque binario de longitud 8, para lo cual empleamos la función $ascii\_to\_bin(m)$:

```
In: mensajebin=ascii_to_bin(mensaje);
    vectorbin=vector(GF(2), [GF(2)(str(list(mensajebin)[i])) for i
                                in range(len(list(mensajebin)))]);
```

Ahora, si el cuerpo finito sobre el que estamos trabajando, $\mathbb{F}_{p^m}$, tiene característica distinta a 2, es decir, es un cuerpo no binario, por lo que $p \neq 2$, tenemos que calcular la longitud necesaria, que almacenamos en la variable mm, de un vector formado por elementos del cuerpo $\mathbb{F}_p$ para poder expresar los 256 posibles caracteres ASCII unívocamente:

```
In: mm=1
    while True:
        if  self.K.characteristic()**(mm-1)< ZZ(256) and
                            self.K.characteristic()**(mm)>= ZZ(256):
            break
        else:
            mm+=1
```

Obviamente, si el cuerpo no es binario, la longitud del vector que necesitamos es menor que 8, es decir, $mm < 8$ y en este caso es necesario transformar cada bloque binario de longitud 8 en un bloque de longitud mm formado por elementos de $\mathbb{F}_p$, denotado $base\_K$. Para ello, transformamos el bloque binario a su correspondiente identificación numérica mediante el empleo de la función $ascii\_integer()$ y después transformamos dicho número al vector que le corresponde, almacenado en la variable $base\_K$, mediante la obtención de los restos de dividir por p:

```
In: dividir=self.K.characteristic();
    bb=vector(GF(2),vectorbin[i*8:(i+1)*8])
    numero_bb= ascii_integer(bb)
    q=ZZ(numero_bb)
```

```
    base_K=[]
    while q>=dividir:
        (q,r) = q.quo_rem(dividir);
        base_K+=[r]
    if q<dividir:
        (q,r) = q.quo_rem(dividir);
        base_K+=[r]
    if(len(base_K)<mm):
        incrementar_ceros=mm-len(base_K)
        for j in range(incrementar_ceros):
            base_K+=[0]
        base_K.reverse()
    else:
        base_K.reverse()
```

Ahora, en los casos en que el cuerpo empleado para la construcción del criptosistema, $\mathbb{F}_{p^m}$, no tengamos que $m = 1$, es decir, que el cuerpo no se construye directamente a partir de un número primo, luego es una extensión, es necesario transformar el vector que tenemos actualmente sobre el cuerpo $\mathbb{F}_p$ a un vector sobre el cuerpo $\mathbb{F}_{p^m}$ para poder realizar el cifrado, ya que la matriz pública G' esta formada por elementos del cuerpo $\mathbb{F}_{p^m}$. Por tanto, es necesario ir realizando la transformación de un vector de longitud m sobre el cuerpo $\mathbb{F}_p$ a su identificación de un elemento de $\mathbb{F}_{p^m}$. Si ocurre que la longitud del vector sobre $\mathbb{F}_p$ no es un múltiplo de m, añadimos la cantidad de elementos nulos necesarios para que lo sea, los cuales podremos identificar en el descifrado fácilmente.

```
In: if iterar1.is_integer():
        for i in range(iterar1):
            bb=vector(self.K.base_ring(),[a for a in
                                vector_base_K[i*self.pp:(i+1)*self.pp]])
            listacuerpoascen+=[self.K(bb)]
    else:
        for i in range(iterar1.ceil()):
            if i<>(iterar1.ceil()-1):
                bb=vector(self.K.base_ring(),[a for a in
                                vector_base_K[i*self.pp:(i+1)*self.pp]])
                listacuerpoascen+=[self.K(bb)]
            else:
                tamrest=len(vector_base_K)-i*self.pp
                numeroceros= self.pp-tamrest;
                listrest=list(vector_base_K[len(vector_base_K)
```

```
                                        -tamrest:len(vector_base_K)]);
                listrest+=[0 for i in range(numeroceros)];
                listrest=vector(self.K.base_ring(),listrest)
                listacuerpoascen+=[self.K(listrest)]
    vectorCC=vector(self.K,listacuerpoascen)
```

Una vez realizado todo este proceso de conversión, para realizar ahora el cifrado simplemente tenemos que aplicar el proceso que hemos explicado en la sección 5.3.1, del cual ya hemos mostrado su implementación al comienzo de esta sección. Para ello, tenemos que ir dividiendo nuestro vector total en vectores de longitud k. De igual forma, si la longitud del vector total no es múltiplo de k, añadimos la cantidad de elementos nulos necesarios para que lo sea.

Si la cadena de caracteres ASCII se ha leído desde un fichero de texto, el correspondiente cifrado de dicha cadena de caracteres se almacena en un nuevo fichero creado, cuyo nombre se obtiene de la concatenación del nombre del fichero de texto que ciframos y la cadena de caracteres $" - cif"$.

5.4. Descifrado

5.4.1. Proceso de descifrado

Para realizar el descifrado del mensaje $c' = (c'_1, ..., c'_n)$ recibido, podemos observar que como tenemos $c' = c + e = mG' + e$, entonces si multiplicamos dicho vector por la matriz inversa de la matriz de permutación, P^{-1} obtenemos: $c^* = c'P^{-1} = mSG + eP^{-1}$.

Siempre es posible realizar el paso anterior ya que toda matriz de permutación es invertible.

Ahora podemos emplear el algoritmo eficiente de descodificación de t errores, ϑ, para obtener mSG. El algoritmo se puede emplear ya que la matriz SG es también una matriz generatriz del código de Goppa y tanto el vector e como el vector eP^{-1} tienen el mismo peso.

Después, obtenemos mS resolviendo el sistema de ecuaciones correspondiente a realizar la descodificación de los códigos de Goppa, ya que mS se corresponde con otro posible mensaje.

Por último, obtenemos el bloque descifrado m a partir de la inversa de la matriz S: $m = mSS^{-1}$.

5.4.2. Implementación en Sage

Hemos construido una función llamada $descifrado(self, mensaje, ASCII = True, fichero = False)$ también dentro de la clase *McEliece* para poder realizar el descifrado de las formas implementadas en la sección 5.3.2. Para ello, en dicha función, si el parámetro opcional $fichero$ es *False*, cuyo valor viene dado por defecto, debemos introducir en el parámetro *mensaje* el vector cifrado que queremos descifrar. En el parámetro booleano $ASCII$, el cual también es opcional, de forma que si no se indica ningún valor para este se asignará automáticamente el valor *True*, debemos indicar si deseamos transformar el mensaje descifrado a la cadena de caracteres correspondiente.

Por tanto, si queremos descifrar un mensaje y no queremos convertirlo a caracteres ASCII, debemos indicar en la variable $ASCII$ el valor *False*. En caso contrario, debemos indicar el valor *True* o no indicar ningún valor.

De igual forma, si queremos leer un fichero de texto que contiene el mensaje cifrado que queremos descifrar, tenemos que indicar en el parámetro $fichero$ el valor *True* y en el parámetro *mensaje* el nombre del fichero. Entonces se guarda el vector almacenado en el fichero y se realiza su descifrado, donde dependiendo del parámetro $ASCII$ se recupera una cadena de caracteres o el vector descifrado, que se almacena en otro fichero nuevo cuyo nombre se obtiene de concatenar del nombre del fichero de texto que desciframos y la cadena de caracteres $" - des"$.

Dicha función, divide el vector cifrado en bloques de tamaño n, donde en este caso si debe ocurrir que el tamaño del vector total sea un múltiplo de n, ya que se supone que esta formado por un conjunto de vectores cifrados de longitud n. Después, tenemos que realizar el proceso explicado en la sección 5.4.1, para lo cual, distinguimos entre si el código empleado por el criptosistema de McEliece es de Goppa, en cuyo caso tenemos que emplear el algoritmo de descodificación de Patterson que ya implementamos en la sección 4.3.3, si es un código de producto de matrices, donde empleamos un algoritmo de descodificación que explicaremos e implementaremos en la sección 7.3 y si es cualquier otro código lineal que hemos visto en el capítulo 3, donde emplearemos como algoritmo de descodificación la función $decode\_to\_message()$ ya implementada en *Sage*. Hay que destacar como vimos en la sección 3.2, que si en la descodificación realizada por dicha función tenemos que obtener un vector cuyas últimas coordenadas se corresponden con elementos nulos, la función nos devuelve el vector obtenido a partir de la eliminación de dichas últimas coordenadas nulas. Por tanto, en estos casos es necesario la introducción de nuevo de dichas coordenadas nulas para obtener un descifrado total del mensaje correcto:

```
In: for i in range(iterar):
        trozomensaje= vector( self.K ,[a for a in
```

```
                    mensaje[i*filas:(i+1)*filas]*self.P.inverse()]);
        if not self.is_Goppa:
            if trozomensaje==0:
                trozomensaje1=vector( self.K ,[0 for i in
                                                range(self.S.nrows())]);
            else:
                if self.is_MPC:
                    trozomensaje1=vector( self.K ,[a for a in
                        self.codigo.decodificar_MPC(trozomensaje,True)]);
                else:
                    trozomensaje1=vector( self.K ,[a for a in
                             self.codigo.decode_to_message(trozomensaje)]);
            if len(trozomensaje1)<>self.S.nrows():
                listatrozo=list(trozomensaje1);
                cantidad_ceros=self.S.nrows()-len(listatrozo);
                for k in range(cantidad_ceros):
                    listatrozo+=[0]
                trozomensaje1=vector( self.K ,listatrozo);
            trozomensaje = vector( self.K ,[a for a in trozomensaje1
                                             *self.S.inverse()]);
        lista += [a for a in trozomensaje];
        if self.is_Goppa:
        vec=vector(self.K.base_ring(),lista);
        vec=self.codigo.decodificar(vec,False,True);
        men=[];
        filas=self.S.nrows();
        iterar1=RR(len(vec))/RR(filas);
        for i in range(iterar1.floor()):
            trozovec = vec[i*filas : (i+1)*filas]* self.S.inverse();
            men+=[a for a in trozovec];
        men=vector(self.K,men)
        else:
        vec=[]
        men=vector(self.K,lista);
```

Una vez realizado el proceso anterior, ya hemos descifrado el mensaje, por tanto, si el parámetro booleano *ASCII* es *False* devolvemos el resultado obtenido. Sin embargo, si el parámetro *ASCII* tiene como valor *True* es necesario realizar el proceso contrario al implementado en la sección 5.3.2, es decir, convertir el mensaje descifrado, que se puede encontrar en el caso mas complejo en el cuerpo $\mathbb{F}_{p^m}$, al vector correspondiente sobre el cuerpo $\mathbb{F}_2$, para poder realizar su posterior conversión a los caracteres ASCII

correspondientes.

Para realizar este proceso, en primer lugar tenemos que comprobar si el código empleado por el criptosistema no es de Goppa y que no se esta empleando un cuerpo construido directamente sobre un número primo, es decir, si estamos empleando un código construido sobre $\mathbb{F}_{p^m}$. En caso afirmativo, tenemos que convertir cada elemento $\mathbb{F}_{p^m}$ del vector descifrado en un vector de tamaño m realizando un descenso del cuerpo. Para ello, recurrimos de nuevo a la clase implementada por *Sage* llamada *embedding*() dentro de un tipo de códigos implementados denotados como *codes.SubfieldSubcode*(). Dicha clase nos permite obtener la conversión que buscamos de un elemento de $\mathbb{F}_{p^m}$ al correspondiente vector formado por elementos de $\mathbb{F}_p$ empleando la función *relative_field_representation*():

```
In: if self.pp<>1 and not self.is_Goppa:
        C=codes.random_linear_code(self.codigo.base_field()
                ,self.matriz_publica.ncols(),self.matriz_publica.ncols()-1)
        Csubfield=codes.SubfieldSubcode(C, C.base_field().base_ring())
        E=Csubfield.embedding()
        menSF=[]
        for i in range(len(men)):
            elemento=E.relative_field_representation(men[i])
            menSF+=[a for a in elemento]
        menSF=vector(self.codigo.base_field().base_ring(),menSF)
    else:
        menSF=men
```

Ahora ya tenemos el vector descifrado sobre el cuerpo $\mathbb{F}_p$, por tanto, tenemos que realizar su conversión a un vector sobre el cuerpo $\mathbb{F}_2$, si aun no se encuentra en este. Para ello, dividimos el vector en bloques cuya longitud vale mm, donde mm era la variable que almacenaba la longitud necesaria para guardar un caracter ASCII mediante elementos de $\mathbb{F}_p$. Después, realizamos la conversión de cada bloque de longitud mm a su valor numérico y seguidamente transformamos dicho número a su representación binaria mediante el empleo de un bloque de longitud 8:

```
In: if mm<>8:
        menbin=[]
        iterar4=RR(len(menSF))/RR(mm);
        continuar1=True;
        i=0;
        while i<iterar4.ceil() and continuar1:
            entra=False;
            if i<>(iterar4.ceil()-1):
```

```
                entra=True;
            else:
                if iterar4.is_integer():
                    entra=True
            if entra:
                trozovector=menSF[i*mm : (i+1)*mm];
                if vector(GF(self.K.characteristic())
                                        ,[a for a in trozovector])==0:
                    continuar1=False;
                else:
                    numero=ZZ(0);
                    for k in range(mm):
                        numero+= ZZ(trozovector[k])
                                    *self.K.characteristic()**(mm-k-1)
                    q=ZZ(numero)
                    base_bin=[]
                    while q>=2:
                        (q,r) = q.quo_rem(2);
                        base_bin+=[r]
                    if q<2:
                        (q,r) = q.quo_rem(2);
                        base_bin+=[r]
                    if(len(base_bin)<8):
                        incrementar_ceros=8-len(base_bin)
                        for j in range(incrementar_ceros):
                            base_bin+=[0]
                        base_bin.reverse()
                    else:
                        base_bin.reverse()
                    menbin+=base_bin
            i+=1;
    else:
        menbin=menSF
```

Para finalizar, tenemos que convertir el vector que se encuentra ya sobre el cuerpo $\mathbb{F}_2$ a los caracteres ASCII correspondientes, para ello, tras dividir el vector total obtenido en bloques de longitud 8, empleamos la función *bin_to_ascii*() para convertir cada bloque en el carácter correspondiente y después ya devolvemos el mensaje completo:

```
In: h=RR(len(menbin))/RR(8);
    i=0;
```

```
continuar=True;
ascii='';
while i<h.ceil() and continuar:
    if i<>(h.ceil()-1):
        caracterbin=menbin[i*8:(i+1)*8];
        if vector(GF(2),[a for a in caracterbin])==0:
            continuar=False;
        else:
            ascii+=bin_to_ascii(caracterbin);
    else:
        if h.is_integer():
            caracterbin=menbin[i*8:(i+1)*8];
            ascii+=bin_to_ascii(caracterbin);
    i+=1;
return ascii;
```

Es importante remarcar que el lenguaje de programación de *Sage* utiliza un formato de codificación interno a partir del cual no es capaz de reconocer caracteres especiales ASCII. Por tanto, si ciframos una cadena de caracteres ASCII que contenga algún carácter especial, como por ejemplo la letra "ñ", tras realizar su descifrado no obtendremos estos caracteres especiales.

Sin embargo, al tratarse de un problema interno de *Sage*, cuando realizamos el cifrado de un fichero de texto que puede contener distintos caracteres especiales y después su correspondiente descifrado, como almacenamos dicho mensaje descifrado en otro fichero de texto, al abrirlo de forma externa a *Sage* si que se reconocerán los distintos caracteres especiales.

5.5. Otras funciones

Además de las funciones de cifrado, descifrado e introducir errores que hemos explicado hasta el momento, también hemos implementado un conjunto de funciones que nos permiten obtener propiedades fundamentales del criptosistema de McEliece creado:

- *code_mceliece*() es una función que nos proporciona el código empleado para la construcción del criptosistema.
- *matriz_invertible*() es una función que nos devuelve la matriz aleatoria invertible generada para la construcción del criptosistema.

- *matriz_permutacion*() es una función que nos proporciona la matriz aleatoria de permutación generada para la construcción del criptosistema.
- *public_matrix*() es una función a partir de la cual obtenemos la matriz pública del criptosistema, G'.
- *capacidad_correctora*() es una función que nos devuelve el valor t introducido, el cual nos indica el número de errores que se introducen durante el proceso de cifrado.
- *dimension*() es una función que nos proporciona la dimensión del código empleado en el criptosistema.
- *base_field*() es una función a partir de la cual obtenemos el cuerpo sobre el que trabaja tanto el código como el criptosistema.

5.6. Ventajas y desventajas del criptosistema

El criptosistema de McEliece presenta como una de las principales ventajas, un rápido proceso de codificación y descodificación, los cuales son mas rápidos que los procesos que realizan actualmente muchos de los sistemas mas empleados, como es el caso del sistema RSA.

Otra de sus ventajas es que, como ya hemos comentado, es un criptosistema postcuántico, es decir, es resistente a ataques realizados mediante ordenadores cuánticos que emplean el algoritmo de Shor.

El algoritmo de Shor, es un algoritmo cuántico que fue desarrollado en 1994 y permite factorizar un número entero en factores primos en tiempo polinomial. Por ello, dicho algoritmo nos permitiría romper el criptosistema RSA y el del logaritmo discreto entre otros, los cuales, son los mas empleados actualmente a la hora de realizar cualquier cifrado asimétrico.

A pesar de esto, aunque el criptosistema presenta pocas desventajas, la mas importante es que el tamaño de las claves, tanto pública como privada es muy grande, al ser matrices de elevadas dimensiones.

Por ejemplo, los tamaños de parámetros sugeridos por McEliece fueron $n = 1024$, $k = 524$, $t = 50$, los cuales dan lugar a una clave pública de tamaño cercano a 2^{19} bits.

Sin embargo, debido al avance tecnológico producido en estos años, el tamaño de los parámetros sugerido ha sido elevado a $n = 2048$, $k = 1750$, $t = 27$, de esta forma

podemos obtener 80 bits de seguridad, es decir, es necesario realizar 2^{80} operaciones para "*romper*" el criptosistema. Esto permite evitar ataques realizados por fuerza bruta con los ordenadores actuales. Sin embargo, para resistir ataques realizados mediante ordenadores cuánticos, el tamaño de estos parámetros se debe elevar hasta $n = 6960$, $k = 5400$, $t = 119$, los cuales nos proporcionan un tamaño de clave pública cercano a 2^{23} bits.

Incrementar el tamaño de los parámetros esta relacionado con el incremento de los tamaños de las claves pública y privada empleadas para el criptosistema. Entonces, como estos parámetros se han incrementado notablemente en relación a los sugeridos en la versión original, la cual ya presentaba un tamaño de claves alto, tenemos como resultado un tamaño de claves muy elevado.

Otro inconveniente que presenta este criptosistema es que no se puede emplear para producir firmas digitales, aunque este problema ha sido solventado empleando el criptosistema Niederreiter, el cual es un criptosistema dual al criptosistema de McEliece que mostramos en [1, Sección 5.2] del trabajo fin de grado de matemáticas, que permite la realización de firmas digitales [7].

5.7. Posibles ataques

Consideramos un criptosistema de McEliece construido a partir del código C de parámetros $[n, k, d]$ y matrices S y P. Vamos a mostrar muy brevemente los dos posibles ataques que se pueden realizar contra el criptosistema de McEliece, al ser un criptosistema que emplea códigos para su construcción:

- *Ataques genéricos de descodificación*: Se trata de intentar recuperar el mensaje original, m, que hemos cifrado obteniendo el vector y, mediante el empleo del código C' que se obtiene al considerar la matriz pública del criptosistema, G', como matriz generatriz para dicho código.

  Para ello, los mejores resultados se obtienen empleando algoritmos que mejoran la descodificación del conjunto de información. Esta descodificación trata de encontrar k coordenadas del vector y que no presenten ningún error. Entonces, tomando dichas coordenadas del vector y y la matriz inversa formada por las k columnas seleccionadas de la matriz G' podemos recuperar el mensaje original m.

  Uno de los algoritmos de descodificación del conjunto de información mas conocido es el de *Lee-Brickell* [36].

- *Ataques contra la estructura del código*: En este caso se trata de intentar recuperar, las matrices G, S y P a partir de las cuales se ha realizado la construcción del

criptosistema de McEliece empleado. Para este tipo de ataques los criptosistemas de McEliece que emplean códigos de Goppa son los que presentan una elevada seguridad.

Por otro lado, el ataque por filtración que mostramos en el capítulo 6, es un ejemplo de ataque contra la estructura del código cuando se emplean los códigos Reed-Solomon generalizados, ya que permite recuperar en este caso los parámetros del código C'. También existen otros tipos de ataques contra la estructura del código para otros códigos, como Reed-Muller, BCH, cuasi-cíclicos entre otros.

Para finalizar el capítulo, vamos a realizar la construcción de un criptosistema de McEliece que emplea un código de Goppa para su construcción.

Ejemplo 5.1. Para la construcción del criptosistema, realizamos en primer lugar la construcción del código de Goppa empleando unos parámetros muy pequeños en comparación con los sugeridos, para poder mostrar de esta forma las matrices que obtenemos.

Por tanto, generamos el código de Goppa sobre el cuerpo $\mathbb{F}_{2^5}$, con máxima longitud $n = 32$ y capaz de corregir $t = 5$ errores:

```
In: goppa=Goppa(2**5,5,5)
```

El polinomio de Goppa empleado para su construcción es:

```
In: goppa.polinomio_g()

Out: x^5 + (a^3 + a^2 + 1)*x^2 + (a^2 + a + 1)*x + a^4 + a^2
```

Ahora obtenemos la matriz generatriz del código de Goppa obtenido:

```
In: goppa.generator_matrix()

Out: [1 0 0 0 0 0 0 1 1 0 1 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 0 0 1 1 0 1]
     [0 1 0 0 0 0 0 0 1 0 0 1 1 0 0 1 0 1 0 1 0 0 0 1 0 0 1 1 0 1 1 1]
     [0 0 1 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 1 0 0 1 0 1 1 0 1 0 0 1 0 1]
     [0 0 0 1 0 0 0 1 1 1 1 1 1 1 0 0 0 1 0 0 1 1 1 0 1 0 1 1 0 1 0 0]
     [0 0 0 0 1 0 0 1 0 0 1 1 0 1 1 1 1 0 1 0 1 0 1 0 0 0 0 1 1 1 0 1]
     [0 0 0 0 0 1 0 1 0 1 1 1 1 1 1 1 0 0 1 1 1 1 0 1 1 1 0 1 1 0 1 1]
     [0 0 0 0 0 0 1 0 0 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 1 1 1 0 0 1 0]
```

Mediante este código de Goppa, construimos el criptosistema de McEliece, donde indicamos que se introduzcan $t = 5$ errores durante el proceso de cifrado, que se corresponden con la capacidad correctora del código de Goppa:

```
In: codigo_mceliece=McEliece(goppa,5)
```

Mostramos ahora la matriz pública, G', junto con la matriz invertible, S, y la matriz de permutación, P, del criptosistema de McEliece construido:

```
In: G_pub=codigo_mceliece.public_matrix()
    G_pub

Out: [0 0 1 0 1 0 1 1 0 0 0 1 0 0 0 0 0 0 1 1 0 1 0 0 0 0 1 1 0 0 0 1]
     [0 1 0 0 1 1 0 1 0 0 0 1 0 0 0 1 1 0 1 1 1 0 1 0 0 0 0 1 1 0 1 1]
     [1 0 0 1 0 1 1 0 0 0 0 1 0 0 0 0 0 1 1 0 1 1 1 0 1 1 1 0 1 0 0 0]
     [1 0 1 0 0 1 0 0 1 1 1 1 1 0 0 1 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0]
     [1 0 1 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 1 1 0 0 0 1 1 0 0 1 1 1]
     [0 0 0 0 1 0 1 0 1 0 1 1 1 1 1 0 1 1 1 0 0 0 1 0 1 0 1 1 0 0 1 0]
     [0 0 0 1 0 0 0 0 0 1 1 0 1 1 0 1 0 0 0 0 1 1 1 0 1 1 0 1 0 0 0 1 0]
```

```
In: P=codigo_mceliece.matriz_permutacion();
    P

Out: 32 x 32 dense matrix over Finite Field of size 2
```

```
In: S=codigo_mceliece.matriz_invertible();
    S

Out: [0 0 0 0 0 0 1]
     [1 0 0 1 1 1 0]
     [1 0 1 0 1 0 0]
     [0 0 0 0 0 1 1]
     [1 1 0 0 0 0 1]
     [0 0 1 0 1 0 0]
     [1 0 1 0 0 0 0]
```

Podemos observar que no se muestra la matriz de permutación cuadrada de tamaño 32 debido a que ya presenta un elevado tamaño. Para que el lenguaje de programación *Sage* nos devuelva la matriz, independientemente del tamaño que presenta, podemos emplear la función *.str()*.

Por último, realizamos la codificación y descodificación de los dos tipos de mensajes admitidos. En primer lugar, de la palabra *"hola"*:

```
In: m='Hola';

In: cifrado_errores=codigo_mceliece.cifrado(m);
    cifrado_errores
```

```
Out: (1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0,
0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1,
0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1,
0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1,
0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1,
1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1,
0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0)
```

```
In: codigo_mceliece.descifrado(cifrado_errores)
```

```
Out: 'Hola'
```

Para finalizar, realizamos en segundo lugar la codificación y descodificación de un vector aleatorio de longitud $k = 7$:

```
In: vector_aleatorio=vector(GF(2),[GF(2).random_element() for i in range(7)])
    vector_aleatorio
```

```
Out: (0, 1, 0, 1, 0, 1, 0)
```

```
In: cifrado_aleatorio=codigo_mceliece.cifrado(vector_aleatorio);
    cifrado_aleatorio
```

```
Out: (1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0,
1, 1, 1, 1, 0, 0, 0, 0, 1)
```

```
In: codigo_mceliece.descifrado(cifrado_aleatorio,False)
```

```
Out: (0, 1, 0, 1, 0, 1, 0)
```

Podemos ver que al realizar la descodificación del segundo caso, es necesario añadir el parámetro opcional $ASCII = False$ como dijimos en la sección 5.4.2.

Capítulo 6

Ataque contra los códigos GRS

6.1. Introducción

Como vimos en la sección 3.3, los códigos Reed-Solomon y Reed-Solomon generalizados presentan muy buenas propiedades y por tanto son ampliamente usados en teoría de códigos y sus aplicaciones. Debido a esto fueron propuestos en diferentes criptosistemas como el de Niederreiter hasta que fueron criptoanalizados de forma exitosa. El criptoanalisis mas conocido de estos códigos es el llamado ataque de Sidelnikov y Shestakov [8], el cual permite recuperar la estructura de cualquier código Reed-Solomon generalizado en tiempo polinomial, obteniendo para ello el par de vectores a y b que definen el código Reed-Solomon generalizado. Para realizar este ataque es necesario calcular en una primera etapa las palabras del código de peso mínimo.

En su lugar, vamos a explicar otro ataque, llamado ataque por filtración donde no es necesario realizar esa primera etapa. Para dicho ataque, mostramos una forma estándar que nos permite recuperar los vectores a, b de un código GRS_k mediante el conocimiento de las matrices generatrices de GRS_k y de GRS_{k-1}. Por último, mostramos una modificación del ataque estándar, que se explica en [32], a partir del cual si seremos capaces de recuperar los vectores a y b empleando solo la matriz generatriz de GRS_k o pública de un criptosistema de McEliece construido a partir de un código Reed-Solomon generalizado.

Este ataque es menos eficiente que el ataque de Sidelnikov y Shestakov desde el punto de vista computacional ya que mientras la complejidad del ataque por filtración es $O(k^2n^3 + k^3n^2)$, la del ataque de Sidelnikov y Shestakov es $O(k^3 + k^2n)$. Sin embargo, el ataque por filtración, ademas de ser un ataque válido para los códigos Reed-Solomon generalizados, también se puede emplear en otros códigos como los códigos Reed-Muller, o los códigos algebraico-geométricos, en los cuales, la etapa de cálculo de palabras de código de peso mínimo se realiza en tiempo subexponencial

y exponencial respectivamente, y por tanto, el ataque de Sidelnikov y Shestakov no sería factible para criptoanalizar dichos códigos.

Por otro lado, otro objetivo de este capítulo es mostrar que aunque el criptosistema de McEliece parezca ser resistente frente a ataques realizados mediante un ordenador cuántico, para ello es necesario emplear códigos que actualmente sigan siendo seguros, ya que si no, podremos criptoanalizarlos exitosamente sin necesidad de emplear ningún ordenador cuántico.

Antes de mostrar los ataques, vamos a introducir un conjunto de definiciones y propiedades esenciales de la teoría de códigos que necesitamos emplear cuando desarrollemos el ataque por filtración.

Definición 6.1. Sean $x = (x_1, ..., x_n)$ e $y = (y_1, ..., y_n)$ dos elementos de $\mathbb{F}_q^n$. El producto por componentes o producto estrella, denotado por $x \star y$, se define como el producto escalar de las respectivas componentes de ambos vectores:

$$x \star y = (x_1 y_1, ..., x_n y_n)$$

La potencia i-esima del vector x, denotado por $x^{\star i}$, será el resultado de realizar el producto estrella i veces, es decir, $x^{\star i} = x \star x \star \cdots \star x$.

Por último, dado un vector $x = (x_1, ..., x_n)$ que cumple que $x_i \neq 0 \, \forall i = 1, ..., n$, definimos elemento inverso del vector x como, $x^{\star -1} = (x_1^{-1}, ..., x_n^{-1})$

Una vez definido el producto por componentes de dos vectores, resulta bastante intuitiva la definición de producto de códigos que mostramos ahora:

Definición 6.2. Dados C_1 y C_2, dos códigos de igual longitud. Diremos que el código de producto estrella obtenido a partir de los códigos C_1 y C_2, denotado por $C_1 \star C_2$, es el espacio vectorial obtenido a partir de todos los productos por componentes donde el primer vector es un elemento de C_1 y el segundo vector es un elemento de C_2:

$$C_1 \star C_2 = \{x \star y \mid x \in C_1 \, , y \in C_2\}$$

Nota 6.3. También, si las dimensiones de C_1 y C_2 son k_1 y k_2 respectivamente y los vectores $a_1, ..., a_{k_1}$ forman una base del código C_1 y los vectores $b_1, ..., b_{k_2}$ forman una base del código C_2, el código de producto estrella de C_1 y C_2 esta generado por todos los productos estrellas de un vector de una base con otro vector de la otra base:

$$C_1 \star C_2 = \langle a_i \star b_j \mid i = 1, ..., k_1 \, , \, j = 1, ..., k_2 \rangle$$

Definición 6.4. Llamaremos código cuadrado de C_1, denotado por $C_1 \star C_1$ o $C_1^{\star 2}$, al código obtenido a partir del producto estrella del mismo código, es decir, considerando en la definición anterior que $C_2 = C_1$.

Proposición 6.5. *Dados C_1 y C_2, dos códigos de igual longitud, se verifica que:*

1) $dim(C_1 \star C_2) \leq dim(C_1) \cdot dim(C_2)$

2) $dim(C_1^{\star 2}) \leq \binom{dim(C_1)+1}{2}$

Proposición 6.6. *Sean $x, y, y' \in \mathbb{F}_q^n$, se cumple que:*

1) *Si $k + k' - 1 \leq n$ entonces $GRS_k(x, y) \star GRS_{k'}(x, y') = GRS_{k+k'-1}(x, y \star y')$.*

2) *Si $2k - 1 \leq n$ entonces $GRS_k(x, y)^{\star 2} = GRS_{2k-1}(x, y \star y)$.*

Nota 6.7. Si tenemos que la dimensión del GRS verifica que $2k - 1 > n$, la segunda parte de la proposición 6.6 también se cumple ya que, considerando su código dual, sabemos por la proposición 3.23 que la dimensión será $k' = n - k$ y entonces como $k > \frac{n+1}{2}$ se tiene que $-k \leq \frac{-n-1}{2}$ y por tanto $k' \leq \frac{n-1}{2} \leq \frac{n+1}{2}$.

Para la primera parte de la proposición 6.6 también se puede realizar un razonamiento similar teniendo en cuenta tanto la dimensión k como k', pero este caso no lo mostramos debido a que no lo necesitaremos emplear después.

Por tanto, como principal conclusión obtenemos que la dimensión de un código cuadrado de un *GRS* de dimensión k es $2k - 1$.

Ahora que sabemos la dimensión del código cuadrado de un *GRS*, vamos a introducir una importante proposición, la cual nos permite conocer con alta probabilidad la dimensión del cuadrado de un código lineal aleatorio.

Proposición 6.8. *Sea C un código lineal aleatorio de parámetros $[n, k]$ sobre el cuerpo $\mathbb{F}_q$ tal que $n > \binom{k+1}{2}$. Entonces se verifica que*

$$Pr\left(dim(C^{\star 2}) = \binom{k+1}{2}\right) = 1$$

donde $Pr(S)$ indica la probabilidad de que ocurra el suceso S.

Debido a esto, tenemos que la dimensión del código cuadrado de un código aleatorio lineal debe encontrarse en torno al $min\{\binom{k+1}{2}, n\}$. Por tanto, esta técnica nos permite diferenciar un código *GRS* de un código lineal aleatorio, lo cual es muy útil ya que, si por ejemplo, tenemos la clave pública de un criptosistema de McEliece, podremos conocer si dicho criptosistema esta empleando códigos *GRS* y por tanto aplicar el ataque por filtración que veremos mas adelante para criptoanalizar dicho criptosistema de McEliece.

6.2. Ataque por filtración a códigos GRS

6.2.1. Ataque estándar

En un principio, para el ataque estándar vamos a considerar que conocemos los códigos Reed-Solomon generalizados de dimensión k y $k-1$, $GRS_k(a,b)$ y $GRS_{k-1}(a,b)$ respectivamente, donde a y b son elementos de $\mathbb{F}_q^n$ y veamos que si disponemos de estos códigos, es posible obtener el código Reed-Solomon generalizado de dimensión $k-2$ construido a partir de los vectores a y b, lo cual es un paso clave para nuestro ataque.

Proposición 6.9. *Dados los códigos $GRS_k(a,b)$ y $GRS_{k-1}(a,b)$, es posible calcular el código $GRS_{k-2}(a,b)$.*

Nota 6.10. Para la obtención de forma práctica del código $GRS_{k-2}(a,b)$, tenemos que calcular una base del código $(GRS_{k-1}(a,b)^{\star 2})^{\perp}$ y después resolver el sistema construido a partir de las ecuaciones que tienen la forma $g_i \star h_j$, donde g_i hace referencia a la fila i-esima de una matriz generatriz de $GRS_k(a,b)$ y h_j hace referencia a la fila j-esima de una matriz de control, H, de $GRS_{k-1}(a,b)^{\star 2}$. Esto es debido a que se cumple en primer lugar que $(c \star g_i) \in GRS_{k-1}(a,b)^{\star 2}$ si y solo si $(c \star g_i) \cdot H^T = 0$, es decir, $(c \star g_i) \cdot h_j = 0$, y en segundo lugar, se verifica que $(c \star g_i) \cdot h_j = c \cdot (g_i \star h_j)$.

Por tanto, a partir de la proposición 6.9, podemos reiterar el proceso con los dos códigos Reed-Solomon generalizados que tengamos de dimensión mas baja hasta obtener el código $GRS_1(a,b)$. Como $GRS_1(a,b) = \{\lambda b \mid \lambda \in \mathbb{F}_q\} = \langle b \rangle$ debido a que sus elementos se obtienen evaluando en constantes y multiplicándolas por el elemento b, a partir de este proceso, conocido como proceso de filtración, podemos obtener el elemento b.

Para ello, hemos implementado en *Sage* una función llamada $reducir\_codigos(C, C1, reC = False)$, la cual va obteniendo los códigos Reed-Solomon generalizados con dimensiones menores a la del código $C1$ aplicando la proposición 6.9, donde la implementación se ha realizado según la forma expuesta en la nota 6.10, es decir, primero obtenemos el código cuadrado del código $C1$ empleando el siguiente código:

```
In: baseC1cuadrado=[];
    n=C1.generator_matrix().row(0).length();
    for i in range(C1.generator_matrix().nrows()):
        for j in range(i,C1.generator_matrix().nrows()):
            baseC1cuadrado+=[vector(F,[C1.generator_matrix().row(i)[h]*
            C1.generator_matrix().row(j)[h] for h in range(n)]) ];
    V= VectorSpace(F,n)
```

```
    subV=V.subspace(baseC1cuadrado)
    Gcuadrada=subV.basis_matrix()
    C1cuadrado= codes.LinearCode(Gcuadrada);
    if C1cuadrado.dimension() <> (2*C1.dimension()-1):
        raise TypeError('Deben emplearse códigos GRS para realizar el
                                                                ataque')
```

Podemos observar que una vez obtenemos el código cuadrado de $C1$, también validamos que su dimensión se corresponda con los visto en la proposición 6.6. Después resolvemos el correspondiente sistema de ecuaciones a partir de las siguientes líneas de código:

```
In: n=len(C1.random_element());
    lista=[]
    for i in range(C.generator_matrix().nrows()):
       for j in range(C1cuadrado.parity_check_matrix().nrows()):
            lista+=[list(vector(F,[F(C.generator_matrix().row(i)[h]*
                        C1cuadrado.parity_check_matrix().row(j)[h]) for
                                                    h in range(n)]))]
    AAA=matrix(F,lista)
    matrizG_codigo=AAA.right_kernel().basis_matrix()
    C2=codes.LinearCode(matrizG_codigo)
```

Todo el proceso mostrado anteriormente se repite con los dos códigos de menor dimensión que tengamos hasta obtener un código de dimensión 1, cuya matriz generatriz se corresponde con el vector λb.

El parámetro opcional *recG* se emplea únicamente para saber si la función debe devolver solo el vector λb o también la parte de la matriz generatriz, del código de dimensión 2 calculado, en forma estándar que no se corresponde con la matriz identidad. Su utilidad la veremos mas adelante.

Si ahora obtenemos el elemento a, tendríamos determinado por completo el código Reed-Solomon generalizado y por tanto un ataque exitoso contra dicho código. Para la obtención del elemento a, vamos a emplear el código Reed-Solomon de dimensión 2, que ya tenemos calculado por la filtración realizada.

A partir de este código, calculamos una matriz generatriz para él, donde n hace referencia al tamaño de los elementos codificados, que es conocido:

$$G = \begin{pmatrix} g_{11} & g_{12} & \cdots & g_{1n} \\ g_{21} & g_{22} & \cdots & g_{2n} \end{pmatrix}$$

Como sabemos por la sección 3.25 otra matriz generatriz de $GRS_2(a,b)$ es

$$G' = \begin{pmatrix} b_1 & b_2 & \cdots & b_n \\ a_1b_1 & a_2b_2 & \cdots & a_nb_n \end{pmatrix}$$

Entonces, como la matriz generatriz en forma estándar es única, ambas matrices generatrices del mismo código deben ser equivalentes a la misma matriz generatriz en forma estándar. Por tanto, realizando eliminación de Gauss-Jordan sobre ambas matrices, tenemos que $rref(G) = rref(G')$, donde $rref(A)$ hace referencia a la forma escalonada reducida de la matriz A.

Conocemos completamente $rref(G)$. Ademas, $rref(G')$ tiene la siguiente forma:

$$G' = \begin{pmatrix} 1 & 0 & \frac{(a_2-a_3)b_3}{(a_2-a_1)b1} & \cdots & \frac{(a_2-a_n)b_n}{(a_2-a_1)b1} \\ 0 & 1 & \frac{(a_3-a_1)b_3}{(a_2-a_1)b_2} & \cdots & \frac{(a_n-a_1)b_n}{(a_2-a_1)b_2} \end{pmatrix}$$

donde el elemento b ya lo tenemos calculado. Por tanto igualando ambas matrices, obtenemos un sistema de $2n-4$ ecuaciones lineales con n incógnitas, donde al menos $n-2$ ecuaciones serán linealmente independientes. Además, como sabemos que en un código Reed-Solomon generalizado siempre se pueden fijar tres puntos del vector a según [27], entonces se puede resolver dicho sistema y por tanto obtener un único vector que se corresponde con el vector a que nos quedaba por calcular.

En *Sage* hemos implementado la función $ataque\_estandar(CCC, CCC1)$, la cual llama en primer lugar a la función $reducir\_codigos(CCC, CCC1, True)$ para obtener el vector λb y la parte de la matriz $rref(G)$ que no se corresponde con la matriz identidad. Después se realiza la resolución del sistema, el cual siempre presenta la misma estructura:

```
In: def ataque_estandar(CCC,CCC1):
        [bb,G]=reducir_codigos(CCC,CCC1,True)
        F=CCC.base_ring()
        AA=matrix(F,G.ncols()*2,bb.ncols())
        for i in range(AA.nrows()):
            for j in range(AA.ncols()):
                if j==0 and i<G.ncols():
                    AA[i,j]=bb[0,0]*G[0,i]
                if j==1 and i<G.ncols():
                    AA[i,j]=bb[0,i+2]-bb[0,0]*G[0,i]
                if j==i+2 and i<G.ncols():
                    AA[i,j]=-bb[0,i+2]
                if j==0 and i>(G.ncols()-1):
```

```
                AA[i,j]=bb[0,1]*G[1,i-G.ncols()]-bb[0,i+2-G.ncols()]
            if j==1 and i>(G.ncols()-1):
                AA[i,j]=-bb[0,1]*G[1,i-G.ncols()]
            if j==i+2-G.ncols() and i>(G.ncols()-1):
                AA[i,j]= bb[0,i+2-G.ncols()]
    A=matrix(F,[[AA[i,j] for j in range(2,AA.ncols())] for i in
                                                    range(G.ncols())])
    z=vector(F,[-AA[i,0]*1-AA[i,1]*2 for i in range(G.ncols())]);
    aa=A.inverse()*z
    a=vector(F,[1,2]+[aaa for aaa in aa])
    b=vector(F,[bb[0,i] for i in range(bb.ncols())])
    return [a,b];
```

Podemos observar que siempre se fijan las dos primeras posiciones del vector a con los mismos valores, $a_1 = 1$ y $a_2 = 2$, y se resuelve un sistema de ecuaciones muy simple que nos permite recuperar el vector a

Ejemplo 6.1. Vamos a emplear los mismos códigos que utilizamos para mostrar el ataque estándar de forma teórica en [1, Ejemplo 6.3] del trabajo fin de grado de matemáticas.

Por tanto, vamos a construir los códigos $GRS_4(a,b)$ y $GRS_3(a,2b)$, con valores $a = (1,2,9,3,10,8,5,4) \in \mathbb{F}_{11}$ y $b = (3,10,5,9,9,10,5,2) \in \mathbb{F}_{11}$. Únicamente suponemos que conocemos una matriz generatriz de cada código, por ello empleamos el siguiente código para realizar la construcción de los códigos y mostrar las matrices que empleamos:

```
In: F=GF(11);
    a=[F(i) for i in [1,2,9,3,10,8,5,4]]; #3,9.....
    b=[F(i) for i in [3,10,5,9,9,10,5,2]];
    b1=[F(i)*2 for i in [3,10,5,9,9,10,5,2]];
    k=5
    C=codes.GeneralizedReedSolomonCode(a, k, b);
    C.generator_matrix()
```

```
Out: [ 3 10  5  9  9 10  5  2]
     [ 3  9  1  5  2  3  3  8]
     [ 3  7  9  4  9  2  4 10]
     [ 3  3  4  1  2  5  9  7]
     [ 3  6  3  3  9  7  1  6]
```

```
In: C1=codes.GeneralizedReedSolomonCode(a, k-1, b1);
    C1.generator_matrix()
```

```
Out: [ 6  9 10  7  7  9 10  4]
     [ 6  7  2 10  4  6  6  5]
     [ 6  3  7  8  7  4  8  9]
     [ 6  6  8  2  4 10  7  3]
```

Ahora empleamos la función *ataque_estandar*() que hemos implementado para obtener los valores de los vectores a y b u otros valores que también nos permiten reconstruir el código de partida:

```
In: parametros=ataque_estandar(C,C1)
    parametros
```

```
Out: [(1, 2, 9, 3, 10, 8, 5, 4), (1, 7, 9, 3, 3, 7, 9, 8)]
```

Por tanto, en este ejemplo podemos observar que recuperamos el mismo vector a, aunque no tiene porque ocurrir. En cuanto al vector b obtenido, podemos observar que es $4b$. Entonces, aunque en este caso se ve claramente, al tener el mismo vector a, que los valores obtenidos van a generar el mismo código del que partíamos ya que sabemos que $GRS_k(a,b) = GRS_k(a,\lambda b)$, vamos a comprobar que la matriz generatriz estándar del código de partida y el obtenido son iguales:

```
In: C.systematic_generator_matrix()
```

```
Out: [1 0 0 0 0 2 2 7]
     [0 1 0 0 0 7 8 4]
     [0 0 1 0 0 8 9 9]
     [0 0 0 1 0 6 7 1]
     [0 0 0 0 1 3 1 7]
```

```
In: CCCC=codes.GeneralizedReedSolomonCode([F(a1) for a1 in parametros[0]],
                                          k, [ F(b1) for b1 in parametros[1]]);
    CCCC.systematic_generator_matrix()==C.systematic_generator_matrix()
```

```
Out: True
```

6.2.2. Ataque al criptosistema de McEliece

Ahora vamos a explicar un ataque de filtración que funciona correctamente contra criptosistemas de McEliece construidos a partir de códigos Reed-Solomon generalizados y también directamente contra códigos Reed-Solomon generalizados de los cuales conocemos únicamente una matriz generatriz. Dicho ataque fue mostrado en el año 2014 en [32].

Para ello, partimos del código $GRS_k(a,b)$ y suponemos que la dimensión del código es menor o igual que la mitad de la longitud del código, n, es decir, $k \leq \frac{n}{2}$. Si esto no ocurre, por la proposición 3.23 podemos aplicar el procedimiento sobre el código dual, que si cumplirá nuestra restricción y después recuperar, a partir de los del dual, los vectores a y b de partida.

En *Sage* hemos implementado una función llamada $ataque\_filtracion(G\_pub)$, que realiza el ataque que vamos a explicar, donde el parámetro G_pub se corresponde con la matriz generatriz del código $GRS_k(a,b)$ o la matriz pública de un criptosistema de McEliece construido a partir de $GRS_k(a,b)$. Dicha función recurre también al dual si es necesario, de forma que su implementación permite realizar el ataque a códigos $GRS_k(a,b)$ de cualquier dimensión.

Comenzamos fijando las dos primeras coordenadas del vector a con valores 0 y 1 respectivamente, lo cual es posible ya que por [27] sabemos que se pueden fijar 3 puntos del vector a.

Debido a esto, todos los polinomios con 0 como raíz tendrán una codificación con primera coordenada nula. De igual forma ocurre con los polinomios que tengan a 1 como raíz, cuya codificación tendrá la segunda coordenada nula.

Entonces, si construimos un subcodigo de $GRS_k(a,b)$ formado únicamente a partir de las codificaciones de polinomios con una raíz de cierta multiplicidad en 0, dicho subcodigo se corresponde con el código recortado en la primera posición a partir de $GRS_k(a,b)$. De igual forma ocurre con la raíz en 1.

Denotamos por $C(i,j)$ el subcodigo de $GRS_k(a,b)$, con $i,j > 0$ y $i+j \leq k-1$ formado solamente a partir de las codificaciones de polinomios, $f(x)$, que tengan a 0 como raíz con multiplicidad al menos i y a 1 como raíz con multiplicidad al menos j, es decir, $x^i(x-1)^j | f(x)$.

Por tanto, se cumple que $C(1,0) = S_1(GRS_k(a,b))$, $C(0,1) = S_2(GRS_k(a,b))$, $C(1,1) = S_{\{1,2\}}(GRS_k(a,b))$ y denotamos $C(0,0) = GRS_k(a,b)$. A partir de esto, vamos a probar una proposición muy parecida a la proposición 6.9, que nos permitía realizar el ataque estándar, la cual también nos proporciona el método clave para comenzar el ataque por filtración.

Proposición 6.11. *Sea $k \leq \frac{n}{2}$ y el código $GRS_k(a,b)$, entonces $\forall i,j > 0$ tales que $i+j \leq k-2$ se verifica que*

$$C(i+1,j) \star C(i-1,j) = C(i,j)^{\star 2} \tag{6.1}$$

Nota 6.12. Todos los códigos $C(i,j)$ son Reed-Solomon generalizados ya que se obtienen mediante la proposición 6.11 a partir de los códigos $C(1,0)$, $C(0,1)$, $C(1,1)$ y

$C(0,0)$, los cuales sabemos que son Reed-Solomon generalizados y por tanto a partir de la ecuacion (6.1) y la proposición 6.9 estamos calculando los mismos códigos.

6.2.2.1. Ataque

Una vez mostrada la proposición 6.11 y la notación empleada, vamos a explicar el ataque, mostrando también el código empleado para la realización de cada paso.

1) En primer lugar, empleamos los códigos $C(0,0)$ y $C(1,0)$ para obtener, aplicando la ecuación (6.1) $k-2$ veces, el código $C(k-1,0)$, cuya dimensión es 1 y por tanto su matriz generatriz, $G_{(k-1)0}$ se corresponde solamente con un vector. El código empleado para su realización es:

```
In: C_Lin1=codes.LinearCode(G_pub)
    F=C_Lin1.base_field()
    n=C_Lin1.length()
    k=C_Lin1.dimension()
    k1=C_Lin1.dimension()
    if k>(n/2):
        C_Lin=codes.LinearCode(C_Lin1.parity_check_matrix())
        k1=C_Lin.dimension()
    else:
        C_Lin=codes.LinearCode(G_pub)
    C_10=C_Lin.shortened([0])
    CC=C_Lin.punctured([0])
    c=reducir_codigos(CC,C_10)
    ccc=vector(F,[c[0,i] for i in range(1,c.ncols())])
```

Obviamente, tenemos que el código $C(k-1,0)$ esta formado mediante la codificación de polinomios de la forma λx^{k-1}, ya que por definición solo se codifican polinomios que tengan en 0 una raíz de multiplicidad al menos $k-1$ y como la dimensión es k el grado de los polinomios no puede ser superior a $k-1$. Entonces, si denotamos $c = G_{(k-1)0}$, tenemos que $c = \lambda(a^{k-1} \star b)$.

2) Por otro lado, calculamos el código $C(k-2,1)$ de nuevo aplicando la ecuación (6.1) $k-3$ veces a partir de los códigos $C(0,1)$ y $C(1,1)$. Para ello, empleamos las siguientes líneas de código:

```
In: C_01=C_Lin.shortened([1])
    C_11=C_Lin.shortened([0,1])
    CC_01=C_01.punctured([0])
    c_1=reducir_codigos(CC_01,C_11)
    ccc_1=vector(F,[c_1[0,i] for i in range(c_1.ncols())])
```

De nuevo el código $C(k-2,1)$ tiene dimensión 1 y denotando $c' = G_{(k-2)1}$, siendo $G_{(k-2)1}$ su matriz generatriz, tenemos que el vector c' es de la forma $\alpha a^{k-2} \star (a-1) \star b$.

3) Entonces, el vector c tiene únicamente su primera coordenada nula y el vector c' tiene solo las dos primeras coordenadas nulas. Por tanto, esta bien definida la división $\frac{c'}{c}$ para todas las coordenadas excepto las dos primeras, la cual se corresponde con $\frac{\beta(a-1)}{a}$. Ahora, como solo hemos fijado las dos primeras coordenadas del vector a y por [27] sabemos que se pueden fijar 3 elementos, podemos seleccionar un valor arbitrario para β que denotamos por β_0. Entonces tenemos que la aplicación

$$f : \mathbb{F}_q \setminus \{0,1\} \longrightarrow \mathbb{F}_q$$
$$x \longmapsto \frac{\beta_0(x-1)}{x}$$

es una biyección que representa las distintas coordenadas obtenidas a partir del cociente de c y c' mediante la evaluación de la correspondiente coordenada de a. Por tanto, si calculamos la función inversa

$$f^{-1} : \mathbb{F}_q \setminus \{\beta_0\} \longrightarrow \mathbb{F}_q$$
$$y \longmapsto \frac{\beta_0}{\beta_0 - y} = \frac{1}{1 - \frac{1}{\beta_0} y}$$

y evaluamos en las coordenadas de $\frac{c'}{c}$ podemos recuperar todas las coordenadas del vector a excepto las dos primeras, lo cual no supone ningún problema ya que las hemos fijado al comienzo del ataque.

Hay que tener en cuenta que puede ocurrir que alguna coordenada de $\frac{c'}{c}$ tenga el valor β_0 y por tanto no se podría aplicar el paso anterior. Sin embargo, como nosotros elegimos el valor de β_0, siempre podremos seleccionar un cierto valor de β_0 de forma que la función f^{-1} este bien definida realizando su evaluación en todas las coordenadas de $\frac{c'}{c}$. Esto se debe a que todos las coordenadas del vector a deben ser distintas y por tanto también lo deben ser las de $\frac{c'}{c}$, luego existirá al menos un elemento del cuerpo empleado que no sea una coordenada de $\frac{c'}{c}$ y por tanto, dicho elemento sería un valor posible para β. El código empleado para la obtención del vector a, una vez que conocemos los vectores c y c', siguiendo los pasos mostrados es:

```
In: division=vector(F,[F(ccc_1[h]/ccc[h]) for h in range(n-2)])
    division1=vector(F,[a for a in division])
```

```
        numero=ZZ(1)
        while 1 in division1:
            division1=vector(F,[a*F.list()[numero] for a in division])
            numero+=1
        aa=vector(F,[F(1/(1-division1[h])) for h in range(n-2)])
        aaa=vector(F,[0,1]+[ff for ff in aa])
```

4) Para finalizar, una vez que tenemos el vector a podemos recuperar fácilmente el vector b ya que sabemos que $c = \lambda(a^{k-1} \star b)$. Por tanto, si realizamos la división por coordenadas del vector c entre a^{k-1} recuperamos todas las coordenadas de b excepto la primera, ya que $a_1 = 0$ y por tanto la división no esta definida. Entonces para recuperar la coordenada restante, podemos ir asignándola distintos elementos del cuerpo $\mathbb{F}_q$ empleado hasta obtener un vector que sea una palabra del código $GRS_k(a, b)$ de partida. El código implementado para la obtención del vector b es:

```
In: bbb=vector(F,[F(c[0,h]/aaa[h+1]**(k1-1)) for h in range(n-1)])
    obtener_bbb1=0
    for i in range(1,F.cardinality()):
        obtener_bbb=vector(F,[F.list()[i]]+[a for a in bbb])
        if C_Lin.syndrome(obtener_bbb)==0:
            obtener_bbb1=vector(F,[a for a in obtener_bbb])
            break
```

Por tanto, ahora que ya conocemos los vectores a y b del código empleado, también disponemos del algoritmo de corrección de errores del código. Entonces, si una persona realiza el cifrado de un mensaje m obteniendo y, nosotros aplicamos el algoritmo de corrección de errores que conocemos y nos permite recuperar c, ahora obteniendo una matriz cuadrada invertible de la matriz pública de partida, podemos recuperar el mensaje m de partida. En *Sage* también construimos una función llamada $recuperar\_mensaje(G\_pub, codigo, y)$, donde el parámetro G_pub hace referencia a la matriz pública que emplea el criptosistema de McEliece construido, $codigo$ se corresponde con el código obtenido mediante el empleo de la función $ataque\_filtraciom()$, del cual podemos emplear su algoritmo de corrección de errores, y por último el parámetro y hace referencia al vector cifrado que queremos descifrar para recuperar el mensaje. Para el descifrado del vector y, dicha función se construye mediante el siguiente código:

```
In: def recuperar_mensaje(G_pub,codigo,y):
        F=G_pub.base_ring()
        y_sin_err=codigo.decode_to_code(y)
```

```
        columnas=[]
        G_escalonada=G_pub.echelon_form()
        for i in range(G_escalonada.nrows()):
            pos=0
            while G_escalonada[i,pos]==F(0):
                pos+=1
            columnas+=[pos]
        Gcuadrada=transpose(matrix(F,[G_pub.column(a) for a in columnas]))
        y1_sin_err=vector(F,[y_sin_err[i] for i in columnas])
        return y1_sin_err*Gcuadrada.inverse()
```

Nota 6.13. Destacamos que en el paso 3 el valor de β no siempre puede ser 1, como sugiere el artículo [32], ya que puede ocurrir que el vector $\frac{c'}{c}$ tenga alguna coordenada con valor 1 y entonces la función f^{-1} no estaría bien definida. Por ejemplo, si realizamos el ataque de filtración sobre el código $GRS_4(a,b)$ construido en $\mathbb{F}_{17}$, con $a = (0,1,2,3,4,5,6,7,8,9)$ y $b = (2,3,4,5,6,7,8,9,10,11)$, obtendremos como vector $\frac{c'}{c} = (7,15,2,1,6,12,8,3)$ si tomamos $\beta = 1$. En cambio, tomando $\frac{1}{\beta} = 2$ si se puede realizar ya que $\beta = \frac{1}{2} = 9$ no se corresponde con ninguna coordenada de $\frac{c'}{c}$.

Ejemplo 6.2. Vamos a construir un criptosistema de McEliece empleando un código Reed-Solomon generalizado $GRS_4(a,b)$ sobre el cuerpo $\mathbb{F}_{17}$, donde el valor de los vectores $a = (0,1,2,3,4,5,6,7,8,9)$ y $b = (2,3,4,5,6,7,8,9,10,11)$ es desconocido. Para ello construimos en primer lugar el código $GRS_4(a,b)$:

```
In: F=GF(17);
    a=[F(i) for i in [F.list()[i] for i in range(10)]]; #3,9.....
    b=[F(i) for i in [F.list()[i] for i in range(2,12)]];
    k=4;
    C=codes.GeneralizedReedSolomonCode(a, k, b);
    C

Out: [10, 4, 7] Generalized Reed-Solomon Code over GF(17)
```

Ahora, cargamos el fichero que contiene la clase *McEliece* para poder construir el criptosistema:

```
In: load("./McEliece.sage");
```

Entonces, construimos el criptosistema indicando que queremos introducir 3 errores por cada bloque que ciframos, lo cual se corresponde con la capacidad correctora del código $GRS_4(a,b)$ que estamos empleando:

```
In: crip_mceliece=McEliece(C,3)
    crip_mceliece

Out: Criptosistema de McEliece construido a partir de [10, 4, 7]
     Generalized Reed-Solomon Code over GF(17) que introduce 3 errores
```

Mostramos las matrices de permutación e invertibles empleadas, asi como la matriz pública que emplearemos para realizar el cifrado de mensajes:

```
In: crip_mceliece.matriz_invertible()

Out: [ 6  2  0 12]
     [ 6 10  2  9]
     [ 0 16  6 11]
     [15  9 13  0]

In: crip_mceliece.matriz_permutacion()

Out: [0 0 1 0 0 0 0 0 0 0]
     [0 0 0 0 0 0 0 0 0 1]
     [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 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 1 0 0 0 0]
     [0 0 0 0 0 0 0 1 0 0]
     [0 0 0 0 1 0 0 0 0 0]

In: G_pub=crip_mceliece.public_matrix()
    G_pub

Out: [16  0 12 14  0 11  4  1  4  9]
     [16 14 12  6  5  7 13  8 15 13]
     [15 16  0  6  7  7 16  2 15 14]
     [ 0  7 13 13  8  9 12 10  9  9]
```

A partir de la matriz pública del criptosistema, empleamos la función *ataque_filtracion*() para recuperar el código empleado:

```
In: [a2,b2,codigo2]=ataque_filtracion(G_pub)
    codigo2.generator_matrix()
```

```
Out: [ 2  1  7  8  7 14  9  3  2  7]
     [ 0  1  6  1 11  2 10 10  7  9]
     [ 0  1 10 15 10 10 13  5 16 14]
     [ 0  1 11  4  6 16  5 11  5  1]
```

Validamos que el código obtenido es el mismo que el correspondiente a la matriz pública del criptosistema mediante sus matrices generatrices:

```
In: codigo2.systematic_generator_matrix()==G_pub.echelon_form()

Out: True
```

Por último, supongamos que se realiza el cifrado del mensaje $m = (11, 1, 11, 1)$ obteniendo el vector $y = (0, 2, 4, 1, 5, 12, 7, 8, 12, 3)$. Si ahora nosotros interceptamos el mensaje cifrado, y, somos capaces de recuperar el mensaje de partida m empleando la función *recuperar_mensaje*() que hemos implementado:

```
In: m=vector(F,[F.random_element() for i in range(k)])
    m

Out: (11, 1, 11, 1)

In: y=crip_mceliece.cifrado(m)
    y

Out: (0, 2, 4, 1, 5, 12, 7, 8, 12, 3)

In: recuperar_mensaje(G_pub,codigo2,y)

Out: (11, 1, 11, 1)
```

Capítulo 7

Códigos de producto de matrices

7.1. Introducción

Los códigos de producto de matrices fueron introducidos por Blackmore y Norton en un artículo desarrollado en 2001 [21]. Dichos códigos surgieron a partir de la generalización de la construcción de Plotkin, también conocida como $(u \mid u+v)$.

La principal característica de estos códigos es que se construyen a partir de un conjunto de códigos mas pequeños, lo cual resulta interesante para la teoría de codificación.

Definición 7.1. Sea $A = [a_{ij}]$ una matriz $M \times N$ con entradas en $\mathbb{F}_q$, con $M \leq N$ y sean $C_1, ..., C_M$ códigos de longitud n sobre el cuerpo $\mathbb{F}_q$. Diremos que el código de producto de matrices, denotado por $[C_1 \cdots C_M] \cdot A$ es el conjunto de todos los productos de matrices $[c_1 \cdots c_M] \cdot A$, donde $c_i \in C_i$ hace referencia a un vector columna de tamaño n, $\forall i = 1, ..., M$. Por tanto, las palabras del código $[C_1 \cdots C_M] \cdot A$ son matrices $n \times N$ de la forma

$$c = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1M} \\ c_{21} & c_{22} & \cdots & c_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nM} \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M1} & a_{M2} & \cdots & a_{MN} \end{pmatrix} =$$

$$\begin{pmatrix} c_{11}a_{11} + \cdots + c_{1M}a_{M1} & \cdots & c_{11}a_{1N} + \cdots + c_{1M}a_{MN} \\ \vdots & \ddots & \vdots \\ c_{n1}a_{11} + \cdots + c_{nM}a_{M1} & \cdots & c_{n1}a_{1N} + \cdots + c_{nM}a_{MN} \end{pmatrix}$$

Nota 7.2. Podemos ver que la j-esima columna en la palabra del código también se puede escribir para simplificar como $\sum_{i=1}^{M} c_{hi}a_{ij}$ para $1 \leq h \leq n$, donde $1 \leq j \leq N$.

Además, si denotamos por $c_i = [c_{1i}, ..., c_{ni}]^T$ entonces podemos escribir la j-esima columna como $\sum_{i=1}^{M} c_i a_{ij}$.

Entonces, las palabras del código de $[C_1 \cdots C_M] \cdot A$ se pueden ver como vectores escritos de la forma

$$c = [\sum_{i=1}^{M} c_i a_{i1}, \sum_{i=1}^{M} c_i a_{i2}, ..., \sum_{i=1}^{M} c_i a_{iN}] \in \mathbb{F}_q^{nN} \tag{7.1}$$

donde la matriz se ha escrito en el conocido como orden de tamaño de mayor columna. En esta forma, observamos que el código tiene tamaño nN

Entonces si $C_1, ..., C_M$ son códigos lineales, se puede encontrar una matriz generatriz, G, para el código de producto de matriz, cuya definición es evidente a partir de como hemos realizado la construcción de los códigos de producto de matrices:

$$G = \begin{pmatrix} G_1 a_{11} & \cdots & G_1 a_{1N} \\ \vdots & \ddots & \vdots \\ G_M a_{M1} & \cdots & G_M a_{MN} \end{pmatrix}$$

donde G_i es la matriz generatriz del código C_i para $i = 1, ..., M$.

Además, por la matriz generatriz del código podemos ver que la dimensión del código de producto de matriz es $k = k_1 + ... + k_M$ donde k_i es la dimensión del código C_i, $i = 1, ..., M$.

Aquí hay que remarcar que esta igualdad solamente ocurre en el caso en que la matriz A tenga el máximo rango, lo cual supondremos a partir de ahora salvo que se diga lo contrario. En el caso en que la matriz A no tenga máximo rango entonces se tiene que $k < k_1 + ... + k_M$.

De igual forma que hicimos en el capítulo 4 con los códigos de Goppa, para los códigos de producto de matrices también tenemos que realizar su implementación completamente, al no tener nada construido *Sage* acerca de estos códigos.

Para ello, creamos una clase llamada *MatrixProducCodes*, la cual tiene un constructor, $\_\_init\_\_(self, A, lista\_codigos)$, en el que debemos indicar en el parámetro A dicha matriz y en el parámetro *lista_codigos* el conjunto de M códigos a partir de los cuales realizamos la construcción del código de producto de matrices.

Nada mas comenzar la ejecución del constructor realizamos una serie de validaciones que deben cumplir los parámetros introducidos para generar un código de pro-

ducto de matrices de forma correcta y para el que además seamos capaces de aplicar el algoritmo de descodificación que veremos en la sección 7.3.

Después, únicamente calcula la matriz generatriz del código que esta construyendo mediante el siguiente código:

```
In: M=A.nrows(); N=A.ncols();
    n=lista_codigos[0].generator_matrix().ncols();
    filas=0;
    for a in lista_codigos:
        filas+=a.dimension();
    generatriz=matrix(A.base_ring(),filas,N*n);
    h=-1;
    for i in range(M):
        for k in range(lista_codigos[i].dimension()):
            h+=1;
            for j in range(N*n):
                generatriz[h,j]=A[i,floor(j/n)]*lista_codigos[i].
                                                generator_matrix()[k,j%n];
```

Por último, también calculamos la matriz de control del código:

```
In: Gtran=transpose(generatriz);
    H=Gtran.left_kernel().basis_matrix();
```

Definición 7.3. Sea A una matriz $M \times N$, denotaremos $A(j_1, ..., j_t)$ a la matriz cuadrada de tamaño t formada a partir de las t primeras filas de la matriz A y las columnas $j_1, ..., j_t$, donde $1 \leq j_1 < \cdots < j_t \leq N$ con $1 \leq t \leq M$.

Proposición 7.4. *Sea A una matriz $M \times N$. Si tenemos una matriz formada por M columnas de A que es no singular entonces se verifica que*

$$|[C_1 \cdots C_M] \cdot A| = |C_1| \cdots |C_M|$$

A partir de lo que hemos visto hasta ahora, podemos observar que hemos obtenido tanto la longitud del código $[C_1 \cdots C_M] \cdot A$, como su cardinalidad y dimensión. Sin embargo, para obtener la distancia mínima de dicho código vamos a introducir primero el concepto de matriz no singular por columnas y matriz triangular, que nos será útil mas adelante.

Definición 7.5. Dada una matriz A de tamaño $M \times N$, diremos que A es una matriz no singular por columnas (NSC) si $A(j_1, ..., j_t)$ es no singular para cada $1 \leq t \leq M$ y $1 \leq j_1 < \cdots < j_t \leq N$.

Obviamente se verifica que toda matriz no singular por columnas es también una matriz no singular, sin embargo el reciproco no es cierto en muchas ocasiones, por ejemplo, si consideramos la matriz

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

es evidente que es no singular ya que $det(A) = -1$, y no es NSC ya que es necesario que en la primera fila todos sus elementos sean no nulos.

En *Sage* empleamos el siguiente código para validar si una matriz dada es no singular por columnas dentro de la clase *MatrixProductCodes*.

```
In [ ]: validar=True;
        for i in range(1,A.nrows()+1):
            if validar:
                for orden in Subsets(A.ncols(),i):
                    AA=matrix(A.base_ring(),[[ A[h,j-1] for j in orden] for
                                                              h in range(i)])
                    print(AA)
                    if not AA.is_invertible():
                        validar=False;
                        break;
```

En la sección 7.3 se muestra el funcionamiento de la función *Subsets*()

Definición 7.6. Diremos que un código de producto de matrices es NSC si la matriz A asociada al código es NSC.

Definición 7.7. Diremos que una matriz A es triangular si existe alguna permutación por columnas, π, que transforma la matriz en una matriz triangular superior, es decir, si $a_{i\pi(j)} = 0 \, \forall i > \pi(j)$

Proposición 7.8. *Una matriz triangular NSC A de tamaño $M \times N$ tiene exactamente $i-1$ ceros en la i-esima fila, con $1 \leq i \leq M$.*

Ahora vamos a introducir la proposición que nos permitirá conocer la distancia mínima de un código de producto de matrices, pero antes de ello, denotamos $R_i = (a_{i1}, ..., a_{iN}) \in \mathbb{F}_q^N$, es decir, R_i es un vector de tamaño N formado por los elementos de la i-esima fila de la matriz A, los cuales pertenecen a $\mathbb{F}_q$.

A partir de ello, construimos ahora los códigos C_{R_i} generados a partir de los vectores R_j con $j = 1, ..., i$, es decir, generados por las i primeras filas de la matriz A. Por tanto, $C_{R_i} = \langle R_1, ..., R_i \rangle$, y denotamos por D_i la distancia del código C_{R_i}.

Proposición 7.9. *Dado un código de producto de matrices $C = [C_1 \cdots C_M] \cdot A$, entonces se cumple la siguiente desigualdad para la mínima distancia:*

$$d(C) \geq min\{d_1 \cdot D_1, d_2 \cdot D_2, ..., d_M \cdot D_M\}$$

donde d_i hace referencia a la distancia del código C_i y D_i a la distancia del código C_{R_i}.

Nota 7.10. Como C_{R_i} esta generado a partir de los vectores $R_1, ..., R_i$, es decir, a partir de las i primeras filas de la matriz A, es evidente que $C_{R_i} \subset C_{R_{i+1}}$ y por tanto también que $D_i \geq D_{i+1}$. Por tanto, para obtener un código de producto de matrices con buenos parámetros es conveniente elegir los códigos C_j de manera que la distancia mínima de estos se vaya incrementando o al menos sea igual, es decir, $d_{j+1} \geq d_j$, para obtener de esta forma la mayor distancia posible en el código de producto de matrices a partir de los códigos que lo generan, lo cual sabemos que permite corregir una mayor cantidad de errores. También es aconsejable coger C_{j+1} con una dimensión menor o igual que la de C_j.

Por tanto, para que pueda ocurrir lo descrito anteriormente será necesario que C_1 tenga una dimensión grande y que el último de los códigos tenga una distancia mínima grande.

Ademas de intentar maximizar la distancia mínima del código de producto de matrices, como esta viene delimitada a partir de una desigualdad, es importante también saber exactamente como de grande es la distancia mínima del código, para conocer de esta forma, la cantidad máxima exacta de errores que podemos corregir.

Por ello, vamos a ver a continuación dos formas de obtener la igualdad en la desigualdad que tenemos de la distancia.

Para la primera forma, consideramos que los códigos $C_1, ..., C_M$ son encajados, es decir, $C_M \subset C_{M-1} \subset \cdots \subset C_1$. Estos códigos verifican que $d_M \geq d_{M-1} \geq \cdots \geq d_1$, luego no estamos restringiendo de forma excesiva el conjunto de códigos a escoger al ser del tipo deseado para maximizar la distancia del código de producto de matrices. Para los conjuntos de códigos con esta propiedad se verifica el siguiente teorema, con la igualdad deseada para la distancia mínima del código de producto de matrices.

Teorema 7.11. *Dado un código de producto de matrices $C = [C_1 \cdots C_M] \cdot A$ donde C_i con $i = 1, ..., M$ son códigos que verifican $C_M \subset C_{M-1} \subset \cdots \subset C_1$ y A es una matriz de tamaño $M \times N$. Entonces se cumple que*

$$d(C) = min\{d_1 * D_1, d_2 * D_2, ..., d_M * D_M\}$$

En cuanto a la segunda forma, imponemos que la matriz A sea NSC y triangular, lo cual nos permite obtener la igualdad en la desigualdad de la distancia para los códigos de producto de matrices que tenemos como podemos ver en el siguiente teorema.

Teorema 7.12. *Si A es una matriz $M \times N$ NSC y el código NSC es $C = [C_1 \cdots C_M] \cdot A$, entonces se cumple que:*

1) $|C| = |C_1| \cdots |C_M|$

2) $d(C) \geq d^* = min\{Nd_1, (N-1)d_2, ..., (N-M+1)d_M\}$

3) Si ademas A es también una matriz triangular entonces $d(C) = d^*$

En *Sage* construimos una función que nos proporciona la distancia mínima del código de producto de matrices construido llamada *minimum_distance*(), la cual se obtiene según lo visto en el teorema 7.11, al cumplirse para todos los códigos de producto de matrices que construimos las hipótesis del teorema.

```
In: for i in range(self._A.nrows()):
        G=matrix(self._A.base_ring(),[[self._A[j,k] for k in
                        range(self._A.ncols())] for j in range(i+1)])
        D.append(codes.LinearCode(G).minimum_distance())
        d.append(self._lista_codigos[i].minimum_distance())
        if(d[i]*D[i]<distancia):
            distancia=d[i]*D[i];
    return distancia;
```

7.2. Codificación

Para realizar el proceso de codificación de los códigos de producto de matrices, al igual que hacíamos para codificar los códigos lineales en la sección 3.2, simplemente tenemos que multiplicar un mensaje, m, con longitud la dimensión del código, k, por la matriz generatriz del código, G, es decir, $c = mG$, donde c hace referencia al mensaje codificado. En *Sage* creamos una función llamada *encode*(*self*, *mensaje*) que realiza dicho proceso para codificar el parámetro *mensaje*:

```
In: def encode(self, mensaje):
       if mensaje.length()==self._k and
                        mensaje.base_ring()==self._G.base_ring():
           return mensaje*self._G;
```

Además de la forma clásica para codificar un mensaje, para los códigos de producto de matrices también se podría dividir cada trozo del mensaje en bloques de tamaño correspondiente a la dimensión de cada código empleado dentro del MPC y codificar por separado cada bloque. Después si ponemos las palabras codificadas de cada uno de los subcodigos como columnas y los multiplicamos por la matriz A según vimos en la definición 7.1 obtendríamos el mensaje codificado.

7.3. Descodificación

Después de mostrar los códigos de productos de matrices así como la obtención de sus parámetros, de forma exacta, si empleamos una matriz A no singular por columnas y triangular gracias al teorema 7.12, o si utilizamos códigos incrustados para la generación de los códigos de producto de matrices según lo visto en el teorema 7.11.

Ahora nos preguntamos si podemos emplear los códigos de productos de matrices para realizar la construcción de criptosistemas de McEliece. Por tanto, lo que nos falta por obtener es como realizar una descodificación para dichos códigos.

Por ello, vamos a mostrar un algoritmo que permita realizar la descodificación de los códigos de producto de matrices, para el cual es necesario que los códigos que empleamos para la obtención de los códigos de producto de matrices sean encajados, así como que la matriz A sea no singular por columnas, lo cual, vimos anteriormente que no es una condición tan restrictiva ya que queremos formar códigos de producto de matrices con la mayor mínima distancia posible. También es necesario que los códigos empleados para generar el código de producto de matrices dispongan cada uno de ellos de un algoritmo de descodificación.

Teniendo en cuenta lo anterior, consideramos, $C = [C_1 \cdots C_M] \cdot A$, un código de producto de matrices formado por la matriz A no singular por columnas y por los códigos $C_1, ..., C_M$ encajados y de longitud n, donde cada código C_i, con $i = 1, ..., M$, posee un algoritmo de descodificación, que denotamos por DC_i, el cual es capaz de corregir hasta un máximo de $t_i = \frac{d_i - 1}{2}$ errores de una palabra del código C_i.

Vamos a mostrar un método de descodificación que nos permita corregir siempre la máxima capacidad correctora del código, es decir, $t = \frac{d(C)-1}{2}$, siendo $d(C)$ la distancia del código de producto de matrices C, obtenida a partir del teorema 7.11 o el teorema 7.12.

Antes de mostrar el método de descodificación, mostramos la función implementada en *Sage* llamada $introducir\_errores(self, mensajecodificado, cantidad)$, la cual realiza un proceso muy semejante a la función $introducir\_errores()$ de los códigos de Goppa que vimos en la sección 4.3, con la diferencia de que en dicha función solo se introducian errores en vectores binarios, mientras que en esta función se introducen errores en vectores que se encuentran en cualquier cuerpo finito. Por tanto, en este caso tenemos que generar elementos aleatorios para las distintas posiciones del vector de errores. Esta generación de elementos aleatorios lo hacemos empleando la función $random\_element()$ ya implementada en *Sage* sobre distintos cuerpos, entre ellos los cuerpos finitos que nosotros empleamos.

```
In: if mensajecodificado.is_vector() and boolG and
```

```
                    mensajecodificado.base_ring()== self._F and
                    len(mensajecodificado)>= cantidad and
                    len(mensajecodificado)>= self._n and cantidad>2:
        entrar=True;
        while entrar:
            mensajeerror=vector(self._F,[mensajecodificado[p] for p
                                        in range(len(mensajecodificado))]);
            indices =range(len(mensajeerror));
            posiciones_error=[];
            for i in range(cantidad):
                error=self._F(0);
                while error==self._F(0):
                    error=self._F.random_element();
                posicion=indices.pop(randint(0,len(indices)-1));
                mensajeerror[posicion]+= error;
                posiciones_error.append(posicion)
            bloque_longitud=self._n/self._A.nrows()
            nosale=False
            for i in range(self._A.nrows()):
                contar=0;
                for a in posiciones_error:
                    if i*bloque_longitud <= a and a<(i+1)*bloque_longitud:
                        contar+=1;
                if contar<>0 and contar<3:
                    nosale=True
            if not nosale:
                entrar=False;
        return mensajeerror;
    elif mensajecodificado.is_vector() and mensajecodificado.base_ring()==
                            self._F and len(mensajecodificado)>= cantidad
                            and len(mensajecodificado)>= self._n:
        mensajeerror=vector(self._F,[mensajecodificado[p] for p
                                    in range(len(mensajecodificado))]);
        indices =range(len(mensajeerror));
        for i in range(cantidad):
            error=self._F(0);
            while error==self._F(0):
                error=self._F.random_element();
            mensajeerror[indices.pop(randint(0,len(indices)-1))]+= error;
        return mensajeerror;
```

Podemos observar también en las sentencias mostradas que distinguimos entre los códigos de producto de matrices construidos a partir de al menos un código de Goppa y los construidos sin emplear ningún código de Goppa, empleando la función $is\_Gopppa()$ que implementamos en dichos códigos cuyo valor es almacenado en la variable $boolG$ que utilizamos.

El motivo de esta distinción se encuentra como vimos en 4.3.2, que el algoritmo de descodificación implementado en los códigos de Goppa no es capaz de corregir únicamente uno o dos errores. Por tanto, en este caso tenemos que restringir las posiciones aleatorias de los errores, de forma que en cada bloque que pertenezca a un código de Goppa haya o cero errores o mas de dos.

Con respecto al método de descodificación, vamos a emplear los algoritmos de descodificación de los M códigos que generan el código de producto de matrices. Mostramos a continuación el algoritmo que permite descodificar una palabra recibida $y = c + e$ donde $c \in C$ y e es un vector de errores de longitud nN con un peso menor o igual a la capacidad correctora del código, es decir, $\omega(e) \leq t$. Para ello, dicho algoritmo debe recibir como parámetros tanto la matriz A como los M códigos junto con sus algoritmos de descodificación, además de la palabra y que queremos descodificar. Para la función implementada en *Sage* llamada $decodificar\_MPC(self, y1, decodificarV = True)$, únicamente es necesario la introducción de la palabra y en el parámetro llamado $y1$, al estar el resto de datos almacenados en la propia clase del código de producto de matrices construido. Además, dicha función también tiene el parámetro booleano opcional $decodificarV$ que nos permitirá indicar si queremos descodificar el vector introducido o únicamente corregir los errores de dicho vector. Si no se indica ningún valor al parámetro $decodificarV$, este recibirá automáticamente el valor *True* y por tanto, realizará la descodificación del vector introducido.

Algoritmo 7.13.

1 : $y' = y$;
2 : $A' = A$;
3 : $for\ \{i_1, ..., i_M\} \subset \{1, ..., N\}$:
4 : $\quad y = y'$;
5 : $\quad A = A'$;
6 : $\quad for\ j = 1, ..., M$:
7 : $\quad\quad y_{i_j} = DC_j(y_{i_j})$;
8 : $\quad\quad if\ y_{i_j} =$ ”fallo” :
9 : $\quad\quad\quad$ *break* #Rompemos el for y tomamos otro $\{i_1, ..., i_M\}$ en la linea 2
10 : $\quad\quad for\ k = j + 1, ..., M$:
11 : $\quad\quad\quad y_{i_k} = y_{i_k} - \frac{a_{ji_k}}{a_{ji_j}} y_{i_j}$;

12 : $columna_{i_k}(A) = columna_{i_k}(A) - \frac{a_{ji_k}}{a_{ji_j}} columna_{i_j}(A)$;
13 : Recuperamos $(c_1, ..., c_M)$;
14 : $y = [c_1, ..., c_M] \cdot A$;
15 : $if\ y \in C$ and $\omega(y - y') \leq [\frac{d(C)-1}{2}]$:
16 : return y;

Si y es la palabra recibida, el algoritmo considera $\{i_1, ..., i_M\} \subset \{1, ..., N\}$ un subconjunto ordenado de indices, de forma que el vector de errores $e = (e_1, ..., e_N)$ satisfaga que $\omega(e_{i_j}) \leq t_j\ \forall j \in \{1, ..., M\}$. Si el subconjunto ordenado de indices no verifica esto en todos los indices, por ejemplo, supongamos que no se cumple para el indice i_{j_0}, entonces el correspondiente algoritmo de descodificación DC_{j_0} no es capaz de corregir todos los errores que tiene ese bloque, y por tanto, puede ocurrir que no encuentre ninguna palabra del código C_{j_0} que se encuentre a distancia menor que t_{j_0} del bloque $y_{i_{j_0}}$ y en ese caso, podemos suponer que devuelve una respuesta de "fallo", o que por el contrario si encuentre una palabra que pertenezca al código C_{j_0} y entonces el algoritmo nos devolverá como resultado una palabra del código que no es la correcta. En esta situación, si no existe otro indice para el que el respectivo algoritmo de descodificación proporcione como respuesta "fallo", podemos percatarnos de que la descodificación realizada es errónea comprobando si la palabra descodificada, constituida por todos los bloques descodificados, es una palabra del código de producto de matrices C y comprobando también que no se han corregido mas de t errores. En cualquiera de los dos casos, habría que considerar otro subconjunto distinto de indices y realizar el mismo procedimiento.

Además, podemos observar que es suficiente con tomar un subconjunto de indices de M elementos dentro de los N posibles ya que como la matriz A es no singular por columnas, de las N columnas, M son linealmente independientes y las $N - M$ restantes se pueden obtener como combinación lineal de las otras.

Para ir tomando todos los posibles subconjuntos de indices de un conjunto dado de mayor tamaño, *Sage* tiene implementada ya dos funciones, la primera se llama $Permutations(a, b)$, la cual nos proporciona todos los subconjuntos posibles de tamaño b a partir de los elementos que componen la lista a, o también a puede denotar un entero, que indica desde el uno, todos los enteros que constituyen el conjunto a considerar. También presenta otro tipo de parámetros para realizar otras construcciones, pero no los vemos ya que no es necesario emplearlos.

En cuanto a la segunda función, se llama $Subsets(a, b)$, la cual nos proporciona de igual forma que la función $Permutations()$, todos los subconjuntos posibles de tamaño b a partir de los elementos que componen la lista a, o también a puede denotar un entero, que indica desde el uno, todos los enteros que constituyen el conjunto a con-

siderar, de forma que dichos subconjuntos presenten sus elementos ordenados de forma creciente. Por tanto, la diferencia se encuentra en que mientras que con la función *Permutations*() podemos tener los subconjuntos $[1,2]$ y $[2,1]$, con *Subsets*() únicamente tendríamos el subconjunto $\{1,2\}$.

Por tanto, nosotros necesitamos emplear ahora la función *Permutations*(), de forma que genere todas las posibles permutaciones de tamaño M sobre el conjunto de tamaño N que empleamos.

```
In:  ordenes= Permutations(self._A.ncols(),self._A.nrows())
     for orden in ordenes:
```

Si denotamos para cada $i = 1, ..., N$ por $y_i = \sum_{j=1}^{M} a_{ji}c_j + e_i$ el i-esimo bloque de la palabra recibida y. Entonces, como el bloque i_1 tiene un vector de error menor que la capacidad correctora del código C_1 y además al ser códigos incrustados sabemos que cualquier bloque de c es una palabra del código C_1. Empleando el algoritmo de descodificación DC_1 podemos obtener el bloque i_1 sin errores, ademas del bloque i_1 del vector de errores.

Por tanto, realizamos una distinción entre si el bloque que vamos a descodificar pertenece a un código de Goppa u otro código lineal de los que vimos en el capítulo 3, ya que las funciones de descodificación se denotan de distinta forma:

```
In: try:
        boolG=self._lista_codigos[j-1].is_Goppa();
    except:
        boolG=False;
    yy[(orden[j-1]-1)]=y[(orden[j-1]-1)*tam:(orden[j-1])*tam];
    try:
        if boolG:
          y[(orden[j-1]-1)*tam:(orden[j-1])*tam]=self._lista_codigos[j-1]
                          .decodificar(y[(orden[j-1]-1)*tam:(orden[j-1])
                                                        *tam], False, False);
        else:
          y[(orden[j-1]-1)*tam:(orden[j-1])*tam]=self._lista_codigos[j-1]
                  .decode_to_code(y[(orden[j-1]-1)*tam:(orden[j-1])*tam]);
    except:
        salir=True;
    if salir:
        break;
```

Podemos observar que la descodificación la realizamos dentro de la sentencia *try*, de modo que si la función de descodificación nos devuelve algún error, salimos del bucle para escoger otra permutación posible.

Para eliminar los errores en el resto de bloques vamos tomando el vector $y^{k)}$, con $k = 2, ..., M$, formado por las componentes

$$y_i^{k)} = y_i^{k-1)} - \frac{a_{(k-1)i}^{k-1)}}{a_{(k-1)i_{k-1}}^{k-1)}}(y_{i_{k-1}}^{k-1)} - e_{i_{k-1}}) = \sum_{j=k}^{M} a_{ji}^{k)} c_j + e_i \ \forall i \neq i_1, ..., i_{k-1}$$

donde

$$a_{ji}^{k)} = a_{ji}^{k-1)} - \frac{a_{(k-1)i}^{k-1)}}{a_{(k-1)i_{k-1}}^{k-1)}} a_{ji_{k-1}}^{k-1)}$$

y

$$y_{i_1}^{k)} = y_{i_1}^{k-1)}, ..., y_{i_{k-2}}^{k)} = y_{i_{k-2}}^{k-1)}, y_{i_{k-1}}^{k)} = y_{i_{k-1}}^{k-1)} - e_{i_{k-1}}$$

Por tanto, en *Sage* empleamos las siguientes sentencias para actualizar la matriz A y el vector y:

```
In: for k in [j+1..len(self._lista_codigos)]:
        y[(orden[k-1]-1)*tam:(orden[k-1])*tam]=y[(orden[k-1]-1)
                          *tam:(orden[k-1])*tam]-(A[j  -1,orden[k-1]  -1]
                          /A[j  -1,orden[j-1]  -1]*y[(orden[j-1]-1)
                                                *tam:(orden[j-1])*tam]);
        columna=A.column(orden[k-1]  -1)-(A[j  -1,orden[k-1]  -1]
                       /A[j  -1,orden[j-1]  -1]*A.column(orden[j-1]  -1));
        for h in range(A.nrows()):
          A[h,orden[k-1]  -1]=columna[h];
```

Como por construcción, si $i \in \{1, ..., M\} \setminus \{i_1, ..., i_{k-1}\}$, tenemos que el bloque i-esimo de $y^{k)}$ es una palabra del código C_k mas el i-esimo bloque del vector error, por tanto, usando el algoritmo de descodificación DC_k sobre el i_k-esimo bloque, obtenemos el bloque de error i_k y $\sum_{j=k}^{M} a_{ji_k}^{k)} c_j$.

Ahora, como tenemos los bloques de error e_i, con $i \in \{i_1, .., i_M\}$, podemos recuperar los bloques de la palabra de código $\sum_{j=k}^{M} a_{ji}^{k)} c_j$, con $i \in \{i_1, .., i_M\}$, y por tanto, tenemos el vector $(\sum_{j=k}^{M} a_{ji_1}^{k)} c_j, ..., \sum_{j=k}^{M} a_{ji_M}^{k)} c_j)$ sin errores, el cual es igual al resultado que se obtiene al multiplicar $[c_1 \cdots c_M] \cdot A(i_1, ..., i_M)$, siendo $A(i_1, ..., i_M)$ la submatriz de tamaño M de la matriz A formada a partir de las columnas $i_1, ..., i_M$ de la matriz A. Como la matriz $A(i_1, ..., i_M)$ es invertible al tener rango máximo por ser A no singular por columnas, podemos obtener los vectores $c_1, ..., c_M$.

Por ultimo, realizando la multiplicación $[c_1 \cdots c_M] \cdot A$ podemos recuperar los $N - M$ bloques restantes del vector c, es decir, los $N - M$ bloques restantes del vector y sin errores, como queríamos.

Obviamente, si nos encontramos en el caso particular en que $N = M$ obtenemos directamente todo el mensaje codificado c sin errores, por lo que no es necesario realizar lo visto en los dos últimos párrafos. En cambio si $N \neq M$ si es necesario realizarlos para obtener la palabra entera sin errores:

```
In: for p in range(len(ee)):
        ee[p]=yy[p]- y[p*tam:(p+1)*tam];
        devolver[p*tam:(p+1)*tam]=devolver[p*tam:(p+1)*tam]-ee[p];
    if self._A.ncols()<> self._A.nrows():
        devolver1=vector(devolver.base_ring(),[a for a in devolver])
        orden_ordenado=[]
        for a in [1..self._A.ncols()]:
            if a in orden:
                orden_ordenado+=[a];
        Acuadrada=transpose(matrix(self._F,[self._A.column(a-1) for a
                                                    in orden_ordenado]));
        partemensaje=transpose(matrix(self._F,[[ a for a in
                devolver1[(p-1)*tam:(p)*tam]] for p in orden_ordenado]));
        partemensajedescifrado=partemensaje*Acuadrada.inverse();
        corregidoerrores=partemensajedescifrado*self._A
        lista=[]
        for a in range(corregidoerrores.ncols()):
            lista+=[ corregidoerrores.column(a)[p] for p in
                                range(len(corregidoerrores.column(a)))]
        devolver=vector(self._F,lista)
    error_r=devolver2-devolver
```

Para finalizar la ejecución de la función $decodificar\_MPC()$, en caso de que el valor booleano introducido en el parámetro $decodificarV$ sea *True* y que la palabra sobre la que hemos corregido los errores, sea una palabra del código de producto de matrices y la cantidad de errores menor que la capacidad correctora, se realiza la descodificación de forma semejante a la vista en los códigos de Goppa en la sección 4.3.3:

```
In: if self._H*devolver==0 and error_r.hamming_weight()<=tt:
        if decodificarV:
            Gcuadrada=matrix(self._F,self._k);
            while Gcuadrada.rank()<> self._k or Gcuadrada.ncols()<>self._k:
                columnas=[];
```

```
            totalcolumnas=range(self._n);
            Gcuadrada=self._G;
            for i in range(self._k):
                aleatorio=randint(0,len(totalcolumnas)-1);
                columnas.append(totalcolumnas.pop(aleatorio));
            columnas.sort();
            Gcuadrada=transpose(matrix(self._F,[self._G.column(a) for
                                                a in columnas]));
        partemensaje=vector(self._F,[devolver[j] for j in columnas]);
        mensajedescifrado=partemensaje*Gcuadrada.inverse();
        return mensajedescifrado;
```

Destacamos que el algoritmo 7.13 que hemos visto siempre funciona, cuando se cumplen las exigencias impuestas, para al menos un subconjunto $\{i_1, .., i_M\}$ de todos los posibles dentro del conjunto $\{1, ..., N\}$ posible.

7.3.1. Otras funciones

El resto de funciones implementadas en la clase *MatrixProductCodes* que nos permiten recuperar información acerca del código construido son:

- *matriz_A*() es una función que nos proporciona la matriz A no singular por columnas que se ha empleado para construir el código.
- *lista_codigos*() es una función que nos devuelve el conjunto de códigos empleados para realizar la construcción del código de producto de matrices.
- *dimension*() es una función que nos indica la dimensión del código construido.
- *length*() es una función que nos proporciona la longitud del código construido.
- *generator_matrix*() es una función que nos devuelve la matriz generatriz del código de producto de matrices.
- *base_field*() es una función que nos indica el cuerpo sobre el que se ha construido el código.
- *is_MPC*() es una función que nos permite diferenciar un código de producto de matrices de cualquier otro.

Vamos a realizar un ejemplo para aplicar las distintas funciones que hemos visto.

Ejemplo 7.1. Para realizar la construcción del código de producto de matrices de este ejemplo, emplearemos dos códigos Reed-Solomon generalizados sobre el cuerpo $\mathbb{F}_{13}$, denotados C_2 y C_3, ambos de longitud $n_2 = n_3 = 7$ y dimensiones $k_2 = 4$ y $k_3 = 2$. Como tienen que ser códigos encajados, generamos ambos mediante los mismos vectores a y b que caracterizan a dichos códigos, siendo $a = (1, 2, 8, 9, 10, 11, 12)$ y $b = (1, 3, 1, 10, 12, 7, 5)$. Mostramos también la matriz generatriz de los códigos construidos:

```
In: F = GF(13)
    a = [F.list()[i] for i in range(1,3)+range(8,13)]
    b = [F.list()[i] for i in [1,3,1,10,12,7,5]]
    C3 = codes.GeneralizedReedSolomonCode(a, 2, b)
    C3.generator_matrix()

Out: [ 1  3  1 10 12  7  5]
     [ 1  6  8 12  3 12  8]

In: C2 = codes.GeneralizedReedSolomonCode(a, 4, b)
    C2.generator_matrix()

Out: [ 1  3  1 10 12  7  5]
     [ 1  6  8 12  3 12  8]
     [ 1 12 12  4  4  2  5]
     [ 1 11  5 10  1  9  8]
```

Ahora definimos la matriz A no singular por columnas que nos falta para poder construir el código de producto de matrices, la cual no consideramos cuadrada para que cuando apliquemos la función de descodificación implementada, *decodificar_MPC*(), ejecutemos todos los pasos realizados en la construcción del algoritmo:

```
In: A=matrix(GF(13),[[1,1,1],[0,2,1]]);
    A

Out: [1 1 1]
     [0 2 1]
```

Entonces a partir de la matriz A y los códigos de Reed-Solomon generalizados, C_2 y C_3 construimos el código de producto de matrices y mostramos su matriz generatriz empleando la función *generator_matrix*():

```
In: MPCGRS=MatrixProductCodes(A,[C2,C3])
    generatriz=MPCGRS.generator_matrix()
    generatriz
```

```
Out: [ 1  3  1 10 12  7  5  1  3  1 10 12  7  5  1  3  1 10 12  7  5]
     [ 1  6  8 12  3 12  8  1  6  8 12  3 12  8  1  6  8 12  3 12  8]
     [ 1 12 12  4  4  2  5  1 12 12  4  4  2  5  1 12 12  4  4  2  5]
     [ 1 11  5 10  1  9  8  1 11  5 10  1  9  8  1 11  5 10  1  9  8]
     [ 0  0  0  0  0  0  0  2  6  2  7 11  1 10  1  3  1 10 12  7  5]
     [ 0  0  0  0  0  0  0  2 12  3 11  6 11  3  1  6  8 12  3 12  8]
```

Podemos observar a partir de la matriz generatriz, que la dimensión del código de producto de matrices es $k = k_2 + k_3 = 6$ y su longitud es $n = 3 * 7 = 21$. Dichos parámetros fundamentales también los podríamos obtener mediante las funciones *dimension*() y *length*() vistas en la sección 7.3.1

Ahora vamos a realizar la codificación, c, de un vector, m, de $\mathbb{F}_{13}$ cuya longitud es 6. Para ello, emplearemos la función *encode*() vista en la sección 7.2:

```
In: m=vector(GF(13),[0,1,2,3,4,5]);
    m

Out: (0, 1, 2, 3, 4, 5)

In: c=MPCGRS.encode(m);
    c

Out: (6, 11, 8, 11, 1, 4, 3, 11, 4, 5, 3, 10, 11, 6, 2, 1, 0, 7, 12, 1, 11)
```

Para realizar la descodificación del vector c, primero vamos a introducir t errores, siendo t la capacidad correctora del código. Por tanto, primero llamamos a la función *minimum_distance*() para poder calcular el valor máximo que puede tomar t:

```
In: MPCGRS.minimum_distance()

Out: 12
```

Como la mínima distancia es 12 entonces tenemos que t puede valer como mucho 5, luego introducimos 5 errores al vector c mediante la función implementada obteniendo el vector y:

```
In: y=MPCGRS.introducir_errores(c,5)
    y

Out: (6, 11, 7, 11, 1, 4, 3, 2, 4, 5, 3, 10, 1, 11, 2, 1, 0, 9, 12, 1, 11)
```

Por último, aplicamos el algoritmo de descodificación sobre el vector y para corregir únicamente los errores:

```
In: palabra_codigo=MPCGRS.decodificar_MPC(y,False);
    palabra_codigo

Out: (6, 11, 8, 11, 1, 4, 3, 11, 4, 5, 3, 10, 11, 6, 2, 1, 0, 7, 12, 1, 11)
```

Y también para corregir y descodificar el vector y:

```
In: descifrado=MPCGRS.decodificar_MPC(y,True);
    descifrado

Out: (0, 1, 2, 3, 4, 5)
```

7.4. Nuevo criptosistema de McEliece

En esta sección vamos a construir el criptosistema de McEliece visto en el capítulo 5 empleando en lugar de los códigos de Goppa binarios, los códigos de producto de matrices gracias al algoritmo de descodificación de dichos códigos visto en la sección 7.3.

De todas formas, usaremos códigos de Goppa para la construcción del código de producto de matrices que empleamos para generar el criptosistema de McEliece, ya que sabemos que los códigos de Goppa no han sido criptoanalizados exitosamente en la actualidad.

Sin embargo, podemos observar que si empleamos los códigos de producto de matrices utilizando la matriz generatriz obtenida a partir de la forma vista en la nota 7.2, seguimos teniendo la principal desventaja del criptosistema de McEliece que tenemos independientemente del código que empleemos para su construcción, correspondiente al tamaño de las claves pública y privada. Sin embargo, vamos a presentar a continuación una nueva versión del criptosistema de McEliece, la cual permite reducir el tamaño de las claves pública y privada cuando empleamos para la construcción del criptosistema de McEliece un código de producto de matrices.

Hemos visto que la matriz generatriz del código de producto de matrices se generaba a partir de las matrices generatrices de los subcodigos empleados y la matriz A. A partir de esto, mostramos como es posible generar la clave pública a partir de dicha matriz A y las matrices generatrices enmascaradas de los subcodigos.

Para ocultar dichas matrices generatrices, emplearemos para cada matriz generatriz, una matriz invertible aleatoria S' y una matriz de permutación π. Dicha matriz de permutación coincide para todas las matrices generatrices, lo cual es posible al tener todos los subcodigos empleados en el código de producto de matrices la misma longitud. Sin embargo, las matrices invertibles S' son distintas incluso en el tamaño, ya que

son matrices cuadradas cuyo tamaño debe coincidir con la dimensión del código cuya matriz generatriz queremos ocultar.

Vamos a introducir ahora la forma para obtener las claves, así como el proceso a seguir para realizar el cifrado y descifrado.

7.4.1. Generacion de claves

Para la generación de las claves, vamos a emplear un código producto de matrices $[C_1 \cdots C_M] \cdot A$ formado por una matriz A no singular por columnas cuadrada de tamaño M y M códigos de Goppa binarios encajados, todos ellos de longitud n, de los cuales conocemos un algoritmo de descodificación que hemos visto en la sección 4.3.1. Los códigos de Goppa encajados, ademas de haber una gran cantidad de ellos, son fáciles de conseguir, ya que para obtenerlos, según la nota 4.3, solo hay que estudiar la divisibilidad entre los polinomios de Goppa de los distintos códigos.

Ahora obtenemos una matriz generatriz, G_i, de cada uno de los M códigos de Goppa. Generamos una matriz de permutación aleatoria cuadrada de tamaño n, π, la cual emplearemos para permutar las n columnas de todas las matrices generatrices.

También generamos para cada matriz generatriz G_i, una matriz invertible aleatoria cuadrada de tamaño k_i, S_i, donde k_i hace referencia a la dimensión del código de Goppa C_i.

A partir de las matrices G_i, S_i y π obtenemos la clave pública, que constará de la matriz A, las M matrices obtenidas al enmascarar las matrices G_i, es decir, $G_i' = S_i \cdot G_i \cdot \pi$, con $1 \leq i \leq M$ y la capacidad correctora del código de producto de matrices C, t.

En cuanto a la clave privada, estará formada por la matriz de permutación π, las M matrices generatrices, las M matrices invertibles S_i y el algoritmo de descodificación del código de producto de matrices, ϑ, por tanto, consta de los algoritmos de descodificación de los M subcodigos.

Todo el proceso mostrado para la generación de las claves pública y privada con tamaño reducido lo hemos implementado en *Sage* mediante la función *McEliece_publica_privada_reducida*$(self)$ dentro de la clase *MatrixProductCodes*:

```
In: def McEliece_publica_privada_reducida(self):
       n=self._lista_codigos[0].length();
       columnas= range(n);
       P_bloques= matrix(self._lista_codigos[0].base_field()
                                                .base_ring(),n);
```

```
for i in range(n):
    posicion=randint(0,len(columnas)-1);
    P_bloques[i,columnas.pop(posicion)]=1;
S_bloques=[];
G_pub=[];
for i in range(len(self._lista_codigos)):
    k=self._lista_codigos[i].dimension();
    S1=random_matrix(self._lista_codigos[i].base_field()
                                          .base_ring(),k);
    while rank(S1)<k:
        S1=random_matrix(self._lista_codigos[i].base_field()
                                          .base_ring(),k);
    S_bloques.append(S1);
    G_pub.append(S1*self._lista_codigos[i].generator_matrix()
                 *P_bloques.change_ring(self._lista_codigos[i]
                                        .base_field()));
return [S_bloques,P_bloques,self._A,G_pub];
```

Podemos observar que dicha función nos proporciona todas las matrices invertibles, S_i, generadas aleatoriamente, con $1 \leq i \leq M$, la matriz de permutación, π, también construida de forma aleatoria, la matriz A empleada para la construcción del código, así como el conjunto de M matrices generatrices enmascaradas.

En resumen, tenemos que las claves pública y privada, denotadas respectivamente por κ_p y κ_s están constituidas por:

$$\kappa_p = (A, G'_1, , \cdots, G'_M, t) \text{ y } \kappa_s = (G_1, \cdots, G_M, \pi, S_1, \cdots, S_M, \vartheta)$$

Por tanto, podemos observar que el tamaño de dichas claves es inferior en cuanto trabajemos con códigos con parámetros elevados al tener que almacenar matrices de tamaño inferior respecto a la matriz generatriz enmascarada, G', del código de producto de matrices y las matrices S y P que es necesario emplear en el criptosistema original de McEliece.

7.4.2. Cifrado

Para realizar el cifrado de un mensaje empleando dicho criptosistema tenemos que emplear las distintas matrices que nos proporcionan en la clave pública para generar una matriz que se corresponde con la matriz generatriz enmascarada G'.

Para ello simplemente construimos la matriz de igual forma que se realizaba la construcción de la matriz generatriz de los códigos producto de matrices visto en la nota 7.2

$$\begin{pmatrix} G'_1 & 0 & \cdots & 0 \\ 0 & G'_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & G'_M \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M1} & a_{M2} & \cdots & a_{MM} \end{pmatrix} =$$

$$\begin{pmatrix} G_1 a_{11} & G_1 a_{12} & \cdots & G_1 a_{1M} \\ \vdots & \vdots & \ddots & \vdots \\ G_M a_{M1} & G_M a_{M2} & \cdots & G_M a_{MM} \end{pmatrix} = G'$$

donde $A = (a_{ij})_{1 \leq i,j \leq M}$ y hemos empleado un abuso de la notación al realizar una multiplicación de un elemento perteneciente a $\mathbb{F}_2$, a_{ij}, por una matriz, G_i.

Entonces, una vez obtenida la matriz G' simplemente multiplicamos el mensaje por dicha matriz para obtener su codificación, es decir, $c = m \cdot G'$.

Ahora a partir de la capacidad correctora de errores del código producto de matrices empleado, t, introducimos un vector de errores e de igual longitud que el vector c y con peso de Hamming inferior o igual a t. Por tanto obtenemos el mensaje cifrado tras sumar dicho vector de errores con el mensaje codificado, $y = c + e$.

7.4.3. Descifrado

Para realizar el descifrado del vector y, empleamos las matrices de la clave privada para generar una matriz de permutación cuadrada, Π, de tamaño $n * M$ y una matriz invertible S, las cuales se construyen como se muestra a continuación:

$$\Pi = \begin{pmatrix} \pi & 0 & \cdots & 0 \\ 0 & \pi & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \pi \end{pmatrix}, \quad S = \begin{pmatrix} S_1 & 0 & \cdots & 0 \\ 0 & S_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & S_M \end{pmatrix},$$

donde π es la matriz de permutación cuadrada de tamaño n y S_i es la matriz invertible cuadrada de tamaño k_i.

Para la construcción de estas matrices Π y S, así como para la construcción de la matriz generatriz pública total, G', que mostramos en la sección 7.4.2, también hemos implementado en *Sage* una función llamada *McEliece_publica_privada*$(self, S\_bloques, P\_bloques)$ dentro de la clase *MatrixProductCodes*, la cual construye las matrices a partir de la introducción de las M matrices invertibles S_i, con $1 \leq i \leq M$,y la matriz de permutación π:

```
In: def McEliece_publica_privada(self,S_bloques,P_bloques):
        S=matrix(S_bloques[0].base_ring(),self._k);
        P=matrix(P_bloques.base_ring(),self._n);
        B=identity_matrix(self._A.nrows());
        for i in range(self._k):
            principio=0; fin=0;
            for h in range(len(S_bloques)):
                if h==0:
                    fin += S_bloques[h].nrows();
                else:
                    principio = fin;
                    fin += S_bloques[h].nrows();
                for j in range(fin-principio):
                    if i<fin and i>=principio:
                        S[i,j+principio] = S_bloques[h][i-principio,j];
        h=-1;
        n=P_bloques.nrows();
        for i in range(B.ncols()):
            for k in range(n):
                h+=1;
                for j in range(B.ncols()*n):
                    P[h,j]=B[i,floor(j/n)]*P_bloques[k,j%n];
        return [S,P,S*self._G*P];
```

A partir de estas matrices podemos realizar ya la descodificación de forma semejante a la versión original presentada por McEliece, visto en la sección 5.4.1, es decir, multiplicando el mensaje cifrado por la matriz de permutación inversa, Π^{-1}, obteniendo que $y \cdot \Pi^{-1} = m \cdot S \cdot G + e \cdot \Pi^{-1}$.

Como $e \cdot \Pi^{-1}$ sigue siendo un vector de peso de Hamming inferior o igual a t y además tenemos impuestas todas las condiciones necesarias para poder emplear el algoritmo de descodificación de los códigos de producto de matrices visto en la sección 7.3, aplicamos dicho algoritmo de descodificación y obtenemos $m \cdot S \cdot G$. Ahora realizamos la descodificación obteniendo el vector $m \cdot S$ y aplicando la matriz inversa de S sobre dicho vector recuperamos el mensaje.

De esta forma, hemos mostrado una versión del criptosistema de McEliece que reduce notablemente el tamaño de las claves pública y privada, manteniendo la seguridad de dicho criptosistema. Por tanto, se elimina de esta forma la principal desventaja que presentaba dicho criptosistema.

En cuanto a los códigos empleados para la construcción del código de producto de

matrices, se expone en esta versión el empleo de códigos de Goppa binarios debido a las buenas propiedades que hemos visto que presentan. Sin embargo, es posible emplear cualquier otro tipo de códigos con el nuevo criptosistema presentado siempre que podamos garantizar, en cierta medida, que la seguridad no resulta comprometida al emplearlos.

Por último, se recomienda escoger los códigos de Goppa de forma que se maximice la distancia del código producto de matrices vista en la proposición 7.9, y que los polinomios que generan dichos códigos de Goppa sean separables, para que de esa forma, la capacidad correctora de dicho código sea lo mas grande posible por lo visto en la proposición 4.6.

Ejemplo 7.2. Vamos a mostrar un ejemplo para el cual generamos las claves pública y privada de tamaño reducido y las comparamos con el mismo sin realizar el proceso de reducción de claves. Para ello, emplearemos la misma matriz A y los dos códigos de Goppa construidos en [1, Ejemplo 9.1] del trabajo fin de grado de matemáticas. Por tanto, empleamos la matriz A siguiente:

```
In: A=matrix(GF(2),[[1,1],[0,1]]);
    A

Out: [1 1]
     [0 1]
```

Ahora construimos el código de Goppa C_1 generado a partir del vector L formado por todos los elementos del cuerpo $\mathbb{F}_{2^5}$ y el polinomio irreducible $g_1(x) = x^3 + (\alpha^4 + \alpha^2 + 1)x + \alpha^3 + \alpha^2 + \alpha + 1$, siendo α un elemento primitivo de $\mathbb{F}_{2^5}$:

```
In: F.<a>=GF(2**5);
    a=F.gen();
    PolRing.<x>=PolynomialRing(F)
    x=PolRing.gen()
    g1=PolRing(x^3 + (a^4 + a^2 + 1)*x + a^3 + a^2 + a + 1) ;
    C1=Goppa(2**5,5,g1);
```

De igual forma, construimos el código de Goppa C_2 a partir del mismo vector L y de un polinomio separable, g_2, que verifique que $g_1|g_2$, para obtener de esta forma que C_2 sea un subcodigo de C_1. Por tanto tomamos $g_2(x) = x^6 + (\alpha^3 + \alpha^2)x^4 + (\alpha^2 + \alpha)x^3 + (\alpha^4 + \alpha^3 + \alpha + 1)x^2 + (\alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1)x + \alpha^4 + \alpha^3$:

```
In: F.<a>=GF(2**5);
    g2=x^6 + (a^3 + a^2)*x^4 + (a^2 + a)*x^3 + (a^4 + a^3 + a + 1)*x^2
                          + (a^4 + a^3 + a^2 + a + 1)*x + a^4 + a^3 ;
```

```
In: C2=Goppa(2**5,5,g2)
```

Obtenemos las matrices generatrices tanto del código C_1 como del código C_2:

```
In: C1.generator_matrix()
```

```
Out: [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 1 0 0 0 1 1 1 1]
     [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 0 0 1 0 1 1 1]
     [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 0 1 1 0 1 1 0 0 0 1]
     [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 0 0 1 0 1 1 1 1 0]
     [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 1 1 0 1 1 0 0]
     [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 1 1 0 1 0 1]
     [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 1 1 0 0 1 1 0 1 1 0]
     [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 1 1 1 1 0 1]
     [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 1 0 1 1 0 0 0 0 0 1 1 1 1]
     [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 0 1 1 1 1 1 1 0 0]
     [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 1 1 1 0 1 1 1 1]
     [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 1 0]
     [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 1 1 1 0 1 1 0]
     [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 1 1 0 1 0 1 0 0 1 0 1]
     [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 1 0 1 0 1 0 0 1 0]
     [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 1 1 0 1 0 1 1 1 0 1 0]
     [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 0 1 1 0 1 1 0 1 0 0]
```

```
In: C2.generator_matrix()
```

```
Out: [1 0 1 0 0 1 0 1 1 0 0 0 1 1 1 1 0 0 1 0 0 1 1 0 0 1 0 0 0 0 1 0]
     [0 1 1 0 0 0 1 1 1 0 1 0 1 0 0 0 0 1 0 1 1 1 1 1 1 1 1 1 0 1 1 1]
     [0 0 0 1 1 0 1 1 1 0 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 1 1 1 0 1]
```

A partir de estas matrices, obtenemos que los parámetros fundamentales de C_1 y C_2 son respectivamente $[32, 17, 7]$ y $[32, 3, 13]$.

Con los códigos C_1 y C_2, junto con la matriz A construimos el código de producto de matrices y obtenemos su matriz generatriz, pero debido al gran tamaño de esta, *Sage* solo nos indica sus dimensiones y el cuerpo al que pertenecen sus elementos:

```
In: codigo_MPC=MatrixProductCodes(A,[C1,C2]);
```

```
In: codigo_MPC.generator_matrix()
```

```
Out: 20 x 64 dense matrix over Finite Field of size 2
```

Con esto, obtenemos que los parámetros fundamentales de *codigo_MPC* son $[64, 20, 13]$, lo cual coincide con lo visto a lo largo de la teoría. Si queremos que *Sage* nos muestre la matriz generatriz habría que incorporar, como ya hemos comentado anteriormente, la función *str*() después de la función *generator_matrix*().

Ahora llamamos a la función *McEliece_publica_privada_reducida*() y almacenamos en la variable *claves_reducidas* el conjunto de valores que nos proporciona la función. Por tanto, en esta variable tenemos en *claves_reducidas*[0] las matrices invertibles cuadradas aleatorias de tamaños 17 y 3 respectivamente:

```
In: claves_reducidas=codigo_MPC.McEliece_publica_privada_reducida();
    claves_reducidas[0]
```

```
Out: [
     [0 1 0 0 1 0 0 0 0 1 0 1 1 1 1 0 1]
     [0 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1]
     [0 1 1 0 0 0 1 1 0 0 1 1 0 0 0 0 0]
     [0 1 0 1 0 1 0 0 0 0 0 1 0 0 0 0 1]
     [1 1 0 0 1 1 0 0 1 1 1 0 0 1 0 1 1]
     [0 1 1 1 1 1 0 0 0 0 0 0 1 1 1 0 0]
     [0 1 0 1 0 1 0 0 1 1 1 1 0 0 0 1 1]
     [0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1]
     [1 1 1 0 1 0 0 1 1 1 0 1 0 0 0 1 1]
     [1 0 0 0 1 0 0 1 1 0 0 0 0 1 1 1 0]
     [1 1 0 0 0 1 1 0 1 1 1 1 0 0 0 1 0]
     [0 0 1 1 1 1 1 1 1 0 0 1 0 1 0 1 1]
     [1 1 0 1 0 0 0 1 0 0 1 1 1 1 0 0 1]
     [0 0 0 1 1 0 0 0 1 1 0 1 0 1 1 0 1]
     [1 1 1 0 1 0 0 1 0 1 0 0 1 1 0 1 1]  [1 1 1]
     [0 1 1 0 0 1 0 1 0 1 0 1 0 0 1 0 0]  [1 1 0]
     [0 1 1 0 0 0 1 1 0 1 0 1 1 1 1 0 1], [1 0 1]
     ]
```

En *claves_reducidas*[1] la matriz de permutación cuadrada aleatoria de tamaño 32:

```
In: claves_reducidas[1]
```

```
Out: 32 x 32 dense matrix over Finite Field of size 2
```

En *claves_reducidas*[2] la matriz A y en *claves_reducidas*[3] las dos matrices generatrices enmascaradas de tamaños 17×32 y 3×32 respectivamente:

```
In: claves_reducidas[3]
```

```
Out: [
     [1 1 0 0 0 1 1 0 0 1 0 0 0 0 1 1 0 1 0 0 0 1 1 0 1 0 0 0 1 0 0 1]
     [0 1 1 0 0 0 0 0 1 1 0 0 1 1 1 0 1 1 0 1 1 0 1 0 1 0 1 0 1 1 1 0]
     [0 0 0 1 0 1 0 0 0 0 0 0 1 0 1 0 0 1 0 1 1 0 1 1 0 1 1 1 1 0 1 1]
     [0 0 1 0 1 1 0 0 1 1 0 0 0 1 0 0 0 1 1 0 1 0 1 1 1 0 1 0 1 1 0 1]
     [1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 0 1 1 0 1 0 0 0 0 1 0 0 0 1 0 0 1]
     [1 0 0 0 1 1 1 0 0 0 0 0 1 0 1 1 0 1 0 0 1 1 0 1 1 0 0 0 0 1 0 1]
     [0 1 0 0 1 0 0 0 0 1 1 0 0 1 0 0 1 1 1 1 1 0 1 1 0 0 0 0 0 1 1 1]
     [0 0 1 1 1 0 0 0 0 1 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0 0 1]
     [0 1 0 1 0 0 1 1 1 1 1 1 1 1 1 0 1 1 0 0 0 0 1 1 0 1 1 0 0 0 0 0]
     [1 0 0 0 0 0 1 1 0 0 1 0 0 1 1 1 1 0 1 0 0 0 0 1 1 1 0 0 1 0 1 0]
     [0 1 1 0 1 1 0 1 1 0 1 1 0 0 0 0 1 1 1 1 0 0 1 0 1 0 0 1 0 0 1 1]
     [1 0 0 0 1 1 1 0 1 1 1 1 1 1 1 0 1 0 1 0 0 0 1 1 1 1 0 1 1 1 1 0]
     [1 0 1 0 0 0 0 1 0 1 0 1 0 0 1 0 0 1 1 1 0 1 1 0 1 1 0 0 1 1 1 1]
     [1 1 0 1 0 0 1 0 0 1 0 0 0 0 0 1 1 0 0 0 1 0 1 0 1 0 0 0 1 1 0 0]
     [1 1 0 0 0 0 1 1 0 1 1 1 1 0 0 0 0 1 0 0 1 1 0 0 1 1 1 0 0 0 0 1]
     [0 1 1 0 1 1 0 0 1 0 0 0 1 1 1 1 0 1 0 0 1 0 1 1 1 1 1 0 1 0 1 0]
     [1 1 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0 1 0 0 0 1 1 1 0 1 0 1 0 0 1 1],

     [1 0 1 1 1 0 1 1 0 0 1 0 0 1 1 1 1 1 1 0 1 1 1 0 0 1 1 0 0 1 0 0]
     [1 0 1 1 1 0 0 1 0 1 1 1 0 0 0 1 0 1 1 1 1 0 0 1 0 0 1 1 1 0 1 0]
     [1 0 0 0 1 1 1 1 1 1 1 1 1 0 1 1 0 0 0 1 1 0 1 1 0 0 0 1 1 1 1 1]
     ]
```

Por último, empleando los valores de *claves_reducidas*[0] y *claves_reducidas*[1] obtenemos las matrices S, P y G' en ese orden en las posiciones 0, 1 y 2 de la variable *claves* mediante la ejecución de la función *McEliece_publica_privada*():

```
In: claves=codigo_MPC.McEliece_publica_privada(claves_reducidas[0],
                                               claves_reducidas[1])

In: claves[0]

Out: 20 x 20 dense matrix over Finite Field of size 2

In: claves[1]

Out: 64 x 64 dense matrix over Finite Field of size 2

In: claves[2]

Out: 20 x 64 dense matrix over Finite Field of size 2
```

Por tanto, podemos reducir el tamaño de la clave que tenemos que almacenar ya que de tener que guardar una matriz de tamaño 20×64 en este caso, tenemos que almacenar una matriz de tamaño 17×32 y otra de tamaño 3×32 correspondientes a cada uno de las matrices generatrices de los códigos de Goppa enmascarados, junto con la matriz A de tamaño 2×2 para este ejemplo.

Entonces, de la primera forma tendríamos que almacenar un total de 1280 bits o 160 bytes mientras que en la segunda forma tenemos que almacenar un total de $544 + 96 + 4 = 644$ bits o $[80{,}5] = 81$ bytes lo que reduce prácticamente a la mitad el tamaño de la clave pública que se debe transmitir y almacenar, además vemos que esto sucede en el peor de los casos, es decir, como mínimo se reducirá a la mitad el tamaño de la clave, ya que si tomamos una matriz A que admita una mayor cantidad de códigos obtendremos una clave pública total de gran tamaño mientras que cada matriz generatriz enmascarada de cada código que compone el MPC resultará muy pequeña en comparación a la matriz total.

En cuanto a la clave privada, tenemos que almacenar las matrices G, S y P junto con el algoritmo de descodificación del código, el cual ocupa el mismo espacio en ambas formas. En cuanto a la matriz G el tamaño se reduce de igual forma que en la clave pública al corresponderse con matrices de iguales tamaños, es decir, 1280 bits de la primera forma y 644 bits de la segunda. Además, en cuanto a las matrices S y P, su tamaño de la primera forma es de 400 bits y 4096 bits respectivamente. Sin embargo, a partir de la reducción de las claves es necesario solamente almacenar 289 bits de S_1 y 9 bits de S_2 obteniendo un tamaño total de 298 bits a almacenar para las matrices invertibles. En cuanto a la matriz de permutación reducida son necesarios almacenar 1024 bits, donde podemos observar que también se produce una reducción drástica de los tamaños a almacenar.

Capítulo 8

Pruebas del código implementado

En este capítulo vamos a realizar un análisis del conjunto de funciones implementadas en *Sage* que presentan una gran relevancia y complejidad, donde para cada una de ellas comprobaremos que su ejecución es la esperada mediante la ejecución de distintos ejemplos, midiendo el tiempo de ejecución que emplean.

Construiremos 3 ejemplos para cada tipo de código, donde intentaremos que los parámetros fundamentales de estos códigos sean $n = 4096, 2048, 1024$, $k = 2660, 1750, 524$ en los distintos tipos. Se han escogido estos tamaños debido a que eran aproximadamente los aconsejados para construir el criptosistema de McEliece en distintos periodos de tiempo que vimos en la sección 5.6.

En cuanto a los dos ataques de filtración contra los códigos Reed-Solomon generalizados, únicamente realizamos una medición del tiempo de ejecución que emplean las funciones implementadas para realizar el ataque contra un criptosistema de McEliece construido con un código Reed-Solomon generalizado, ya que se trata del ataque que tendremos que realizar siempre en una situación real. Además, reducimos los parámetros fundamentales del código Reed-Solomon generalizado empleado a $n = 256, 128, 64$ ya que su complejidad es mas elevada que todas las funciones medidas anteriormente, por tanto, el tiempo que necesitan emplear es bastante mas elevado.

Además de los ejemplos que mostramos en este capítulo y los mostrados a lo largo de los capítulos anteriores, en los archivos con formato "*.ipynb*" donde se encuentra el código implementado, se encuentran también otros ejemplos adicionales que se han realizado para comprobar el correcto funcionamiento de todas las funciones.

8.1. Goppa

Comenzamos con la clase que permite realizar la construcción de los códigos de Goppa. Para realizar la medición del tiempo de ejecución que emplea cada función, empleamos la función *time*() de *Sage*. Para ello es necesario incorporar la librería *time* que la contiene empleando la sentencia:

$$"from\ time\ import\ time"$$

Ahora vamos a construir los códigos de Goppa empleando los parámetros que ya hemos dicho, para ello vamos a separar el tiempo necesario para calcular el polinomio irreducible necesario para la construcción de cada uno de ellos:

- Para el cálculo del polinomio irreducible de grado 119 sobre el cuerpo $\mathbb{F}_{2^{12}}$ se han necesitado 36,54 segundos.
- Para el cálculo del polinomio irreducible de grado 27 sobre $\mathbb{F}_{2^{11}}$ se han necesitado 0,39.
- Por último, para calcular el polinomio irreducible de grado 50 sobre $\mathbb{F}_{2^{10}}$ se han necesitado 1,09 segundos.

Podemos observar una notable reducción del tiempo empleado para el cálculo de los polinomios irreducibles a medida que disminuimos el grado del polinomio a generar, así como el cuerpo $\mathbb{F}_q$ sobre el que vamos a construir cada código, donde únicamente tenemos un tiempo algo elevado en parámetros cercanos a los necesarios para resistir ataques cuánticos.

Una vez calculados los polinomios irreducibles de los grados deseados vamos a generar los correspondientes códigos de Goppa:

- Para el código de Goppa de parámetros $[4096, 2668, 239]$, que denotamos C_1, se han necesitado 219,24 segundos
- Para el código de Goppa de parámetros $[2048, 1751, 55]$, que denotamos C_2, se han necesitado 46,74 segundos
- Para el código de Goppa de parámetros $[1024, 524, 101]$, que denotamos C_3, se han necesitado 12,14 segundos

Ahora vamos a realizar la codificación y descodificación de un vector formado por elementos de $\mathbb{F}_2$ cuyo tamaño sea justamente la dimensión de los tres tipos de códigos construidos.

En cuanto a los tiempos de codificado, obtenemos unos resultados de 0,07, 0,04 y 0,18 para C_1, C_2 y C_3 respectivamente los cuales son muy pequeños, pero esto es de esperar ya que únicamente tenemos que realizar la multiplicación de un vector por una matriz sobre le cuerpo $\mathbb{F}_2$.

Ahora, realizamos en primer lugar la descodificación sin introducir errores:

- Para el código de Goppa C_1, se han necesitado 50,95 segundos
- Para el código de Goppa C_2, se han necesitado 21,19 segundos
- Para el código de Goppa C_3, se han necesitado 2,12 segundos

Por último, introducimos la máxima cantidad de errores posible sobre cada mensaje codificado, es decir, 119, 27 y 50 respectivamente y ejecutamos la función de descodificación:

- Para el código de Goppa C_1, se han necesitado 56,83 segundos
- Para el código de Goppa C_2, se han necesitado 21,53 segundos
- Para el código de Goppa C_3, se han necesitado 2,68 segundos

Obviamente los tiempos de descodificación al introducir errores son mayores, ya que es necesario corregirlos. Además, realizando la resta de los tiempos obtenidos para ambos casos, podemos obtener una referencia del tiempo empleado aproximadamente por el algoritmo de Patterson en cada descodificación realizada. Dichos tiempos son 5,88, 0,34 y 0,56 segundos para los códigos C_1, C_2 y C_3 respectivamente. Vemos que el tiempo de aplicación del algoritmo de Patterson es muy pequeño para los códigos C_2 y C_3 en comparación con el de C_1, debido al incremento del cuerpo finito empleado.

8.2. Códigos de producto de matrices

Para la construcción de estos códigos mediante el empleo de la clase *MatrixProductCodes* vamos a realizar la construcción de seis de ellos empleando parámetros parecidos a los de Goppa, y realizando su construcción mediante el empleo de códigos de Goppa y también Reed-Solomon generalizados.

Para construir los códigos de producto de matrices con Goppa separamos al igual que antes la ejecución necesaria para obtener los polinomios irreducibles que empleamos para construir los códigos de Goppa:

- Para el cálculo de los polinomios irreducibles de grado 40 y 70 sobre el cuerpo $\mathbb{F}_{2^{11}}$ se han necesitado 25,79 segundos.

- Para el cálculo de los polinomios irreducibles de grado 10 y 25 sobre el cuerpo $\mathbb{F}_{2^{10}}$ se han necesitado 0,81 segundos.
- Por último, para calcular los polinomios irreducibles de grado 15 y 35 sobre el cuerpo $\mathbb{F}_{2^9}$ se han necesitado 3,96 segundos.

Ahora construimos los dos códigos de Goppa necesarios para construir cada código de producto de matrices:

- Para los códigos de Goppa necesarios para la construcción del código de producto de matrices $[4096, 2446, 162]$, que denotamos MPC_1, se han necesitado 132,95 segundos.
- Para los códigos de Goppa necesarios para la construcción del código de producto de matrices $[2048, 1598, 42]$, que denotamos MPC_2, se han necesitado 23,49 segundos.
- Para los códigos de Goppa necesarios para la construcción del código de producto de matrices $[1024, 439, 62]$, que denotamos MPC_3, se han necesitado 5,58 segundos.

Por último realizamos la construcción de los códigos de producto de matrices obtenidos:

- Para la construcción del código de producto de matrices MPC_1, se han necesitado 129,5 segundos.
- Para la construcción del código de producto de matrices MPC_2, se han necesitado 25,12 segundos.
- Para la construcción del código de producto de matrices MPC_3, se han necesitado 6,78 segundos.

Por tanto, los tiempos totales para realizar la construcción de MPC_1, MPC_2 y MPC_3 son 288,24, 49,45 y 16,32 segundos respectivamente. Luego podemos observar que para todos los parámetros los tiempos de obtención de los códigos respecto a los códigos de Goppa de la sección 8.1 son semejantes.

Ahora realizamos la construcción de los códigos de producto de matrices empleando los códigos Reed-Solomon generalizados, los cuales denotamos $MPCGRS_i$, con $i = 1, 2, 3$:

- Para la construcción del código de producto de matrices $MPCGRS_1$, se han necesitado 541,30 segundos.

- Para la construcción del código de producto de matrices $MPCGRS_2$, se han necesitado 53,33 segundos.
- Para la construcción del código de producto de matrices $MPCGRS_3$, se han necesitado 26,81 segundos.

Por lo que su construcción es mas lenta que la realizada empleando códigos de Goppa, esto se debe principalmente a que ahora estamos trabajando realmente sobre el cuerpo $\mathbb{F}_{2^m}$ correspondiente en lugar de $\mathbb{F}_2$.

Ahora vamos a realizar la codificación y descodificación de un vector cuyo tamaño es justo la dimensión de cada código para comprobar los tiempos necesarios para ello:

- Para la codificación de los códigos de producto de matrices con Goppa MPC_1, MPC_2 y MPC_3 se han necesitado 0,15, 0,04 y 0,01 segundos respectivamente.
- Para la codificación de los códigos de producto de matrices con Reed-Solomon generalizados $MPCGRS_1$, $MPCGRS_2$ y $MPCGRS_3$ se han necesitado 28,98, 11,55 y 0,88 segundos respectivamente.

Podemos observar que los tiempos de codificación cuando se emplean códigos de producto de matrices con Goppa vuelven a ser extremadamente pequeños, mientras que al realizarlo con los construidos a partir de los códigos Reed-Solomon generalizados si que se incrementa algo el tiempo de ejecución. Esto vuelve a ser normal ya que las multiplicaciones las realizamos en lugar de en $\mathbb{F}_2$ en $\mathbb{F}_{2^{11}}$, $\mathbb{F}_{2^{10}}$ y $\mathbb{F}_{2^9}$.

En cuanto a la descodificación de los mensajes codificados sin introducción de errores tenemos:

- Para la descodificación sin errores de los códigos de producto de matrices con Goppa MPC_1, MPC_2 y MPC_3 se han necesitado 47,34, 17,35 y 1,52 segundos respectivamente.
- Para la descodificación sin errores de los códigos de producto de matrices con Reed-Solomon generalizados $MPCGRS_1$, $MPCGRS_2$ y $MPCGRS_3$ se han necesitado 196,46, 58,9 y 7,21 segundos respectivamente.

Hay que tener en cuenta que *Sage* emplea un algoritmo de GAP limitado al calculo de la mínima distancia para cuerpos inferiores a $2^8 = 256$, por tanto aunque la matriz A empleada para la construcción del código de producto de matrices tenga una mínima distancia que sepamos calcular con facilidad, nos devuelve un error de implementación interno el programa si empleamos la función $minimum\_distance()$ ya

definida. Por ello se ha construido la función de *_minima_distancia*(), la cual no presenta dicha limitación y por tanto es posible realizar la descodificación anterior de los códigos $MPCGRS_1$, $MPCGRS_2$ y $MPCGRS_3$.

Ahora introducimos la máxima cantidad de errores posible para los mensajes codificados con los códigos MPC_i, $i = 1, 2, 3$, es decir, 80, 20 y 30 respectivamente y para los códigos $MPCGRS_i$, $i = 1, 2, 3$, es decir, 298, 24 y 146 respectivamente. Entonces calculamos la descodificación para todos ellos:

- Para la descodificación con errores de los códigos de producto de matrices con Goppa MPC_1, MPC_2 y MPC_3 se han necesitado 50,75, 30,42 y 2,62 segundos respectivamente.
- Para la descodificación con errores de los códigos de producto de matrices con Reed-Solomon generalizados $MPCGRS_1$, $MPCGRS_2$ y $MPCGRS_3$ se han necesitado 238,3, 62,5 y 8,14 segundos respectivamente.

8.3. Criptosistema de McEliece

En cuanto al criptosistema de McEliece, vamos a realizar el mismo estudio que en las secciones anteriores. Por tanto, para los parámetros $n = 4096, 2048, 1024$ construimos el criptosistema de McEliece empleando códigos Reed-Solomon generalizados, Goppa y códigos de producto de matrices con Goppa y con Reed-Solomon generalizados.

Si realizamos un análisis acerca del proceso realizado por el constructor del criptosistema de McEliece, vemos que prácticamente bastaría con validar que podemos obtener la matriz invertible de tamaño k y el código correspondiente que empleamos para construir el criptosistema de McEliece de forma eficiente y en un tiempo razonable, al ser las sentencias que mas tiempo de ejecución necesitan emplear.

Para ello, medimos por separado el tiempo necesario para la construcción del código que vamos a emplear:

- Para la construcción de los códigos de Goppa $\Gamma(L_1, g_1)$, $\Gamma(L_2, g_2)$ y $\Gamma(L_3, g_3)$, correspondientes a los parámetros con longitudes $n = 4096, 2048, 1024$ respectivamente, se han necesitado 308,64, 49,90 y 43,98 segundos respectivamente.
- Para la construcción de los códigos Reed-solomon generalizados GRS_1, GRS_2 y GRS_3, correspondientes a los parámetros con longitudes $n = 4096, 2048, 1024$ respectivamente, se han necesitado 0,52, 0,07 y 0,03 segundos respectivamente.

- Para la construcción de los códigos de producto de matrices mediante códigos Goppa, MPC_1, MPC_2 y MPC_3, correspondientes a los parámetros con longitudes $n = 4096, 2048, 1024$ respectivamente, se han necesitado 208,39, 46,98 y 15,72 segundos respectivamente.

- Para la construcción de los códigos de producto de matrices mediante códigos Reed-Solomon generalizados, $MPCGRS_1$, $MPCGRS_2$ y $MPCGRS_3$, correspondientes a los parámetros con longitudes $n = 4096, 2048, 1024$ respectivamente, se han necesitado 535,29, 56,14 y 27,84 segundos respectivamente.

Ahora en cuanto al tiempo necesario para realizar la construcción del criptosistema tenemos:

- Para la construcción de los criptosistema con códigos de Goppa, McE_1, McE_2 y McE_3, cuyo código es $\Gamma(L_1, g_1)$, $\Gamma(L_2, g_2)$ y $\Gamma(L_3, g_3)$ respectivamente, se han necesitado 0,16, 0,05 y 0,02 segundos respectivamente.

- Para la construcción de los criptosistema con códigos de Reed-Solomon generalizados, $McEGRS_1$, $McEGRS_2$ y $McEGRS_3$, cuyo código es GRS_1, GRS_2 y GRS_3 respectivamente, se han necesitado 69,68, 18,58 y 1,84 segundos respectivamente.

- Para la construcción de los criptosistema con códigos de producto de matrices, $McEMPC_1$, $McEMPC_2$ y $McEMPC_3$, cuyo código es MPC_1, MPC_2 y MPC_3 respectivamente, se han necesitado 0,17, 0,04 y 0,02 segundos respectivamente.

- Para la construcción de los criptosistema con códigos de producto de matrices, $McEMPCGRS_1$, $McEMPCGRS_2$ y $McEMPCGRS_3$, cuyo código es $MPCGRS_1$, $MPCGRS_2$ y $MPCGRS_3$ respectivamente, se han necesitado 8,9, 3,02 y 0,2 segundos respectivamente.

Por tanto, podemos observar que el tiempo de obtención de la matriz invertible es insignificante para el criptosistema de McEliece con Goppa y el McEliece con MPC, suponiendo prácticamente el tiempo de construcción del código también el de creación del criptosistema. Sin embargo, para el criptosistema de McEliece que emplea códigos Reed-Solomon generalizados si necesita emplear cierto tiempo para el cálculo de la matriz invertible. También ocurre para el criptosistema de McEliece que emplea códigos de producto de matrices construidos a partir de Reed-Solomon generalizados. Esto es debido a que para los dos primeros casos, tiene que trabajar sobre un cuerpo mas pequeño ($\mathbb{F}_2$), por lo que es capaz de obtener una matriz invertible mas rápidamente. En la tabla 8.1, podemos ver el tiempo total necesario para la construcción de cada criptosistema.

	$n = 1024$, $k = 524$	$n = 2048$, $k = 1750$	$n = 4096$, $k = 2660$
McE	44	49,95	308,8
McEGRS	1,87	18,59	70,2
McEMPC	15,74	47,02	208,56
McEMPCGRS	27,86	59,16	544,19

Tabla 8.1: Tiempos de construcción del criptosistema en segundos.

Ahora vamos a realizar el proceso de cifrado y descifrado para cada tipo de criptosistema creado, de un mismo mensaje, que es: *" Cifrado y descifrado de un mensaje con el criptosistema de McEliece"*

Para los cuatro códigos empleados para construir el criptosistema, el proceso de cifrado en los distintos parámetros es el siguiente:

- Para el cifrado del mensaje correspondiente a los criptosistemas McE_1, McE_2 y McE_3, se han necesitado 1,9, 1,85 y 1,8 segundos respectivamente.
- Para el cifrado del mensaje correspondiente a los criptosistemas $McEGRS_1$, $McEGRS_2$ y $McEGRS_3$, se han necesitado 1,79, 1,82 y 1,71 segundos respectivamente.
- Para el cifrado del mensaje correspondiente a los criptosistemas $McEMPC_1$, $McEMPC_2$ y $McEMPC_3$, se han necesitado 2,52, 1,88 y 1,82 segundos respectivamente.
- Para el cifrado del mensaje correspondiente a los criptosistemas $McEMPCGRS_1$, $McEMPCGRS_2$ y $McEMPCGRS_3$, se han necesitado 148,66, 36,84 y 6,86 segundos respectivamente.

Ahora realizamos el proceso de descifrado para los distintos cifrados obtenidos de cada criptosistema construido:

- Para el descifrado del mensaje correspondiente a los criptosistemas McE_1, McE_2 y McE_3, se han necesitado 63,07, 23,21 y 5,32 segundos respectivamente.
- Para el descifrado del mensaje correspondiente a los criptosistemas $McEGRS_1$, $McEGRS_2$ y $McEGRS_3$, se han necesitado 205,44, 39,85 y 8,35 segundos respectivamente.
- Para el descifrado del mensaje correspondiente a los criptosistemas $McEMPC_1$, $McEMPC_2$ y $McEMPC_3$, se han necesitado 88,79, 29,25 y 7,32 segundos respectivamente.

- Para el descifrado del mensaje correspondiente a los criptosistemas $McEMPCGRS_1$, $McEMPCGRS_2$ y $McEMPCGRS_3$, se han necesitado 589,87, 130,61 y 22,38 segundos respectivamente.

Podemos ver que para el proceso de descifrado del mensaje cifrado empleando el criptosistema construido a partir de cualquiera de los códigos usados, el tiempo de descifrado se va reduciendo según vamos reduciendo la longitud del parámetro empleado. Además, vemos que los criptosistemas que realizan los procesos de cifrado y descifrado mas rápidamente son los que emplean códigos Goppa y códigos de producto de matrices construidos a partir de códigos Goppa.

8.4. Ataque por filtración

Como comentamos, en esta sección reducimos los parámetros que empleamos para realizar el ataque debido a la limitación de los recursos tecnológicos y de software que disponemos. Por ello, los códigos de Reed-Solomon generalizados sobre los que realizamos el ataque tienen una longitud de $n = 256$, 128, 64 y la dimensión que tomamos para cada código se corresponde con la mitad de la longitud del código, para medir de esta forma el caso mas desfavorable que se podría presentar para dichas longitudes.

Si observamos el código generado para realizar el ataque, es claro que el proceso que mas tiempo de ejecución emplea se corresponde con la función *reducir_codigos*(), la cual se debe emplear en primer lugar $k-2$ veces para obtener el vector c y en segundo lugar $k-3$ veces para obtener el vector c', a partir de los cuales podemos obtener los vectores a y b. Como $k < \frac{n}{2}$ podemos suponer que se debe realizar hasta n veces, es decir, tiene una complejidad de $O(n)$ operaciones. Por otro lado, la complejidad de la función *reducir_codigos*() se corresponde con $O(k^2n^2)$ para obtener la matriz generatriz del código cuadrado. Después, para obtener el sistema es necesario multiplicar k vectores de tamaño n por $n-2k$ vectores de tamaño n correspondientes con las distintas filas de las matrices generatrices de ambos códigos, por tanto se emplean $O(nk(n-2k)$ operaciones, que para resolver el sistema necesitamos multiplicar por k obteniendo un total de $O(k^3n)$ operaciones. Entonces, la complejidad total de la función *reducir_codigos*() es $O(k^2n^2+k^3n)$ y como debemos emplearla hasta un máximo de n veces, obtenemos que la complejidad del ataque es de $O(k^2n^3+k^3n^2)$ operaciones.

Ahora mostramos los tiempos de ejecución que hemos necesitado emplear para realizar el ataque a los códigos ya mencionados:

- Para realizar el ataque al criptosistema de McEliece construido a partir de un código Reed-Solomon generalizado con longitud $n = 256$ y dimensión $k = 128$, se

han necesitado 21694,64 segundos, es decir, 6 horas, 1 minuto y 34,65 segundos.

- Para realizar el ataque al criptosistema de McEliece construido a partir de un código Reed-Solomon generalizado con longitud $n = 128$ y dimensión $k = 64$, se han necesitado 1045,59 segundos, es decir, 17 minutos y 25 segundos.
- Para realizar el ataque al criptosistema de McEliece construido a partir de un código Reed-Solomon generalizado con longitud $n = 64$ y dimensión $k = 32$, se han necesitado 55,27 segundos.

En estos resultados podemos apreciar como se va incrementando rapidamente el tiempo, a medida que vamos incrementando los parámetros n y k del código empleado, debido a la complejidad que hemos visto que presenta el ataque, $O(k^2n^3 + k^3n^2)$.

Por último indicamos que para la medición de todos los tiempos de ejecución realizados a lo largo del capítulo se ha empleado un procesador $intelCorei7 - 4712MQ$ que funciona a una velocidad de hasta 2,30 GHz pero cuyo rendimiento en el cálculo de operaciones de *Sage* esta limitado al 20 % de su capacidad para cada proceso ejecutado. Debido a esto, podemos observar que aunque hayamos obtenido algunos tiempos elevados, si empleáramos otro procesador mas potente que funcionará con el 100 % de rendimiento o incluso emplearemos un conjunto de procesadores que trabajaran interconectados, los resultados se reducirían notablemente, posibilitando por ejemplo la realización del ataque por filtración a criptosistemas con parámetros cuyo tamaño se emplea actualmente.

Capítulo 9

Conclusiones

En este proyecto nos hemos centrado en realizar un estudio tanto teórico como práctico de una serie de códigos, los cuales empleamos después para realizar la construcción del criptosistema de McEliece. En adicción, realizamos un posible ataque contra el criptosistema de McEliece construido a partir de los códigos Reed-Solomon generalizados y por último, realizamos una fase de investigación mediante la cual somos capaces de crear un nuevo método que nos permite reducir el tamaño de las claves del criptosistema de McElice.

Debido al desarrollo del proyecto, se ha despertado un gran interés en el ámbito relacionado con la seguridad de la información de las comunicaciones debido a su gran importancia, ya que actualmente cualquier persona realiza comunicaciones diarias empleando algún dispositivo electrónico, las cuales contienen en ocasiones información muy personal, donde la mayoría desconoce la seguridad de estas comunicaciones.

Con el he tenido la oportunidad de poner en práctica los conocimientos adquiridos a lo largo de la carrera relacionados principalmente con la teoría de grupos, seguridad informática y manejo de distintos lenguajes de programación. También se realizó una pequeña introducción el segundo año de la carrera a la teoría de códigos, aunque los conocimientos adquiridos mediante la asignatura eran mucho mas elementales que los necesarios para el desarrollo del proyecto. En adicción, ha permitido adquirir el aprendizaje necesario tanto del lenguaje de programación de *Sage* como el de Python, que no se habían estudiado a lo largo de la carrera, así como un incremento en el aprendizaje del lenguaje de edición de textos *latex*, lo cual era un objetivo que intentábamos alcanzar una vez finalizado el presente documento.

Si tenemos en cuenta el resto de objetivos prefijados en la sección 1.2 a conseguir tras la realización del proyecto, podemos observar que se han alcanzado prácticamente en su totalidad.

En primer lugar, hemos realizado tanto un estudio teórico como práctico de los códigos lineales, códigos Reed-Solomon y Reed-Solomon generalizados, códigos cíclicos, códigos punteados y recortados a lo largo de todo el capítulo 3.

Después, también se realiza un estudio tanto de los códigos de Goppa como de los códigos de producto de matrices y para ambos se realiza una implementación completa empleando *Sage* que nos permite realizar su construcción, así como las operaciones elementales empleando dichos códigos.

También se estudia detalladamente el criptosistema de McEliece de forma teórica en el capítulo 5, para el cual hacemos relevancia a la elevada seguridad que parece presentar dicho criptosistema ya que, a pesar de ser uno de los mas antiguos sigue siendo resistente la versión original que emplea códigos de Goppa a la gran cantidad de critoanalisis que se le han intentado realizar, principalmente en estos últimos años debido a la gran importancia adquirida por ser un sistema resistente al algoritmo de Shor.

En relación con el criptosistema de McEliece, también se implementa un criptoanalisis cuando se realiza su construcción empleando códigos Reed-Solomon y Reed-Solomon generalizados, a partir de lo cual se intenta mostrar que aunque dicho criptosistema parece ser seguro frente ataques realizados por un ordenador cuántico, debemos tener especial cuidado en realizar su construcción empleando códigos que le permitan ser seguro frente a los distintos ataques conocidos actualmente contra ciertos tipos de códigos. En adicción, no se ha realizado un análisis de seguridad mediante el empleo de otros códigos, frente a los dos tipos de ataques posibles contra el criptosistema que vimos brevemente en la sección 5.7.

Para finalizar, volvemos a destacar el nuevo criptosistema de McEliece desarrollado, que nos permite reducir el tamaño de las claves del criptosistema manteniendo en un principio la seguridad que presentaba mediante el empleo de los códigos de Goppa.

9.1. Trabajo futuro

Para terminar el capítulo, mostramos a continuación una serie de aspectos que serían convenientes realizar mas adelante para completar aún mas el proyecto desarrollado:

- Realizar un estudio profundo acerca de la seguridad del criptosistema de McEliece, así como de los posibles ataques existentes para distintas variantes del criptosistema, que nos permitan adquirir una serie de conocimientos a partir de los cuales podamos identificar cuales son las mayores debilidades del criptosistema.

- Incrementar la implementación desarrollada para la construcción de los códigos de Goppa de forma que no se restrinjan dichos códigos únicamente a los construidos sobre cuerpos binarios. Para ello, solo sería necesario implementar un algoritmo de descodificación eficiente válido para todos ellos, como por ejemplo el algortimo de Berlekamp-Massey o el de Euclides extendido que se pueden analizar en [24] [26] y [23] [25] respectivamente.

- También se podría crear un fichero que contenga una gran cantidad de polinomios irreducibles cuyo grado coincida con los parámetros mas empleados y de elevado tamaño, de forma que a la hora de generar la construcción de un código de Goppa se reduzca su tiempo de obtención, al tener ya calculados previamente los polinomios irreducibles. Obviamente, sería necesario ir actualizando los polinomios irreducibles almacenados en el fichero para no generar siempre construcciones de códigos de Goppa mediante el mismo polinomio. Esto en principio, volvería a suponer el incremento del tiempo para la generación de los códigos, sin embargo podemos observar que dicho proceso se podría realizar en instantes de tiempo en que no se este empleando el procesador, de forma que no suponga un incremento del tiempo de generación de las claves.

- Con respecto a la nueva versión del criptosistema de McEliece presentada, para los códigos de producto de matrices podemos observar que la matriz A y los códigos se deben encontrar en el mismo cuerpo, entonces solo es posible realizar la construcción del nuevo criptosistema actualmente empleando la matriz correspondiente a la construcción de Plotkin. Por tanto, aunque para esta construcción vimos que se reducía notablemente el tamaño de las claves del criptosistema, una vez implementado el algoritmo de descodificación de los códigos de Goppa sobre cualquier cuerpo, habría que analizar la reducción de las claves ocasionada empleando códigos de producto de matrices que empleen una matriz A de mayor tamaño, donde se prevee que dicha reducción será aún mayor que para la construcción de Plotkin, aunque habría que realizar un análisis acerca de si merece la pena incrementar mas la reducción de las claves, ya que al tener que emplear códigos de Goppa no binarios, se pierde la mitad de la capacidad correctora del código como vimos en la proposición 4.6.

- Otro aspecto importante que faltaría por analizar, es si la seguridad del criptosistema de McEliece se ha visto mermada mediante el empleo del nuevo método desarrollado. Para ello, habría que realizar un estudio comparativo acerca de todos los posibles ataques conocidos hasta el momento contra el criptosistema de McEliece, así como los posibles ataques nuevos que podrían surgir al criptosistema debido al empleo de los códigos de producto de matrices.

Bibliografía

[1] DAVID MORENO CENTENO, *Análisis de la teoría de códigos: McEliece y una nueva versión*, Trabajo fin de grado matemáticas 2019.

[2] SHAMIR A, *A polynomial time algoritm for breaking the basic Merckle-Hellman cryptosystems*, IEEE trans on inform 1984.

[3] MERKLE, R. C., *A digital signature based on a conventional encryption function*, Conference on the Theory and Application of Cryptographic Techniques 1987.

[4] NICOLAS COURTOIS, ALEXANDER KLIMOV, JACQUES PATARIN Y ADI SHAMIR, *Efficient algorithms for solving overdefined systems of multivariate polynomial equations*, Advances in cryptology—EUROCRYPT 2000.

[5] JUSTESEN JØRN, HØHOLDT TOM., *A Course in Error-Correcting Codes*, European Mathematical Society Publishing House 2004.

[6] YUAN XING LI, ROBERT H. DENG, AND XIN MEI WANG, *On the equivalence of McElieces and Niederreiters public-key cryptosystems*, IEEE Transactions on Information Theory 1994.

[7] NICOLAS COURTOIS, MATTHIEU FINIASZ, AND NICOLAS SENDRIER, *How to achieve a mceliece-based digital signature scheme*, Advances in Cryptology, ASIACRYPT 2001.

[8] V. M. SIDELNIKOV AND S. O. SHESTAKOV, *On the insecurity of cryptosystems based on generalized Reed-Solomon codes*, Discrete Math. Appl 1992.

[9] FAUGÈRE, J., GAUTHIER-UMANA, V., OTMANI, A., PERRET, L., TILLICH, J., *A distinguisher for high rate McEliece cryptosystems*, Proceedings IEEE Information Theory Workshop 2011.

[10] P. GABORIT, *Shorter keys for code based cryptography*, International Workshop on Coding and Cryptography 2005.

[11] SAN LING, *On the algebraic structure of quasi-cyclic codes .I. Finite fields*, Proceedings IEEE Information Theory Workshop 2001.

[12] T.A. GULLIVER, V.K. BHARGAVA, *A binary quasi-cyclic code*, Appl. Math. Lett. 1995.

[13] KRISTINE LALLY Y PATRICK FITZPATRICK, *Algebraic structure of quasicyclic codes*, Workshop on Coding and Cryptography 1999.

[14] JACQUES STERN, *A method for finding codewords of small weight*, Lecture Notes in Computer Science 1989.

[15] A. OTMANI, J.P. TILLICH, AND L. DALLOT, *Cryptanalysis of two McEliece cryptosystems based on quasi-cyclic codes*, preprint 2008.

[16] T. P. BERGER, P. CAYREL, P GABORIT, AND A. OTMANI, *Reducing Key Length of the McEliece Cryptosystem*, AFRICACRYPT 2009.

[17] J.-C. FAUGÈRE, A. OTMANI, L. PERRET, AND J.-P. TILLICH, *Algebraic cryptanalysis of McEliece variants with compact keys*, Proc. Adv. Cryptol. 2010.

[18] V. G. UMAÑA AND G. LEANDER, *Practical key recovery attacks on two McEliece variants*, Association Cryptol. Res. (IACR) 2009.

[19] J.-C. FAUGÈRE, A. OTMANI, L. PERRET, F. DE PORTZAMPARC, AND J.-P. TILLICH, *Structural cryptanalysis of McEliece schemes with compact keys*, Designs codes Cryptography 2016.

[20] J.-C. FAUGÈRE, A. OTMANI, L. PERRET, F. DE PORTZAMPARC, AND J.-P. TILLICH, *Folding alternant and Goppa Codes with non-trivial automorphism groups*, IEEE Trans. Inf. Theory 2016.

[21] BLACKMORE, T., NORTON, G.H., *Matrix-product codes over* $\mathbb{F}_q$, Appl. Algebra Eng. Commun. Comput 2001.

[22] MACWILLIAMS, F.J., SLOANE, N.J.A., *The Theory of Error-Correcting Codes*, North-Holland Mathematical Library 1977.

[23] J. GATHEN, AND J. GERHARD, *Modern computer algebra*, Cambridge University 2013.

[24] —, *Algebraic Coding Theory*, New York: McGraw-Hill 1968.

[25] SHOUP AND, VICTOR, *A Computational Introduction to Number Theory and Algebra*, Cambridge University 2008.

[26] E. R. BERLEKAMP, *Goppa codes*, IEEE Trans. Inform. Theory 1973.

[27] R. PELLIKAAN, X. WU, S. BULYGIN AND, R. JURRIUS, *Codes, Cryptology and Curves with computer algebra*, Cambridge University 2018.

[28] R. THOMAS, *How sage helps to implement Goppa codes and McEliece PKCSs*, 2011.

[29] D. J. BERNSTEIN, *List decoding for binary Goppa codes*, Third International Workshop 2011.

[30] *Post-Quantum Cryptography | CSRC*, Computer Security division, Information Technology Laboratory, National Institute of Standars and Technology,.

URL: `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography`

[31] *Página web Indeed*

URL: `https://www.indeed.es/salaries/`

[32] A. COUVREUR, P. GABORIT, V. GAUTHIER-UMAÑA, A. OTMANI Y, J-P. TILLICH, *Distinguisher-Based Attacks on Public-Key Cryptosystems Using Reed-Solomon Codes*, 2014.

[33] I. MÁRQUEZ-CORBELLA, N. SENDRIER Y, M. FINIASZ, *Attack against GRS codes*, France université numérique, FUN-MOOC 2013.

URL:`https://www.fun-mooc.fr/courses/course-v1:inria+41006+archiveouvert/about`

[34] A. COUVREUR, A. OTMANI Y, J-P. TILLICH, *POLYNOMIAL TIME ATTACK ON WILD MCELIECE OVER QUADRATIC EXTENSIONS*, IEEE Transactions on Information Theory 2016.

[35] C. MUNUERA Y, J. TENA, *Codificación de la Información*, Universidad de Valladolid 1997.

[36] P. J. LEE Y, E. F. BRICKELL, *An observation on the security of McEliece's public-key cryptosystem*, Advances in cryptography- EUROCRYPT 88 1988.

Apéndice A

Contenido del CD

El CD con formato -R que se encuentra junto con el presente documento contiene como podemos ver en la imagen A.1 la memoria del mismo en formato *.pdf* así como una carpeta llamada *"codigo Sage"* que contiene todo el código que hemos implementado para la realización del proyecto.

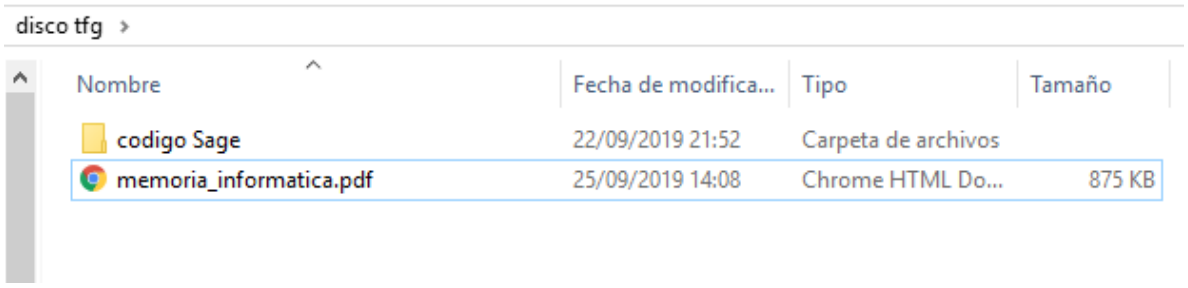

Imagen A.1: Contenido global del CD

Después, si entramos dentro de la carpeta *"codigo Sage"* se encuentran todos los ficheros que hemos empleado en el proyecto.

Por un lado, tenemos los ficheros con formato *.ipynb* que podemos ver en la imagen A.2, los cuales contienen todo el código implementado y comentado, así como un conjunto de pruebas y ejemplos realizados para validar su correcto funcionamiento.

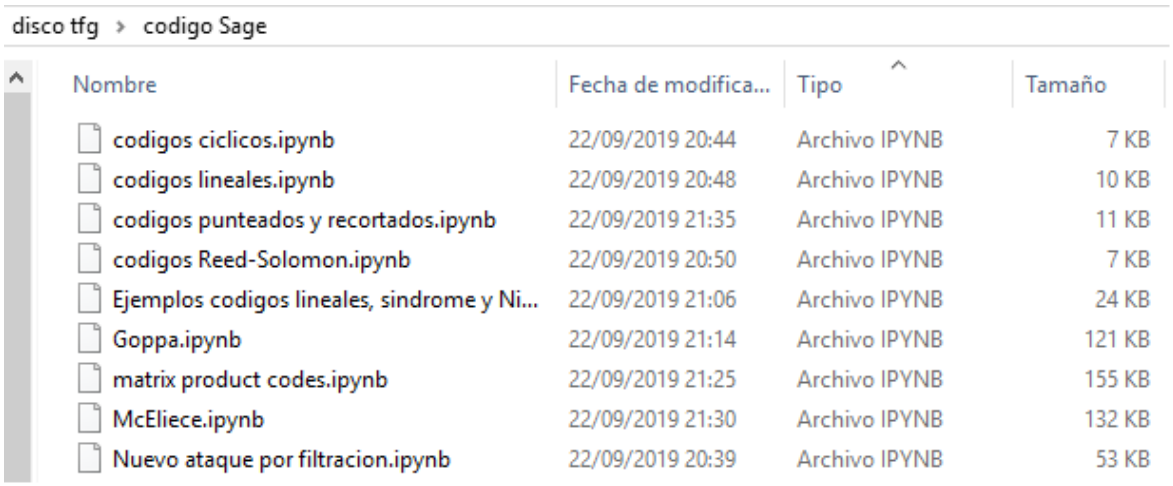

Imagen A.2: Ficheros con formato *.ipynb*

Después, también tenemos una serie de ficheros con formato *.sage* que podemos ver en la imagen A.3. Dichos ficheros contienen únicamente el código implementado, los cuales empleamos cuando queramos realizar la construcción de algún código de Goppa, código de producto de matrices, criptosistema de McEliece o ataque de un criptosistema de McEliece que emplee códigos Reed-Solomon generalizados, de forma que no sea necesario incorporar manualmente dichas funciones cuando se vayan a emplear en otros archivos *.ipynb*. El proceso de incorporación de dichos archivos se muestra en la sección B.2.

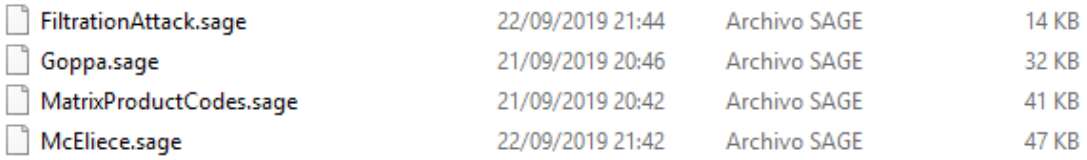

Imagen A.3: Ficheros con formato *.sage*

Por último, tenemos una serie de ficheros con formato *.txt* como podemos ver en la imagen A.4, los cuales podemos emplear para introducir directamente el texto que deseemos cifrar mediante los distintos ejemplos de construcción de criptosistemas de McEliece realizados en el archivo *McEliece.ipynb*, donde el tipo de código que emplean se corresponde con el que hace referencia el nombre de cada fichero. Obviamente se puede generar otro archivo con formato *.txt*, cuyo nombre sea distintos a los presentados, el cual queramos cifrar.

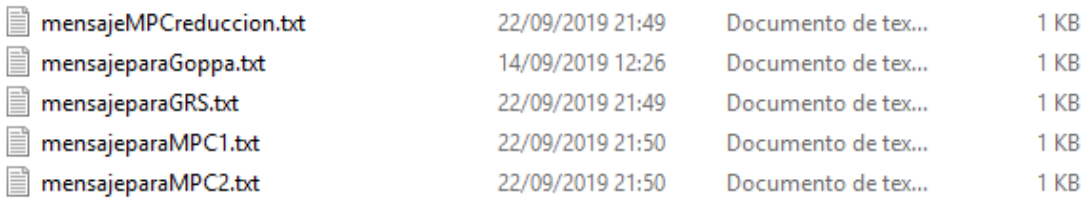

Imagen A.4: Ficheros con formato *.txt*

Apéndice B

Manual de usuario

B.1. Instalación de Sage y acceso a Jupyter

Ahora vamos a mostrar el manual que permite utilizar las distintas clases implementadas en *Sage*, pero antes de ello veamos un breve proceso para explicar como instalar *Sage*, para lo cual es necesario disponer de al menos 4 GB disponibles en el disco duro del ordenador que vamos a emplear.

Para ello simplemente tenemos que acceder a la página web `http://www.sagemath.org/download.html`, donde se muestran distintos servidores para realizar la descarga. Es aconsejable emplear el servidor RedIRIS Research Network que se encuentra en España. Una vez realizado esto, tenemos que indicar un enlace de entre los posibles dependiendo del sistema operativo que estamos empleando. Hay que destacar que para los sistema operativos de MAC y Linux se suele emplear una distribución binaria que permite ver las fuentes de los paquetes de *Sage*, conocidas como SPKG, accediendo a un directorio mediante el empleo de la sentencia *SAGE_ROOT/spkg/standard*.

A partir de ahora, seguimos la instalación solo para el sistema operativo Windows. Para el sistema operativo MAC OS X hacemos referencia a un archivo de texto de la propia página donde se muestra el proceso a seguir tras realizar la descarga: `http://ftp.rediris.es/mirror/sagemath/osx/README.txt`. Una vez seleccionado el sistema Windows simplemente seleccionamos la versión de *Sage* deseada entre todas las disponibles y la descargamos. Para la implementación realizada en este proyecto se ha empleado la versión *SageMath* − 8,4. Después de descargar el archivo ejecutable *Sage* y seguir los pasos del asistente de instalación, se instala *Sage* junto con el entorno Jupyter.

Para acceder a Jupyter tenemos que ejecutar el programa *SageMath Notebook*, del cual obtendremos un acceso directo en el escritorio, y entonces se abrirá una pestaña

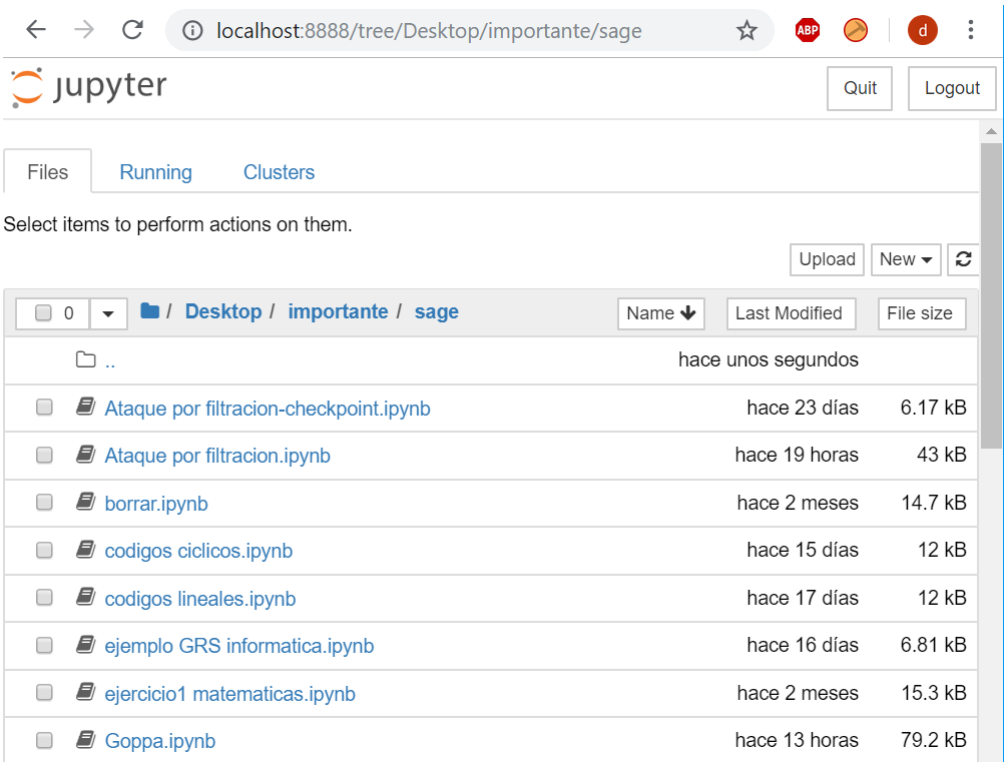

Imagen B.1: Vista general del entorno Jupyter

del navegador, que tenga por defecto el ordenador, con el entorno Jupyter. En dicha pestaña se muestra el conjunto de directorios y archivos del ordenador como podemos ver en la imagen B.1. Una vez nos encontremos sobre el directorio deseado podemos abrir un archivo en formato *.ipynb* deseado o crear un nuevo archivo para implementar código de *Sage*, mediante el empleo del botón *New*, que vemos también en la imagen B.1, y después seleccionamos la opción *SageMath* 8,4.

B.2. Manual de usuario

Una vez que sabemos crear un archivo en el que emplear el lenguaje de *Sage* tenemos que indicar como emplear las distintas clases y funciones implementadas. Para ello, es necesario emplear el comando $load(a)$, donde en la variable de texto a debemos indicar la ruta relativa en la que se encuentra el archivo que queremos emplear. Por ello, es aconsejable realizar la construcción del fichero con formato *.ipynb* junto con los ficheros con formato *.sage* que contienen las clases y funciones implementadas. De esta forma la ruta de acceso al directorio siempre es *./* y después tenemos que indicar el nombre del archivo a emplear. Por ejemplo, si queremos utilizar la clase de los códi-

gos de Goppa, la ruta que tenemos que emplear es *./Goppa.sage*, de esta forma ya se puede llamar a la clase que *Goppa* que contiene el fichero.

Por último, para cada clase que hemos implementado en *Sage* junto con sus correspondientes funciones se ha realizado una documentación que nos permita conocer rápidamente su funcionamiento. Dicha documentación es accesible desde el propio lenguaje *Sage* mediante el empleo de la sentencia $help(a)$ o $a?$, donde a hace referencia al nombre de la clase o función de la que queremos conocer su documentación. Si a es una clase, además de la documentación de la clase, muestra en orden alfabético la documentación de todas las funciones, que no sean "privadas", de esa clase. Por otro lado, si indicamos únicamente una función de una clase, solo se muestra la documentación propia de esa función. Por ejemplo, si queremos obtener la documentación de la función $codificar()$ de la clase *Goppa*, tenemos que emplear la sentencia $help(Goppa.codificar)$ o $Goppa.codificar?$.

En adicción, dichas funciones de ayuda también se pueden emplear sobre una variable que contenga un objeto correspondiente a una cierta clase, es decir, si generamos un código de Goppa que asignamos a la variable *codigo_goppa*, podemos emplear la sentencia $help(codigo\_goppa.codificar)$ o $codigo\_goppa.codificar?$ para obtener la misma documentación de ayuda.

Mostramos a continuación la información que se obtiene a partir de dichas funciones de ayuda para comprender el funcionamiento de cada clase y función. Normalmente, cada clase o función presenta una estructura donde la información se divide en 4 partes, que se corresponden con una breve descripción de la clase o función, los parámetros que puede emplear, la salida o posibles salidas que puede generar y al menos un ejemplo de uso:

B.2.0.1. Goppa

Para el fichero que contiene toda la implementación de los códigos de Goppa comenzamos mostrando la documentación de la propia clase llamada *Goppa*:

1) **Descripción:** Representación de un código de Goppa de la forma: $Goppa(self, n, m, t, LL = None)$

2) **Parámetros:**

   - $"n"$ – número positivo menor o igual a 2^m que indica la longitud del código de Goppa.
   - $"m"$ – número positivo que indica la potencia del número 2, cuyo valor indica el cuerpo F_{2^m} en que se encuentra el código de Goppa.

- $"t"$ – Dos posibles representaciones:
  a) número positivo mayor que dos que indica el grado del polinomio de Goppa que se emplea para construir el código.
  b) polinomio separable sobre el cuerpo F_{2^m} que se emplea como polinomio de Goppa para construir el código.
- $"LL"$ – $(default : "None")$ numero entero no nulo menor que 2^m que indica la primera potencia del elemento generador del cuerpo F_{2^m} en que se encuentra el código de Goppa, el cual se emplea para realizar la construcción del vector L a emplear para construir el código de Goppa.

3) **Salida:** Informa de los parámetros fundamentales que tiene el código de Goppa construido:

   $[n, k, d]$ código de Goppa sobre Finite Field in a of size 2^m

   siendo k la dimensión del código y d la distancia mínima del código.

4) **Ejemplos:** Normalmente, se suele realizar la construcción de un código de Goppa tomando la máxima longitud posible para cada parámetro m, es decir, $n = 2^m$ y tomando el parámetro t como un numero:

   $sage : C = Goppa(2 * *4, 4, 3)$

   $sage : C$

   $[16, 4, 7]$ codigo de Goppa sobre Finite Field in a of size 2^4

   Aunque también, en ocasiones se pueden construir códigos de Goppa con longitud menor a la máxima y tomando el parámetro t como un polinomio separable:

   $sage : F. < a >= GF(2 * *5);$

   $sage : PolRing = PolynomialRing(F,' x');$

   $sage : x = PolRing.gen();$

   $sage : g = PolRing.irreducible\_element(3);$

   $sage : C = Goppa(20, 5, g)$

   $sage : C$

   $[20, 5, 7]$ codigo de Goppa sobre Finite Field in a of size 2^5

Ahora mostramos la documentación dada en las distintas funciones de la clase *Goppa*:

- $\_\_repr\_\_(self)$

  1) **Descripción:** Devuelve una representación de texto de la clase 'self'.

3) **Salida:** Informa de los parámetros fundamentales que tiene el código de Goppa construido:
$[n, k, d]$ código de Goppa sobre Finite Field in a of size 2^m
siendo k la dimensión del código y d la distancia mínima del código.

4) **Ejemplos:**
$sage : C = Goppa(2 * *4, 4, 3)$
$sage : C$
$[16, 4, 7]$ codigo de Goppa sobre Finite Field in a of size 2^4

- $\_codificar\_vector(self, vectorbin)$

1) **Descripción:** Permite realizar la codificación de un vector formado por elementos de F_2.
2) **Parámetros:**
   - $"vectorbin"$ – vector de elementos en F_2 a transformar en un vector formado por la unión de un conjunto de palabras del código de Goppa.
3) **Salida:** Un vector de elementos en F_2

- $codificar(self, mensaje)$

1) **Descripción:** Permite realizar la codificación de una lista o cadena de caracteres ASCII o también de un vector formado por elementos de F_2.
2) **Parámetros:**
   - $"mensaje"$ – vector de elementos en F_2 , lista o cadena de caracteres ASCII a codificar.
3) **Salida:** Un vector de elementos en F_2
4) **Ejemplos:**
$sage : C = Goppa(2 * *4, 4, 3);$
$sage : C.codificar('H')$
$(0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1)$
$sage : C1 = Goppa(2 * *4, 4, 3);$
$sage : mensaje = vector(GF(2), [GF(2).random\_element() \ for \ i \ in \ range(8)])$
$sage : mensaje$
$(1, 0, 0, 1, 0, 1, 1, 0)$
$sage : C1.codificar(mensaje)$
$(0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0)$

- $introducir\_errores(self, mensajecodificado, cantidad)$

  1) **Descripción:** Permite introducir una cierta cantidad de errores en un vector formado por elementos de F_2 cuyo tamaño tiene que ser mayor que la cantidad de errores y mayor o igual que la longitud del código de Goppa pero siempre múltiplo de la longitud del código.
  2) **Parámetros:**
     - $"mensajecodificado"$ – vector de elementos en F_2 que dividimos en bloques de tamaño la longitud del código e introducimos en cada bloque $"cantidad"$ errores.
     - $"cantidad"$ – número entero positivo que indica el número de errores a introducir en el vector.
  3) **Salida:** Un vector de elementos en F_2 con $p * "cantidad"$ errores incorporados, siendo p el numero de bloques en que se divide el vector.
  4) **Ejemplos:**
     $sage : C = Goppa(2 * *4, 4, 3);$
     $sage : m = C.codificar('H')$
     $sage : m$
     $(0,1,0,1,0,0,0,1,0,1,1,1,0,0,1,1,1,0,0,1,1,1,0,1,1,1,1,1,1,1,1,1)$
     $sage : C.introducir\_errores(m, 3)$
     $(0,1,0,1,1,1,0,0,1,0,1,1,1,1,0,1,0,1,0,0,1,1,1,0,1,1,1,0,0,1,1,0,1)$

- $\_dividir\_polinomio(self, p)$

  1) **Descripción:** Divide el polinomio p en la raíz de los coeficientes de los términos con parte par p_0 e impar p_1 tal que se cumple que:

  $$p(z) = p_0(z)^2 + zp_1(z)^2$$

  2) **Parámetros:**
     - $"p"$ – polinomio, normalmente sobre el cuerpo F_{2^m} empleado por el código de Goppa, que deseamos dividir.
  3) **Salida:** Una tupla de longitud 2 con la parte par e impar respectivamente del polinomio p

- $\_longitud\_norma(self, a, b)$

1) **Descripción:** Calcula la longitud del vector (a, b), siendo a, b elementos de $F_{2^m}[x]$, que por la definición dada por Bernstein es $|a^2 + x * b^2|$, donde $|c|$ denota la norma de un polinomio c de $F_{2^m}[x]$, que por la definición dada por Bernstein es $|c| = 2^{deg(c)}$ si c no es 0 y $|c| = 0$ si $c = 0$.
2) **Parámetros:**
   - $"a"$ – primer polinomio de $F_{2^m}[x]$ al que se quiere calcular su longitud respecto al polinomio b.
   - $"b"$ – segundo polinomio de $F_{2^m}[x]$ al que se quiere calcular su longitud respecto al polinomio a.
3) **Salida:** Un numero entero con el valor de la longitud entre a y b.

- $\_reduccion\_base(self, s)$

1) **Descripción:** Calcula los únicos polinomios a, b de $F_{2^m}[x]$ que verifican que $s * b \equiv a \, mod(g)$, siendo s el polinomio introducido y g el polinomio de Goppa del código, y la longitud entre los polinomios a, b es menor que 2^t, siendo t la capacidad correctora del código, i.e, $|(a, b)| <= 2^t$.

   Es necesario que la capacidad correctora del código sea mayor a 2 para que funcione correctamente aunque como indica Bernstein los códigos que no lo verifican son pocos e inútiles ya que presentan malos parámetros.
2) **Parámetros:**
   - $"s"$ – polinomio de $F_{2^m}[x]$ a partir del cual se quiere calcular la congruencia del tercer paso del algoritmo de Patterson.
3) **Salida:** Una tupla de longitud 2 formada por los polinomios a y b respectivamente que verifican la congruencia y tienen la menor longitud.

- $decodificar(self, mensaje, ASCII = False, decodificarV = True)$

1) **Descripción:** Realiza la descodificación de vectores binarios sin errores o con errores, donde el peso del vector de errores no sea superior a la capacidad correctora del código de Goppa en cada bloque en que se divide el mensaje y sea superior a 2.
2) **Parámetros:**
   - $"mensaje"$ – vector binario cuya longitud es múltiplo de la longitud del código de Goppa.
   - $"ASCII"$ – $(default : "False")$ valor booleano que permite realizar o no la transformación del vector descodificado en caracteres ASCII.

- $"decodificarV"$ – $(default : "True")$ valor booleano que permite realizar o no la descodificación del vector una vez corregidos los errores.

3) **Salida:** Existen tres posibles salidas:
   a) Si $ASCII = True$ y $decodificarV = True$, devuelve una cadena de caracteres correspondiente al mensaje descodificado.
   b) Si $ASCII = False$ y $decodificarV = True$ devuelve un vector binario obtenido a partir de la corrección de errores y descodificación de $"mensaje"$.
   c) Si $ASCII = False$ y $decodificarV = False$ devuelve un vector binario obtenido a partir de la corrección de errores de $"mensaje"$

4) **Ejemplos:**

$sage : C = Goppa(2 * *4, 4, 3);$
$sage : c = codigo\_goppa.codificar('H')$
$sage : c$
$(0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1)$
$sage : y = codigo\_goppa.introducir\_errores(c, 3)$
$sage : y$
$(0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1)$
$sage : codigo\_goppa.decodificar(y, True, True)$
$'H'$
$sage : codigo\_goppa.decodificar(y, False, True)$
$(0, 1, 0, 0, 1, 0, 0, 0)$
$sage : codigo\_goppa.decodificar(y, False, False)$
$(0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1)$

- $minimum\_distance(self)$

1) **Descripción:** Devuelve la mínima distancia entre dos palabras cualquiera del código de Goppa, es decir, la distancia mínima del código.

   Como en los códigos de Goppa, su construcción se realiza a partir de la capacidad correctora del código, t, tenemos que la distancia mínima es $2 * t + 1$ ya que empleamos códigos de Goppa binarios.

4) **Ejemplos:**

$sage : C = Goppa(2 * *4, 4, 3);$
$sage : C.minimum\_distance()$
7

$sage : C1 = Goppa(2**6, 6, 6);$
$sage : C1.minimum_distance()$
13

- $generator\_matrix\_nobin(self)$

  1) **Descripción:** Devuelve la matriz generatriz sobre el cuerpo F_{2^m} del código de Goppa, aunque en realidad no se emplea debido a que siempre trabajamos con la matriz generatriz sobre el cuerpo F_2, ver la función $generator\_matrix()$.

- $parity\_check\_matrix\_nobin(self)$

  1) **Descripción:** Devuelve la matriz de control sobre el cuerpo F_{2^m} del código de Goppa, aunque en realidad no se emplea debido a que siempre trabajamos con la matriz de control sobre el cuerpo F_2, ver la función $parity\_check\_matrix()$.

- $generator\_matrix(self)$

  1) **Descripción:** Devuelve la matriz generatriz sobre el cuerpo F_2 del código de Goppa.
  4) **Ejemplos:**
  $sage : C = Goppa(2**4, 4, 3);$
  $sage : C.generator\_matrix()$
  [1001110000111010]
  [0100110110100010]
  [0010000100101111]
  [0000001011111110]

- $parity\_check\_matrix(self)$

  1) **Descripción:** Devuelve la matriz de control sobre el cuerpo F_2 del código de Goppa.
  4) **Ejemplos:**
  $sage : C = Goppa(2**4, 4, 3);$
  $sage : C.parity\_check\_matrix()$
  [1001010111100111]
  [1010111000010001]

[0100000100110101]
[0100010110001101]
[0111011011100110]
[1011000001101100]
[1100111111010100]
[0000110100000100]
[0111011100111010]
[0110110001000100]
[1101100011000100]
[1010100100011100]

- $cuerpo\_finito(self)$

  1) **Descripción:** Devuelve el cuerpo finito F_2 que emplea el código de Goppa.
  4) **Ejemplos:**
  $sage: C = Goppa(2**4, 4, 3);$
  $sage: C.cuerpo\_finito()$
  Finite Field of size 2

- $polinomio\_g(self)$

  1) **Descripción:** Devuelve el polinomio de Goppa, g, sobre el cuerpo F_{2^m} que emplea el código de Goppa.
  4) **Ejemplos:**
  $sage: C = Goppa(2**4, 4, 3);$
  $sage: C.polinomio\_g()$
  $x^3 + a^2$
  Obviamente si realizamos la construcción a partir de un polinomio separable, obtenemos el mismo polinomio como resultado:
  $sage: F. < a >= GF(2**4);$
  $sage: PolRing = PolynomialRing(F,' x');$
  $sage: x = PolRing.gen();$
  $sage: g = x^3 + F.gen()^2 * x + F.gen();$
  $x^3 + a^2 * x + a$
  $sage: C = Goppa(2**4, 4, g);$
  $sage: C.polinomio\_g()$
  $x^3 + a^2 * x + a$

- $dimension(self)$

  1) **Descripción:** Devuelve la dimension, k, que tiene el código de Goppa. Normalmente dicho valor viene dado por: $k = n - mt$

  4) **Ejemplos:**
  $sage : C = Goppa(2 * *4, 4, 3);$
  $sage : C.dimension()$
  4

- $length(self)$

  1) **Descripción:** Devuelve la longitud, n, que tiene el código de Goppa.

  4) **Ejemplos:**
  $sage : C = Goppa(2 * *4, 4, 3);$
  $sage : C.length()$
  16
  $sage : C = Goppa(250, 8, 5);$
  $sage : C.length()$
  250

- $base\_field(self)$

  1) **Descripción:** Devuelve el cuerpo finito F_{2^m} que emplea el código de Goppa.

  4) **Ejemplos:**
  $sage : C = Goppa(2 * *4, 4, 3);$
  $sage : C.base\_field()$
  Finite Field in a of size 2^4
  $sage : C = Goppa(2 * *5, 8, 4);$
  $sage : C.base\_field()$
  Finite Field in a of size 2^8

- $is\_Goppa(self)$

  1) **Descripción:** Devuelve el valor True, de forma que nos permite diferenciar este código de otros tipos de códigos.

4) **Ejemplos:**

$sage : C = Goppa(2 * *4, 4, 3);$
$sage : boolG = False;$
$sage : try :$
$sage : \quad boolG = C.is\_Goppa();$
$sage : except :$
$sage : \quad boolG = False;$
$sage : boolG$
$True$
$sage : C1 = codes.random\_linear\_code(GF(7), 5, 3)$
$sage : C1$
$[5, 3]$ linear code over $GF(7)$
$sage : boolG = False;$
$sage : try :$
$sage : \quad boolG = C.is\_Goppa();$
$sage : except :$
$sage : \quad boolG = False;$
$sage : boolG$
$False$

B.2.0.2. Código de producto de matrices

Para el fichero que contiene toda la implementación de los códigos de producto de matrices también vamos a mostrar en primer lugar la documentación de la propia clase llamada *MatrixProductCodes*:

1) **Descripción:** Representación de un código de producto de matrices formado a partir de códigos encajados y una matriz NSC, todos sobre el mismo cuerpo F_{p^m}, el cual es de la forma: $MatrixProductCodes(self, A, lista\_codigos)$

2) **Parámetros:**

- "A" – matriz de tamaño $M \times N$ sobre el cuerpo F_{p^m}, la cual debe ser no singular por columnas (NSC).
- "$lista_codigos$" – lista que contiene el conjunto de códigos encajados a partir de los cuales se realiza la construcción del código de producto de matrices. La longitud de la lista debe coincidir con el numero de filas de la matriz A y los códigos deben encontrarse sobre el cuerpo F_{p^m}

3) **Salida:** Informa de los parámetros fundamentales que tiene el código de producto de matrices construido:

   $[n, k, d]$ código de producto de matrices sobre Finite Field in a of size 2^m

   siendo k la dimensión del código y d la distancia mínima del código.

4) **Ejemplos:**

   $sage : A = matrix(GF(2), [[1,1],[0,1]]);$

   $sage : F. < a >= GF(2 * *5);$

   $sage : PolRing = PolynomialRing(F,' x');$

   $sage : x = PolRing.gen();$

   $sage : C1 = Goppa(2 * *5, 5, 3);$

   $sage : g1 = C1.polinomio\_g()$

   $sage : g1$

   $x^3 + (a^4 + a^2 + 1) * x + a^3 + a^2 + a + 1$

   $sage : g3 = PolRing.irreducible\_element(3);$

   $sage : g2 = g1 * g3;$

   $sage : g2$

   $x^6 + (a^3 + a^2) * x^4 + (a^2 + a) * x^3 + (a^4 + a^3 + a + 1) * x^2 + (a^4 + a^3 + a^2 + a + 1) * x + a^4 + a^3$

   $sage : C2 = Goppa(2 * *5, 5, g2)$

   $sage : C = MatrixProductCodes(A, [C1, C2]);$

   $sage : F = GF(13)$

   $sage : a = [F.list()[i] for\ i\ in\ range(1,3) + range(8,13)]$

   $sage : b = [F.list()[i] for\ i\ in\ [1,3,1,10,12,7,5]]$

   $sage : C3 = codes.GeneralizedReedSolomonCode(a, 2, b)$

   $sage : C2 = codes.GeneralizedReedSolomonCode(a, 4, b)$

   $sage : A = matrix(GF(13), [[1,1,1],[0,2,1]]);$

   $sage : MPCGRS = MatrixProductCodes(A, [C2, C3])$

Ahora mostramos también la documentación dada en las distintas funciones de la clase *MatrixProductCodes*:

- $\_\_repr\_\_(self)$

1) **Descripción:** Devuelve una representación de texto de la clase 'self'.

3) **Salida:** Informa de los parámetros fundamentales que tiene el código de producto de matrices construido:
$[n, k, d]$ código de producto de matrices sobre Finite Field in a of size p^m
siendo k la dimensión del código y d la distancia mínima del código.

4) **Ejemplos:**
$sage : F = GF(13)$
$sage : a = [F.list()[i]\ for\ i\ in\ range(1,3) + range(8,13)]$
$sage : b = [F.list()[i]\ for\ i\ in\ [1,3,1,10,12,7,5]]$
$sage : C3 = codes.GeneralizedReedSolomonCode(a,2,b)$
$sage : C2 = codes.GeneralizedReedSolomonCode(a,4,b)$
$sage : A = matrix(GF(13),[[1,1,1],[0,2,1]]);$
$sage : MPCGRS = MatrixProductCodes(A,[C2,C3])$
$sage : MPCGRS$
$[21,6,12]$ codigo de producto de matrices sobre Finite Field of size 13

- $introducir\_errores(self, mensajecodificado, cantidad)$

1) **Descripción:** Permite introducir una cierta cantidad de errores en un vector formado por elementos de F_{p^m} cuyo tamaño tiene que ser mayor que la cantidad de errores y mayor o igual que la longitud del código empleado.
Si nos encontramos en el caso en que algún código, empleado para la construcción del código de producto de matrices, es de Goppa se garantiza que en cada bloque no haya ni 1 ni 2 errores.

2) **Parámetros:**
   - $"mensajecodificado"$ – vector de elementos en F_{p^m} en el que introducimos $"cantidad"$ errores.
   - $"cantidad"$ – numero entero positivo que indica el numero de errores a introducir en el vector.

3) **Salida:** Un vector de elementos en F_{p^m} con $"cantidad"$ errores incorporados.

4) **Ejemplos:**
$sage : F = GF(13)$
$sage : a = [F.list()[i]\ for\ i\ in\ range(1,3) + range(8,13)]$
$sage : b = [F.list()[i]\ for\ i\ in\ [1,3,1,10,12,7,5]]$
$sage : C3 = codes.GeneralizedReedSolomonCode(a,2,b)$
$sage : C2 = codes.GeneralizedReedSolomonCode(a,4,b)$

$sage : A = matrix(GF(13), [[1,1,1],[0,2,1]]);$
$sage : MPCGRS = MatrixProductCodes(A, [C2, C3])$
$sage : MPCGRS.minimum\_distance()$
12
$sage : m = vector(GF(13), [0,1,2,3,4,5]);$
$sage : c = MPCGRS.encode(m);$
$sage : c$
$(6,11,8,11,1,4,3,11,4,5,3,10,11,6,2,1,0,7,12,1,11)$
$sage : MPCGRS.introducir\_errores(c, 5)$
$(6,11,8,11,1,4,3,11,0,8,3,9,11,6,2,1,2,7,12,1,3)$

- $encode(self, mensaje)$

  1) **Descripción:** Permite transformar un vector de longitud k formado por elementos de F_{p^m}.
  2) **Parámetros:**
     - *"mensaje"* – vector de elementos en F_{p^m} y longitud k a codificar.
  3) **Salida:** Un vector de elementos en F_{p^m} y longitud n.
  4) **Ejemplos:**
     $sage : F = GF(13)$
     $sage : a = [F.list()[i] \; for \; i \; in \; range(1,3) + range(8,13)]$
     $sage : b = [F.list()[i] \; for \; i \; in \; [1,3,1,10,12,7,5]]$
     $sage : C3 = codes.GeneralizedReedSolomonCode(a, 2, b)$
     $sage : C2 = codes.GeneralizedReedSolomonCode(a, 4, b)$
     $sage : A = matrix(GF(13), [[1,1,1],[0,2,1]]);$
     $sage : MPCGRS = MatrixProductCodes(A, [C2, C3])$
     $sage : m = vector(GF(13), [0,1,2,3,4,5]);$
     $sage : c = MPCGRS.encode(m);$
     $sage : c$
     $(6,11,8,11,1,4,3,11,4,5,3,10,11,6,2,1,0,7,12,1,11)$

- $McEliece\_publica\_privada\_reducida(self)$

  1) **Descripción:** Calcula una matriz de permutación cuadrada aleatoria, P_0, de tamaño n_0, siendo n_0 la longitud de cualquier código empleado para la construcción del código de producto de matrices $C = [C_1, ..., C_M] \cdot A$, junto con M matrices invertibles $S_1, ..., S_M$ cuadradas aleatorias de tamaño

$k_1, ..., k_M$ respectivamente, siendo $k_1, ...k_M$ las dimensiones de los códigos $C_1, ..., C_M$. También nos proporciona la matriz A y las M matrices $G'_i = S_i * G_i * P_0$ donde G_i es una matriz generatriz del código C_i, $i = 1, ..., M$.

3) **Salida:** Una lista de longitud 4 formada, en primer lugar por una lista con las M matrices invertibles aleatorias S_i, en segundo lugar por la matriz de permutación P_0, en tercer lugar por la matriz A y en cuarto lugar por una lista con las M matrices G'_i.

4) **Ejemplos:**

$sage : F = GF(13)$
$sage : a = [F.list()[i]\ for\ i\ in\ range(1,3) + range(8,13)]$
$sage : b = [F.list()[i]\ for\ i\ in\ [1,3,1,10,12,7,5]]$
$sage : C3 = codes.GeneralizedReedSolomonCode(a,2,b)$
$sage : C2 = codes.GeneralizedReedSolomonCode(a,4,b)$
$sage : A = matrix(GF(13),[[1,1,1],[0,2,1]]);$
$sage : MPCGRS = MatrixProductCodes(A,[C2,C3])$
$sage : claves_reducidas = MPCGRS.McEliece\_publica\_privada\_reducida();$
$sage : claves\_reducidas[0]$
[
[9 3 1 10]
[8 10 2 9]
[2 0 10 6][8 11]
[1 10 6 3], [8 0]
]
$sage : claves\_reducidas[1]$
[0000001]
[0100000]
[1000000]
[0010000]
[0000010]
[0000100]
[0001000]
$sage : claves\_reducidas[2]$
[111]
[021]
$sage : claves\_reducidas[3]$
[

[4 11 9 11 9 1 10]
[1 12 12 7 1 0 3]
[9 10 3 4 10 5 5][5 12 4 11 6 12 6]
[12 12 2 9 10 4 7], [8 11 2 1 4 5 8]
]

- $McEliece\_publica\_privada(self, S\_bloques, P\_bloques)$

  1) **Descripción:** Permite calcular la matriz invertible total S, la matriz de permutación P, y la matriz publica $G' = S * G * P$ a partir de las M matrices S_i y la matriz de permutación P_0 de tamaño n cuadrada.
  2) **Parámetros:**
     - "$S_bloques$" – lista de tamaño M compuesta por las matrices invertibles cuadradas S_i ,$i = 1, ..., M$ de tamaños $k_1, ..., k_M$ siendo k_i la dimension del código C_i.
     - "$P_bloques$" – matriz de permutación cuadrada aleatoria, P_0, de tamaño n_0, siendo n_0 la longitud de cualquier código empleado para la construcción del código de producto de matrices.
  3) **Salida:** Una lista de longitud 3 formada, en primer lugar por la matriz invertible aleatoria S generada a partir de las M matrices S_i, en segundo lugar por la matriz de permutación P, en tercer lugar por la matriz G'.
  4) **Ejemplos:**
     $sage: F = GF(13)$
     $sage: a = [F.list()[i]\ for\ i\ in\ range(1,3) + range(8,13)]$
     $sage: b = [F.list()[i]\ for\ i\ in\ [1,3,1,10,12,7,5]]$
     $sage: C3 = codes.GeneralizedReedSolomonCode(a,2,b)$
     $sage: C2 = codes.GeneralizedReedSolomonCode(a,4,b)$
     $sage: A = matrix(GF(13),[[1,1,1],[0,2,1]]);$
     $sage: MPCGRS = MatrixProductCodes(A,[C2,C3])$
     $sage: claves_reducidas = MPCGRS.McEliece\_publica\_privada\_reducida();$
     $sage: claves = MPCGRS.McEliece\_publica\_privada(claves\_reducidas[0], claves\_reducidas$
     $sage: claves[0]$
     [9 3 1 10 0 0]
     [8 10 2 9 0 0]
     [2 0 10 6 0 0]
     [1 10 6 3 0 0]

[0 0 0 0 8 11]
[0 0 0 0 8 0]
$sage : claves[1]$
21×21 dense matrix over Finite Field of size 13
$sage : claves[2]$
[4 11 9 11 9 1 10 4 11 9 11 9 1 10 4 11 9 11 9 1 10]
[1 12 12 7 1 0 3 1 12 12 7 1 0 3 1 12 12 7 1 0 3]
[9 10 3 4 10 5 5 9 10 3 4 10 5 5 9 10 3 4 10 5 5]
[12 12 2 9 10 4 7 12 12 2 9 10 4 7 12 12 2 9 10 4 7]
[0 0 0 0 0 0 0 10 11 8 9 12 11 12 5 12 4 11 6 12 6]
[0 0 0 0 0 0 0 3 9 4 2 8 10 3 8 11 2 1 4 5 8]

- $minimum\_distance(self)$

  1) **Descripción:** Devuelve la mínima distancia del código de producto de matrices. Como los códigos de producto de matrices verifican el teorema 6,11 entonces tenemos que la mínima distancia se obtiene:
  $d = \min d1 * D1, ..., d_M * D_M$ donde d_i hace referencia a la mínima distancia de los M códigos que forman el código de producto de matrices y D_i hace referencia a la mínima distancia del código lineal formado a partir de las i primeras filas de la matriz A.

  4) **Ejemplos:**
  $sage : F = GF(13)$
  $sage : a = [F.list()[i]\ for\ i\ in\ range(1,3) + range(8,13)]$
  $sage : b = [F.list()[i]\ for\ i\ in\ [1,3,1,10,12,7,5]]$
  $sage : C3 = codes.GeneralizedReedSolomonCode(a,2,b)$
  $sage : C2 = codes.GeneralizedReedSolomonCode(a,4,b)$
  $sage : A = matrix(GF(13),[[1,1,1],[0,2,1]]);$
  $sage : C = MatrixProductCodes(A,[C2,C3])$
  $sage : C.minimum\_distance()$
  12

- $decodificar\_MPC(self, y1, decodificarV = True)$

  1) **Descripción:** Realiza la descodificación de vectores en el cuerpo F_{p^m} sin errores o con errores, donde el peso del vector de errores no sea superior a la capacidad correctora del código de producto de matrices.
  Si estamos empleando algún código de Goppa, es necesario que el numero de errores introducidos sea mayor que dos o 0 en ese bloque.

2) **Parámetros:**
   - $"y1"$ – vector binario cuya longitud es la longitud del código de producto de matrices, n.
   - $"decodificarV"$ – $(default : "True")$ valor booleano que permite realizar o no la descodificación del vector una vez corregidos los errores.

3) **Salida:** Existen dos posibles salidas:
   a) Si $decodificarV = True$ devuelve un vector en el cuerpo F_{p^m} obtenido a partir de la corrección de errores y descodificación de $"mensaje"$.
   b) Si $decodificarV = False$ devuelve un vector en el cuerpo F_{p^m} obtenido a partir de la corrección de errores de $"mensaje"$.

4) **Ejemplos:**

$sage : F = GF(13)$
$sage : a = [F.list()[i]\ for\ i\ in\ range(1,3) + range(8,13)]$
$sage : b = [F.list()[i]\ for\ i\ in\ [1,3,1,10,12,7,5]]$
$sage : C3 = codes.GeneralizedReedSolomonCode(a,2,b)$
$sage : C2 = codes.GeneralizedReedSolomonCode(a,4,b)$
$sage : A = matrix(GF(13),[[1,1,1],[0,2,1]]);$
$sage : C = MatrixProductCodes(A,[C2,C3])$
$sage : C.minimum\_distance()$
12
$sage : m = vector(GF(13),[0,1,2,3,4,5]);$
$sage : c = C.encode(m);$
$sage : c$
$(6,11,8,11,1,4,3,11,4,5,3,10,11,6,2,1,0,7,12,1,11)$
$sage : y = C.introducir\_errores(c,5)$
$sage : C.decodificar\_MPC(y,False)$
$(6,11,8,11,1,4,3,11,4,5,3,10,11,6,2,1,0,7,12,1,11)$
$sage : C.decodificar\_MPC(y,True)$
$(0,1,2,3,4,5)$

- $matriz\_A(self)$

1) **Descripción:** Devuelve la matriz A que se emplea para construir el código de producto de matrices.

4) **Ejemplos:**
$sage: F = GF(13)$
$sage: a = [F.list()[i]\ for\ i\ in\ range(1,3) + range(8,13)]$
$sage: b = [F.list()[i]\ for\ i\ in\ [1,3,1,10,12,7,5]]$
$sage: C3 = codes.GeneralizedReedSolomonCode(a,2,b)$
$sage: C2 = codes.GeneralizedReedSolomonCode(a,4,b)$
$sage: A = matrix(GF(13),[[1,1,1],[0,2,1]]);$
$sage: C = MatrixProductCodes(A,[C2,C3])$
$sage: C.minimum\_A()$
[1 1 1] [0 2 1]

- $lista\_codigos(self)$

1) **Descripción:** Devuelve una lista con el conjuntos de códigos que forman el código de producto de matrices

4) **Ejemplos:**
$sage: F = GF(13)$
$sage: a = [F.list()[i]\ for\ i\ in\ range(1,3) + range(8,13)]$
$sage: b = [F.list()[i]\ for\ i\ in\ [1,3,1,10,12,7,5]]$
$sage: C3 = codes.GeneralizedReedSolomonCode(a,2,b)$
$sage: C2 = codes.GeneralizedReedSolomonCode(a,4,b)$
$sage: A = matrix(GF(13),[[1,1,1],[0,2,1]]);$
$sage: C = MatrixProductCodes(A,[C2,C3])$
$sage: C.lista\_codigos()$
$[[7,4,4]$ Generalized Reed-Solomon Code over $GF(13)$, $[7,2,6]$ Generalized Reed-Solomon Code over $GF(13)]$

- $dimension(self)$

1) **Descripción:** Devuelve la dimension, k, que tiene el código de producto de matrices. Dicho valor viene dado por: $k = k_1 + ... + k_n$, donde k_i hace referencia a la dimension del código i-esimo que forman el código de producto de matrices.

4) **Ejemplos:**
$sage: F = GF(13)$
$sage: a = [F.list()[i]\ for\ i\ in\ range(1,3) + range(8,13)]$
$sage: b = [F.list()[i]\ for\ i\ in\ [1,3,1,10,12,7,5]]$

$sage: C3 = codes.GeneralizedReedSolomonCode(a, 2, b)$
$sage: C2 = codes.GeneralizedReedSolomonCode(a, 4, b)$
$sage: A = matrix(GF(13), [[1,1,1],[0,2,1]]);$
$sage: C = MatrixProductCodes(A, [C2, C3])$
$sage: C.dimension()$
6

- $length(self)$

1) **Descripción:** Devuelve la longitud, n, que tiene el código de producto de matrices. Dicho valor se obtiene multiplicando la longitud que tienen los códigos que forman el código de producto de matrices, por el número de columnas de la matriz A

4) **Ejemplos:**
$sage: F = GF(13)$
$sage: a = [F.list()[i] \ for \ i \ in \ range(1,3) + range(8,13)]$
$sage: b = [F.list()[i] \ for \ i \ in \ [1,3,1,10,12,7,5]]$
$sage: C3 = codes.GeneralizedReedSolomonCode(a, 2, b)$
$sage: C2 = codes.GeneralizedReedSolomonCode(a, 4, b)$
$sage: A = matrix(GF(13), [[1,1,1],[0,2,1]]);$
$sage: C = MatrixProductCodes(A, [C2, C3])$
$sage: C.length()$
21

- $generator\_matrix(self)$

1) **Descripción:** Devuelve la matriz generatriz del código de producto de matrices.

4) **Ejemplos:**
$sage: F = GF(13)$
$sage: a = [F.list()[i] \ for \ i \ in \ range(1,3) + range(8,13)]$
$sage: b = [F.list()[i] \ for \ i \ in \ [1,3,1,10,12,7,5]]$
$sage: C3 = codes.GeneralizedReedSolomonCode(a, 2, b)$
$sage: C3.generator\_matrix()$
[1 3 1 10 12 7 5]
[1 6 8 12 3 12 8]
$sage: C2 = codes.GeneralizedReedSolomonCode(a, 4, b)$

$sage: C2.generator\_matrix()$
[1 3 1 10 12 7 5]
[1 6 8 12 3 12 8]
[1 12 12 4 4 2 5]
[1 11 5 10 1 9 8]
$sage: A = matrix(GF(13), [[1,1,1],[0,2,1]]);$
$sage: C = MatrixProductCodes(A, [C2, C3])$
$sage: C.generator\_matrix()$
[1 3 1 10 12 7 5 1 3 1 10 12 7 5 1 3 1 10 12 7 5]
[1 6 8 12 3 12 8 1 6 8 12 3 12 8 1 6 8 12 3 12 8]
[1 12 12 4 4 2 5 1 12 12 4 4 2 5 1 12 12 4 4 2 5]
[1 11 5 10 1 9 8 1 11 5 10 1 9 8 1 11 5 10 1 9 8]
[0 0 0 0 0 0 0 2 6 2 7 11 1 10 1 3 1 10 12 7 5]
[0 0 0 0 0 0 0 2 12 3 11 6 11 3 1 6 8 12 3 12 8]

- $base\_field(self)$

  1) **Descripción:** Devuelve el cuerpo finito F_{p^m} que emplea el código de producto de matrices.

  4) **Ejemplos:**
  $sage: F = GF(13)$
  $sage: a = [F.list()[i] \; for \; i \; in \; range(1,3) + range(8,13)]$
  $sage: b = [F.list()[i] \; for \; i \; in \; [1,3,1,10,12,7,5]]$
  $sage: C3 = codes.GeneralizedReedSolomonCode(a, 2, b)$
  $sage: C2 = codes.GeneralizedReedSolomonCode(a, 4, b)$
  $sage: A = matrix(GF(13), [[1,1,1],[0,2,1]]);$
  $sage: C = MatrixProductCodes(A, [C2, C3])$
  $sage: C.base\_field()$
  Finite Field of size 13

- $is\_MPC(self)$

  1) **Descripción:** Devuelve el valor *True*, de forma que nos permite diferenciar este código de otros tipos de códigos.

  4) **Ejemplos:**
  $sage: F = GF(13)$
  $sage: a = [F.list()[i] \; for \; i \; in \; range(1,3) + range(8,13)]$

$sage : b = [F.list()[i]\ for\ i\ in\ [1,3,1,10,12,7,5]]$
$sage : C3 = codes.GeneralizedReedSolomonCode(a,2,b)$
$sage : C2 = codes.GeneralizedReedSolomonCode(a,4,b)$
$sage : A = matrix(GF(13),[[1,1,1],[0,2,1]]);$
$sage : C = MatrixProductCodes(A,[C2,C3])$
$sage : boolMPC = False;$
$sage : try :$
$sage : \quad boolMPC = C.is\_MPC();$
$sage : except :$
$sage : \quad boolMPC = False;$
$sage : boolMPC$
$True$

B.2.0.3. McEliece

Para el fichero que contiene toda la implementación del criptosistema de McEliece seguimos mostrando la documentación de la propia clase en primer lugar, llamada *McEliece*:

1) **Descripción:** Representación del criptosistema de McElice formado a partir de la introducción de un código y el número de errores que se desean incorporar durante el proceso de cifrado, el cual normalmente coincide con la capacidad correctora del código. El critosistema se construye de la forma: $McEliece(self, code, t, reduction\_MPC = None)$

2) **Parámetros:**

- "*code*" – código lineal que se emplea para la construcción del criptosistema.
- "t" – numero entero no negativo que indica la cantidad de errores que se deben añadir durante el proceso de cifrado. Normalmente se le da el valor de la capacidad correctora, al ser la máxima cantidad de errores que se pueden introducir y corregir.
- "*reduction_MPC*" – $(default : reduction\_MPC = None)$ lista de longitud 2 compuesta en primer lugar por una matriz invertible cuadrada, S, de tamaño la dimension del código "*code*" cuyos elementos están en el cuerpo F_{p^m} que el código emplea. En segundo lugar ocupa una matriz de permutación cuadrada, P, de tamaño la longitud del código también sobre el cuerpo F_{p^m} que el código emplea.

3) **Salida:** Informa de los parámetros fundamentales que tiene el código empleado, "*code*", para la construcción del criptosistema así como de la cantidad de errores que se introducen durante el proceso de cifrado, "t":

   Criptosistema de McEliece construido a partir de "*code*" que introduce "t" errores

4) **Ejemplos:**

   Normalmente, se suele realizar la construcción del criptosistema de McEliece empleando únicamente los parámetros "*code*" y "t", por tanto, se generan aleatoriamente tanto la matriz de permutación, P, como la matriz invertible, S, sobre el cuerpo que se este empleando:

   $sage : C = codes.GeneralizedReedSolomonCode(GF(16,'a').list(), 6)$

   $sage : C$

   $[16, 6, 11]$ Reed-Solomon Code over $GF(16)$

   $sage : C.minimum\_distance()$

   11

   $sage : codigo\_mceliece1 = McEliece(C, 5)$

   Aunque también, en ocasiones se puede construir el criptosistema de McEliece empleando en adicción el parámetro opcional "*reduction_MPC*", donde indicamos la matriz invertible, S, y la matriz de permutación, P, para la generación del criptosistema. Dicha opción es importante para construir el nuevo criptosistema de McEliece mediante las claves con tamaño reducido, cuando empleamos los códigos de producto de matrices:

   $sage : F = GF(13)$

   $sage : a = [F.list()[i] \; for \; i \; in \; range(1, 3) + range(8, 13)]$

   $sage : b = [F.list()[i] \; for \; i \; in \; [1, 3, 1, 10, 12, 7, 5]]$

   $sage : C3 = codes.GeneralizedReedSolomonCode(a, 2, b)$

   $sage : C2 = codes.GeneralizedReedSolomonCode(a, 4, b)$

   $sage : A = matrix(GF(13), [[1, 1, 1], [0, 2, 1]]);$

   $sage : MPCGRS = MatrixProductCodes(A, [C2, C3])$

   $sage : MPCGRS.minimum\_distance()$

   12

   $sage : McEliece\_MPC22 = McEliece(MPCGRS, 5)$ #Podemos corregir hasta 5 errores

sage : #Obtenemos las claves reducidas del codigo de producto de matrices

sage : $claves\_reducidas = MPCGRS.McEliece\_publica\_privada\_reducida();$

sage : #Ahora a partir de ellas generamos las matrices totales para poder generar el criptosistema de McEliece.

sage : $claves = MPCGRS.McEliece\_publica\_privada(claves\_reducidas[0],$
$claves\_reducidas[1])$

sage : $McEliece\_reduccion = McEliece(MPCGRS, 5, [claves[0], claves[1]])$

Ahora mostramos la documentación dada en las distintas funciones de la clase *McEliece*:

- $\_\_repr\_\_(self)$

  1) **Descripción:** Devuelve una representación de texto de la clase 'self'.
  3) **Salida:** Informa de los parámetros fundamentales que tiene el código empleado, "*code*", para la construcción del criptosistema así como de la cantidad de errores que se introducen durante el proceso de cifrado, "t":
     Criptosistema de McEliece construido a partir de "*code*" que introduce "t" errores
  4) **Ejemplos:**
     sage : $C = codes.GeneralizedReedSolomonCode(GF(16,'a').list(), 6)$
     sage : C
     $[16, 6, 11]$ Reed-Solomon Code over $GF(16)$
     sage : $C.minimum\_distance()$
     11
     sage : $codigo\_mceliece1 = McEliece(C, 5)$
     sage : $codigo\_mceliece1$
     Criptosistema de McEliece construido a partir de $[16, 6, 11]$ Reed-Solomon Code over $GF(16)$ que introduce 5 errores

- $introducir\_errores(self, mensajecodificado1, cantidad)$

  1) **Descripción:** Permite introducir una cierta cantidad de errores en un vector formado por elementos del cuerpo empleado por el código, F_{p^m}, cuyo tamaño tiene que ser mayor que la cantidad de errores y mayor o igual que la longitud del código empleado.

Si nos encontramos en el caso en que el código empleado para la construcción del criptosistema de McEliece, es de Goppa se recurre a la función $introducir\_errores()$ de su propia clase para ello.

Si nos encontramos en el caso en que el código empleado para la construcción del criptosistema de McEliece, es un código de producto de matrices se recurre a la función $introducir\_errores()$ de su propia clase para ello.

2) **Parámetros:**
   - $"mensajecodificado"$ – vector de elementos en F_{p^m} en el que introducimos $"cantidad"$ errores.
   - $"cantidad"$ – numero entero positivo que indica el numero de errores a introducir en el vector.

3) **Salida:** Un vector de elementos en F_{p^m} con $"cantidad"$ errores incorporados.

4) **Ejemplos:**

$sage: C = codes.GeneralizedReedSolomonCode(GF(16,'a').list(),6)$

$sage: C$

$[16,6,11]$ Reed-Solomon Code over $GF(16)$

$sage: C.minimum\_distance()$

11

$sage: codigo\_mceliece1 = McEliece(C,5)$

$sage: vector1 = C.random\_element()$

$sage: vector1$

$(a^2+a, a^2+a+1, a^3+a^2+a, a^3+a^2, a^3+a^2+a, a^3, a+1, 0, a^2+1, a^3+1, a^3+a, 1, a+1, a^3, a^3+a^2, a^2+a)$

$sage: vector1\_errores = codigo\_mceliece1.introducir\_errores(vector1,5)$

$sage: vector1\_errores$

$(a^2+a, a^2+a+1, a^3+a^2+a, a^3+a^2+a+1, a^3+a^2+a, a^2, a^3+a+1, 0, a^2+a+1, a^3+1, a^3+a, a^3+a, a+1, a^3, a^3+a^2, a^2+a)$

- $cifrado(self, mensaje, fichero = False)$

1) **Descripción:** Permite realizar el cifrado de una lista o cadena de caracteres ASCII de cualquier longitud o también de un vector de longitud k y elementos del cuerpo empleado por el código usado.

   En adicción también es capaz de leer un fichero de texto con formato $".txt"$ y con caracteres ASCII para realizar después el cifrado de todo el fichero.

2) **Parámetros:**

- $"mensaje"$ – vector de elementos en F_{p^m} y longitud k , lista o cadena de caracteres de cualquier longitud ASCII que cifrar.
- $"fichero"$ – $(default : "False")$ – valor booleano mediante el que indicamos si se desea cifrar el texto $"mensaje"$ o leer y cifrar el fichero con nombre $"mensaje"$.

3) **Salida:** Existen dos posibles salidas:
   a) Un vector de elementos en F_{p^m} y longitud n o múltiplo de n si el valor de $"fichero"$ es $"False"$.
   b) Si el valor de $"fichero"$ es $"True"$ se crea un fichero con nombre $"mensaje" - cif$ en el cual se encuentra el fichero con nombre $"mensaje"$ cifrado.

4) **Ejemplos:**
   $sage : C = codes.GeneralizedReedSolomonCode(GF(16,' a').list(), 6)$
   $sage : C.minimum\_distance()$
   11
   $sage : codigo\_mceliece1 = McEliece(C, 5)$
   $sage : cifrado\_errores = codigo\_mceliece1.cifrado('Hola');$
   $sage : cifrado\_errores$
   $(a^3 + a^2 + 1, a^2 + 1, a^3 + 1, a^2, a^2 + a + 1, 1, a^2 + 1, 0, a^3 + 1, a^3 + a, a^3 + a^2, a, a^2 + 1, 1, a^3 + a^2 + 1, 0, a, a + 1, a^2, a^3, a^2 + 1, a^3 + a^2 + a + 1, a^3 + a^2 + 1, a + 1, a^3 + 1, a^3, a^3 + 1, a^3 + 1, a + 1, a^2, 1, 1)$
   $sage : codigo\_mceliece1.cifrado('mensajeparaGRS.txt', True);$
   (' El mensaje cifrado se ha guardado en el archivo con nombre: ', $'mensajeparaGRS - cif.txt'$)
   $sage : vector1 = vector(GF(16,' a'), [GF(16,' a').random\_element() \; for \; i \; in \; range(6)])$
   $sage : vector1$
   $(a^3 + a, a^3 + a + 1, 1, a + 1, a, a^3 + a + 1)$
   $sage : cifrado = codigo\_mceliece1.cifrado(vector1)$
   $sage : cifrado$
   $(a, a^3 + a^2 + a + 1, 1, a^3 + 1, a^3 + a^2 + a, a^2, a^3 + a^2 + a + 1, a^3 + a, a^3 + a^2 + a, a + 1, 1, a^3 + a^2 + a + 1, a^3 + a^2 + a + 1, a^3 + a^2, 1, a^2 + a + 1)$

- $descifrado(self, mensaje, ASCII = True, fichero = False)$

1) **Descripción:** Realiza el descifrado de vectores sobre F_{p^m}, donde el peso del vector de errores no sea superior a la capacidad correctora del código empleado para la construcción del criptosistema de McEliece en cada bloque en que se divide el mensaje.

En adicción también es capaz de leer un fichero de texto con formato ".*txt*" que contenga el mensaje cifrado para realizar el correspondiente descifrado de todo el fichero.

2) **Parámetros:**
   - "*mensaje*" – vector con elementos en F_{p^m} cuya longitud es múltiplo de la longitud del código empleado.
   - "$ASCII$" – ($default$: "*False*") valor booleano que permite realizar o no la transformación del vector descifrado en caracteres ASCII.
   - "$fichero$" – ($default$: "*False*") valor booleano que permite realizar o no la lectura de un fichero que queremos descifrar.

3) **Salida:** Existen cuatro posibles salidas:
   a) Si $ASCII = True$ y $fichero = False$ devuelve una cadena de caracteres ASCII correspondiente al mensaje descifrado.
   b) Si $ASCII = False$ y $fichero = False$ devuelve un vector en F_{p^m} obtenido tras realizar el descifrado de "*mensaje*".
   c) Si $ASCII = True$ y $fichero = True$, crea un fichero en donde almacena una cadena de caracteres ASCII correspondiente al mensaje descifrado.
   d) Si $ASCII = False$ y $fichero = True$, crea un fichero en donde almacena un vector en F_{p^m} obtenido tras realizar el descifrado de "*mensaje*".

4) **Ejemplos:**

$sage : C = codes.GeneralizedReedSolomonCode(GF(16,'a').list(),6)$

$sage : C.minimum\_distance()$

11

$sage : codigo\_mceliece1 = McEliece(C,5)$

$sage : vector1 = vector(GF(16,'a'),[GF(16,'a').random\_element()\ for\ i\ in\ range(6)])$

$sage : vector1$

$(a^3+a, a^3+a+1, 1, a+1, a, a^3+a+1)$

$sage : cifrado = codigo\_mceliece1.cifrado(vector1)$

$sage : cifrado$

$(a, a^3+a^2+a+1, 1, a^3+1, a^3+a^2+a, a^2, a^3+a^2+a+1, a^3+a, a^3+a^2+a, a+1, 1, a^3+a^2+a+1, a^3+a^2+a+1, a^3+a^2, 1, a^2+a+1)$

$sage : codigo\_mceliece1.descifrado(cifrado, False)$

$(a^3+a, a^3+a+1, 1, a+1, a, a^3+a+1)$

$sage : cifrado1 = codigo\_mceliece1.cifrado('hola');$

$sage : cifrado1$

$(a^3+a^2+a, a^3+a^2+a, a+1, a^3+a, a^2+1, 1, a^3+a^2, a^3, a^2+1, a^3+a, a^2+a, a^3+a^2, a^2+a+1, a^2+1, a^3, a^3+1, a^2+1, a+1, a^2, a^3, 1, a^3+a^2+a+1, a^3+1, a+1, a^3+1, a, a^3+1, a^3+1, a+1, a+1, a^2+1, 0)$

$sage : codigo\_mceliece1.descifrado(cifrado1)$

$'hola'$

$sage : codigo\_mceliece1.cifrado('mensajeparaGRS.txt', True);$

(' El mensaje cifrado se ha guardado en el archivo con nombre: ', $'mensajeparaGRS - cif.txt'$)

$sage : codigo\_mceliece1.descifrado('mensajeparaGRS - cif.txt', fichero = True)$

(' El mensaje descifrado se ha guardado en el archivo con nombre: ', $'mensajeparaGRS - cif - des.txt'$)

- $code\_mceliece(self)$

1) **Descripción:** Devuelve el código empleado para la construcción del criptosistema de McEliece.

4) **Ejemplos:**

$sage : C = codes.GeneralizedReedSolomonCode(GF(16,'a').list(), 6)$

$sage : C.minimum\_distance()$

11

$sage : codigo\_mceliece1 = McEliece(C, 5)$

$sage : codigo\_mceliece1.code\_mceliece()$

$[16, 6, 11]$ Reed-Solomon Code over $GF(16)$

- $matriz\_invertible(self)$

1) **Descripción:** Devuelve la matriz invertible aleatoria generada para la construcción del criptosistema de McEliece.

4) **Ejemplos:**

$sage : C = codes.GeneralizedReedSolomonCode(GF(16,'a').list(), 6)$

$sage : C.minimum\_distance()$

11

$sage : codigo\_mceliece1 = McEliece(C, 5)$

$sage : codigo\_mceliece1.matriz\_invertible()$

$[0\,1\,0\,a^2+a\,a^3\,a^2]$

$[a^3+a\,a^2+a\,a+1\,0\,1\,a^3]$

$[a^2\ a^2+1\ 0\ a^3+a^2\ 1\ a^3]$
$[a^3+a^2+a+1\ 0\ a^3+a\ 0\ a+1\ a^2+a]$
$[a^3+1\ a^3+1\ a^3+a^2+1\ a\ a^2+a+1\ a^3+a^2+a]$
$[a\ a^3+a^2+a\ a^3+a+1\ a^3+a^2+a\ a^3+a^2+a+1\ a^3+a^2+1]$

- *matriz_permutacion*(*self*)

  1) **Descripción:** Devuelve la matriz de permutación aleatoria generada para la construcción del criptosistema de McEliece.

  4) **Ejemplos:**
  sage : $C = codes.GeneralizedReedSolomonCode(GF(16,'a').list(),6)$
  sage : *C.minimum_distance*()
  11
  sage : *codigo_mceliece*1 = *McEliece*(*C*,5)
  sage : *codigo_mceliece*1.*matriz_permutacion*()
  [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 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
  [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
  [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 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]
  [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
  [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
  [0 0 0 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 0 0 0 0 0 0 1]
  [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]

- *public_matrix*(*self*)

  1) **Descripción:** Devuelve la matriz publica generada para la construcción del criptosistema de McEliece.

4) **Ejemplos:**
$sage: C = codes.GeneralizedReedSolomonCode(GF(16,'a').list(),6)$
$sage: C.minimum\_distance()$
11
$sage: codigo\_mceliece1 = McEliece(C,5)$
$sage: codigo\_mceliece1.public\_matrix()$
$[a^3 + a\ a^3 + a + 1\ a^3 + a + 1\ a^3 + a^2\ a\ a^3 + a^2 + a + 1\ a^3 + 1\ a^2 + 1\ a^2\ 0\ 0\ a^2 + a + 1\ a^2 + a + 1\ a^3 + a + 1\ 0\ a^3]$
$[a^3 + a^2 + a + 1\ a^3 + a^2 + a\ a^3 + a^2\ a^2 + 1\ a^2\ a^3\ a + 1\ a^3\ a^3 + a^2 + a + 1\ a^3 + a\ a\ a^3 + a^2\ a + 1\ a^2 + a\ a^3 + a^2\ a^3 + a^2 + 1]$
$[a^3 + a^2 + a\ a^2\ a^2\ a\ a^2 + 1\ a^2\ a^3 + a^2 + 1\ a^3 + a^2 + a\ a^3 + a^2\ a^2\ a^3 + 1\ a\ a\ a^2\ a^3 + a^2\ a^2 + a + 1]$
$[a^2 + a + 1\ a^2\ a^3 + a^2\ a^3 + 1\ a^3\ a^3 + a\ a^3 + a\ a^2 + 1\ a^3 + a\ a^3 + a^2 + a + 1\ a^2\ a + 1\ a^3\ 0\ 1\ 0]$
$[0\ a^2 + a + 1\ 0\ a^2 + a\ a^2 + a\ a^3 + a^2\ a^2\ a^3 + a^2 + a + 1\ a^3 + a^2\ a^3 + 1\ a^3 + a^2 + 1\ 1\ a^3 + a^2 + 1\ a^2 + a\ a^3 + a^2 + a + 1\ a^3 + a^2 + 1]$
$[a^3 + a + 1\ a^2 + a\ a^2 + a\ a^3 + a^2 + 1\ a^3 + a^2 + 1\ a\ a^3 + a^2 + 1\ a^3 + a^2 + a\ a^2 + a + 1\ a\ a\ a^2\ a^3 + a^2\ a^3 + a + 1\ a + 1\ a^3 + a^2 + 1]$

- $capacidad\_correctora(self)$

1) **Descripción:** Devuelve el numero de errores que se introducen durante el proceso de cifrado de un vector de tamaño k.

4) **Ejemplos:**
$sage: C = codes.GeneralizedReedSolomonCode(GF(16,'a').list(),6)$
$sage: C.minimum\_distance()$
11
$sage: codigo\_mceliece1 = McEliece(C,5)$
$sage: codigo\_mceliece1.capacidad\_correctora()$
5

- $dimension(self)$

1) **Descripción:** Devuelve la dimension, k, que tiene el código empleado por el criptosistema de McEliece construido.

4) **Ejemplos:**
$sage: C = codes.GeneralizedReedSolomonCode(GF(16,'a').list(),6)$
$sage: C.minimum\_distance()$

11
$sage: codigo\_mceliece1 = McEliece(C, 5)$
$sage: codigo\_mceliece1.dimension()$
6

- $base\_field(self)$

  1) **Descripción:** Devuelve el cuerpo finito F_{p^m} que emplea el código utilizado para construir el criptosistema de McEliece.
  4) **Ejemplos:**
  $sage: C = codes.GeneralizedReedSolomonCode(GF(16,'a').list(), 6)$
  $sage: C.minimum\_distance()$
  11
  $sage: codigo\_mceliece1 = McEliece(C, 5)$
  $sage: codigo\_mceliece1.base\_field()$
  Finite Field in a of size 2^4

B.2.0.4. Ataque por filtración

Por último mostramos la documentación de las cuatro funciones implementadas para realizar el ataque por filtración estándar y el ataque al criptosistema de McEliece que se encuentran en el fichero *"AtaqueFiltracion.ipynb"* para editarlas y en el fichero *"AtaqueFiltracion.sage"* para emplearlas. En primer lugar mostramos las dos funciones necesarias para realizar el ataque estándar:

- $reducir\_codigos(C, C1, recG = False)$

  1) **Descripción:** Permite obtener una matriz generatriz del código $GRS_1(a, b)$, y por tanto recuperar el parámetro b de los códigos $GRS_k(a, b)$ y $GRS_{k-1}(a, b)$ introducidos como parámetros de partida, mediante la obtención de los distintos códigos $GRS_i(a, b)$, con $i = k - 2, ..., 1$. Para ello se realiza el proceso expuesto en la proposición 6,9 y nota 6,10.
  2) **Parámetros:**
     - *"C"* – código $GRS_k(a, b)$ del cual queremos recuperar el vector b.
     - *"C1"* – código $GRS_{k-1}(a, b)$ del cual queremos recuperar el vector b.
     - *"recG"* – (default: "False") valor booleano que permite a la función saber si es necesario devolver en adicción la matriz G, que hace referencia a la parte de la matriz generatriz en forma estándar del código $GRS_2(a, b)$ que no es la matriz identidad.

3) **Salida:** Existen dos posibles salidas:
   a) Si $recG = False$ únicamente devolvemos la matriz con b.
   b) Si $recG = True$ devolvemos una lista de longitud dos compuesta por la matriz b y también por la matriz G.

4) **Ejemplos:**
   $sage : F = GF(11);$
   $sage : a = [F(i)\ for\ i\ in\ [1,2,9,3,10,8,5,4]];$
   $sage : b = [F(i)\ for\ i\ in\ [3,10,5,9,9,10,5,2]];$
   $sage : b1 = [F(i) * 2\ for\ i\ in\ [3,10,5,9,9,10,5,2]];$
   $sage : k = 5$
   $sage : C = codes.GeneralizedReedSolomonCode(a,k,b);$
   $sage : C1 = codes.GeneralizedReedSolomonCode(a,k-1,b1);$
   $sage : reducir\_codigos(C,C1)$
   $[1\,7\,9\,3\,3\,7\,9\,8]$
   $sage : reducir\_codigos(C,C1,True)$
   $[$
   $\qquad\qquad [3\,8\,9\,2\,6\,6]$
   $[1\,7\,9\,3\,3\,7\,9\,8], [4\,4\,7\,7\,2\,5]$
   $]$

- $ataque\_estandar(CCC, CCC1)$

1) **Descripción:** Permite recuperar los parámetros a y b de los códigos $GRS_k(a,b)$ y $GRS_{k-1}(a,b)$ introducidos como parámetros de partida. Para ello emplea la función $reducir\_codigos()$ para obtener el parámetro b y la matriz G que emplea después para resolver un sistema que nos proporciona el parámetro a.

2) **Parámetros:**
   - "CCC" – código $GRS_k(a,b)$ del cual queremos recuperar los vectores a y b.
   - "$CCC1$" – código $GRS_{k-1}(a,b)$ del cual queremos recuperar los vectores a y b.

3) **Salida:** Devolvemos una lista de longitud dos compuesta por el vector a y el vector b respectivamente.

4) **Ejemplos:**
   $sage : F = GF(11);$

$sage: a = [F(i)\ for\ i\ in\ [1,2,9,3,10,8,5,4]]$;
$sage: b = [F(i)\ for\ i\ in\ [3,10,5,9,9,10,5,2]]$;
$sage: b1 = [F(i) * 2\ for\ i\ in\ [3,10,5,9,9,10,5,2]]$;
$sage: k = 5$
$sage: C = codes.GeneralizedReedSolomonCode(a,k,b)$;
$sage: C1 = codes.GeneralizedReedSolomonCode(a,k-1,b1)$;
$sage: parametros = ataque\_estandar(C,C1)$
$sage: parametros$
$[(1,2,9,3,10,8,5,4),\ (1,7,9,3,3,7,9,8)]$
$sage: CCCC = codes.GeneralizedReedSolomonCode([F(a1)\ for\ a1\ in\ parametros[0]], k, [F(b1)\ for\ b1\ in\ parametros[1]])$;
$sage: CCCC.systematic\_generator\_matrix() == C.systematic\_generator\_matrix()$
True

Ahora mostramos la documentación de las otras dos funciones implementadas para poder realiza el ataque por filtración al criptosistema de McEliece, donde la primera de ellas también emplea la función *reducir_codigos()* mostrada anteriormente:

- $ataque\_filtracion(G_pub)$

  1) **Descripción:** Permite recuperar los parámetros a y b que emplea la matriz generatriz o pública G_{pub} que introducidos como parámetros de partida, así como el código $GRS_k(a,b)$ que genera el mismo código. Para ello emplea la función *reducir_codigos()* varias veces y sigue el proceso mostrado en la sección 6,2,2.
     Si $k > \frac{n}{2}$ se recurre al código dual para emplear el método.
  2) **Parámetros:**
     - "G_pub" – matriz generatriz del código $GRS_k(a,b)$ o matriz pública de un criptosistema de McEliece construido a partir de un código $GRS_k(a,b)$, de la cual queremos recuperar los parámetros a, b y el código $GRS_k(a,b)$.
  3) **Salida:** Devolvemos una lista de longitud tres compuesta por el vector a, el vector b y el código $GRS_k(a,b)$ respectivamente.
  4) **Ejemplos:**
     $sage: F = GF(16,'a')$;

$sage: a = [F(i)\ for\ i\ in\ [F.list()[i]\ for\ i\ in\ range(1,11)]];$
$sage: b = [F(i)\ for\ i\ in\ [F.list()[i]\ for\ i\ in\ range(3,13)]];$
$sage: k = 6;$
$sage: C = codes.GeneralizedReedSolomonCode(a,k,b);$
$sage: load("./McEliece.sage");$
$sage: crip\_mceliece = McEliece(C,2)$
$sage: crip\_mceliece$
Criptosistema de McEliece construido a partir de $[10,6,5]$ Generalized Reed-Solomon Code over $GF(16)$ que introduce 2 errores
$sage: G\_pub = crip\_mceliece.public\_matrix()$
$sage: [a2,b2,codigo2] = ataque\_filtracion(G\_pub)$
$sage: codigo2.systematic\_generator\_matrix() == G\_pub.echelon\_form()$
True

- $recuperar\_mensaje(G\_pub, codigo, y)$

  1) **Descripción:** Permite recuperar un mensaje a partir del mensaje cifrado.
  2) **Parámetros:**
     - $"G_pub"$ – matriz generatriz del código $GRS_k(a,b)$ o matriz pública de un criptosistema de McEliece construido a partir de un código $GRS_k(a,b)$, del cual queremos recuperar el mensaje.
     - $"codigo"$ – código $GRS_k(a,b)$ a partir del cual aplicamos el algoritmo de corrección de errores para descifrar el mensaje. Este código se corresponde con el obtenido a partir de la función $ataque\_filtracion()$.
     - $"y"$ – vector que contiene el mensaje cifrado que queremos descifrar.
  3) **Salida:** Devolvemos el vector correspondiente con el mensaje descodificado.
  4) **Ejemplos:**
     $sage: F = GF(16,'a');$
     $sage: a = [F(i)\ for\ i\ in\ [F.list()[i]\ for\ i\ in\ range(1,11)]];$
     $sage: b = [F(i)\ for\ i\ in\ [F.list()[i]\ for\ i\ in\ range(3,13)]];$
     $sage: k = 6;$
     $sage: C = codes.GeneralizedReedSolomonCode(a,k,b);$
     $sage: load("./McEliece.sage");$
     $sage: crip\_mceliece = McEliece(C,2)$
     $sage: crip\_mceliece$

Criptosistema de McEliece construido a partir de $[10, 6, 5]$ Generalized Reed-Solomon Code over $GF(16)$ que introduce 2 errores

$sage: G\_pub = crip\_mceliece.public\_matrix()$

$sage: [a2, b2, codigo2] = ataque\_filtracion(G\_pub)$

$sage: m = vector(F, [F.random\_element() \; for \; i \; in \; range(k)])$

$sage: m$

$(a^2 + 1, a^3 + a^2, a^2 + a, a^3 + a^2 + a, a^2, 1)$

$sage: y = crip\_mceliece.cifrado(m)$

$sage: y$

$(a^3 + a^2, a^3 + a^2 + a + 1, a^3 + a^2 + a, a, a, a^3 + a^2 + 1, a^3 + a^2, a^3, a^3 + a + 1, a^2 + a + 1)$

$sage: recuperar\_mensaje(G\_pub, codigo2, y)$

$(a^2 + 1, a^3 + a^2, a^2 + a, a^3 + a^2 + a, a^2, 1)$