Universidad de Valladolid

Escuela de Ingeniería Informática (SG)

Trabajo Fin de Grado

Grado en Ingeniería Informática de Servicios y Aplicaciones

Acceso confidencial a bases de datos: Implementación en SAGE

Autor: Sergio Díez García

Tutor: José Ignacio Farrán Martín

Índice general

Agradecimientos

A Diego Ruano y Jose Ignacio Farrán, tutores de los dos trabajos fin de grado por ayudarme durante todo el año en la elaboración de los mismos.

Introducción

En la actualidad, el acceso a información almacenada en bases de datos es algo que realizamos continuamente cuando, por ejemplo, estamos navegando en internet, o accediendo a cualquier servicio on-line. Como es lógico, existen gran cantidad de medidas de seguridad para garantizar que estos procedimientos se realicen de manera segura.

Entre ellas, se usan herramientas para protegernos contra una posible tercera persona que esté "escuchando" nuestras operaciones con el fin de robar nuestros datos. También las bases de datos cuentan con sistemas de autorización y autenticación para restringir el acceso a parte de los datos almacenados en su base de datos, de forma que sólo las personas autorizadas para ello sean capaces de extraer cierta información. Sin embargo, en el caso de proteger al propio usuario contra el propietario de la base de datos no se toma ninguna medida de seguridad.

Existen muchos casos donde este nivel de seguridad podría ser realmente útil. Un primer ejemplo podría ser el de un régimen opresor, donde que un usuario consulte cierta información sensible hace que dicho usuario sea añadido a una lista negra. Otro ejemplo podría ser el de una empresa que esté considerando la adquisición de otra más pequeña. Para ello desea consultar el valor de las acciones de esta, además de otros datos financieros. Sin embargo, estas consultas al no ser anónimas, pueden advertir a la empresa pequeña de los posibles planes.

Con esta idea se empezó a investigar recientemente la forma de crear modelos de bases de datos y procedimientos que sean capaces de garantizar este nivel de seguridad a los usuarios. A estos métodos que fueron apareciendo se los denominó protocolos PIR, del inglés Private Information Retrieval (Recuperación Privada de la Información).

En el caso de una base de datos formada por un único servidor, el único protocolo PIR posible consiste en descargar la totalidad de la base de datos, cosa que queremos evitar ya que resulta muy costoso. Es por ello, que a la hora de desarrollar un protocolo PIR, inicialmente debemos desarrollar un modelo de base de datos en el que existan varios servidores. A partir de este modelo se explica una manera de realizar solicitudes a los diferentes servidores para recuperar cierto archivo que deseamos. Desde el punto de vista de los servidores de la base de datos, la información que se le está solicitando no guarda ninguna relación con ninguno de los archivos que contiene. Sin embargo, desde el punto de vista del usuario, al agrupar las respuestas a las solicitudes de cierta forma, éste será capaz de recuperar el archivo que deseaba de forma privada.

Pongamos un primer ejemplo con una base de datos formada por dos servidores. La base de datos estará formada por m archivos, donde cada uno de ellos es un elemento de un cuerpo finito $\mathbb{F}_q$. Podemos representar dicha base de datos como un vector x de longitud m sobre

$\mathbb{F}_q$. Supondremos entonces que cada uno de los dos servidores tienen almacenados la totalidad de los datos, es decir, el vector x. El siguiente procedimiento nos permitirá recuperar el archivo que deseemos x_f.

En primer lugar, empezamos generando un vector aleatorio , $u \in \mathbb{F}_q^m$. A partir de este vector le haremos una solicitud al primero de los servidores. Le solicitaremos el producto escalar entre u y los datos que él posee. Su respuesta será por tanto $r_1 = x \cdot u \in \mathbb{F}_q$. El tamaño de la solicitud es pequeño ya que está formada por el vector u únicamente. Desde el punto de vista de este servidor, no estamos interesados en ninguno de los archivos en concreto, sino en una combinación de todos los datos, que no guarda ninguna relación con alguno de ellos particularmente. Al segundo servidor, en cambio, le solicitaremos el producto escalar entre sus datos y el vector $u + e_f$, donde e_f se trata del vector canónico con un uno en la posición f. Desde el punto de vista de este servidor tampoco estamos interesados en ningún archivo en concreto. Su respuesta será $r_2 = x \cdot (u + e_f) \in \mathbb{F}_q$. Sin embargo, al restar las dos respuestas podremos obtener x_f, ya que $r_2 - r_1 = x \cdot (u + e_f) - x \cdot u = x \cdot e_f = x_f$.

El principal problema de este protocolo es que si ambos servidores se comunican entre ellos, al comparar las dos solicitudes que han recibido, podrían llegar a deducir, usando el mismo procedimiento que hemos hecho nosotros, que estamos interesados en el archivo x_f. A estos dos servidores que son capaces de comunicarse entre ellos los llamamos servidores cooperantes y presentan un grave problema en la elaboración de estos protocolos. En este ejemplo en concreto, nuestro protocolo solo funciona cuando la cantidad de servidores cooperante es a lo sumo $b = 1$.

Si para este mismo ejemplo $b = 2$ la única forma válida de obtener el archivo x_f de forma privada sería descargarse la base de datos al completo, ya que cualquier conclusión que podamos sacar nosotros a partir de las respuestas, también podrían obtenerla los propios servidores. Por la misma razón, en cualquier base de datos que esté formada por un único servidor, esta será la única solución.

Es por ello que estos protocolos solamente funcionan en bases de datos formadas por varios servidores, las llamadas bases de datos distribuidas. De hecho, este tipo de base de datos es la más frecuente hoy en día ya que facilita el acceso a datos en cualquier parte del mundo.

En este ejemplo que hemos visto se usa una base de datos en la que toda la información está almacenada en cada uno de los servidores. Se conoce como una base de datos replicada. Este tipo de bases tiene la ventaja de que contactando con cualquier servidor podremos acceder a la totalidad de la información. No obstante, el principal problema de este tipo, es que si el tamaño de dicha base es muy elevado, replicarla en muchos servidores puede ser muy costoso.

Por ello, se empezó a usar un segundo tipo de bases de datos distribuidas. En este nuevo tipo, cada servidor únicamente contiene parte de la información total, así que en todos los casos, para obtener la totalidad de un archivo deberemos contactar con varios servidores. Para garantizar ciertas propiedades, la réplica de la información no está hecha de forma aleatoria, sino usando un código lineal. Cada archivo se divide en varios fragmentos que son posteriormente codificados usando un código lineal. Gracias a ello, el usuario podrá recuperar la información que desee contactando con cierto número de servidores cualesquiera. Además el código lineal nos dará también la opción de corregir errores y borrones que puedan surgir en la transmisión de la información.

Este documento será un proyecto de investigación en relación a los protocolos PIR con

una base de datos con las características explicadas. En él, nos centraremos en explicar detalladamente, analizar e implementar los protocolos descritos en cuatro artículos:

- R. Tajeddine, O.W. Gnilke, S. El Roayheb *Private Information Retrieval From MDS Coded Data in Distributed Storage Systems*, IEEE Transactions on Information Theory 64 (2018) 7081-7093 [5]
- R. Freij-Hollanti, O. Gnilke, C. Hollanti, D. Karpuk, *Private Information Retrieval from Coded Databases with Colluding Servers*, SIAM J. Appl. Algebra Geometry 1 (2017), 647–664 [6]
- R. Tajeddine, O.W. Gnilke, D. Karpuk, R. Freij-Hollanti, C. Hollanti, *Robust Private Information Retrieval from Coded Systems with Byzantine and Colluding Servers*, arXiv 1802.03731 [7]
- R. Tajeddine, O.W. Gnilke, D. Karpuk, R. Freij-Hollanti, , *Private Information Retrieval from Coded Storage Systems with Colluding, Byzantine, and Unresponsive Servers*, IEEE Transactions on Information Theory 65 (2019) 3898 - 3906 [8]

En cuanto a la organización de este documento, se hará de la siguiente forma.

El primer capítulo analizaremos la organización que realizaremos durante la elaboración del proyecto. En primer lugar decidiremos el tipo de metodología que usaremos en el desarrollo de la investigación e implementación. A continuación identificaremos cuales serán las distintas etapas en el ciclo de vida del proyecto, dando una explicación de en qué consistirá cada una de ellas. Más adelante explicaremos las herramientas de trabajo que usaremos en el transcurso de este proyecto de investigación. Por último analizaremos el trabajo a realizar identificando los roles más importantes que se deben asumir y realizaremos la distribución del trabajo a lo largo del tiempo.

En el segundo capítulo explicaremos un conjunto de conocimientos iniciales necesarios para la comprensión del funcionamiento de los protocolos PIR. Inicialmente introduciremos el software que usaremos en la implementación de los protocolos *Sage* 2.1 y su entorno principal de trabajo. El siguiente paso consiste en la explicación de la teoría de códigos más básica. Además acompañaremos a los nuevos introducidos con su implementación en *Sage* por librerías ya creadas. Por último indagaremos en una sección de la teoría de códigos, los códigos MDS, donde veremos dos tipos de códigos que usaremos muy a menudo en otras secciones del proyecto.

Una vez realizada la explicación de los conceptos necesarios para la comprensión de los protocolos, empezaremos con su investigación. En el tercer capítulo nos centraremos en los protocolos PIR en general, viendo concretamente en qué consisten y qué los caracteriza. Además analizaremos especialmente el tipo de base de datos que necesitamos para poder llevar a cabo recuperación de información de forma privada. Tras esto caracterizaremos cual es modelo que usaremos de base de datos en los protocolos que estudiaremos a lo largo del documento. Por último identificaremos las etapas comunes a todos los protocolos PIR para realizar una implementación de las mismas.

En el cuarto capítulo empezaremos viendo casos concretos de protocolos PIR. En este caso veremos 3 protocolos para el caso en el que la base de datos utilizada ha sido codificada mediante un código MDS y sistemático, introducidos en [5]. Para cada uno de ellos

analizaremos las diferencias que guardan entre ellos en cuanto a eficiencia y protección contra servidores cooperantes. Veremos cómo se realizan las solicitudes y el procedimiento mediante el cual somos capaces de recuperar un archivo tras recibir las respuestas. En ambos casos realizaremos una implementación en *Sage* que realice dichas tareas.

En el siguiente capítulo ampliaremos la casuística mediante un cuarto protocolo, que nos permitirá le recuperación privada de información para una base de datos codificada usando un código arbitrario. Sin embargo para poder comprender el funcionamiento de este y el resto de protocolos de secciones posteriores introduciremos una nueva operación entre vectores y códigos, el producto estrella. Esta operación nos dará una mayor libertad en la creación de protocolos PIR, resultando en la creación de este cuarto protocolo [6]. Al igual que con los tres anteriores identificaremos las diferencias con los protocolos ya vistos y explicaremos la creación de las solicitudes y recuperación del archivo, además de realizar una implementación de ambas operaciones. Por último investigaremos en el caso concreto en el que el código empleado para codificar la base de datos se trate de un código GRS.

En el sexto capítulo introduciremos los dos últimos protocolos. Estos dos nuevos protocolos funcionan para un caso más real en el que ruido en el canal puede alterar las respuestas de los servidores, creando errores y borrones en las mismas. Analizaremos las distintas formas en las que podemos resolver este problema para concluir con los protocolos 5 [7] y 6 [8]. De la misma forma que anteriormente, explicaremos los rasgos más importantes de estos protocolos, concluyendo con la implementación de los mismos. En este caso también investigaremos el caso particular en el que el código utilizado es un código GRS.

Además de la explicación de los protocolos, en todos los casos se han acompañado de ejemplos para facilitar su comprensión. En algunos casos, los ejemplos han sido realizados en *Sage* para realizar de forma más rápida los cálculos.

Al final del documento, se incluye además un anexo con un manual de usuario para todas las funciones implementadas a lo largo del proyecto de investigación.

Objetivos

A continuación presentamos los objetivos principales que se desean alcanzar durante la realización de este documento:

- Estudiar la teoría básica de códigos lineales
- Conocer los códigos Reed-Solomon y Reed-Solomon Generalizados, tanto su construcción como sus principales algoritmos de codificación y decodificación.
- Aprender los conceptos básicos de teoría de información necesarios para comprender el concepto de "privacidad" en los protocolos PIR.
- Aprender los conceptos básicos del lenguaje de programación Python
- Conocer la funcionalidad que ofrece el lenguaje de programación *Sage* [9] en cuanto a códigos lineales, códigos Reed-Solomon y otras funciones básicas del mismo.
- Implementar en *Sage* métodos básicos para la codificación de archivos en bases de datos usando códigos lineales.

- Leer los cuatro artículos [5],[6],[7],[8] donde se describen los principales protocolos PIR para bases de datos distribuidas codificadas usando un código lineal. Para cada uno de estos protocolos se realizará una descripción detallada de la base de datos, se explicará la construcción de las solicitudes y del análisis de las respuestas. Por último se demostrará su funcionamiento.
- Analizar la eficiencia de cada uno de estos protocolos y estudiar la protección que se realiza contra la presencia de servidores cooperantes.
- Realizar la implementación en *Sage* de cada uno de estos protocolos PIR.

Capítulo 1

Metodología de trabajo

En este capítulo inicial realizaremos la planificación del trabajo, estructurando las diferentes etapas en las que lo dividiremos.

En primer lugar, tendremos que elegir el tipo de metodología que usaremos en el desarrollo. La metodología elegida será una de las metodologías tradicionales, ya que se adecuan más al tipo de trabajo que debemos hacer. Más en concreto, usaremos un desarrollo iterativo. Esto se debe a que empezaremos estudiando e implementando las funciones fundamentales que necesitaremos en otras etapas del proyecto. Una vez desarrolladas estas funciones iniciales, cada protocolo nuevo que veamos supondrá una iteración que añadirá nueva funcionalidad.

Este tipo de metodología además será una metodología secuencial. Cada una de estas iteraciones que iremos realizando con cada uno de los protocolos supondrán un incremento en la funcionalidad del programa, pero en todos los casos, estos incrementos necesitarán los conocimientos adquiridos y funciones realizadas en iteraciones anteriores.

1.1. Ciclo de vida del proyecto

Dividiremos el proyecto en las siguientes etapas:

1. En la primera etapa realizaremos un estudio de los conceptos necesarios para entender y desarrollar los distintos esquemas PIR. Esta etapa tiene un desarrollo secuencial, ya que en primer lugar realizaremos un estudio sobre códigos lineales, que luego necesitaremos para comprender qué es y como funciona un código GRS, o cómo realizar el producto estrella entre dos códigos lineales o dos códigos GRS.

   De forma simultánea a este estudio, estudiaremos las carácteristicas más básicas de Python, para también estudiar también el lenguaje Sage. En este último veremos la funcionalidad ofrecida por las distintas librerías en relación a los códigos lineales.

2. En la segunda etapa nos centraremos de forma más concreta en los protocolos PIR. En primer lugar los estudiaremos de forma teórica, específicamente en qué los caracteriza y cómo podemos definir el concepto de privacidad. También realizaremos un estudio sobre el tipo de base de datos que debemos usar para uno de estos protocolos y qué características debe tener. A continuación analizaremos como se puede realizar

una implementación de un protocolo PIR general en Sage. Tras este análisis seremos capaces de identificar procesos comunes a todos los protocolos PIR, en concreto tres de ellos, el proceso de creación de la base de datos, el del cálculo de las respuestas a las solicitudes y el de decodificación de un archivo.

Por último estudiaremos como podemos realizar la implementación de cada uno de estos métodos de forma consistente entre ellos. Tras realizar la implentación de cada uno de ellos, realizaremos una serie de pruebas para comprobar que todos funcionan de forma deseada.

3. En esta etapa empezaremos a estudiar casos concretos de protocolos PIR. En primer lugar nos centraremos en los casos más simples de bases de datos. En estos casos, la base de datos será codificada usando un código MDS y sistemático. Estudiaremos tres protocolos distintos. Para cada uno de ellos estudiaremos como realizar las solicitudes a la base de datos y como recuperar el archivo que deseamos a partir de las respuestas a estas solicitudes. Además analizaremos las principales diferencias entre estos tres protocolos en dos ámbitos concretos, la protección que ofrecen contra servidores cooperantes y la eficiencia de los mismos.

   Además analizaremos cómo realizar la implementación de las funciones de creación de solicitudes y recuperación del archivo para un protocolo arbitrario, de forma que los argumentos y salidas sean consistentes con los de las funciones ya implementadas en etapas anteriores. Tras este análisis realizaremos la implementación de estos dos métodos para cada uno de los tres protocolos que hemos estudiado en esta etapa.

4. Ahora en la cuarta etapa, abordaremos el caso de una base de datos más general codificada usando un código C arbitrario. Estudiaremos como podemos adaptar los protocolos que hemos estudiado para este caso. Para lograrlo necesitaremos estudiar una nueva operación entre códigos lineales, el producto estrella. Además de estudiar su funcionamiento y propiedades, también estudiaremos qué nuevas propiedades surgen cuando los códigos involucrados son códigos GRS.

   A continuación seremos capaces de estudiar teóricamente un nuevo protocolo PIR para una base de datos con estas propiedades. Veremos al igual que en casos anteriores cómo crear las solicitudes y recuperar el archivo a partir de las respuestas. Tras ello, realizaremos la implentación de cada uno de estos métodos de forma consistente con lo ya implementado.

5. En esta etapa nos centraremos en un caso más real, en el que se pueden cometer errores y borrones en la transmisión de las respuestas de los servidores. En primer lugar veremos como podemos adaptar las funciones básicas ya implementadas para soportar este caso. Como es lógico los protocolos vistos en etapas anteriores no son efectivos en estos casos y por tanto será necesario desarrollar nuevos. En concreto, estudiaremos dos protocolos PIR que nos permitirán corregir errores y borrones en este caso.

   Al igual que en etapas anteriores, realizaremos la implementación de tanto la función para generar las solicitudes, como la función para recuperar el archivo a partir de las respuestas.

6. En la sexta y última etapa hemos realizado una gran variedad de ejemplos y pruebas para poder confirmar el funcionamiento correcto de las funciones implementadas

1.2. Herramientas utilizadas

En el desarrollo de este trabajo, hemos realizado dos principales herramientas

1. ***Sage* y el entorno de trabajo *Jupyter***: En el desarrollo de este trabajo hemos usado el lenguaje de programación *Sage*, un lenguaje basado fundamentalmente en *Python*, además de incluir una gran cantidad de otros lenguajes de gran utilidad en las matemáticas como *Singular* o *GAP*. Para desarrollar los distintos programas hemos usado el entorno de trabajo *JupyterNotebook*. Simplemente tras la instalación de *Sage*, tendremos disponible este entorno en el programa *SageNotebook*. Este entorno de trabajo se despliega en el puerto 8888 de nuestro ordenador y se accede a través de cualquier navegador web. En él, podemos crear hojas de trabajo. En estos documentos con formato *.ipynb* podemos introducir fragmentos de código en *Sage* para obtener su salida, y también fragmentos de texto en lenguaje LaTeX para añadir explicaciones a los cálculos que realizamos. Una gran utilidad que posee este entorno es la posibilidad de descargar las hojas de trabajo como documentos en otras extensiones como *PDF*, o LaTeX, este último en especifico muy útil en la creación de este documento.
2. ***TexMaker***: Este programa es un editor de texto con el que podemos crear documentos a partir del lenguaje LaTeX. Es un editor gratuito que además de LaTeX, soporta una gran cantidad de lenguajes e idiomas. En este editor se ha escrito este documento.

1.3. Planificación y distribución del trabajo

En la elaboración de este proyecto solamente participa un trabajador, que será el encargado de realizar todas las etapas y tareas ya especificadas. Podemos diferenciar tres roles principales que debe asumir este trabajador. En primer lugar, debe ejercer como jefe de proyecto. Las tareas fundamentales que debe realizar con respecto a este rol son el estudio de las distintas fuentes de documentación sobre el tema a tratar, la selección del contenido a tratar en el proyecto, la planificación de las distintas etapas del proyecto y la distribución de las tareas a lo largo del tiempo.

El segundo rol que debe asumir el trabajador es el de analista. Es necesario un estudio profundo de la documentación para decidir cual es la forma óptima de implementación. Además debe realizar el proceso de documentación de la teoría en la que se basan los protocolos, así como la realización del manual de usuario para las funciones implementadas.

El tercer y último rol que debe asumir el trabajador es de programador. Las labores de este rol consiste en la programación de las funciones indicadas en el proceso de análisis.

En cuanto a la distribución del trabajo, se ha empleado una distinta cantidad de horas dependiendo del mes de año. La distribución aproximada de horas que se ha seguido es la siguiente.

Entre los meses de octubre y noviembre del año 2019 se ha destinado unas 30-35 horas semanales de media. A continuación, a partir de diciembre y hasta febrero el tiempo fue reducido hasta una media de 5-6 horas semanales. Entre los meses de marzo y julio se

aumentó ligeramente la cantidad de horas empleadas en la elaboración del proyecto hasta 10-12 horas semanales. Por último entre los meses de julio y septiembre se volvieron a aumentar las horas invertidas en el desarrollo hasta 40-45 horas semanales. En total la cantidad de horas invertidas han sido entre 715 y 837 aproximadamente.

Capítulo 2

Códigos Lineales y SageMath

2.1. SageMath

Como se ha explicado en el capítulo anterior el lenguaje de programación que se usará para la implementación de los diversos métodos en este trabajo es SageMath o Sage. Este lenguaje de programación es un software libre basado en Python, por lo que podremos usar las funciones habituales de Python además de las específicas de Sage. Hemos elegido este lenguaje ya que cuenta con un conjunto de librerías realmente útiles para trabajar con códigos lineales y demás estructuras algebraicas, algo que necesitaremos en gran medida. Usaremos Sage para simular una base de datos distribuida basada en un código lineal, a la que podremos hacer solicitudes. Este código lineal que usaremos podrá ser un código lineal cualquiera o también un código Reed-Solomon o Reed-Solomon Generalizado, códigos que en ambos casos están ya implementados en Sage. Utilizaremos Sage para desarrollar unos protocolos que nos permitirán a partir de este código crear una serie de solicitudes que haremos a la base de datos para obtener de forma privada parte de los datos almacenados en ella.

Para desarrollar los diferentes protocolos en el lenguaje Sage usaremos el entorno de trabajo *Jupyter Notebook*. Este entorno de trabajo es una interfaz web que se despliega en el puerto 8888 de localhost, desde donde podremos crear, editar y ejecutar cualquier documento con la extensión adecuada. Esta extensión debe ser .ipynb. Una vez abierto un documento con dicha extensión veremos ciertas celdas donde podremos escribir. En una celda podemos escribir tanto código para ser ejecutado como texto plano. Podemos ejecutar las celdas con código mendiante el atajo de teclado ctrl + enter y veremos bajo la celda los valores devueltos tras la ejecución. Las celdas de texto plano podemos usarlas para realizar anotaciónes entre diversas partes de código. Podemos ver un ejemplo de esta distribución en la figura 2.1.

Como hemos dicho, Sage cuenta con una gran cantidad de librerías para implementar diversas estructuras y procedimientos matemáticos. En nuestro caso, usaremos únicamente una muy pequeña parte de estas librerías. Las más importantes que usaremos y que veremos en más detalle más adelante son las siguientes:

- **sage.coding**: Esta librería contiene la implementación de los códigos lineales y sus métodos, así como diversos subconjuntos de códigos lineales de gran interés como

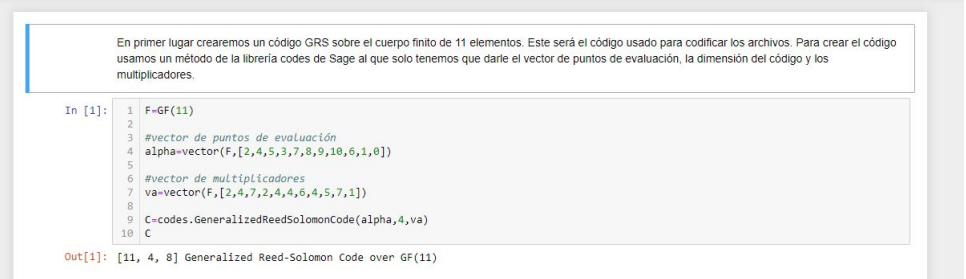

Figura 2.1: Entorno gráfico Jupyter Notebook con una primera celda de texto plano y una celda de código con su output

los códigos Reed-Solomon y códigos Reed-Solomon Generalizados y opraciones sobre códigos como puntear.

- **sage.rings**: Esta librería contiene la implementación de la estructura algebraica de los anillos. Nosotros únicamente estamos interesados en los cuerpos, ya que todo código lineal se construye en referencia a uno de ellos, sin embargo, en Sage, un cuerpo está definido como una especificación de un anillo por lo que podemos realizar sobre éste cualquier operación que también podamos sobre un anillo.
- **sage.matrix**: Esta librería contiene la implementación de las matrices junto con un conjunto de métodos muy útiles para manipularlas.
- **sage.functions** Esta librería contiene una implementación de funciones entre dos espacios algebraicos. En algunos casos necesitaremos definir funciones entre dos cuerpos distintos y para ello haremos uso de esta librería.

Por último, Sage usa otros lenguajes de programación para el desarrollo de ciertas funciones. Estos lenguajes también pueden ser usados a través del propio Sage. En este trabajo no usaremos explícitamente ninguno de estos pero dado su importancia en el mundo de las matemáticas cabe destacar *Singular, GAP y R.*

2.2. Cuerpos y Códigos Lineales

Una de las librerías más importantes que tiene Sage y con la que trabajaremos constantemente es la biblioteca de **sage.rings**. Representaremos los elementos de la base de datos mediante elementos de un cuerpo finito y por tanto debemos de ser capaces de trabajar con ellos. En este documento daremos la definición de un cuerpo finito y algunas de las propiedades pero no haremos demasiado hincapié en el tema.

Definición 2.1. Un cuerpo es una estructura algebraica formada por una terna $\mathbb{F} = (A, +, \cdot)$, donde A es un conjunto y $+, \cdot$ son funciones $+ : A \times A \to A$, $\cdot : A \times A \to A$ que cumplen las propiedades de conmutatividad, asociatividad y de existencia de elemento neutro y de inverso, además de la propiedad distributiva.

Teorema 2.2. *La cardinalidad de un cuerpo finito es una potencia de un número primo.*

Los cuerpos finitos se definen únicamente por su número de elementos u orden, es decir, dos cuerpos finitos con el mismo orden son isomorfos. Si el orden de un cuerpo $\mathbb{F}$ es primo el cuerpo se trata de $\mathbb{Z}$ modulo $orden(\mathbb{F})$. En caso de ser una potencia de un numero primo $q = p^n$, el cuerpo se trata del cociente del conjunto de polinomios sobre el cuerpo de orden p, modulo $p(x)$, donde $p(x)$ puede ser cualquier polinomio irreducible de grado n. Como hemos dicho, la elección de $p(x)$ es arbitraria pero por sus buenas propiedades se suelen usar los llamados *polinomios de Conway*.

En *Sage* podemos crear un cuerpo finito con el siguiente comando:

```
In [ ]: GF(order,name,modulus,impl)
```

donde el único parámetro obligatorio es el primero, el orden o cardinalidad del cuerpo finito. El parámetro *name* es un String que nos permite darle un nombre a la variable de los polinomios que forman parte del cuerpo. *modulus* nos permite especificar el polinomio $p(x)$ con el que tomaremos cociente. Si no se especifica se toma por defecto el polinomio de Conway. Por último *impl* nos permite especificar si deseamos implementar los elementos de una forma distinta a la explicada anteriormente. En este trabajo siempre usaremos los valores por defecto, especificando únicamente el orden. Este método devolverá dicho cuerpo finito que podremos guardar en una variable.

Teorema 2.3. *Sean $\mathbb{F}_1, \mathbb{F}_2$ dos cuerpos finitos de ordenes $q_i = p_i^{n_i}$ $i = 1, 2$ respectivamente. Entonces $\mathbb{F}_1 \subseteq \mathbb{F}_2$ si y solo si $p_1 = p_2$ y $n_2 = n_1 \cdot a$ para cierto entero a. En este caso podemos expresar $\mathbb{F}_2$ como un espacio vectorial sobre $\mathbb{F}_1$ de dimensión a.*

Sage interpreta la contención automáticamente, es decir que interpretará un elemento de $\mathbb{F}_1$ dependiendo del cuerpo en el que estamos trabajando. Por ejemplo, si multiplicamos dos elementos de dos cuerpos F, G con $F \subseteq G$, *Sage* usará la imagen del elemnto de F por la aplicación de contención antes de realizar el producto.

Además *Sage* también nos permite obtener la representación de un cuerpo G como espacio vectorial sobre otro cuerpo F de la siguiente forma:

```
In [ ]: G.vector_space(F,map=true)
```

Este método devuelve tres valores, en primer lugar el espacio vectorial V sobre F al que G es isomorfo, en segundo lugar el isomorfismo desde G a V y en tercer lugar su aplicación inversa. Gracias a esta forma podremos alternar entre las diferentes implementaciones de G. Para usar las aplicaciones devueltas por la función lo haremos usando paréntesis:

```
In [ ]: funcion(x)
```

donde x es el elemento del que queremos obtener la imagen por la aplicación $funcion$.

A continuación empezaremos a explicar los códigos lineales y sus propiedades, parte indispensable en la que se basa gran parte del trabajo.

Definición 2.4. Sea $\mathbb{F}$ un cuerpo finito y sea $\mathbb{F}^n$ el conjunto de n-uplas sobre $\mathbb{F}$. Entonces definimos un código lineal C como un subespacio vectorial no vacío de $\mathbb{F}^n$. Llamamos una palabra código a cualquier elemento de C. Definimos la longitud de un código lineal como la longitud de las palabras código, es decir, n. La dimensión k de C la definimos como su dimensión como espacio vectorial. Dados estos parámetros, podemos decir que C es un código de tipo $[n, k]$.

En *Sage* tenemos dos formas de definir un nuevo código lineal.

- Usando una matriz de elementos de la base: Podemos agrupar los k elementos de una base del código lineal que deseamos crear en una matriz de tamaño $k \times n$ con un elemento por cada fila (Matriz Generatriz, ver 2.8). A partir de esta matriz G podemos crear el código C de la siguiente forma:

  ```
  In [ ]: codes.LinearCode(G)
  ```

- Usando los parámetros que deseamos que tenga. Hemos visto tres parámetros, el cuerpo base $\mathbb{F}$, la dimensión k y la longitud n. Estos parámetros no definen unívocamente un único código, pero podemos obtener uno aleatorio que con esos parámetros de la siguiente forma:

  ```
  In [ ]: codes.random_linear_code(F,n,k)
  ```

Una vez creado un código C podemos recuperar sus parámetros de forma muy sencilla usando los siguientes métodos:

```
In [ ]: C.base_field()
In [ ]: C.dimension()
In [ ]: C.length()
```

Definición 2.5. Sea C un código lineal de tipo $[n, k]$. Un codificador de C es una aplicación inyectiva

$$\mathcal{E} : \mathbb{F}_q^k \to \mathbb{F}_q^n$$

de forma que $C = \mathcal{E}(\mathbb{F}_q^k)$. De esta forma existe una única *palabra fuente* $m \in \mathbb{F}_q^k$ tal que $c = \mathcal{E}(m)$.

De la definición podemos deducir que existen muchos codificadores distintos. Algunos específicos vienen ya implementados en *Sage*. Dependiendo del tipo de código habrá una mayor o menor cantidad de ellos. Para ver los disponibles podemos usar el siguiente método:

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

En el caso de los códigos lineales, solamente hay dos: GeneratorMatrix y Systematic. Podemos obtener cualquiera de los codificadores disponibles usando el siguiente comando, donde *encoder_name* debe ser uno de los codificadores disponibles y *args* son los argumentos necesarios que variarán dependiendo del codificador deseado.

```
In [ ]: C.encoder(encoder_name,args)
```

Una vez elegido un codificador existen dos operaciones fundamentales que podemos hacer con él: codificar y obtener la palabra fuente correspondiente a una palabra código (decodificar es una operación distinta). Podemos realizar ambas operaciones desde el propio código o también a partir de un codificador de la siguiente forma:

- A partir del propio código

  ```
  In [ ]: C.encode(word,encoder_name,args)
  In [ ]: C.unencode(word,encoder_name,args)
  ```

  En este caso los dos últimos parámetros son opcionales. Se usará el codificador predeterminado para el código si no se especifica otro en los parámetros.

- A partir de un codificador E

  ```
  In [ ]: E.encode(word)
  In [ ]: E.unencode(word)
  ```

Definición 2.6. La distancia de Hamming entre dos vectores $x, y \in \mathbb{F}^n$ se define como el numero de posiciones en el que difieren

Definición 2.7. La distancia mínima d de un código C se define como:

$$d = \text{mín}\{d(x, y)|x, y \in C, x \neq y\}$$

donde la distancia usada es la distancia de Hamming

La distancia mínima es una métrica de cómo de distintas son las palabras código y nos permitirá corregir errores y borrones en ellas. Podemos obtener la distancia mínima de un código en Sage de la siguiente forma:

```
In [ ]: E.minimun_distance(algorithm)
```

donde *algorithm* es un parámetro opcional para indicar un algoritmo que no sea el predeterminado. Es importante explicar que la distancia mínima es un parámetro computacionalmente muy difícil de calcular en un código arbitrario, por ello, es algo que debemos evitar en todo momento y solo usar cuando sea necesario. Sin embargo, una vez calculada por primera vez, el valor se guardará en el código, accediendo a él si se desea volver a llamar al método. De forma contraria, para algunos grupos de códigos, como los Reed-Solomon, ésta se puede calcular a partir de sus parámetros principales de forma muy sencilla y por tanto no supone ningún problema realizar llamadas al método.

A principio del capítulo, vimos que una de las formas de definir un código lineal era agrupando los distintos generadores del subespacio lineal en una matriz. Esta es la llamada matriz generatriz del código.

Definición 2.8. Una matriz de tamaño $k \times n$ con entradas en $\mathbb{F}_q$ se denomina una matriz generatriz de un código lineal si sus filas forman una base de C

De la propia definición vemos que esta matriz generatriz no es única ya que hay múltiples bases de un subespacio lineal. De hecho, si ya conocemos una matriz generatriz, todas las matrices equivalentes por filas también generaran el mismo subespacio vectorial y por tanto el mismo código. Sin embargo sabemos que hay únicamente una matriz escalonada por filas que es equivalente a todas las matrices generatrices. Decimos que esta matriz es la matriz en forma sistemática del código. Podemos obtenerla de la siguiente forma:

```
In [ ]: E.systematic_generator_matrix()
```

Definición 2.9. Decimos que dos códigos C, D son equivalentes si existe una permutación σ de n elementos tal que $C = \{\sigma(d) | d \in D\}$

Definición 2.10. Un código es sistemático si y solo si las primeras k columnas de su matriz generatriz son linealmente independientes

Ahora podemos combinar estas tres últimas definiciones para obtener un resultado bastante importante

Proposición 2.11. *Todo código lineal es equivalente a un código sistemático. Además la matriz generatriz se puede conseguir sistemática.*

La idea de la prueba se basa en que al ser la matriz generatriz de rango k siempre habrá k columnas independientes que podemos mover al principio mediante una permutación, obteniendo un código sistemático.

A continuación veremos que la matriz generatriz no es la única matriz capaz de definir un código. Un subespacio siempre se puede definir explícitamente (usando sus generadores) o implictamente (dando una serie de ecuaciones que cumplen todos los puntos). Usando la segunda forma obtenemos lo siguiente

Definición 2.12. Una matriz de tamaño $(n-k) \times n$ de rango máximo es una matriz de control de un código C de tipo $[n, k]$ si para todo $c \in \mathbb{F}_q^n$ se cumple que $Hc^T = 0$ si y solo si $c \in C$.

En sage podemos obtener la matriz de control de un código mediante el siguiente método:

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

La siguiente proposición nos proporciona un método de calcular la distancia mínima más eficaz que comprobar todos los pares de palabras-

Proposición 2.13. *Sea C con matriz generatiz sistemática. Entonces las matrices generatriz G y de control H son de la siguiente forma para cierta matriz A:*

$$G = \begin{pmatrix} I & | & A \end{pmatrix} \quad H = \begin{pmatrix} -A^T & | & I \end{pmatrix}$$

Proposición 2.14. *Sea H la matriz de control de cierto código C. Entonces la distancia mínima d de C es el menor entero d tal que existen d columnas linealmente dependientes en H*

Definición 2.15. El código dual de un código C se define de la siguiente forma:

$$C^{\perp} = \{x \in \mathbb{F}_q^n | c \cdot x = 0, \forall c \in C\}$$

En *Sage* podemos obtener el código dual de un código C mediante el siguiente método:

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

Sin embargo usando la siguiente proposición resulta muy sencillo obtenerlo a partir de la matriz de control de C

Proposición 2.16. *La matriz de control de un código C genera su código dual $C^{\perp}$*

Los códigos lineales se denominan en muchas fuentes códigos correctores y esto se debe (como el nombre indica) a que tienen ciertas propiedades que les permite corregir errores y borrones. En las siguientes páginas veremos como esto se consigue.

Para ello nos podemos poner en el contexto de una transmisión de información. Hemos recibido una palabra $r' \in \mathbb{F}_q^n$ que ha sido codificada a partir de un código C y queremos obtener la palabra fuente original de $\mathbb{F}_q^k$. Como en una transmisión puede haber interferencias el vector que hemos recibido puede no ser el que ha sido enviado $r \in \mathbb{F}_q^n$. Es posible que las interferencias hayan modificado ciertas posiciones del vector. A estas modificaciones las llamaremos errores. Podemos representarlos en un vector de errores e. Este será un vector de $\mathbb{F}_q^n$ definido como $r' - r$. Este vector tendrá un 0 en todas la coordenadas que hemos recibido correctamente y será distinta de 0 en las que no, pero, como es lógico nosotros no sabemos de cuales se tratan. Daremos una definición para poder dar rigurosamente el número de errores.

Definición 2.17. Llamamos el peso de Hamming de una palabra $c \in \mathbb{F}_q^n$ al número de coordenadas no nulas de c

El número de errores será el peso de Hamming del vector e. Llamaremos decodificar a obtener la palabra original del código que fue enviada. Como es lógico no podemos saber el número de errores que se ha cometido y por tanto cualquier palabra pudo ser la enviada si el número de errores fue muy elevado. Sin embargo, si ponemos una cota sobre el número de errores, podremos decodificar la palabra recibida de forma única. El procedimiento para decodificar una palabra r' es hallar la palabra del código más cercana a ella, en términos de la métrica de Hamming. En muchos casos es muy posible que no haya una única palabra más cercana, en cuyo caso diremos que la decodificación ha fallado, pero si imponemos la condición de que el número de errores cometidos es a lo sumo $t = \lfloor (d-1)/2 \rfloor$, entonces la decodificación será única. Esto se debe a que las palabras del código tienen una distancia entre ellas de al menos d, que siempre será mayor que $2t$. A este valor t lo llamaremos la capacidad correctora del código.

De la misma forma que durante la transmisión pueden ocurrir errores, también es posible que haya coordenadas que no podamos identificar, o que simplemente no hemos recibido. A estas las llamaremos borrones e igualmente podemos representarlo mediante un vector de borrones con un cero en las coordenadas sin borrón y un símbolo $\bullet$ en las coordenadas con borrón (otra forma de representarlo es con un vector en $\mathbb{F}_2$).

Proposición 2.18. *Sea C un código lineal con distancia mínima d. Entonces C puede corregir un total de $t = \lfloor (d-1)/2 \rfloor$ errores, $d-1$ borrones o un total de e errores y b borrones simultaneamente siempre que se cumpla la desigualdad $2e + b \leq d - 1$.*

El proceso de decodificación en *Sage* se realiza de forma similiar al proceso de codificación. En primer lugar tenemos una función para ver los decodificadores disponibles, y otra para seleccionar uno de ellos.

```
In [ ]: C.decoders_available()
In [ ]: C.decoder(decoder_name,args)
```

Además existen dos métodos para decodificar que, al igual que en el caso del codificador se pueden llamar desde el decodificador D o desde el propio código C. El primero de estos métodos devuelve la palabra código con errores corregidos y el segundo la palabra fuente original.

- A partir del propio código

  ```
  In [ ]: C.decode_to_code(word,decoder_name,args)
  In [ ]: C.decode_to_message(word,decoder_name,args)
  ```

- A partir de un codificador D

  ```
  In [ ]: D.decode_to_code(word)
  In [ ]: D.decode_to_message(word)
  ```

Definición 2.19. Sea C un código lineal de longitud n. Sea $I \subseteq \{1..n\}$ y sea D el subespacio de $\mathbb{F}_q^{n-1}$ formado por la eliminación de las coordenadas cuyos índices están en I en todas las palabras de C. Decimos que D es el código punteado de C en I.

La construcción de los códigos punteados es muy útil, especialmente en la decodificación de borrones. Para realizarlo en *Sage* tenemos dos formas.

```
In [ ]: C.punctured(I)
In [ ]: codes.PuncturedCode(C,I)
```

donde en ambos casos C es el código que queremos puntear y I es una lista conteniendo los índices que deseamos puntear. El nuevo código pertenece a una subclase de los códigos lineales y contiene nuevos métodos que hace referencia al código original pero que dado que no usaremos no se hará hincapié.

Proposición 2.20. *La matriz generatriz de un código punteado se puede obtener de forma sencilla a partir de la matriz generatriz del código original eliminando las cólumnas con índices en I y eliminando las filas linealmente dependientes.*

Por último damos una desigualdad muy importante sobre los parámetros fundamentales de un código y que será la base de la siguiente sección.

Proposición 2.21. *(Cota de Singleton) Sea C un código de tipo $[n, k, d]$. Entonces*

$$d \leq n - k + 1$$

Ejemplo 2.22. En primer lugar definiremos el cuerpo de 16 elementos

```
In [1]: F=GF(4); F
```

```
Out[1]: Finite Field in z2 of size 2^2
```

No hemos especificado ningun parámetro por lo que el polinoio usado para hacer módulo es el predeterminado, al igual que el nombre de la variable de los polinomios. Podemos ver cual es dicha variable de la siguiente forma

```
In [2]: x=F.gen(); x
```

```
Out[2]: z2
```

Además podemos obtener un elemento aleatorio de dicho cuerpo de la siguiente forma:

```
In [3]: F.random_element()
```

```
Out[3]: 0
```

A continuación definiremos un primer código lineal a partir de sus parámetros principales de forma aleatoria. Será de longitud 5 y dimensión 2.

```
In [4]: C=codes.random_linear_code(F,5,2); C
```

```
Out[4]: [5, 2] linear code over GF(4)
```

Podemos comprobar dichos parámetros y dado que es un código relativeamente pequeño podemos calcualar la distancia mínima.

```
In [5]: C.dimension()
```

```
Out[5]: 2
```

```
In [6]: C.length()
```

```
Out[6]: 5
```

```
In [7]: C.minimum_distance()

Out[7]: 2
```

También podemos obtener la matriz generatriz y la matriz generatriz en forma sistemática

```
In [8]: C.generator_matrix()

Out[8]: [     1     z2      0 z2 + 1      1]
        [     1      0      0      1      0]

In [9]: C.systematic_generator_matrix()

Out[9]: [     1      0      0      1      0]
        [     0      1      0      1 z2 + 1]
```

Cada matriz está formada por dos filas que forman una base del subespacio vectorial.

A continuación obtenemos una matriz de control y comprobamos su propiedad principal. Su producto con cualquier palabra del código es cero. Para ello obtendremos una palabra aleatoria del mismo.

```
In [10]: C.parity_check_matrix()

Out[10]: [ 1  0  0  1 z2]
         [ 0  1  0  0 z2]
         [ 0  0  1  0  0]

In [11]: C.parity_check_matrix()*C.random_element().column()

Out[11]: [0]
         [0]
         [0]
```

Si además creamos el código dual también veremos que su matriz generatriz coincide con la matriz de control de C

```
In [12]: C.dual_code()

Out[12]: [5, 3] linear code over GF(4)

In [13]: C.dual_code().generator_matrix()
```

```
Out[13]: [ 1  0  0  1 z2]
         [ 0  1  0  0 z2]
         [ 0  0  1  0  0]
```

Para construir un código a partir de su matriz generatriz tendremos que primero que crear dicha matriz. Para ello debemos espcecificar el cuerpo de los elementos y dar los valores como una lista de filas donde cada fila es una lista de elementos. Más adelante la podremos usar para crear un código lineal.

```
In [14]: M=Matrix(F,[[1,x,x+1,1,0,x],[0,1,0,x,x+1,1],[x,1,0,0,x,0]]); M
```

```
Out[14]: [     1     z2 z2 + 1      1      0     z2]
         [     0      1      0     z2 z2 + 1      1]
         [    z2      1      0      0     z2      0]
```

```
In [15]: C=codes.LinearCode(M); C
```

```
Out[15]: [6, 3] linear code over GF(4)
```

```
In [16]: C.minimum_distance()
```

```
Out[16]: 3
```

Dado que la distancia mínima es 3 podremos corregir un error. Obtendremos una palabra del código codificando una palabra del espacio fuente y añadiendo a la codificación manualmente un error. Las palabras son vectores y el procedimiento para crearlas es similar al de la matriz.

```
In [17]: v=vector(F,[1,x+1,0]); v
```

```
Out[17]: (1, z2 + 1, 0)
```

```
In [18]: C.encoders_available()
```

```
Out[18]: ['GeneratorMatrix', 'Systematic']
```

```
In [19]: E=C.encoder('GeneratorMatrix'); E
```

```
Out[19]: Generator matrix-based encoder for [6, 3] linear code over GF(4)
```

```
In [20]: v2=E.encode(v); v2
```

```
Out[20]: (1, 1, z2 + 1, 0, z2, 1)
```

```
In [21]: v2[3]=1; v2

Out[21]: (1, 1, z2 + 1, 1, z2, 1)
```

Ahora usaremos un decodificador para decodificar la palabra al código y al espacio fuente

```
In [22]: C.decoders_available()

Out[22]: ['InformationSet', 'NearestNeighbor', 'Syndrome']

In [23]: D=C.decoder('Syndrome'); D

Out[23]: Syndrome decoder for [6, 3] linear code over GF(4) handling
         errors of weight up to 2

In [24]: D.decode_to_code(v2)

Out[24]: (1, 1, z2 + 1, 0, z2, 1)

In [25]: D.decode_to_message(v2)

Out[25]: (1, z2 + 1, 0)
```

Por último realizaremos una serie de operaciones de punteado sobre el código para mostrar su funcionamiento

```
In [26]: I=[1,3]

In [27]: P=C.punctured(I); P

Out[27]: Puncturing of [6, 3] linear code over GF(4) on position(s) [1, 3]
```

Para comprobar que la base de P genera el mismo espacio que la base de C eliminando las columnas 1 y 3 (el índice empieza en 0) comprobamos que ambas matrices son equivalentes por filas. La función echelon_form nos permite realizarlo ya que nos devuelve la matriz en forma fundamental equivalente por filas a la matriz desde donde llamamos al método.

```
In [28]: C.generator_matrix()[:,[0,2,4,5]].echelon_form()==
         P.generator_matrix().echelon_form()

Out[28]: True
```

```
In [29]: P.dimension()

Out[29]: 3

In [30]: P.length()

Out[30]: 4
```

2.3. Códigos MDS

A la hora de elegir un código lineal para corregir errores, estamos interesados en que su distancia mínima sea lo mayor posible. Esto nos permitirá corregir una mayor cantidad de errores. Sin embargo no podemos hacerla tan grande como queramos para longitud y dimensión fija, ya que la cota de Singleton nos pone un límite en ella. En esta sección veremos los códigos que obtienen la igualdad en dicha cota, siendo por por tanto los códigos lineales con la mayor distancia mínima posible, con dimensión y longitud fijas.

Definición 2.23. Llamamos a un código lineal **MDS** (Maxima Distancia Separable) si

$$d = n - k + 1$$

Muy pocos códigos son MDS. En esta sección veremos dos familias de códigos que sí lo son y que usaremos a lo largo del trabajo. Estas dos familias son los códigos Reed-Solomon y Reed-Solomon Generalizados (RS y GRS de forma abreviada).

Definición 2.24. (Códigos Reed-Solomon) Sea $x_1, ..., x_n$ elementos distintos de un cuerpo finito $\mathbb{F}_q$. Para $k \leq n$ consideramos el conjunto $\mathbb{P}_k$ de polinomios sobre $\mathbb{F}_q$ de grado menor que k. Un código de Reed-Solomon consiste de las palabras

$$\{(f(x_1), ..., f(x_n)), \ \ para\ f \in \mathbb{P}_k\}$$

.

Proposición 2.25. *Los códigos Reed-Solomon son lineales y además MDS*

En *Sage* la clase a la que pertenecen los códigos Reed-Solomon hereda de la clase de códigos lineales, por ello todos los métodos de dicha clase pueden ser aplicados a uno de ellos. Además tienen funcionalidad añadida que no puede ser aplicada a un código lineal cualquiera, como el cálculo de la distancia mínima o distintos algoritmos de codificación y decodificación. En el trabajo de matemáticas viene la explicación del decodificador de $BerlekampWelch$, uno de los exclusivos de códigos GRS. La construcción la podemos realizar de la siguiente forma:

```
In [ ]: codes.ReedSolomonCode(base_field,length,dimension,primitive_root)
```

donde $base\_field$ es el cuerpo sobre el que queremos construir el cuerpo, $lenght$ la longitud y $dimension$ su dimensión. Los puntos de evaluación que *Sage* utiliza son las diferentes potencias de una raíz n-ésima primitiva de la unidad. Esta raíz se puede espcificar en el parámetro opcional $primitive\_root$. Si en cambio se deja vacío, *Sage* elige una por defecto.

Definición 2.26. Sea $\mathbb{F}_q$ el cuerpo finito de q elementos. Elegimos n elementos distintos del cuerpo $x = (x_1, ..., x_n)$ al igual que para los códigos Reed-Solomon. Consideramos igualmente los polinomios en $\mathbb{P}_k$. Sea $v = (v_1, ..., v_n)$ un vector de elementos de $\mathbb{F}_q$ no nulos. Definimos entonces el código Reed-Solomon Generalizado como el conjunto de las palabras

$$\{(v_1 f(x_1), v_2 f(x_2), ..., v_n f(x_n)), \quad para\ f \in \mathbb{P}_k\}$$

Lo primero que nos salta a la vista es que los códigos RS definidos anteriormente son simplemente un subconjunto de los códigos GRS donde el vector de multiplicadores está formado por unos. Por ello, *Sage* usa una misma clase para implementar ambos códigos. Para crear un código GRS lo podemos hacer de forma sencilla mediante el siguiente método:

```
In [ ]: codes.GeneralizedReedSolomonCode(eval_points,dimension,column_mult)
```

donde el primer parámetro es el conjunto de puntos donde realizaremos las evaluaciones, *dimension* es la dimensión del código y *column_mult* es el vector de multiplicadores. Si no se especifica un vector de multiplicadores se asume que el deseado está formado por unos, y por tanto el código creado será un código RS.

Estos parámetros fundamentales que hemos usado para crear un código C los podemos recuperar a través de los siguientes métodos:

```
In [ ]: C.evaluation_points()
In [ ]: C.column_multipliers()
```

Proposición 2.27. *El código dual de un código GRS es también GRS.*

Ejemplo 2.28. Creamos inicialmente un código RS de dimiensión 4 y longitud 10 sobre el cuerpo de 7 elementos.

```
In [31]: F=GF(7)
```

```
In [32]: C=codes.ReedSolomonCode(F,10,7)
```

```
---------------------------------------------------------------------

ValueError: A classical Reed-Solomon code has a length which divides the
      field cardinality minus 1
```

Dado que los códigos Reed-Solomon tienen una longitud igual al número de puntos de evaluación que usemos, la longitud nunca puede ser mayor al número de elementos del código y por eso obtenemos un error. Corregiremos nuestro error y usaremos un cuerpo de 11 elementos

```
In [33]: F=GF(11)
         C=codes.ReedSolomonCode(F,10,7); C
```

```
Out[33]: [10, 7, 4] Reed-Solomon Code over GF(11)
```

Comprobamos que efectivamente cumple la cota de Singleton

```
In [34]: C.length()-C.dimension()+1==C.minimum_distance()
```

```
Out[34]: True
```

Podemos recuperar los puntos de evaluación con el siguiente comando. Tambien podemos ver que los multiplicadores son siempre 1 dado que es un código GRS

```
In [35]: C.evaluation_points()
```

```
Out[35]: (1, 2, 4, 8, 5, 10, 9, 7, 3, 6)
```

```
In [36]: C.column_multipliers()
```

```
Out[36]: (1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
```

A continuación crearemos un código GRS. Para ello debemos crear previamente el vector de puntos de evaluación y el de multiplicadores

```
In [37]: a=vector(F,[1,2,3,4,5,6])
```

```
In [38]: b=vector(F,[2,4,2,3,2,2])
```

```
In [39]: C=codes.GeneralizedReedSolomonCode(a,4,b); C
```

```
Out[39]: [6, 4, 3] Generalized Reed-Solomon Code over GF(11)
```

```
In [40]: C.decoders_available()
```

```
Out[40]: ['BerlekampWelch',
          'ErrorErasure',
          'Gao',
          'GuruswamiSudan',
          'InformationSet',
          'KeyEquationSyndrome',
          'NearestNeighbor',
          'Syndrome']
```

Como podemos comprobar tenemos una cantidad mucho mayor de decodificadores

Por último vemos que su dual es también un código GRS usando el método type que nos devuelve la clase de un objeto.

```
In [41]: type(C.dual_code())
```

```
Out[41]: <class 'sage.coding.grs_code.GeneralizedReedSolomonCode_with_category'>
```

Capítulo 3

Conceptos básicos de PIR

El acceso a información que está almacenada en una base de datos, es un procedimiento muy común cuando un usuario accede a una aplicación web. En la mayoría de los casos, existen medidas de seguridad sobre la propia base de datos que restringen la cantidad de información a la que un usuario puede acceder, como por ejemplo la autorización y autenticación. Sin embargo, en la gran mayoría de los casos no existe ningún recurso o procedimiento que permita proteger la privacidad de un usuario en relación a la información que está solicitando. Un ejemplo dónde procedimientos como este podrían ser útiles es el caso de un régimen opresor en el que solicitudes a cierta información pueden hacer que el solicitante sea añadido a una lista negra. Otro ejemplo es el de la solicitud de acciones en la bolsa sin que se sepa en qué empresa es la que estamos interesados en invertir. Para solucionar estos problemas se idearon teóricamente los protocolos PIR (del inglés Private Information Retrieval). Estos algoritmos nos permiten conseguir esta privacidad en la información en bases de datos con unas propiedades específicas. En este capítulo veremos los conceptos fundamentales que definen un protocolo PIR general y qué carácterísticas tiene que tener la base de datos para poder relizar estos protocolos.

3.1. Modelo de base de datos

Hoy en día, la cantidad de peticiones que puede recibir una base de datos en un corto periodo de tiempo puede ser realmente alta. Por ello, es muy común el uso de bases de datos distribuidas para asegurar que todas las peticiones pueden ser respondidas en el menor tiempo posible y sin causar una sobrecarga en el sistema. La principal diferencia entre una base de datos de este tipo y una tradicional es la réplica de los datos. Una base de datos distribuida está formada por varios servidores que almacenan todos o parte de los datos originales, de forma que si deseamos acceder a cierta parte de los datos, existen varios servidores de donde los podemos extraer. Se basan en dos principios fundamentales, la réplica de los datos y el uso de varios servidores. Existen múltiples formas de distribuir las peticiones entre los diferentes servidores. Entre ellas están el uso de balanceadores de carga que redirigen las peticiones a ciertos servidores.

Para poder aplicar un protocolo PIR necesitamos que la base de datos de donde queremos extraer la información sea una base de datos distribuida. Esta base de datos estará formada por cierta cantidad de servidores n. Una condición adicional que debemos imponer

es que no todos los servidores están en contacto entre ellos directamente, es decir, no pueden contrastar entre todos ellos las peticiones que han recibido. La idea principal de estos protocolos es realizar ciertas peticiones a los servidores donde parezca que no se está pidiendo nada en concreto, pero al poner en común las diferentes respuestas que recibimos, obtenemos la información deseada. Más adelante daremos una explicación más detallada de esto.

Otra condición que debemos imponer es que los datos almacenados en la base sean elementos de un mismo cuerpo finito $\mathbb{F}$. Esta restricción, que a priori puede parecer muy restrictiva, no lo es, ya que podemos usar simplemente la representación binaria de los datos ($\mathbb{F}_2$), o la representación ASCII de los datos ($\mathbb{F}_{256}$). Sin embargo, algo que sí que se requiere y que no suelen tener las bases de datos es la opción de devolver una combinación lineal de todos o parte de los datos que posee.

Dados estos requerimientos, para explicar estos protocolos representaremos los datos almacenados en un servidor i como un vector v_i sobre el cuerpo $\mathbb{F}$, cuya longitud es la cantidad de datos m almacenados en dicho servidor, $v_i \in \mathbb{F}^m$. Las solicitudes las representaremos como un vector en el mismo espacio $x \in \mathbb{F}^m$. La respuesta que obtendremos será el producto escalar entre los vectores x y v_i, es decir la combinación lineal de los datos almacenados en v_i donde los diferentes coeficientes vienen dados por x. Así, si por ejemplo deseamos obtener el dato almacenado en cuarta posición, la solicitud necesaria tendrá un uno en dicha posición y ceros en el resto. Si en otro caso necesitamos la suma de los datos tres y cuatro, la solicitud necesaria tendrá un uno en tercera y cuarta posición, y ceros en el resto.

Dados estos requisitos podemos dar un ejemplo de un primer protocolo PIR muy simple donde podemos observar el funcionamiento de la base de datos, a la vez que vemos la estructura más básica de uno de estos protocolos PIR.

Ejemplo 3.1. Sea una base de datos formada por dos servidores. Cada uno de los servidores tiene almacenadas la totalidad de los datos que forman la base de datos. Podemos decir por tanto, que la base de datos está replicada totalmente en dos servidores. Dicha base está formada por diez archivos, $x_1 \dots x_{10}$ representados por un elemento del cuerpo $\mathbb{F}_{512}$. Podemos representar por tanto la información almacenada en cada servidor con un vector formado por los diez archivos $x = [x_1 \cdots x_{10}]^T$.

Supongamos que deseamos obtener el archivo 3 y que ambos servidores no comparten información. Podemos realizar las solicitudes a la base de datos de forma que seremos capaces de obtener el archivo a partir de las respuestas. La forma de hacer esto es la siguiente.

Generamos un vector aleatorio de elementos de $\mathbb{F}_{512}$ y de longitud 10 $v = [v_1 \dots v_{10}]$. Este vector será el que enviaremos como solicitud al primer servidor. Con él estamos pidiendo una combinación lineal aleatoria de los elementos de x. De parte del servidor obtendremos como respuesta el siguiente elemento de $\mathbb{F}_{512}$:

$$r_1 = v_1 \cdot x_1 + \cdots + v_{10} \cdot x_{10}$$

La solicitud que usaremos para el segundo servidor será ligeramente diferente a la del primero. Será la suma del vector aleatorio v y un vector de ceros, excepto un uno en la tercera posición:

$$s_2 = v + [0, 0, 1, 0, 0, 0, 0, 0, 0, 0] = [v_1, v_2, v_3 + 1, v_4, \ldots, v_{10}]$$

La respuesta que recibiremos será en esencia muy similar a la recibida por el primer servidor.

$$r_2 = v_1 \cdot x_1 + v_2 \cdot x_2 + (v_3 + 1) \cdot x_3 + \cdots + v_{10} \cdot x_{10}$$

Ahora desarrollando esta expresión veremos como obtener el archivo que deseabamos:

$$r_2 = v_1 \cdot x_1 + v_2 \cdot x_2 + v_3 \cdot x_3 + \cdots + v_{10} \cdot x_{10} + 1 \cdot x_3 = r_1 + x_3$$

Por tanto si simplemente restamos la primera respuesta a la segunda, el resultado será el dato que queríamos obtener.

Es importante recalcar que si ambos servidores se comunican entre ellos, también pueden realizar el procedimiento que hemos realizado nosotros para averiguar en qué dato estábamos interesados. De hecho, si todos los servidores están comunicados entre ellos, la única forma de obtener información de forma privada es descargando la base de datos entera, que puede ser realmente costoso.

En este ejemplo hemos usado una representación de cada archivo, como un único elemento del cuerpo base. Sin embargo, lo más normal es que un archivo esté formado por muchos símbolos del cuerpo base (p.e. un texto formado por varios caracteres). En estos casos, podemos representar cada uno de los símbolos como un archivo independiente y en el caso de querer recuperar un archivo completo, tendremos que obtener cada uno de ellos. Esto hace que la base de datos tenga un número muy elevado de entradas, lo que hace que las solicitudes tengan un tamaño muy elevado y sea significantemente más alto el coste computacional de las respuestas. Por ello debemos buscar otra solución.

En primer lugar podemos entender la sucesión de símbolos que es un archivo como un vector sobre el cuerpo base $\mathbb{F}$. Dicho vector tendrá un tamaño t igual al número símbolos que lo forman. Como habíamos visto en la proposición 2.3, podemos representar un vector sobre $\mathbb{F}$ de tamaño t como un elemento del cuerpo $\mathbb{F}^t$. De esta forma podremos representar cada archivo como un único elemento de un cuerpo. El problema principal de este método es que el tamaño del cuerpo crece exponencialmente respecto al número de símbolos que forman el archivo. Dado que manejar cuerpos muy grandes no es cómodo y el coste computacional crece mucho, este método tampoco es enteramente efectivo.

La solución es realizar una mezcla de ambos métodos. Podemos dividir el archivo en cierto número l de "subarchivos"del mismo tamaño y usar el segundo método para codificarlos como un elemento de un único cuerpo.

$$x = [x_1 \ldots x_{12}]^T \in \mathbb{F}_p^{12} \Rightarrow x = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ x_5 & x_6 & x_7 & x_8 \\ x_9 & x_{10} & x_{11} & x_{12} \end{bmatrix} \Rightarrow x = [y_1, y_2, y_3, y_4]^T \in \mathbb{F}_{p^4}^4$$

Por último, explicaremos como supondremos que la base de datos almacena la información. En el anterior ejemplo toda la información estaba replicada en muchos servidores. Sin

embargo, en los protocolos PIR que estudiaremos en los siguientes capítulos este no será el caso. Cada servidor solamente tendrá almacenada parte de la información y por tanto en la mayoría de los casos, necesitaremos contactar con varios servidores para obtener los datos deseados.

Una primera idea que nos surge es dividir los archivos en tantas partes como servidores haya y almacenar cada parte en uno de los servidores. Sin embargo habríamos ignorado uno de los principios de las bases de datos distribuidas, la réplica de los datos. Si uno de los servidores no está disponible, entonces parte de la información estará inaccesible. La siguiente idea que se nos puede ocurrir es similar a la anterior pero usando el doble de servidores, de forma que cada parte esté replicada en dos servidores. Esta opción es mejor que la anterior pero sigue teniendo el problema de que si los dos servidores que contienen la misma parte de un archivo dejan de funcionar, no podremos acceder a esa información. La mejor solución para estos problemas es la codificación usando un buen código lineal.

La principal ventaja del uso de estas estructuras es que nos permiten recuperar la información contactando con cualesquiera l servidores de los n totales para cierto número l que dependerá del código. Con que haya l servidores operativos seremos capaces de obtener la totalidad de la información. El proceso para codificar un archivo se realizará de la siguiente forma.

Suponiendo que tenemos n servidores, en primer lugar dividiremos el archivo en k partes, para un $k < n$. Si cada una de estas partes no se trata de un único elemento del cuerpo, los combinamos todos en uno de un cuerpo más grande como antes. Tomamos un código C de tipo $[n, k, d]$. Para mejores resultados debemos usar un código con una distancia mínima d lo mayor posible. Agrupamos las k partes en las que hemos dividido el archivo en un vector de tamaño k y lo multiplicamos por la matriz generatriz de C. De estar forma obtendremos un vector de tamaño n. Guardaremos la primera posición en el primer servidor y así sucesivamente.

Para recuperar la información nos vale con realizar solicitudes a $n-d+1$ servidores. Como hemos visto, esta es la cantidad de borrones que un código puede corregir. Si interpretamos las respuestas de los $n - d + 1$ servidores como un vector con un borrón en las posiciones en las que no hemos realizado solicitud, podemos usar los algoritmos de corrección de borrones y decodificación para recuperar los valores del archivo.

Una opción posible y que se usará en los protocolos es dividir el archivo en un múltiplo de k partes y agrupar cada conjunto de k partes como un archivo separado. De esta forma el tamaño de los cuerpos es aún menor y facilitará los cálculos. A cada uno de estos conjuntos de k partes se les llamará bandas.

En definitiva, un archivo lo podremos representar como una matriz de tamaño $\alpha \times k$ donde α es el número de bandas en las que está dividida el archivo:

$$X^i = \begin{pmatrix} x^i_{11} & \dots & x^i_{1k} \\ \vdots & \ddots & \vdots \\ x^i_{\alpha 1} & \dots & x^i_{\alpha k} \end{pmatrix}$$

Tras codificar el archivo con un código lineal de tipo $[n, k, d]$, la representación del mismo será

$$Y^i = \begin{pmatrix} y^i_{11} & \cdots & y^i_{1n} \\ \vdots & \ddots & \vdots \\ y^i_{\alpha 1} & \cdots & y^i_{\alpha n} \end{pmatrix}$$

Si ahora agrupamos verticalmente todos los m archivos en una misma matriz, obtendremos una representación de la base de datos en una matriz, donde la columna j-ésima de la matriz será la información almacenada por el servidor j. Dicha columna la denominaremos w_j.

$$\begin{pmatrix} y^1_{11} & \cdots & y^1_{1n} \\ \vdots & \ddots & \vdots \\ y^1_{\alpha 1} & \cdots & y^1_{\alpha n} \\ \vdots & \vdots & \vdots \\ y^m_{11} & \cdots & y^m_{1n} \\ \vdots & \ddots & \vdots \\ y^m_{\alpha 1} & \cdots & y^m_{\alpha n} \end{pmatrix}$$

Ejemplo 3.2. Sean $X^1 = 010010101000$, $X^2 = 111100111000$, dos archivos de 12 bits que queremos almacenar en una base de datos distribuida. Dicha base de datos estará formada por 7 servidores. Para reducir el tamaño del cuerpo usado dividiremos cada archivo en 4 bandas. Por último, la codificación se realizara mediante el código lineal de tipo $[7, 3, 5]$ con la siguiente matriz generatriz matriz generatriz

```
In [1]: G=Matrix(GF(2),[[1,0,0,1,1,0,1],[0,1,0,1,0,1,1],[0,0,1,0,1,1,1]])
```

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

```
In [2]: C=codes.LinearCode(G); C
```

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

El primer paso es expresar los archivos como matrices de tamaño 2×3, ya que los debemos dividir en 2 bandas y 3 es la dimensión del código. Esto hace un total de 6 grupos de bits, por tanto cada grupo estará formado por 2 bits, por lo que las entradas de la matriz serán elementos de $\mathbb{F}_{2^2} = \mathbb{F}_4$. Podemos usar las funciones que explicamos en el capítulo anterior para realizar la conversión.

```
In [3]: #Creamos los cuerpos de 2 y 4 elementos
        F=GF(2)
        G=GF(4,"z")
        z=G.gen()

        #Introducimos los archivos
```

```
        x1=vector(F,[0,1,0,0,1,0,1,0,1,0,0,0])
        x2=vector(F,[1,1,1,1,0,0,1,1,1,0,0,0])

        #Creamos los isomorfismos
        [V,map1,map2]=G.vector_space(F, map=true)

        #Creamos una pequeña función que toma de dos en dos los
        #elementos del vector y los convierte a un elemento de G
        def transformar(v,morf):

            #Inicializamos
            X=vector(GF(4,"z"),zero_vector(6))

            for i in range (6):     #Cada uno de los 6 grupos
                x=v[2*i:2*i+2]      #Tomamos las siguientes dos posiciones
                X[i]=morf(x)        #Usamos el morfismo

            return X

        #Usamos dicha función
        X1=transformar(x1,map2)
        X2=transformar(x2,map2)

        X1=Matrix(G,[X1[0:3],X1[3:6]])
        X2=Matrix(G,[X2[0:3],X2[3:6]])

        print(X1)
        print("\n")
        print(X2)
```

```
[z 0 1]
[1 1 0]


[z + 1 z + 1     0]
[z + 1     1     0]
```

El cuerpo de 4 elementos está formado por polinomios de grado menor que 2 sobre el cuerpo de dos elementos. En este caso la variable que hemos usado la hemos dado el nombre de z.

Ahora que los hemos expresado en forma matricial, tenemos que usar el código para obtener los valores que guardaremos en cada servidor. Para ello lo único que tenemos que hacer es multiplicar por la matriz generatriz de dicho código.

```
In [4]: #Combinamos verticalmente ambas matrices
        X=X1.stack(X2)

        Y=X*C.generator_matrix(); Y
```

```
Out[4]: [    z     0     1     z z + 1     1 z + 1]
        [    1     1     0     0     1     1     0]
        [z + 1 z + 1     0     0 z + 1 z + 1     0]
        [z + 1     1     0     z z + 1     1     z]
```

Esta matriz representa la base de datos. Cada columna son los datos que almacena cada servidor. Como vemos, cada uno de ellos almacena solamente 4 elementos del cuerpo de 4 elementos.

Supongamos ahora que deseamos obtener el segundo archivo. Dicho archivo se corresponde con las bandas 3 y 4 de la base de datos, así que debemos solicitar ambas. Como el código que hemos usado es un código con distancia mínima 5, nos basta con contactar a $7 - 5 + 1 = 3$ servidores para poder recuperar los datos originales. Usaremos por ejemplo los servidores 4 ,5 y 6.

En primer lugar construimos las solicitudes. Como deseamos obtener dos de los elementos almacenados, tenemos que enviar dos solicitudes. En el caso de los tres servidores serán las mismas. En la primera obtendremos su valor almacenado en tercera posición, y en la segunda el valor almacenado en la cuarta. Serán las siguientes solicitudes:

$$s1 = [0, 0, 1, 0]$$
$$s2 = [0, 0, 0, 1]$$

Cada servidor procesará las respuestas como el producto escalar de la solicitud y sus datos almacenados. Por ejemplo, en el caso del quinto servidor, los valores que devolverá serán:

```
In [5]: s1=vector(F,[0,0,1,0])
        s2=vector(F,[0,0,0,1])

        #El quinto servidor se corresponde al cuarto índice,
        # los indices empiezan en 0

        r1=s1*Y[:,4] #Producto escalar entre s1 y la cuarta columna de Y
        r2=s2*Y[:,4]

        print(r1)
        print(r2)
```

```
(z + 1)
(z + 1)
```

Suponiendo que hemos obtenido los valores de los servidores 3 y 7 de la misma forma, conocemos 3 valores de cada banda. Si agrupamos estos 3 valores en un vector en las posiciones correspondientes a los servidores obtendremos el siguiente vector en el caso de la tercera banda:

$$(-, -, -, 0, z + 1, z + 1, -)$$

Este vector es una palabra código con 4 borrones y como nuestro código tiene distancia mínima 5 podemos corregir dichos borrones y obtener la palabra fuente original, que se corresponde con los tres primeros valores del archivo. Usaremos el método de corrección de borrones del punteado del código. Dicho método consiste en el punteado del código a uno de longitud menor, eliminando los borrones. Para una explicación más detallada se referencia al capítulo 1 del TFG de matemáticas [1].

```
In [6]: M=Matrix(G,C.generator_matrix())
        M=M[:,3:6]
        N=M.inverse()
        N*vector(G,[0,z+1,z+1])
```

```
Out[6]: (z + 1, z + 1, 0)
```

De la misma forma obtendremos la segunda banda del archivo y con ella el archivo en su totalidad.

3.2. Modelo de PIR

Una vez que hemos explicado cómo será la base de datos con la que trataremos, veremos en qué consiste un protocolo PIR y cómo podemos medir su eficiencia. Estas definiciones que veremos fueron introducidas en [5]. Muchos de los resultados no se demostrarán por su complejidad, si se desea una explicación se puede obtener en dicho artículo o en el TFG de matemáticas [1].

Como hemos visto en la sección anterior, la base de datos a la que se harán las solicitudes debe ser una base de datos distribuida y codificada mediante un código lineal. Esto nos da nuestra primera condición. Existen varios tipos de protocolos PIR, pero en este documento nos centraremos en el que más se ha investigado, los protocolos PIR lineales.

Definición 3.3. Sea una base de datos distribuida formada por m archivos con α bandas cada uno. Un protocolo PIR es lineal sobre un cuerpo $\mathbb{F}_q$ y de dimensión ρ si está formado por las siguientes dos fases:

1. Fase de solicitud: El usuario envía una solicitud a un subconjunto de todos los servidores. Cada una de las solicitudes viene dada por una matriz Q_l sobre $\mathbb{F}_q$ de tamaño $\rho \times m\alpha$. Dicha matriz será la matriz de solicitud dirigida al servidor l.
2. Fase de descarga: Cada servidor que recibe una solicitud la responde enviando la proyección de sus datos sobre la matriz recibida. $R_l = Q_l w_l$ donde w_l es el vector de datos almacenados en dicho servidor.

Cada una de las filas de una solicitud Q_l se puede interpretar como una sub-solicitud, del tipo que usábamos en la sección anterior. De esta forma no enviamos cada una de las sub-solicitudes una a una, sino que las agrupamos todas en dicha matriz y las enviamos todas juntas. Es por ello que en un protocolo PIR lineal todas las solicitudes se pueden determinar antes de recibir la respuesta a alguna de ellas.

De la misma forma, en vez de recibir un único elemento como respuesta a una solicitud, recibiremos un vector con tantas entradas como sub-solicitudes tenía nuestra matriz Q_l. La posición i-ésima de dicho vector se corresponderá con la respuesta a la i-ésima fila de la matriz solicitud.

La dimensión ρ únicamente denotará el número necesario de sub-solicitudes (o filas en la matriz solicitud) para recuperar el archivo que deseamos.

Al principio del capítulo hicimos una pequeña mención sobre que podían existir grupos de servidores que se comunican entre ellos y por tanto podrían poner en común las solicitudes que hemos enviado y descubrir cuál es el archivo f en el que estamos interesado. A estos grupos de servidores los llamaremos servidores **cooperantes**. Todos los protocolos PIR tienen una cota b tal que si la cantidad de servidores cooperantes es menor que b, la privacidad en la recuperación está garantizada. La forma de comprobar dicha garantía es mediante conceptos de teoría de la información. Por brevedad, no se explicará el significado de estos conceptos, pero si se desea más información se puede acudir al capítulo dos de [1].

Definición 3.4. Se dice que un protocolo PIR protege contra b servidores cooperantes si y solo sí se cumple que $H(f|Q_j, j \in J) = H(f)$ para todos los conjuntos $J \subseteq \{1 \dots n\}$ con $|J| = b$. H denota la función entropía.

Por último solo necesitamos una forma de evaluar la eficiencia de un protocolos PIR. Como es lógico, pra garantizar la privacidad en nuestras solicitudes tenemos que pagar un precio en eficiencia. No nos vale con simplemente descargar los elementos en los que estamos interesados, sino que necesitaremos descargar otros, de forma que podamos extraer de forma indirecta aquellos que nos interesan. Esto supone que descargaremos una mayor cantidad de elementos. Tiene sentido que la cantidad de fragmentos que tengamos que descargar para poder realizar el protocolo PIR sea lo menor posible. El cPoP mide dicho aumento en la cantidad de descargas:

Definición 3.5. El precio de la privacidad en la comunicación (cPoP), es el cociente entre la cantidad de símbolos descargados como respuesta a las solicitudes, y la cantidad de símbolos en los que realmente estamos interesados. En muchos artículos se usa un parámetro denominado razón (rate) que es el inverso del cPoP.

A continuación calcularemos el cPoP del ejemplo 3.1 como una muestra. En dicho ejemplo, únicamente estábamos interesados en un símbolo, ya que cada archivo está formado únicamente por uno de ellos. Sin embargo realizamos dos solicitudes, recibiendo dos símbolos, y a partir de ellos, ya fuimos capaces de decodificar el valor del deseado. Por tanto el cPoP será de $2/1 = 2$.

Una vez explicados los conceptos básicos del PIR, veremos en capítulos posteriores protocolos concretos y además realizaremos su implementación en Sage.

3.3. Implementación de los métodos básicos

En esta sección daremos una implementación de todos los métodos que son comunes a todos los protocolos PIR lineales. En primer lugar, tenemos que identificar cuales son estos métodos.

Como explicamos, un protocolo PIR lineal se caracteriza por tener dos fases, una de solicitud y otra de respuesta. En la primera fase se hace una construcción de todas las solicitudes que serán enviadas a los servidores. La forma de estas solicitudes son lo que diferencia los protocolos PIR y por tanto variará entre protocolos. En cambio, en la segunda fase se recibe la respuesta del servidor y a partir de ella se reconstruyen los datos deseados. Esta última parte de reconstrucción variará en función de la estructura de las solicitudes que hemos enviado, y por tanto será diferente entre cada protocolo. Sin embargo, el cálculo de las respuestas por parte del servidor, es algo que no varía entre protocolos PIR ya que se trata del producto entre la solicitud y los datos del servidor. Antes de poder implementar esta parte, debemos también implementar una función que codifique un vector de archivos en una base de datos con las propiedades que enunciamos en la primera sección. Al igual que vamos a realizar un método que realice la codificación, también realizaremos uno que decodifique un archivo en forma matricial sin codificar a su forma original. Estos tres métodos son los comunes a todos los protocolos y los que implementaremos en esta sección.

3.3.1. Creación de la base de datos

En el ejemplo en Sage que se realizó anteriormente (3.2) realizamos una pequeña función que funcionaba dado el caso, pero ahora haremos una que funcione para todos. El primer paso que debemos considerar se trata de identificar los inputs y outputs que debe tener nuestra función.

El output será la base de datos en forma matricial codificada. Es decir una matriz con tantas columnas como servidores se desee y $\alpha \cdot m$ filas, donde α es el número de bandas por archivo y m es el número de archivos que forman la base.

En cuanto a los inputs, el primero que necesitamos son los propios archivos en forma de una secuencia de elementos de un cuerpo. Pediremos que vengan en una matriz en la que cada columna es un archivo. El tamaño de dicha matriz será de $s \times m$ donde s es el máximo tamaño de un archivo. El siguiente parámetro que necesitaremos es el número de bandas en las que dividiremos cada archivo. Por último solo nos falta el código lineal que usaremos para realizar la réplica de los datos. No es necesario que tratemos el número de servidores como un input, ya que la longitud del código tiene que coincidir con dicho número y por tanto la podemos extraer del mismo. Además introduciremos un parámetro extra opcional que nos permita indicar si la codificación que haremos será usando la matriz generatriz por defecto del código o la matriz sistemática del mismo. El nombre que daremos a estos inputs son el siguiente:

- Matriz de archivos: f
- Numero de bandas: $alpha$
- Código lineal: C
- Indicador de codificación sistemática: $systematic = false$

La sintaxis de este último parámetro quiere decir que en caso de no indicarse un valor para $systematic$, se usara el valor $false$.

En primer lugar, extraemos de los inputs datos necesarios para realizar los distintos cálculos. Estamos interesados en los siguientes:

1. K: dimensión del código C
2. N: longitud del código C y número de servidores en los que se realizará la codificación.
3. F: cuerpo base del código C y de la matriz f
4. M: número de archivos y número de columnas de la matriz f
5. *size*: tamaño máximo de los archivos y número de filas de la matriz f

```
def PIREncodeToServers(f, alpha, C, systematic=false):

            K = C.dimension();
            N = C.length();
            M = f.ncols();
            size = f.nrows();
            F = C.base_field();
```

El siguiente paso es empezar a dividir los archivos en las distintas partes para agruparlos en bandas. El número de partes en las que tenemos que dividir un archivo viene dado por el número de bandas que deseamos que tenga el mismo y la dimensión de C. Un total de $\alpha \cdot K$. Por tanto necesitamos poder dividir los archivos de forma exacta en dicho número de partes. Para ello añadiremos ceros al final de cada archivo hasta alcanzar un tamaño divisible entre $\alpha \cdot K$.

El número de símbolos que formarán cada parte viene dado por la parte entera hacia arriba, llamada función techo (en python la función *ceil*()). Una vez obtenido añadiremos los ceros a la matriz f usando el método *stack*() de Sage. La matriz de archivos f tendrá M columnas y un tamaño de filas divisible entre $\alpha \cdot K$.

```
            symbols = ceil(size/(alpha*K));
            zeros = matrix(F, symbols*alpha*K-size, M);
            f=f.stack(zeros);
```

A continuación tenemos que divir los nuevos archivos ampliados en los distintos grupos y combinar los elementos de un mismo grupo en uno de una extensión del mismo. El primer paso es crear la extensión. Como queremos combinar *symbols* elementos en uno el tamaño de la extensión tiene que ser el tamaño del original elevado a *symbols*. Una vez obtenida la extensión, simplemente calculamos el isomorfismo entre ella y el espacio vectorial sobre F usando el método *vector_space*().

```
            G=GF(F.order()^symbols);
            Ggen=G.gen();

            V,m1,m2=G.vector_space(F,map=true);
```

Tras este paso ahora si tenemos que empezar a combinar los distintos elementos. Inicializamos una matriz DB, donde iremos introduciendo los diferentes archivos una vez hagamos

las combinaciones. Las entradas de esta matriz serán en el cuerpo G en vez de F. También crearemos una matriz $File$ para almacenar la codificación de uno de los archivos. Una vez obtenida, volcaremos su contenido en DB para proceder con el siguiente archivo.

El proceso para combinar los elementos es el siguiente. Iteraremos para cada grupo de *symbols* elementos. Manejando los índices correspondientes obtenemos los elementos correspondientes a un grupo, los agrupamos en un vector y usamos el morfismo que calculamos anteriormente. Una vez obtenido el elemento de la extension, lo introducimos en la entrada correspondiente de $File$. Una vez completadas todas ellas usamos el método $stack()$ para añadirlos a DB. Realizamos el mismo proceso con el siguiente archivo hasta que obtenemos una matriz de tamaño $alpha \cdot M \times K$, la base de datos sin codificar.

```
File = Matrix(G,alpha,K);

DB = Matrix(0,K);

for m in range(M):                  #cada archivo
    for a in  range(alpha):     #cada fila de la nueva matriz
        for k in range(K):       #cada columna de la nueva matriz
            index=a*K*symbols+k*symbols;
            v=f[range(index,index+symbols),m];
            v=vector(v);
            File[a,k]=m1(v);

    DB=DB.stack(File);
    File=Matrix(G,alpha, K);
```

El último paso es ya usar el código C para codificar dicha matriz. En caso de tratarse de codificación sistemática usaremos la matriz generatriz en forma sistemática. En caso contrario usaremos la matriz generatriz predefinida. En ambos casos, únicamente debemos multiplicar la base de datos por la matriz correspondiente para obtener la codficación de la misma.

```
if systematic:
    CodedDB=DB*C.systematic_generator_matrix();
else:
    CodedDB=DB*C.generator_matrix();
return CodedDB;
```

Código completo del programa:

```
def PIREncodeToServers(f, alpha, C, systematic=false):

    #f es una matriz donde cada columna representa los datos
    # de un archivo
    #alpha es el numero de bandas
    #C es un código para codificar los datos
    #systematic es un booleano que indica si queremos que la
```

```
#codificación sea sistemática o no

K = C.dimension();
N = C.length();
M = f.ncols();
size = f.nrows();
F = C.base_field();



#Añadir ceros al final de cada archivo hasta que el tamaño sea
#divisible entre alpha*k

symbols = ceil(size/(alpha*K));
zeros = matrix(F, symbols*alpha*K-size, M);  #añadimos ceros
#para poder dividir exactamente el vector
f=f.stack(zeros);

#Dividir cada archivo en las diferentes alpha filas y k columnas
#y concatenarlos todos verticalmente. Cada entrada
#es un vector del mismo número de elementos sobre el cuerpo base.

G=GF(F.order()^symbols);
Ggen=G.gen();

V,m1,m2=G.vector_space(F,map=true);

File = Matrix(G,alpha,K);

DB = Matrix(0,K);

for m in range(M):                  #cada archivo
    for a in  range(alpha):     #cada fila de la nueva matriz
        for k in range(K):       #cada columna de la nueva matriz
            index=a*K*symbols+k*symbols;
            v=f[range(index,index+symbols),m];
            v=vector(v);
            File[a,k]=m1(v);

    DB=DB.stack(File);
    File=Matrix(G,alpha, K);



#Por último usamos el código para codificar la Base de Datos.
if systematic:
    CodedDB=DB*C.systematic_generator_matrix();
else:
    CodedDB=DB*C.generator_matrix();
```

```
        return CodedDB;
```

3.3.2. Cálculo de las respuestas

La segunda función que necesitamos se trata del cálculo de las respuestas por parte del servidor. Como hemos visto, dicho cálculo se realiza únicamente mediante un producto matricial, así que será mucho más simple que en el caso anterior.

Para la realización del programa separaremos dos casos principales, el caso en el que se desee hacer solicitudes a todos los servidores, o el caso en el que solo se desee hacer solicitudes a un subconjunto de ellos. En ambos casos el output será una lista de vectores en las que cada vector contendrá la respuesta de un servidor a sus solicitudes.

En cuanto a los inputs, necesitaremos introducir la base de datos en forma matricial y además el conjunto de solicitudes. El conjunto de solicitudes pediremos que venga dado como una lista de matrices donde cada matriz es una solicitud. Para indicar a qué servidor va dirigida cada matriz usaremos un tercer parámetro. Este último será una lista del mismo tamaño que la anterior indicando los índices de los servidores a las que van dirigidas. Como generalmente las solicitudes se harán a todos los servidores, haremos que en caso de omitir este parámetro se sobreentienda dicha situación. Los nombres que daremos a estos parámetros son los siguientes:

- Base de datos: $CodedDB$
- Lista de solicitudes: $query$
- Lista de indices de los servidores a los que dirigimos las solicitudes: $servers = -1$

En este caso el programa es muy simple. En primer lugar necesitamos crear una lista vacía que iremos rellenando con las respuestas. En el caso de que no se hayan introducido ninguna lista de índices de servidores guardamos en la variable $servers$ la lista con todos los índices. Una vez realizado, solo tenemos que iterar sobre todos los servidores y realizar los productos mientras introducimos el resultado en la lista.

Código completo del programa:

```
def PIRQueryToServer(CodedDB, query, servers=-1):

        #CodedDB es la matriz devuelta por PIREncodeToServers
        #que representa los datos que tiene cada servidor
        #servers son los servidores a los que deseamos hacer las
        #solicitudes
        #query es la matriz solicitud que deseamos hacer al
        #servidor. Tiene que tener alpha*m columnas


        response=[];
        N=len(query);
```

```
    if servers==-1:
        servers=range(N);



    for n in servers:
        data=CodedDB.column(n).column();
        response.append(query[n]*data);
    return response;
```

3.3.3. Decodificación de un archivo

Cuando realicemos un protocolo PIR, los fragmentos del archivo que vamos a obtener, son los fragmentos que se usaron para codificar la base de datos, es decir los fragmentos del archivo en forma matricial. Sin embargo, como es lógico nosotros estamos realmente interesados en obtener el original. Para ello servirá esta función.

El output del programa será por tanto una lista que representará al archivo. El input en cambio serán los siguientes:

- Archivo en forma matricial: $File$
- Cuerpo original en el que estaba representado el archivo original: F

El programa en sí es bastante simple. El primer paso es crear el homomorfismo entre los diferentes cuerpos y usarlo para ir decodificando uno a uno los elementos de $File$. Los fragmentos decodificados los iremos añadiendo en una lista que posteriormente devolveremos.

Código completo del programa:

```
def DecodeToFile(File,F):

    FF=File[0,0].parent()

    V,m1,m2=FF.vector_space(F,map=true);
    file=[];

    for i in range(File.nrows()):
        for j in range(File.ncols()):
            x=File[i,j];
            file=file+list(m2(x))


    return file
```

Capítulo 4

PIR en códigos MDS sistemáticos

En esta sección abordaremos varios de los primeros protocolos PIR que fueron desarrollados. Fueron introducidos por primera vez en [5]. Veremos tres protocolos distintos. Todos ellos tratan el mismo caso, el de una base de datos codifica con un código MDS y sistemático. Recordamos que un código MDS se trata de uno con la mayor distancia mínima posible $(n-k+1)$ y un código sistemático es uno en la que las k columnas iniciales de su matriz generatriz forman la identidad. La primera propiedad de ellas resulta muy útil en una base de datos distribuida ya que reduce le número de servidores a los un usuario tiene que contactar para obtener datos. La segunda de ellas implica que los primeros k servidores poseen un fragmento de los datos originales, y no una combinación lineal de todos ellos como en el caso de los $n-k$ servidores restantes. Por tanto si un archivo sin codificar tiene la forma que ya introdujimos en el capítulo anterior (a la izquierda) entonces su forma codificada será la siguiente (a la derecha):

$$X^i = \begin{pmatrix} x^i_{11} & \dots & x^i_{1k} \\ \vdots & \ddots & \vdots \\ x^i_{\alpha 1} & \dots & x^i_{\alpha k} \end{pmatrix} \quad Y^i = \begin{pmatrix} x^i_{11} & \dots & x^i_{1k} & \sum_{j=1}^n a_{k+1,j}x_{1j} & \dots & \sum_{j=1}^n a_{n,j}x_{1,j} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ x^i_{\alpha 1} & \dots & x^i_{\alpha k} & \sum_{j=1}^n a_{nj}x_{\alpha j} & \dots & \sum_{j=1}^n a_{n,j}x_{\alpha,j} \end{pmatrix}$$

Como hemos recalcado, los tres protocolos funcionan contra un mismo tipo de base de datos pero se diferencian en otro factor muy importante. El primero de ellos no protege contra servidores cooperantes y por tanto no podremos usarlo cuando sepamos de la existencia de alguno. En cambio, tanto el segundo como el tercero protegen contra hasta $d-1$ servidores cooperantes, donde d es la distancia mínima del código. La principal diferencia entre ambos es que el último es ligeramente más eficiente, pero solo se podrá aplicar para ciertos valores de n y k.

En el caso de los tres protocolos, explicaremos su funcionamiento pero no se demostrará por qué funcionan. Si se desea leer una explicación más detallada del funcionamiento, además de su demostración y el cálculo de los servidores contra los que protege, está todo explicado en el capítulo 3 de [1].

4.1. Protocolo PIR sin servidores cooperantes

El protocolo de esta sección es efectivo únicamente en el caso más sencillo, durante la ausencia de servidores cooperantes. Una forma más técnica de denotarlo es que existen $b = 1$ servidores cooperantes. Además como hemos explicado supondremos que la base de datos esta codificada usando un código C de tipo $[n, k, d]$ MDS y sistemático. Por último también supondremos que cada archivo está formado por $\alpha = n - k$ bandas y hay un total de m archivos. La condición sobre las bandas no es estrictamente necesaria pero el protocolo funciona óptimamente cuando este es el número de bandas descargado.

4.1.1. Creación de las solicitudes

Este protocolo además será de dimensión k y por tanto cada matriz solicitud estará formada por k filas o k sub-solicitudes. Como en todos los protocolos PIR las solicitudes estarán formadas por una parte aleatoria, que está destinada a confundir al servidor, y una parte de donde podemos extraer los datos. La parte aleatoria provendrá de una matriz U que generaremos de tamaño $k \times m\alpha$.

La parte que usaremos para extraer los datos será un matriz $E_{f,l}$ de unos y ceros que será distinta en cada servidor l y dependerá también del archivo en el que estamos interesados f. La solicitud a cada servidor estará formada por la suma de U y una de estas matrices. Estas matrices $E_{f,l}$ cumplen tres propiedades fundamentales que garantizan que podamos recuperar el archivo deseado:

- En cada fila y cada columna de cada matriz $E_{f,l}$ habrá como máximo un 1. El propósito de esta restricción es obtener uno de los valores sin obtener un mismo valor varias veces.
- En cada fila, únicamente $n-k$ matrices $E_{f,l}$ tienen un 1. Con esto podremos obtener el producto de U y los datos de la base.
- Si j es el índice de una banda que queremos obtener entonces únicamente k matrices $E_{f,l}$ tiene un 1 en la columna j-ésima. De esta forma obtendremos k valores de cada banda que nos permitirá decodificarla entera.

Para poder definir un conjunto de matrices con estas propiedades primeros tenemos que expresar $n - k$ como suma de cociente y resto tras dividir entre k. Para cociente β y resto r, $n - k = \beta k + r$. Ahora ya podemos definir estas matrices empezando por la primera de ellas:

$$E_{f,1} = \begin{bmatrix} 0_{k\times(f-1)\alpha} & \begin{matrix} I_{r\times r} \\ 0_{(k-r)\times r} \end{matrix} & 0_{k\times\beta k} & 0_{k\times(m-f)\alpha} \end{bmatrix}$$

Dependiendo del servidor que se trate tenemos tres opciones para calcular la matriz $E_{f,l}$ correspondiente.

- En el caso de los servidores con índices $2 \leq l \leq k$, $E_{f,l}$ se obtienen rotando cíclicamente las filas de $E_{f,l-1}$ una posición hacia abajo.

- En el caso de los últimos r servidores, $E_{f,l}$=0.
- En el caso de los servidores con índices $k < l \leq \beta k + k$, dividiremos todos los servidores en β grupos de k servidores. Los servidores que estarán en cada grupo $1 \leq s \leq \beta$ son los que tienen índices $sk + 1 \leq l \leq sk + k$. Las matrices $E_{f,l}$ correspondientes a servidores en el mismo grupo serán iguales y vienen dadas por:

$$E_{f,l} = \begin{bmatrix} 0_{k\times(f-1)\alpha+r+(s-1)k} & I_{k\times k} & 0_{k\times(\beta-s)k+(m-f)\alpha} \end{bmatrix}$$

Una vez calculadas todas estas matrices, la solicitud que enviaremos a cada servidor l viene dada por $Q_l = U + E_{f,l}$. A partir de las respuestas seremos capaces de recuperar el archivo usando el método que veremos más adelante.

Ejemplo 4.1. Veremos ahora en un ejemplo esta primera fase de solicitud. Supondremos que estamos trabajando en el cuerpo $\mathbb{F}_7$ y que deseamos obtener el archivo 1. Para este caso $\alpha = n - k = 4$ así que cada archivo está formado por 4 bandas. La matriz generatriz del código con el que se codificó la base de datos es la siguiente:

$$G = \begin{pmatrix} 1 & 0 & 0 & 1 & 3 & 6 & 3 \\ 0 & 1 & 0 & 4 & 6 & 6 & 4 \\ 0 & 0 & 1 & 3 & 6 & 3 & 1 \end{pmatrix}$$

Primero generamos la matriz aleatoria de tamaño $3 \times 4m$, $U = [u_{ij}]$. Calcularemos ahora las matrices $E_{f,l}$ teniendo en cuenta que en este caso nos queda $r = 1$. Para ahorrar espacio únicamente daremos las primeras 4 columnas ya que el resto es la parte correspondiente a los otros archivos y por tanto es 0.

$$E_{1,1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad E_{1,2} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad E_{1,3} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

$$E_{1,4} = E_{1,5} = E_{1,6} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad E_{1,7} = 0$$

La matriz de solicitud Q_l que enviaremos a un servidor l será la suma de la matriz aleatoria U y $E_{1,l}$.

Por último realizaremos un cálculo del cPoP de este protocolo. Cada archivo está formado por $\alpha = n - k$ bandas. Por tanto el número de símbolos que tenemos que obtener para recuperar el archivo son $(n - k) \cdot k = nk - k^2$. Sin embargo la cantidad de símbolos que recibimos de los servidores son n respuestas por sub-solicitud en un total de k de ellas, lo que hace nk respuesas. Por tanto el cPoP será el cociente de ambas:

$$\frac{nk}{nk - k^2} = \frac{1}{1 - k/n}$$

4.1.2. Recuperación del archivo

El proceso de decodificación se realizara por sub-solicitudes. Para cada una de ellas crearemos un sistema de ecuaciones de los que podemos extraer α fragmentos del archivo. Tras las k sub-solicitudes habremos obtenido los αk fragmentos en los que se divide el archivo deseado. Dicho sistema de ecuaciones viene dado por la propia estructura del código y la forma de las matrices $E_{f,l}$. Supondremos que queremos decodificar la sub-solicitud i.

La respuesta a una sub-solicitud es el producto escalar entre la propia sub-solicitud y el vector de datos almacenados en el propio servidor. Dado que el vector de datos de un servidor l es el producto de la matriz de datos y la columna l de la matriz generatriz para cada sub-solicitud a un servidor tenemos la siguiente ecuación:

$$(U_i + E_{f,l_i}) \cdot X^f G^l = r_{li}$$

donde U_i es la i-ésima fila de U, E_{f,l_i} es la i-ésima fila de $E_{f,l}$, X^f el archivo en el que estamos interesados en forma matricial, G^l la columna l-ésima de la matriz generatriz del código C y r_{li} la respuesta a la i-ésima sub-solicitud del servidor l. Podemos ahora separar la ecuación en dos partes y obtener

$$U_i X^f G^l + E_{f,l_i} X^f G^l = r_{li}$$

El producto de U_i y X^f podemos expresarlo también como la suma de los productos de U_i por cada una de las k filas de X_f. A cada uno de los resultados de estos productos podemos expresarlos mediante una variable I_p con $p \leq k$. Sustituyendo en la ecuación anterior:

$$= \sum_{p}^{k} I_p G^l + E_{f,l_i} X^f G^l = r_{li}$$

Dado que conocemos G_l, r_{li} y E_{f,l_i} se trata de un una ecuación lineal con variables $I_1 \dots I_p, x_{11} \dots x_{\alpha n}$. Agrupando las respuestas de los n servidores a la sub-solicitud i obtendremos un sistema de n ecuaciones sobre las mismas variables. Las propiedades que impusimos sobre las matrices $E_{f,l}$ nos garantizarán que la solución a este sistema existe y es única.

Ejemplo 4.2. Por último realizaremos el proceso de decodificación del archivo 1 para el ejemplo 4.1. Cada solicitud es una matriz de 3 filas, por lo que esta formada por 3 sub-solicitudes y por tanto la respuesta del servidor l será un vector $r_l = Q_l w_l \in \mathbb{F}_{7^\omega}^3$. Realizaremos únicamente la decodificación de los símbolos obtenidos por la primera sub-solicitud.

Sea $I_l = U_1^T w_l$ para $l \leq 3$ donde U_1 es la primera fila de U. Tenemos el siguiente sistema de ecuaciones lineales formado por las respuestas de los servidores:

$$I_1 + x_{11}^1 = r_{11}$$

$$I_2 = r_{21}$$

$$I_3 = r_{31}$$

$$I_1 + 4I_2 + 3I_3 + x_{21} + 4x_{22} + 3x_{23} = r_{41}$$
$$3I_1 + 6I_2 + 6I_3 + 3x_{21} + 6x_{22} + 6x_{23} = r_{51}$$
$$6I_1 + 6I_2 + 3I_3 + 6x_{21} + 6x_{22} + 3x_{23} = r_{61}$$
$$3I_1 + 4I_2 + I_3 = r_{71}$$

La segunda, tercera y última ecuación forman un sistema lineal que puede ser resuelto y de donde podemos obtener el valor de I_1, I_3, I_3. Al obtener estas valores podemos obtener x_{11} de la primera ecuación y x_{21}, x_{22}, x_{23} de la cuarta, quinta y sexta. De esta forma hemos conseguido calcular 4 fragmentos del archivo en está sub-solicitud. Los 8 restantes los obtendremos de la misma forma en las dos sub-solicitudes restantes.

En la siguiente tabla representamos los fragmentos del archivo que tiene cada servidor y cuales y en que sub-solicitud han sido obtenidos de forma privada

	1	2	3	4	5	6	7
Banda 1	1	2	3				
Banda 2				1	1	1	
Banda 3				2	2	2	
Banda 4				3	3	3	

4.1.3. Implementación del algoritmo de creación de solicitudes

El output de nuestro programa debe de coincidir con el input del programa que creamos para procesar solicitudes, y por tanto tiene que ser una lista de matrices de solicitud. En total construiremos una matriz de solicitud por cada servidor.

Los parámetros iniciales que debemos de recibir por parte del usuario serán los siguientes:

- Código que usa la base de datos: C
- Número de bandas que forman cada archivo: $alpha$
- Número de archivos que forman la base de datos: M
- Índice del archivo en el que estamos interesados: f

En primer lugar, al igual que en programas anteriores extraemos parámetros fundamentales para los cálculos a partir de los introducidos por el usuario y comprobaremos que el número de bandas que tiene cada archivo es el mismo que pide el protocolo. Obtenemos los mismos que en casos anteriores:

```
def primera(C,alpha,M,f):

    N=C.length();
    K=C.dimension();
    F=C.base_field();
    x=F.gen();
```

```
d=N-K+1;

if alpha!=N-K:
    print("Number of stripes does not match the required number")
    return -1;
```

El siguiente paso es inicializar una lista vacía *queries* donde iremos introduciendo las matrices de solicitud a medida que las vayamos generando. Después usamos el método *random_matrix*() para obtener la matriz aleatoria de elementos del cuerpo U. A continuación calculamos la descomposición de $n - k$ en cociente y resto usando las operaciones básicas de división entera y módulo.

```
queries=[]

U=random_matrix(F, K, M*alpha);

beta=alpha//K;
r=alpha%K;
```

Ahora podemos introducir la primera de las matrices $E_{f,1}$. Para realizarlo vamos concatenando verticalmente con el método *stack*() y horizontalmente con el método *augment*() varias matrices hasta obtener la deseada.

```
E=Matrix(F,K,(f-1)*alpha)
.augment(identity_matrix(F,r).stack(Matrix(F,K-r,r)))
.augment(Matrix(F,K,beta*K+(M-f)*alpha));
```

El siguiente paso es empezar a calcular las matrices $E_{f,l}$ correspondientes a los servidores del primer grupo e introducir la suma con U en la lista *queries*. Para ello en primer lugar creamos la permutación necesaria para girar las filas de $E_{f,1}$ e ir calculando sucesivamente todas las $E_{f,l}$. Esta permutación es $(1, 2, 3, \ldots, K)$. Esta permutación es la inversa de la primera de las dos permutaciones que genera el grupo simétrico sobre K elementos así que para obtenerla creamos dicho grupo y extraemos su primer generador.

Ahora solo tenemos que iterar sobre los K primeros servidores para obtener sus solicitudes. En el bucle simplemente añadimos a la lista la suma de U y E y usamos el método *permute_rows*() para calcular la siguiente matriz E correspondiente al próximo servidor.

```
permutation=SymmetricGroup(K).gens()[0].inverse();

for n in range(K):
    queries.append(U+E);
    E.permute_rows(permutation);
```

Para crear las matrices correspondientes al siguiente grupo iteraremos sobre el numero de grupos de servidores *beta*. Creamos la matriz E correspondiente al grupo de forma similar

a la anterior y añadimos su suma con U un total de K veces en la lista, una por cada servidor del grupo.

Para los últimos r servidores simplemente introducimos en *queries* r veces la matriz U ya que en su caso $E = 0$. Por último devolvemos la lista de matrices *queries*

```
        for s in range(beta):
             E=Matrix(F,K,(f-1)*alpha+r+s*K)
             .augment(identity_matrix(F,K))
             .augment(Matrix(F,K,(beta-s-1)*K+(M-f)*alpha));
             m=U+E;
             for j in range(K):
                 queries.append(m);

        for n in srange(N-r,N):
            queries.append(U);
        return queries;
```

Código completo del programa:

```
def primera(C,alpha,M,f):

    N=C.length();
    K=C.dimension();
    F=C.base_field();
    x=F.gen();
    d=N-K+1;

    if alpha!=N-K:
        print("Number of stripes does not match the required number")
        return -1;

    queries=[]

    U=random_matrix(F, K, M*alpha);

    beta=alpha//K;
    r=alpha%K;
    E=Matrix(F,K,(f-1)*alpha)
    .augment(identity_matrix(F,r).stack(Matrix(F,K-r,r)))
    .augment(Matrix(F,K,beta*K+(M-f)*alpha));

    permutation=SymmetricGroup(K).gens()[0].inverse();

    for n in range(K):      #Creamos las matrices E para los primeros
           queries.append(U+E);
           E.permute_rows(permutation);
```

```
for s in range(beta):
        E=Matrix(F,K,(f-1)*alpha+r+s*K)
        .augment(identity_matrix(F,K))
        .augment(Matrix(F,K,(beta-s-1)*K+(M-f)*alpha));
        m=U+E;
        for j in range(K):
            queries.append(m);

for n in srange(N-r,N):
        queries.append(U);
return queries;
```

4.1.4. Implementación del algoritmo de recuperación del archivo

El output de esta función será el propio archivo que se solicitó en forma matricial. Los inputs necesarios para lograrlo serán los siguientes:

- Código que se usó para la codificación de la base de datos: C
- Número de bandas en las que se divide cada archivo: *alpha*
- Lista de respuestas recibidas por parte del servidor: R

Comenzamos por obtener los parámetros necesarios como en otros programas. En este caso necesitamos diferenciar entre dos cuerpos. En primer lugar F, el cuerpo base de C y de los elementos del archivo original, y FF, una extensión de F. Los elementos de la base de datos codificada pertenecen a FF y por ello, las respuestas que recibiremos del servidor lo serán también. Además realizamos una comprobación del número de bandas al igual que en el programa anterior:

```
def primera(C,alpha,R):

    N=C.length();
    K=C.dimension();
    F=C.base_field();
    FF=R[0][0,0].parent();
    x=F.gen();
    y=FF.gen();

    beta=alpha//K;
    r=alpha%K;
    rho=R[0].nrows();

    if alpha!=N-K:
        print("Number of stripes does not match the required number")
        return -1;
```

A continuación inicializamos dos matrices $File$ y M. En la primera iremos introduciendo los fragmentos del archivo que vayamos calculando. En la segunda agruparemos todas las respuestas de los servidores de forma que cada fila contenga las respuestas de todos los servidores a la misma sub-solicitud. Por último extraemos la matriz generatriz sistemática de C en la variable G y la replicamos en otra matriz GG que será igual pero los elementos serán elementos de FF en vez de F.

```
File=Matrix(FF,alpha,K);

M=Matrix(F,rho,0);

for i in range(N):
    M=M.augment(R[i]);

G=C.systematic_generator_matrix();

GG=Matrix(FF,G);
```

El siguiente paso es ir iterando sobre cada sub-solicitud (o cada fila de M) e ir obteniendo y resolviendo los sistemas lineales. Sin embargo el proceso no es tan simple como puede parecer. En primer lugar, en la sección anterior se explico cómo obtener el sistema lineal pero no se explicó cómo obtener la matriz del sistema lineal de forma explícita. Tras un estudio de cómo es está matriz, obtuve que dicha matriz tiene la siguiente forma:

$$\left(\begin{array}{c|cccc} & \multicolumn{4}{c}{G} \\ \hline T & \multicolumn{4}{c}{0} \\ \hline 0 & G_1 & 0 & \dots & 0 \\ & 0 & G_2 & \dots & 0 \\ & \vdots & \vdots & \ddots & \vdots \\ & 0 & 0 & \dots & G_\beta \end{array}\right)$$

G es la matriz generatriz del código, los sucesivos $G_1 \dots G_\beta$ son las columnas de G correspondientes a los servidores en el grupo indicado por el subíndice. T por último es una matriz de ceros y unos que únicamente viene un 1 en cada fila y columna. La posición de estos unos depende de la forma de la matriz E en cada sub-solicitud. En esta matriz del sistema lineal, las primeras k filas son las correspondientes a los I_k y el resto de filas se corresponden a diversos fragmentos del archivo que dependerán de la sub-solicitud. En nuestro programa tendremos que construir dicha matriz. Para ello usamos una matriz S. Inicialmente clonamos en S la matriz G y posteriormente le iremos añadiendo las distintas partes que forman la matriz total. En primer lugar le añadimos T y la matriz de ceros contigua e introducimos los 1 en las posiciones correspondientes. A continuación por cada grupo de servidores introducimos una fila de matrices. Estará formada por la concatenación horizontal de una matriz de ceros, una submatriz de G y otra submatriz de ceros. Una vez añadidos todas estas partes, la matriz del sistema estará completa y simplemente tenemos que usar el método $solve_left()$ que nos proporciona Sage para resolverlo.

```
for i in range(rho):
```

```
            S=GG;
            T=Matrix(FF,r,N);

            for j in range(r):
                T[j,(K-j+i)%K]=1;
            S=S.stack(T);

            for j in range(beta):
                a1=Matrix(FF,K,(j+1)*K);
                a2=Matrix(GG.submatrix(0,(j+1)*K,K,K))
                a3=Matrix(FF,K,K*(beta-1-j))
                T=a1.augment(a2).augment(a3);
                S=S.stack(T);

            s=S.solve_left(M.row(i));
```

Sin salir del bucle *for* ahora tenemos que extraer los fragmentos que hemos obtenido en la solución del sistema lineal e introducirlos en la matriz que representa el archivo $File$. Un estudio de las matrices $E_{f,l}$ nos permite identificar los índices de los fragmentos que hemos recuperado para así poderlos introducir. Tras ello simplemente tendremos que devolver el archivo.

```
            for j in range(r):
                File[j,(K+i-j)%K]=s[K+j];

            for j in range(beta):
                File[r+j*K+i,0:K]=s[K+r+j*K:r+(j+2)*K];

        return File
```

Código completo del programa:

```
def primera(C,alpha,R):

    N=C.length();
    K=C.dimension();
    F=C.base_field();
    FF=R[0][0,0].parent();
    x=F.gen();
    y=FF.gen();

    beta=alpha//K;
    r=alpha%K;
    rho=R[0].nrows();

    if alpha!=N-K:
        print("Number of stripes does not match the required number")
```

```
        return -1;

    File=Matrix(FF,alpha,K);

    M=Matrix(F,rho,0);

    for i in range(N):
        M=M.augment(R[i]);

    G=C.systematic_generator_matrix();

    GG=Matrix(FF,G);


    for i in range(rho):

        S=GG;
        T=Matrix(FF,r,N);

        for j in range(r):
            T[j,(K-j+i)%K]=1;
        S=S.stack(T);

        for j in range(beta):
            a1=Matrix(FF,K,(j+1)*K);
            a2=Matrix(GG.submatrix(0,(j+1)*K,K,K))
            a3=Matrix(FF,K,K*(beta-1-j))
            T=a1.augment(a2).augment(a3);
            S=S.stack(T);

        s=S.solve_left(M.row(i));

        for j in range(r):
            File[j,(K+i-j)%K]=s[K+j];

        for j in range(beta):
            File[r+j*K+i,0:K]=s[K+r+j*K:r+(j+2)*K];

    return File
```

4.2. Protocolo PIR para servidores cooperantes

En esta sección veremos un protocolo PIR para servidores cooperantes. En concreto, este protocolo garantizará privacidad en la recuperación de los archivos siempre que la cantidad de servidores cooperantes sea menor que la distancia mínima d del código C MDS usado para codificar la base de datos. Como C se trata de un código MDS entonces protege contra como máximo $n - k$ servidores cooperantes.

Este protocolo fue introducido en el mismo artículo que el anterior y al igual que en el caso anterior para más información se referencian dicho artículo [5] y el TFG de matemáticas [1].

En el caso de que la cantidad de servidores cooperantes sea menor que la cota superior, el protocolo no necesita realizar solicitudes a todos ellos. En esos casos podemos simplemente ignorar los últimos servidores, suponer que el código tiene una longitud menor y situarnos en el caso en el que la cota se alcanza. La cantidad de servidores que contactaremos es de $b + k$. Por ello, únicamente explicaremos el protocolo en dicho caso.

En este caso, no necesitamos que los archivos estén divididos en varias bandas. Por ello, supondremos que cada archivo está formado únicamente por una banda. En caso contrario podremos recuperar todas las bandas del archivo tratando a cada una como un archivo independiente y aplicando el procedimiento en cada uno de ellos. Esto simplifica la notación de la forma matricial de un archivo, ya que pasa de ser una matriz con múltiples filas a una de una única fila. Por ello en vez de usar dos índices para los fragmentos, podemos ignorar el índice referente a la banda y usar uno único. $x^f_{1j} \to x^f_j$

El número de sub-solicitudes necesarias que necesitaremos para recuperar un archivo, o banda es de k, por lo que recuperaremos un fragmento del archivo por sub-solicitud. El procedimiento será el siguiente:

4.2.1. Creación de las solicitudes y recuperación del archivo

Al igual que en el protocolo anterior, esta vez también dividiremos las solicitudes en una parte aleatoria y otra parte de donde extraeremos los datos. Sin embargo la construcción que haremos de las matrices solicitud, será por sub-solicitudes o lo que es lo mismo por filas.

Supongamos entonces que deseamos obtener el archivo f de una base de datos codificada mediante un código MDS C de tipo $[n, k, d]$. Deberemos realizar el siguiente proceso para cada sub-solicitud i.

El primer paso consiste en generar $d - 1$ vectores aleatorios de dimensión m. Estas palabras las podemos agrupar un vector por columna en una misma matriz U_i, que será de dimensiónes $m \times d - 1$. A continuación multiplicaremos U_i por la matriz de control H de C. De esta forma obtendremos una matriz Q'_i que será de dimensiones $m \times n$.

Como H es la matriz generatriz del código dual de C, cada una de las filas de Q'_i es una palabra de dicho código dual. Esta propiedad es la principal en la explicación del funcionamiento de este algoritmo, pero no indagaremos en el tema.

La i-ésima sub-solicitud que enviaremos a un servidor l viene dada por:

$$q_{l,i} = \begin{cases} q'_{l,i} + e_f & \text{si } l = i \\ q'_{l,i} & \text{en caso contrario} \end{cases}$$

Las matrices solicitud Q_l para un servidor l las realizaremos agrupando en una misma matriz las k sub-solicitudes dirigidas al mismo servidor.

Tras recibir las respuestas a todas las solicitudes, podemos empezar la fase de obtención

de los fragmentos, mucho más sencilla en este caso que en el anterior.

Para este caso, únicamente tenemos que sumar todas las respuestas a una misma sub-solicitud para obtener un fragmento del archivo. Más en concreto si se trata de la sub-solicitud i, el fragmento que obtendremos del servidor será x_i^f. Tras realizar el procedimiento en las k sub-solicitudes habremos recuperado los k fragmentos que forman el archivo.

Por último analizaremos el cPoP de este protocolo. Como cada archivo estaba formado por una única banda, solo necesitamos obtener k fragmentos. En cambio hemos usado k sub-solicitudes dirigidas a $b+k$ servidores, lo cual hace un total de $k(b+k)$ respuestas. El cociente de ambas y por tanto el cPoP de este protocolo es de $b+k$.

Ejemplo 4.3. Consideraremos en este ejemplo el mismo caso que en el anterior pero únicamente para 2 servidores cooperantes. En este caso necesitaremos únicamente mandar solicitudes a $k+b=4$ de los 5 servidores. Por ello podemos reducir la longitud de nuestro código en 1, eliminando la última coordenada de cada palabra código. De esta forma el código resultante seguiría siendo un código MDS con matriz generatriz y de control:

$$G' = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \end{pmatrix} \quad H' = \begin{pmatrix} -1 & -1 & 1 & 0 \\ -1 & -2 & 0 & 1 \end{pmatrix}$$

Esta vez solamente necesitaremos dos vectores aleatorios. De la misma forma que antes, la primera sub-solicitud a cada servidor serán:

$$\begin{aligned} q_{1,1} &= -u_{1,1} - u_{1,2} + e_2 \\ q_{1,2} &= -u_{1,1} - 2u_{1,2} \\ q_{1,3} &= u_{1,1} \\ q_{1,4} &= u_{1,2} \end{aligned}$$

Por lo que agrupando las respuestas en un sistema lineal obtendremos:

$$\begin{aligned} -I_{1,1} - I_{1,2} + x_1^2 &= r_{1,1} \\ -I_{2,1} - 2I_{2,2} &= r_{1,2} \\ I_{1,1} + I_{2,1} &= r_{1,3} \\ I_{1,2} + 2I_{2,2} &= r_{1,4} \end{aligned}$$

De aquí podemos obtener el fragmento del archivo sumando todas las respuestas igual que antes.

4.2.2. Implementación del algoritmo de creación de solicitudes

El output de nuestro programa debe ser una lista de matrices solicitud. Los inputs que necesitaremos para calcular las matrices son los siguientes:

- Código con el que se ha codificado la base de datos: C.

- Número de bandas en las que está dividido cada archivo: *alpha*.
- Número de archivos en la base de datos: M.
- Índice del archivo en el que estamos interesados: f

El principio del programa es similar al protocolo anterior. Comenzamos extrayendo los datos necesarios y luego realizamos una comprobación sobre el número de bandas.

```
def segunda(C,alpha,M,f):

    N=C.length();
    K=C.dimension();
    F=C.base_field();
    x=F.gen();
    d=N-K+1;

    if alpha!=1:
        print("Number of stripes does not match the required number")
        return -1;
```

A continuación comprobaremos si debemos reducir el tamaño del código ya que no contactaremos a todos los servidores. Para ello seleccionamos los servidores que habrá que ignorar, que son los que tienen un índice superior a $b + K$ y punteamos el código C para eliminar dichas columnas. Actualizamos los parámetros fundamentales del nuevo código y continuamos.

```
    eliminate=srange(b+K,N)
    C=codes.PuncturedCode(C,eliminate)
    N=C.length();
    d=d-len(eliminate);
```

Posteriormente comenzamos con el proceso para crear las matrices solicitud. Inicalizamos la lista *queries* donde las iremos introduciendo y guardamos la matriz de control en la variable H para su posterior uso. El siguiente paso es llenar la lista *queries* de N matrices nulas que iremos rellenando con los valores necesarios para las solicitudes.

```
    queries=[]

    H=C.parity_check_matrix()
    for i in range(b+K):
        queries.append(zero_matrix(F,K,M))
```

El último paso es ya iterar sobre las k sub-solicitudes para realizar el proceso. En primer lugar generamos la matriz aleatoria U, la multiplicamos por H y añadimos 1 en la posición indicada dependiendo de la sub-solicitud. El último paso es introducir cada una de las columnas del resultado en la matriz y fila correspondiente de *queries*. Una vez realizado devolvemos la lista.

```
        for i in range(K):
            U=random_matrix(F, M, d-1);
            Q=U*H
            Q[f+1,i]=Q[f+1,i]+1

            for j in range(N):
                queries[j][i,:]=Q[:,j].transpose()
        return queries;
```

Código completo del programa:

```
    def segunda(C,alpha,M,f):

        N=C.length();
        K=C.dimension();
        F=C.base_field();
        x=F.gen();
        d=N-K+1;

        if alpha!=1:
            print("Number of stripes does not match the required number")
            return -1;

        eliminate=srange(b+K,N)
        C=codes.PuncturedCode(C,eliminate)
        N=C.length();
        d=d-len(eliminate);

        queries=[]

        H=C.parity_check_matrix()
        for i in range(b+K):
            queries.append(zero_matrix(F,K,M))
        #Para cada sub solicitud
        for i in range(K):
            #Matriz aleatoria
            U=random_matrix(F, M, d-1);
            #Generamos las palabras del dual
            Q=U*H
            #Añadimos el símbolo en una posición
            Q[f+1,i]=Q[f+1,i]+1

            for j in range(N):
                queries[j][i,:]=Q[:,j].transpose()
        return queries;
```

4.2.3. Implementación del algoritmo de recuperación del archivo

En este caso como es muy sencillo, simplemente indicaré el código completo del programa sin ningún comentario.

Cógigo completo del programa:

```
def segunda(C,alpha,R):

    FF=R[0][0,0].parent();
    K=C.dimension();

    File=Matrix(FF,alpha,K);

    if alpha!=1:
        print("Number of stripes does not match the required number")
        return -1;

    for i in range(len(R)):
        File=File+R[i].transpose()
        #El resultado es la suma de todos los valores recibidos

    return File
```

4.3. Protocolo más eficiente en el caso de servidores cooperantes

En esta sección veremos el tercero y último de los protocolos dirigidos a bases de datos codificadas con códigos MDS sistemáticos. La principal diferencia entre los dos primeros era el número de servidores cooperantes contra los que protegían. El primero de ellos únicamente protegía contra servidores aislados y el segundo contra como máximo grupos de $b = d - 1$ servidores cooperantes. En este caso, el tercero de los protocolos también protegerá contra grupos de $b = d - 1$ servidores.

La principal diferencia entre este y el de la sección anterior se trata de la eficiencia, el cPoP. El protocolo que veremos en esta sección es significantemente más eficiente que el anterior. Sin embargo, en muchos casos no seremos capaces de aplicarlo y es por eso que el segundo protocolo sigue teniendo interés.

El principal problema del segundo protocolo es que cuando la cantidad de servidores cooperantes no es demasiado alta, únicamente necesitamos hacer solicitudes a $b+k$ de los n servidores. Para muchas combinaciones de valores la cantidad de servidores que reciben alguna solicitud es muy pequeña sobre el total. El protocolo que vemos en esta sección es una adaptación del anterior en la que no se ignoran a tantos servidores, logrando una mayor eficiencia. Sin embargo, para poder aplicarlo se tiene que cumplir la condición de que $b + k$ sea mucho más pequeño que n.

Para indicar cuanto más pequeño definimos $\delta = \lfloor \frac{n-b}{k} \rfloor$. Este protocolo podremos aplicar-

lo cuando $\delta \geq 2$. Usar esta versión cuando $\delta = 1$ resulta en la versión anterior. La principal idea consiste en dividir los servidores en grupos de k y usar los b servidores restantes como b servidores comunes a todos los grupos. Veremos más en concreto su funcionamiento en la creación de las solicitudes.

4.3.1. Creación de las solicitudes y recuperación del archivo

Sea $\delta = \lfloor \frac{n-b}{k} \rfloor$. Para este protocolo únicamente necesitaremos realizar solicitudes a $b + \delta k$ servidores. Igual que en la sección anterior, supondremos que la longitud del código es $n = b + \delta k$ ya que podemos ignorar las últimas coordenadas para hacer un código con dicha longitud.

Como dijimos brevemente antes, la idea es dividir los δk servidores iniciales en δ grupos de k servidores cada uno. Los últimos b servidores serán unos servidores comunes. Por ello, cada grupo de k servidores y los b servidores comunes forman una agrupación de $b + k$ servidores desde la que podemos aplicar el protocolo segundo que vimos anteriormente. La clave del ahorro en eficiencia consiste en que estamos usando las respuestas de los b servidores comunes en δ protocolos simultáneamente.

Para asegurarnos de que esta forma funciona habría que combrobar que las matrices generatrices de los sub-códigos generados por cada grupo de $b + k$ servidores cumplen las propiedades de ser MDS y sistemáticos. Al ser el resultado de puntear un código MDS, también se tratan de códigos MDS, sin embargo inicialmente no son códigos sistemáticos. Sin embargo, por la proposición 2.11 dicho código es equivalente a uno sistemático, por lo que obteniendo dichos códigos podemos aplicarlo.

El algoritmo explícito consiste entonces en inicialmente realizar la división de los servidores. Se obtienen las matrices sistemáticas generatrices de cada subcódigo, así como las de control y se usan para aplicar el protocolo segundo en todos los grupos, usando la misma matriz aleatoria U y la correspondiente matriz de control en cada caso. Dada la estructura de las matriz de control, las solicitudes obtenidas en cada grupo que se enviaran a los servidores comunes son iguales y por tanto con enviarlas una única vez su respuesta nos vale para todos los grupos. Se agrupan todas las sub-solicitudes en una matriz y ya se puede enviar a los servidores.

Como queremos obtener en total k símbolos (una única banda) y en cada sub-solicitud recuperaremos δ fragmentos (uno por cada grupo de servidores), el total de sub-solicitudes que necesitaremos es $\lceil k/\delta \rceil$.

Para calcular el cPoP nos fijamos en que en cada sub-solicitud recuperamos δ fragmento, pero para ello recibimos $b + \delta k$ respuestas. El cociente de ambas es:

$$cPoP = \frac{b + \delta}{k}$$

El proceso de decodificación es también casi idéntico al caso anterior. Simplemente tenemos que identificar los grupos y sumar las respuestas de los servidores correspondientes a un mismo grupo. El resultado serán los diferentes fragmentos del archivo.

4.3.2. Implementación del algoritmo de creación de solicitudes

Al ser una función que crea las solicitudes, el output será el mismo que en los casos anteriores, una lista de matrices solicitud. En este caso el input será también el mismo que en los casos anteriores:

- Código con el que se ha codificado la base de datos: C.
- Número de bandas en las que está dividido cada archivo: $alpha$.
- Número de archivos en la base de datos: M.
- Índice del archivo en el que estamos interesados: f

Inicialmente obtenemos los distintos parámetros, realizamos la comprobación habitual sobre el número de bandas y calculamos el valor de δ.

```
def tercera(C,alpha,M,f):

    N=C.length();
    K=C.dimension();
    F=C.base_field();
    x=F.gen();
    d=C.minimum_distance();

    if alpha!=1:
        print("Number of stripes does not match the required number")
        return -1;
```

El siguiente paso es eliminar los servidores sobrantes, al igual que hicimos en el protocolo anterior y luego empezar a separar los servidores en los distintos grupos. En primer lugar seleccionamos los b últimos como los comunes.

```
    delta=floor((N-b)/K)

    eliminate=srange(b+delta*K,N)
    C=codes.PuncturedCode(C,eliminate)
    N=C.length();
    d=d-len(eliminate);

    comunes=srange(N-b,N)
    todos=range(N)
```

A continuación iremos identificando cada uno de los *delta* grupos. Para cada uno de ellos seleccionamos sus servidores y punteamos el código C para obtener el subcódigo asociado con el grupo de servidores. De él extraeremos la matriz generatriz sistemática y usaremos la propiedad de la proposición 2.13 para obtener la parte de la matriz de control correspondiente a los servidores no comunes de cada grupo. Almacenaremos cada una de estas sumbatrices en una lista $PartityMatrices$ para usarlas posteriormente.

```
        ParityMatrices=[]

        for i in range(delta):
            group=srange(K*i,K*(i+1))+comunes
            quitar=[item for item in todos if item not in group]
            D=codes.PuncturedCode(C,quitar);
            H=-D.systematic_generator_matrix()[:,K:K+b].transpose();
            ParityMatrices.append(H);
```

Ahora ya podemos empezar a calcular las matrices de solicitud. Para ello inicializamos una lista de matrices de solicitud e iteramos sobre el número de subsolicitudes. En cada una de ellas generamos una matriz aleatoria U. El siguiente paso es multiplicar la matriz de control de cada sub-código por U. Dado que las últimas b columnas de todas las matrices de control son la matriz identidad, las últimas columnas de UH coinciden con las de U. Podemos entonces añadir las últimas b columnas de U como subsolicitudes a estos servidores. Luego iteramos por los *delta* grupos. Calculamos los productos UH, sumamos uno en la posición correspondiente y trasladamos las diversas filas como sub-solicitudes. Por último solo nos falta devolver la lista.

```
        queries=[]
        for i in range(N):
            queries.append(zero_matrix(F,ceil(K/delta),M))

        for i in range(ceil(K/delta)):
            U=random_matrix(F, M, b);
            for j in range(b):
                queries[N-b+j][i,:]=U[:,j].transpose()
            for j in range(delta):
                Q=U*ParityMatrices[j]
                Q[f+1,i]=Q[f+1,i]+1
                for k in range(K):
                    queries[j*K+k][i,:]=Q[:,k].transpose()
        return queries
```

Código completo del programa:

```
def tercera(C,alpha,M,f):

    N=C.length();
    K=C.dimension();
    F=C.base_field();
    x=F.gen();
    d=C.minimum_distance();

    if alpha!=1:
        print("Number of stripes does not match the required number")
        return -1;
```

```
delta=floor((N-b)/K)

eliminate=srange(b+delta*K,N)
#Nos quedamos con el código formado por las primeras
#b+delta*k coordenadas
C=codes.PuncturedCode(C,eliminate)
#Adaptamos los parámetros al nuevo código
N=C.length();
d=d-len(eliminate);

#Indices para los servidores comunes a todos los grupos
comunes=srange(N-b,N)
todos=range(N)

ParityMatrices=[]

#Para cada grupo
for i in range(delta):
    group=srange(K*i,K*(i+1))+comunes
    quitar=[item for item in todos if item not in group]
    #Nos quedamos con el código generado por dichos servidores
    D=codes.PuncturedCode(C,quitar);
    #Obtenemos la parte inicial de la matriz de control a partir
    #de la generatriz
    H=-D.systematic_generator_matrix()[:,K:K+b].transpose();
    #Las almacenamos

    ParityMatrices.append(H);
queries=[]
for i in range(N):
    #Inicializamos las solicitudes
    queries.append(zero_matrix(F,ceil(K/delta),M))

for i in range(ceil(K/delta)):
    #Creamos la matriz aleatoria
    U=random_matrix(F, M, b);
    for j in range(b):
        #La parte correspondiente a los servidores comunes
        #es la propia matriz
        queries[N-b+j][i,:]=U[:,j].transpose()
    for j in range(delta):
        Q=U*ParityMatrices[j]
        #El resto lo calculamos igual que en el protocolo anterior
        Q[f+1,i]=Q[f+1,i]+1
        for k in range(K):
            #Cada columna es una fila de la solicitud iesima
            queries[j*K+k][i,:]=Q[:,k].transpose()
return queries
```

4.3.3. Implementación del algoritmo de recuperación del archivo

El output de esta función será la habitual forma matricial del archivo. En el caso de los inputs serán también los mismos que en ocasiones anteriores:

- Código que se usó para la codificación de la base de datos: C
- Número de bandas en las que se divide cada archivo: *alpha*
- Lista de respuestas recibidas por parte del servidor: R

El comienzo resulta igual que en otras ocasiones obteniendo los datos necesarios, inicializando variables y punteando el código para reducirlo con los servidores a los que vamos a enviar una solicitud:

```
def tercera(C,alpha,R):

    N=C.length();
    K=C.dimension();
    F=C.base_field();
    FF=R[0][0,0].parent();
    x=F.gen();
    y=FF.gen();

    delta=floor((N-b)/K)

    File=Matrix(FF,alpha,K);

    eliminate=srange(b+delta*K,N)
    D=codes.PuncturedCode(C,eliminate)
    N=D.length();
```

Tenemos ahora que diferenciar los servidores que pertenecen a cada grupo e ir aplicando en ellos el protocolo anterior. Para ello, en primer lugar guardamos en *perGroup* el número de fragmentos que serán recuperados por cada grupo. Ya que hay un total de k fragmentos y *delta* grupos será el cociente de ambos. En el vector *Symbols* iremos introduciendo los fragmentos que conozcamos del archivo codificado. La diferencia entre este protocolo y los anteriores es que en los anteriores siempre recuperábamos los k primeros fragmentos de cada banda, y como el código era sistemático, estos coincidían con los k fragmentos del archivo sin codificar. En esta versión, obtendremos los *perGroup* primeros fragmentos de cada grupo de servidores, que no coincidirán con los primeros de la banda. Las posiciones de los fragmentos que obtendremos las guardaremos en el vector *posicionesTodas*.

```
    perGroup=ceil(K/delta)
    Symbols=vector(FF,delta*perGroup);
```

```
posicionesTodas=[]
```

Ahora iteraremos sobre cada grupo, recuperando los fragmentos. En primer lugar guardamos los primeros *perGroup* índices del grupo. Estos son de los que recuperaremos un fragmento del archivo. Los añadimos a la lista general *posicionesTodas*. Posteriormente realizamos el algoritmo de decodificación del protocolo anterior, es decir sumamos las respuestas de todos los servidores del grupo. Estos son los b últimos (los servidores comunes), y los K del grupo. El resultado serán los fragmentos del archivo que introduciremos en *Symbols*.

```
for i in range(delta):
    posiciones=srange(i*K,i*K+perGroup)
    posicionesTodas=posicionesTodas+posiciones
    for j in range(K):
        Symbols[i*perGroup:(i+1)*perGroup]=
            Symbols[i*perGroup:(i+1)*perGroup]+R[i*K+j].transpose().row(0)
    for j in range(b):
        Symbols[i*perGroup:(i+1)*perGroup]=
            Symbols[i*perGroup:(i+1)*perGroup]+R[N-1-j].transpose().row(0)
```

Una vez realizado el proceso sobre todos los grupos, deberíamos haber recuperado k fragmentos del archivo. Hemos recuperado por tanto k coordenadas de una palabra código de C, podemos interpretar las restantes $d-1$ posiciones como borrones. Entonces podemos usar algoritmos de decodificación para corregir dichos borrones y obtener la palabra fuente original, correspondiente al archivo sin codificar. Dicho algoritmo que usaremos será puntear el código en las posiciones de los borrones y usar el método *unencode*() para recuperar la palabra original.

```
quitar=[item for item in range(N) if item not in posicionesTodas]

C=codes.LinearCode(Matrix(FF,C.systematic_generator_matrix()[:,posicionesTodas]))
File=Matrix(C.unencode(Symbols));
```

Código completo del programa:

```
def tercera(C,alpha,R):

    N=C.length();
    K=C.dimension();
    F=C.base_field();
    FF=R[0][0,0].parent();
    x=F.gen();
    y=FF.gen();

    delta=floor((N-b)/K)
```

```
File=Matrix(FF,alpha,K);

eliminate=srange(b+delta*K,N)
#Reducimos el código eliminando las últimas posiciones
D=codes.PuncturedCode(C,eliminate)
N=D.length();

perGroup=ceil(K/delta)
Symbols=vector(FF,delta*perGroup);
posicionesTodas=[]

for i in range(delta):
    posiciones=srange(i*K,i*K+perGroup)
     posicionesTodas=posicionesTodas+posiciones
     #Sumamos las respuestas obtenidas por cada grupo
     for j in range(K):
       Symbols[i*perGroup:(i+1)*perGroup]=
           Symbols[i*perGroup:(i+1)*perGroup]+R[i*K+j].transpose().row(0)
    for j in range(b):
        Symbols[i*perGroup:(i+1)*perGroup]=
           Symbols[i*perGroup:(i+1)*perGroup]+R[N-1-j].transpose().row(0)

#Tenemos ahora que obtener la palabra que genera los simbolos en Symbols

quitar=[item for item in range(N) if item not in posicionesTodas]

#Para ello volvemos a reducir el código a partir de la matriz generatriz

C=codes.LinearCode(
   Matrix(FF,C.systematic_generator_matrix()[:,posicionesTodas]))
 File=Matrix(C.unencode(Symbols));
```

4.4. Agrupación de los métodos

En este capítulo hemos visto tres protocolos distintos. Cada uno de ellos tenía un distinto algoritmo de creación de solicitudes y decodificación de las respuestas. Sin embargo, existe una estrecha relación entre los tres protocolos. Los tres funcionan únicamente en el caso de una base de datos codificada por ún código MDS y sistemático. La única diferencia es que dependiendo de la cantidad de servidores cooperantes, será más rentable el uso de uno o de otro. Por ello agruparemos todos los métodos de creación de solicitudes y todos los de recuperación del archivo en dos programas conjuntos. En función de la cantidad de servidores cooperantes y de la eficiencia deseada se elegirá uno u otro.

Los programas en sí son muy simples pero por dar completitud al documento se incluirá el código correspondiente

Programa de recuperación del archivo:

```
def PIRQueryMatrices(C, alpha, M, b, f, flag=false):


  queries=[];
  d=C.minimum_distance()

  if b>=d:
      print("PIR cannot be achieved with that many colluding servers")
      return -1

  elif b==1:
      queries=primera(C,alpha,M,f)

  elif  not flag :
      queries=segunda(C,alpha,M,f)

  else:
      queries=tercera(C,alpha,M,f)

  return queries;
```

Programa de recuperación del archivo:

```
def PIRDecodeFromResponses(C,alpha,b,R,flag=false):


   F=C.base_field();

   if b==1:
       File=primera(C,alpha,R)
   elif not flag:
       File=segunda(C,alpha,R)
   else:
       File=tercera(C,alpha,R)

   file=DecodeToFile(File,F)


   return file
```

Capítulo 5

PIR en códigos arbitrarios

En el capítulo anterior abordamos el caso de bases de datos codificadas de forma sistemática con códigos MDS. Sin embargo, únicamente una pequeña parte de los códigos lineales cumplen ambas condiciones. Por ello tiene sentido tratar de adaptar los protocolos anteriores a una nueva versión donde se pueda aplicar a un código lineal general. Otra opción es desarrollar un protocolo totalmente nuevo que nos permita realizar dicho proceso.

En este capítulo explicaremos un protocolo con dichas características. Este protocolo fue introducido en [6] y es una generalización del segundo protocolo descrito en el capítulo anterior. Veremos como podemos aplicarlo en el caso de un código lineal arbitrario y además en el caso de que el código usado para codificar la base de datos se trate de un código GRS.

Para poder explicar el funcionamiento del código debemos antes introducir un nuevo concepto que nos permitirá crear las solicitudes de una forma distinta. Este concepto es el producto estrella entre dos vectores y entre dos códigos.

5.1. Producto estrella

Entre las distintas operaciones que podemos realizar entre dos vectores de misma dimensión están la suma/resta y el producto escalar. La suma de dos vectores consiste en la suma componente a componente de los mismos, en cambio, el producto escalar consiste en la suma de los productos entre las distintas componentes de los vectores. Es lógico plantearse un producto de dos vectores con un funcionamiento similar al de la suma, componente a componente.

Definición 5.1. Sea $\mathbb{F}_q^n$ un espacio vectorial de dimensión n sobre el cuerpo $\mathbb{F}_q$. Entonces el producto de Hadamard o producto estrella entre dos vectores $v = [v1, \ldots, v_n]$ y $w = [w_1, \ldots, w_n]$ se define como el producto componente a componente de los dos vectores, es decir, $v \star w = [v_1 w_1, \ldots, v_n w_n]$.

Ahora podemos extender esta operación entre vectores a una operación entre subespacios de la misma forma que podemos hacer con la suma. Dado que un código es en esencia un subespacio vectorial, tenemos la siguiente definición.

Definición 5.2. Sean V, W dos subespacios vectoriales de $\mathbb{F}_q^n$. Entonces definimos el producto estrella (de Shur) de V y W como el subespacio generado por el producto de Hadamard de los elementos de V y W

$$V \star W = span(\{v \star w | v \in V, w \in W\})$$

Al igual que en el caso de la suma de espacios vectoriales, un conjunto generador de $V \star W$ se puede obtener a partir de los productos entre los generadores de V y W.

Esta operación que puede parecer muy simple tiene propiedades que resultan muy interesantes en teoría de códigos.ñ

Proposición 5.3. *El producto estrella de dos códigos C y D sobre $\mathbb{F}_q^n$ tiene las siguientes propiedades.*

1. *Si D es el código de repetición de longitud n $Rep(n)_q$, es decir, el código formado por las palabras con todas las componentes iguales, entonces $C \star D = C$.*
2. *Si $(C \star D)^\perp = E$ para cierto código E, entonces $d_E \geq d_{C^\perp} + d_{D^\perp} - 2$ donde d indica la distancia mínima de un código.*
3. *Si C es un código MDS, entonces $(C \star C^\perp)^\perp = Rep(n)_q$*
4. *Si C y D son códigos GRS con parámetros $C = GRS_{k_1}(\alpha, v_1), D = GRS_{k_2}(\alpha, v_2)$. Entonces $C \star D = GRS_{\text{mín}\{k_1+k_2-1,n\}}(\alpha, v_1 \star v_2)$.*

Sage tiene un método que implementa el producto estrella entre dos vectores. Dicho método pertenece a la clase *vector* y se trata de *pairwise_product*(). Para usarlo simplemente tenemos que llamarlo a partir de uno de los vectores involucrados en el producto de la siguiente forma:

```
In [ ]: v.pairwise_product(w)
```

Sin embargo, no existe ningún método en *Sage* que implemente el producto estrella entre dos códigos. Dado que lo vamos a necesitar posteriormente le implementaremos.

Los inputs del programa deben ser dos códigos lineales de los que se desea hallar el producto. El output será un código lineal igual al producto estrella de ambos inputs.

En el programa, lo primero que debemos hacer es comprobar que efectivamente podemos realizar el producto estrella entre los dos códigos. Debemos comprobar que la longitud de los códigos y el cuerpo base coincidan. De la misma manera guardaremos en variables en mayúscula los parámetros fundamentales del primer código C, y los del segundo en parámetros en minúscula.

```
def StarProduct(C,D):

    F=C.base_field()
    N=C.length()
```

```
n=D.length()
K=C.dimension()
k=D.dimension()

if F!=D.base_field() or N!=n:
    raise ValueError('Los códigos no tienen las carácteristicas
       comunes necesarias para realizar el producto')
```

A continuación guardamos en dos matrices *gensC*, *gensD* las dos matrices generatrices de C y D. Cada fila de estas matrices es uno de los generadores de su código. Un sistema generador del producto estrella lo podemos obtener a partir de los productos estrella de los generadores de C con los de D. Si ahora agrupamos en una matriz G por filas cada uno de los resultados de estos productos, la imagen de dicha matriz generará el producto estrella de ambos códigos. Sin embargo G puede no ser su matriz generatriz, ya que es posible que entre las distintas filas haya algunas linealmente dependientes entre ellas. A pesar de ello, podemos usarla como matriz generatriz ya que *Sage* se encargará por nosotros de eliminarlas.

```
gensC=C.generator_matrix()
gensD=D.generator_matrix()
G=Matrix(F,0,n)

for i in range(k):
    for j in range(K):
      G=G.stack(Matrix(F,gensD.row(i).pairwise_product(gensC.row(j))))

return codes.LinearCode(G)
```

5.2. Protocolo general

Con este nuevo concepto de producto estrella somos capaces ya de explicar el nuevo método que nos permitirá obtener PIR en el caso de una base de datos codificada por un código arbitrario. La única condición que pediremos a C es que las primeras c (definido posteriormente) columnas de su matriz de control sean linealmente independientes. Si somos capaces de reordenar los servidores esta condición no es muy restrictiva. La principal diferencia entre este protocolo y los que hemos visto hasta ahora es la selección de la parte aleatoria de la solicitud. Anteriormente tomábamos elementos aleatorios. Sin embargo, en este caso la parte aleatoria serán palabras de un código D que elegiremos con ciertas propiedades.

Supongamos entonces que deseamos recuperar un archivo de una base de datos codificada usando un código lineal C de tipo $[n, k, d]$ sobre $\mathbb{F}_q$. Como hemos dicho usaremos un código D que podremos elegir a nuestro gusto, pero la eficiencia del protocolo dependerá de la elección de dicho código. Sea ahora $d_{C\star D}$ la distancia mínima del producto estrella de C y D, y $c = D_{C\star D}$. Entonces el protocolo logrará descargar c fragmentos por sub-solicitud. El tamaño óptimo en bandas de un archivo entonces será $\alpha = \dfrac{mcm(c,k)}{k}$. Al igual que en los

casos anteriores, supondremos que los archivos están divididos en exactamente α bandas, pero en caso contrario también podríamos efectuar el protocolo. En dicho caso, la cantidad necesaria de solicitudes para recuperar el archivo al completo será $\rho = \dfrac{mcm(c,k)}{c}$.

El papel que desempeña el código D que elijamos es muy grande ya que de él dependerá el cPoP del protocolo y la cantidad máxima de servidores cooperantes contra los que se protege. En concreto este protocolo protegerá contra $b = d_{D^\perp} - 1$ servidores cooperantes y tiene un cPoP de $n/d_{C\star D-1}$

5.2.1. Creación de las solicitudes

Como se ha explicado anteriormente, seremos capaces de obtener c fragmentos del archivo por cada sub-solicitud. Dichos fragmentos provendrán siempre del mismo conjunto de servidores. A este subconjunto le llamaremos J. El tamaño de J dependerá de los valores de c y k:

$$|J| = max\{c, k\}$$

El subconjunto que podemos usar para recuperar los fragmentos puede ser cualquiera del tamaño indicado. Por simplicidad diremos que son los $|J|$ primeros y por tanto $J = 1, \ldots, |J|$. Algo necesario de clarificar es que no en todas las sub-solicitudes recibiremos un fragmento de cada servidor en J ya que esa cantidad siempre será c aunque el tamaño de J sea mayor. El conjunto de servidores de los que recibimos un fragmento en una sub-solicitud i lo denominaremos J_i. Como es lógico, J_i será un subconjunto de J para todos los valores de i.

El proceso para recuperar los fragmentos es por sub-solicitudes, inicialmente generaremos la primera fila de la matriz de solicitud e iremos creándolas una a una hasta obtener las ρ filas. Daremos el algoritmo para generar la primera de ellas y cómo obtener el resto a partir de ella.

El primer paso es generar un total de $m\alpha$ palabras aleatorias del código de las solicitudes D, $d^{p,a} = (d^1_{p,a}, \ldots, d^n_{p,a})$ con $p \leq m, a \leq \alpha$. A partir de ellas definiremos n vectores d_j tomando la coordenada j-ésima de cada una de las palabras código aleatorias.

$$d_j = (d_j^{1,1}, \ldots, d_j^{1,\alpha} \ldots d_j^{m,1}, \ldots, d_j^{m,\alpha}) \in \mathbb{F}_q^{m\alpha}$$

Definimos el primer conjunto de servidores de donde extraeremos fragmentos J_1 como los primeros c servidores. Ahora dividiremos estos c servidores en α grupos de $g = c\alpha$ servidores. Por la definición de α esta división es siempre exacta. Los diferentes grupos serán los siguientes:

$$J_1^1 = \{1, \ldots, g\}, J_1^2 = \{g+1, \ldots, 2g\}, \ldots\ldots, J_1^\alpha = \{(\alpha-1)g+1, \ldots, \alpha g = c\}$$

La sub-solicitud correspondiente a un servidor l vendrá dada por el grupo al que pertenezca de la siguiente forma:

$$q_{l,1} = \begin{cases} d_l + e_{\alpha(f-1)+a} & \text{si } l \in J_1^a \\ d_l & \text{si } l \notin J_1 \end{cases}$$

donde $e_{\alpha(f-1)+a}$ es el vector canónico con un uno en la posición $\alpha(f-1)+a$ y cero en el resto.

El proceso para calcular el resto de solicitudes es muy similar al de la primera. Tendremos que generar inicialmente $m\alpha$ palabras aleatorias y calcular los d_j a partir de ellas. Donde el proceso cambia es en la selección de los J_i^a. Definiremos estos conjuntos de servidores a partir del anterior de la siguiente forma:

$$J_i^a = \{j_1 + g, \ldots, j_g + g\}$$

donde si para alguno de los índices $j_k+g \notin J$ entonces lo sustituiremos por $(j_k+g) \bmod |J|$. Podemos decir pues, que J_i^a se calcula rotando cíclicamente sobre J los elementos de J_{i-1}^a un total de g posiciones. De esta forma ya podremos calcular la subsolicitud correspondiente a un serviodr l de forma similar a la anterior:

$$q_{l,i} = \begin{cases} d_l + e_{\alpha(f-1)+a} & \text{si } l \in J_i^a \\ d_l & \text{si } l \notin J_i \end{cases} \tag{5.1}$$

5.2.2. Recuperación del archivo

Para la decodificación de los archivos usaremos un resultado que nos da la estructura de las respuestas que recibimos.

Proposición 5.4. *La agrupación de las respuestas de los n servidores a la misma sub-solicitud i, $r_i = (r_{i,1}, r_{i,2}, \ldots, r_{i,n})$, esta formada por la suma de una palabra de $C \star D$ más un vector v_{i,J_i} con soporte únicamente en J_i.*

Es por esta razón que el uso de un código lineal en las solicitudes es tan útil. Ahora si multiplicamos las respuestas por la matriz de control dd $C \star D$ obtendremos valores que dependerán únicamente de los fragmentos del archivo correspondientes a los servidores en J_i. Esto se debe a que cualquier palabra de un código multiplicada por su matriz de control es 0. De hecho si H es dicha matriz de control

$$Hr_i = Hz_i + Hv_{i,J_i} = Hv_{i,J_i}$$

Únicamente las c posiciones correspondientes a los servidores de J_i del vector v_{i,J_i} son distintas de 0. Además por definición de c y por la proposición 2.14, todo conjunto de c columnas de H es linealmente independiente. Esto nos dice que el siguiente sistema lineal tiene una única solución, que serán c fragmentos de c correspondientes a los $e_{\alpha(f-1)+a}$ que sumamos en las solicitudes.

$$r_i = (r_{i,1}, r_{i,2}, \ldots, r_{i,n}) = Hv_{i,J_i}$$

Resolviendo este sistema obtendremos c fragmentos y tras ρ sub-solicitudes habremos obtenido el archivo al completo.

5.2.3. Implementación del algoritmo de creación de solicitudes

El output del algoritmo será una lista de matrices de solictudes. Los input que necesitaremos serán los siguientes:

1. Código con el que se ha codificado la base de datos: C
2. Código que usaremos para generar las solicitudes: D
3. Número de bandas en las que se divide el archivo: $alfa$
4. Número de archivos que forman la base de datos: M
5. Índice del archivo que deseamos obtener: f
6. Distancia mínima del código $C \star D$, en caso de no indicarse se calculará manualmente: $d = -1$

El primer paso como suele ser habitual es el de extraer parámetros de los que hemos recibido por parte del usuario y comprobar que el número de bandas coincide con el esperado. Cabe destacar que en caso de que el usuario no haya introducido ningún valor para d, dicho valor se calculará a mano y en muchos casos será un coste computacional muy elevado.

```
def PIRStarQueryMatrices(C,D,alfa,M,f,d=-1):

    N=C.length()
    K=C.dimension()
    F=C.base_field()


    if d==-1:
        Star=StarProduct(C,D)
        d=Star.minimum_distance()
    c=d-1

    alpha=lcm(c,K)/K
    rho=lcm(c,K)/c
    g=c/alfa
    size=max(c,K)

    if alpha!=alfa:
            print("Number of stripes does not match the required number")
            return -1;

    queries=[]
    for i in range(N):
        queries.append(Matrix(F,rho,alfa*M))
```

A continuación comenzamos con el proceso de creación de las solicitudes. Iteramos sobre cada una de las *rho* sub-solicitudes. Empezaremos creando los vectores aleatorios. Crearemos una matriz U con cero columnas e iremos añadiendo las $alfa \cdot M$ palabras aleatorias que iremos generando con el método *random_element*() como filas en la matriz. Los vectores d_j serán las filas de U.

```
        for r in range(rho):
            #Generamos las palabras aleatorias
            U=Matrix(F,N,0)
            for i in range(M):
                for j in range(alfa):
                    U=U.augment(D.random_element().column())
```

Ahora que ya tenemos los vectores d_j procederemos a finalizar el cálculo de las sub-solicitudes. Iteramos sobre cada uno de los servidores. Para cada servidor i, introducimos la fila i de U en la fila de la i-ésima matriz de *queries* correspondiente a la subsolicitud actual. El último paso es sumar uno a la posición correspondiente en el caso de encontrarnos en un servidor de los correspondientes a 5.1, es decir un servidor en J_i. Para calcular si es necesario añadir uno realizamos un cálculo de índices. En la primera sub-solicitud, los servidores correspondientes son los c primeros. En la r-ésima son los servidores con índices $\{r \cdot g, \ldots, (r+1) \cdot g - 1\} mod \quad size$. Por ello para calcular si a un servidor le corresponde añadir uno, restamos $r \cdot g$ a su índice. Si el resultado en menor que c sí tenemos que realizar la resta. Además si realizamos la división entera entre g obtendremos a cuál de los subgrupos pertenece. Con esta información ya podemos usar la fórmula 5.1 para comprobar si tenemos que realizar la resta y cual es la posición correspondiente.

```
            for i in range(N):
                queries[i][r,:]=U[i,:]
                pos=floor(((i-r*g)%size)/g)
                if i<size and pos<alfa:
                    queries[i][r,(f-1)*alfa+pos]+=1

        return queries
```

Código completo del programa:

```
Out[12]: [10, 6] linear code over GF(5)
```

Protocolo 2

```
    def PIRStarQueryMatrices(C,D,alfa,M,f,d=-1):

        N=C.length()
        K=C.dimension()
        F=C.base_field()
```

```
if d==-1:
    Star=StarProduct(C,D)
    d=Star.minimum_distance()
c=d-1

alpha=lcm(c,K)/K
rho=lcm(c,K)/c
g=c/alfa
size=max(c,K)

if alpha!=alfa:
        print("Number of stripes does not match the required number")
        return -1;

queries=[]
for i in range(N):
    queries.append(Matrix(F,rho,alfa*M))

for r in range(rho):
    #Generamos las palabras aleatorias
    U=Matrix(F,N,0)
    for i in range(M):
        for j in range(alfa):
            U=U.augment(D.random_element().column())
    for i in range(N):
        queries[i][r,:]=U[i,:]
        pos=floor(((i-r*g)%size)/g)
        if i<size and pos<alfa:
            queries[i][r,(f-1)*alfa+pos]+=1

return queries
```

5.2.4. Implementación del algoritmo de recuperación del archivo

En el caso de este programa, el output será el archivo completo en su forma original.

Los inputs serán los siguientes:

1. Código con el que se ha codificado la base de datos: C
2. Código resultado del producto estrella entre C y D: $Star$
3. Número de archivos que forman la base de datos: M
4. Lista de respuestas de los servidores a las solicitudes: R
5. Distancia mínima del código $C \star D$, en caso de no indicarse se calculará manualmente: $d = -1$

El principio del programa como en casos anteriores es extraer de los parámetros valores que necesitaremos. En el caso de no haber recibido ningún valor de d realizamos el cálculo. Realizamos también una comprobación sobre el número de bandas y agrupamos la lista de respuestas en una matriz *Response*, donde cada una de las respuestas es una columna. Posteriormente inicializamos dos matrices, *File* donde iremos añadiendo los fragmentos del archivo original en forma matricial sin codificar, y la matriz *Obtenidos* donde iremos introduciendo los fragmentos que recuperemos del archivo en forma matricial codificado. Esta última matriz tiene solo *size* columnas, ya que los fragmentos que recuperaremos serán siempre de los primeros *size* servidores.

```
def PIRStarDecodeFromResponses(C,Star,alfa,M,R,d=-1):
    N=C.length()
    K=C.dimension()
    F=C.base_field()
    FF=R[0][0,0].parent()

    if d==-1:
        d=Star.minimum_distance()

    c=d-1

    alfa=lcm(c,K)/K
    rho=lcm(c,K)/c
    g=c/alpha
    size=max(c,K)

    if alpha!=alfa:
            print("Number of stripes does not match the required number")
            return -1;

    H=Star.parity_check_matrix()
    Response=Matrix(FF,rho,0)
    for i in range(N):
        Response=Response.augment(R[i])

    File=Matrix(FF,0,K)
    Obtenidos=Matrix(FF,alfa,size)
```

Una vez creados y obtenidos estos datos podemos pasar a realizar el procedimiento sobre las respuestas. Realizaremos el mismo proceso con todas las sub-solicitudes, así que iteraremos sobre el número de ellas *rho*. Sin embargo, antes de empezar la iteración crearemos dos variables *downlimit* y *uplimit* que indicarán los índices entre los cuales se encuentran los servidores de J_i. A lo largo del bucle tendremos que diferenciar siempre dos casos, el de $downlimit < uplimit$ en el que los servidores de J_i se encuentran entre estos índices, y el caso contrario, en el que los servidores en J_i tienen índices $\{downlimit, \ldots, size-1\} \cup \{0, \ldots, uplimit\}$. Dependiendo del caso guardaremos los índices de los servidores en la variable *rango*.

```
    downlimit=0
```

```
uplimit=c


for i in range(rho):
    if downlimit < uplimit:
        rango=range(downlimit,uplimit);
    else:
        rango=range(0,uplimit)+range(downlimit,size)
```

El siguiente paso es construir los sistemas lineales. La matriz del sistema lineal es H, pero como el vector solución tendrá soporte únicamente en *range* podemos ignorar el resto. Guardamos en la matriz Hi la matriz resultante de seleccionar estas columnas. El término independiente del sistema es el producto entre la fila i de la matriz *Response* (las respuestas de los servidores a la i-ésima sub-solicitud) y la matriz de control H. Dicho resultado lo guardamos en el vector columna x.

```
Hi=H[:,rango]
Hi=Matrix(FF,Hi)
x=Hi.solve_right(H*Response.row(i).column())
```

El vector x contiene c fragmentos del archivo. Ahora tenemos que identificar cual es la posición de cada uno de estos fragmentos en el archivo original codificado. Sabemos que los servidores de los que hemos recibido uno de los fragmentos son los que tienen índice en *rango* así que tenemos que identificar en que grupo J_i^a se encontraba cada uno de ellos. En todos los casos, el grupo $a = 0$ serán los g primeros desde *downlimit*. En el caso de $a = 1$ los g siguientes y así sucesivamente. Empezaremos introduciendo los valores de x desde la posición *downlimit*.

Indicaremos con una variable *posicionO* la columna de la matriz *Obtenidos* en la que introduciremos el próximo elemento de x. Indicaremos el índice de x cuyo valor será el siguiente en ser introducido en *Obtenidos*, con la variable *posicionR*. Los valores iniciales de estas variables serán *downlimit* en el caso de *posiconO* y para el caso de *posicionR* tendremos que diferenciar los casos anteriores. En ambos casos, el valor que le tenemos que dar es la posición del índice *downlimit* en la lista *rango*.

Una vez dados los valores iniciales, iteramos sobre el número de bandas. En cada una de ellas introduciremos g fragmentos, así que iteramos también sobre g. Introducimos la posición *posicionR* de x en la banda actual y columna *posicionO* de Obtenidos. Preparamos los valores de *posicionR* y *posiconO* para el siguiente valor. En ambos casos simplemente tenemos que incrementar en uno el índice, teniendo en cuenta que en el caso de llegar a los valores máximos posibles para cada uno de ellos, tenemos que volver a la posición 0.

Tras acabar de decodificar una sub-solicitud incrementamos en g los valores de *downlimit* y *uplimit* para proceder con la siguiente.

```
if downlimit < uplimit:
    posicionR=0
else:
    posicionR=uplimit
```

```
    posicionO=downlimit
    for a in range(alfa):
        for j in range(g):
            Obtenidos[a,posicionO]=x[posicionR,0]
            posicionR=(posicionR+1)%c
            posicionO=(posicionO+1)%size
    downlimit=(downlimit+g)%size
    uplimit=(uplimit+g)%size
```

En este momento, tendremos recuperados en *Obtenidos* K valores de cada banda del archivo. Tenemos que obtener a partir de ellos, los K fragmentos del archivo original en forma matricial. Para ello realizamos un proceso similar al protocolo anterior en el que punteamos en las coordenadas que no tenemos un fragmento y decodificamos en el código punteado. Simplemente tenemos que tener cuidado de estar eligiendo las columnas correctas de la matriz *Obtenidos* en cada una de las solicitudes. En la primera banda serán las K primeras. En el caso del resto podemos obtenerlas sumando g a las columnas de la banda anterior.

Para realizar el proceso usamos las variables *downlimit* y *uplimit* para marcar estos dos límites que hemos hablado, que tendrán valores iniciales de 0 y K. Obtenemos los rangos de las columnas diferenciando entre los casos como anteriormente. Ahora guardamos en *simbolos* los valores de la matriz *Obtenidos* en la banda actual y las columnas de *rango*. Son los valores que usaremos para obtener los originales. Punteamos la matriz G y resolvemos el sistema lineal para recuperar la banda del archivo. La añadimos al archivo *File* y actualizamos los valores de *downlimit* y *uplimit* para la siguiente banda.

Tras iterar por todas las bandas tenemos el archivo en forma matricial sin codificar, por lo que podemos usar el método *decodeToFile*() para recuperar el archivo.

```
downlimit=0
uplimit=K

for i in range(alfa):
    if downlimit<uplimit:
        rango=range(downlimit,uplimit)
    else:
        rango=range(0,uplimit)+range(downlimit,size)

    simbolos=Obtenidos[i,rango]
    G=Matrix(FF,C.generator_matrix()[:,rango])
    x=G.solve_left(simbolos)
    File=File.stack(x)
    downlimit=(downlimit+g)%size
    uplimit=(uplimit+g)%size

 file=DecodeToFile(File,F)
```

Código completo del programa

```
def PIRStarDecodeFromResponses(C,Star,alfa,M,R,d=-1):
    N=C.length()
    K=C.dimension()
    F=C.base_field()
    FF=R[0][0,0].parent()

    if d==-1:
        d=Star.minimum_distance()

    c=d-1

    alfa=lcm(c,K)/K
    rho=lcm(c,K)/c
    g=c/alpha
    size=max(c,K)

    if alpha!=alfa:
            print("Number of stripes does not match the required number")
            return -1;

    H=Star.parity_check_matrix()
    Response=Matrix(FF,rho,0)
    for i in range(N):
        Response=Response.augment(R[i])

    File=Matrix(FF,0,K)
    Obtenidos=Matrix(FF,alfa,size)

    #Limite inferior de los servidores de los cuales obtenemos
    #símbolos en la primera subsolicitud
    downlimit=0
    #Limite superior de los servidores de los cuales obtenemos
    #símbolos en la primera subsolicitud
    uplimit=c

    #Recuperamos los valores y los colocamos en una matriz de
    #acuerdo a su posición

    for i in range(rho):
        if downlimit < uplimit:
            rango=range(downlimit,uplimit);
        else:
            rango=range(0,uplimit)+range(downlimit,size)
        Hi=H[:,rango]
        Hi=Matrix(FF,Hi)
        x=Hi.solve_right(H*Response.row(i).column())

        if downlimit < uplimit:
            posicionR=0
```

```
        else:
            posicionR=uplimit

        posicionO=downlimit
        for a in range(alfa):
            for j in range(g):
                Obtenidos[a,posicionO]=x[posicionR,0]
                posicionR=(posicionR+1)%c
                posicionO=(posicionO+1)%size

        downlimit=(downlimit+g)%size
        uplimit=(uplimit+g)%size

    #Decodificamos cada una de las bandas

    #Limite inferior de los servidores de los cuales obtenemos
    #símbolos en la primera banda
    downlimit=0
    #Limite superior de los servidores de los cuales obtenemos
    #símbolos en la primera banda
    uplimit=K

    for i in range(alfa):
        if downlimit<uplimit:
            rango=range(downlimit,uplimit)
        else:
            rango=range(0,uplimit)+range(downlimit,size)

        simbolos=Obtenidos[i,rango]
        G=Matrix(FF,C.generator_matrix()[:,rango])
        x=G.solve_left(simbolos)
        File=File.stack(x)
        downlimit=(downlimit+g)%size
        uplimit=(uplimit+g)%size



    ################## Reconstruct original file ################


    file=DecodeToFile(File,F)


return file
```

5.2.5. Particularidades en códigos GRS

Como dijimos a principio de la sección, este protocolo se puede realizar sobre cualquier código C. Sin embargo, para un código arbitrario las caracterísiticas que tendrá el protocolo pueden ser muy malas. Estas características todas dependen de la elección del código de solicitudes D. Tiene sentido entonces buscar un código D que mejore lo máximo posible el cPoP y los servidores cooperantes contra los que el protocolo protege. El problema es lo impredecible que resulta el producto estrella entre dos códigos arbitrarios.

Es fácil elegir un código D de forma que se proteja contra muchos servidores cooperantes. También es posible, aunque más difícil encontrar uno que nos garantice un $cPoP$ elevado. Sin embargo buscar un código D tratando de maximizar ambos valores a la vez es muy complicado en códigos arbitrarios. Resulta lógico entonces estudiar el comportamiento en códigos que se comporten bien con el producto estrella. Por ello, y por su propiedad de código MDS, nos centraremos en el estudio de los códigos GRS.

Teorema 5.5. *Sea $C = GRS_j(\alpha, v)$ el código usado para codificar una base de datos en n servidores. Entonces para cada $1 \leq b \leq n-k$ existe un código para las solicitudes D tal que el protocolo anterior protege contra b servidores cooperantes y con el que se consigue un cPoP de $n/(n-k_C-b+1)$*

De hecho existirán una gran cantidad de ellos. Será cualquiera de la forma $D = GRS_b(\alpha, u)$ para cualquier vector $u \in \mathbb{F}_q^n$. De esta forma, por las propiedades de los códigos GRS y el producto estrella, $C \star D = GRS_{k+b-1}(\alpha, u \star v)$

Proposición 5.6. *Sean C y D dos códigos, de dimensiones k_C, k_D respectivamente y con soporte en $1 \ldots n$. Sea $E = C \star D$. Entonces*

$$d_E \leq max\{1, n-k_1-k_2+2\}$$

Además si ni C ni D ni $(C \star D)^{\perp}$ es el código de repetición $Rep_q(n)$, entonces se da la igualdad si ambos son códigos GRS sobre la misma n-upla α.

Este resultado lo que nos garantiza además es que si se desea proteger contra b servidores, el mejor cPoP al que podemos aspirar es exactamente el que hemos obtenido, por lo que la elección de D es óptima.

$$cPoP = \frac{n}{d_{C\star D}-1} = \frac{n}{n-k_C-b+1}$$

Para facilitar la obtención de estos códigos, también se ha implementado una pequeña función que para un código GRS C y una cantidad de servidores cooperantes deseada b devuelve un código de solicitud D de forma que se protege contra b servidores cooperantes y el cPoP será de $n/(n-k_C-b+1)$.

Código completo del programa

```
def GetGRSQueryCode(C,b):
    N=C.length()
    K=C.dimension()
```

```
if b>N-K:
    raise ValueError('No se puede realizar PIR para esa cantidad de
       servidores cooperantes')
D=codes.GeneralizedReedSolomonCode(C.evaluation_points(),b,
   vector(C.base_field(),ones_matrix(1,N).row(0)))
Star=codes.GeneralizedReedSolomonCode(C.evaluation_points(),K+b-1,
   C.column_multipliers())
return(D,Star)
```

Capítulo 6

PIR en canales con ruido

Hasta este momento hemos estado describiendo el proceso de comunicación entre usuario y servidores como un proceso perfecto, en el que no puede haber pérdida de información. Sin embargo, en una situación real es muy probable que haya pérdida de información, especialmente si el canal es muy inestable. En este capítulo veremos e implementaremos un primer método que permite recuperar archivos en canales con ruido y más adelante una mejora de dicho método para códigos GRS.

El primero de estos dos métodos fue introducido en [8] y posteriormente fue mejorado por los mismos autores en [7]. En este documento no se explicará en profundidad las razones del funcionamiento de estos protocolos, pero si se desea conocer más información se puede consultar tanto estos dos artículos o una explicación más detallada en el TFG de matemáticas [1].

Una transmisión de información siempre puede ser alterada por ruido que haya en el canal. Este ruido puede provocar dos importantes efectos. En primer lugar parte de la información que ha sido enviada puede quedar irreconocible para el receptor del mensaje. A cada uno de los símbolos del mensaje que hayan quedado irreconocibles los llamamos borrones. Un borrón también puede ocurrir cuando no se recibe respuesta alguna. La segunda opción es que la parte que ha sido afectada por el ruido haya sido alterada de forma que el receptor puede interpretarla de una forma distinta a la que ha sido enviada. A cada uno de estos símbolos que ha sido alterado a otro distinto los denominamos errores.

6.1. Generador de ruido

Hasta ahora nunca habíamos considerado el caso de recibir respuestas con ruido. Por eso, el método que computa las respuestas de los servidores no tiene esta funcionalidad. Crearemos un nuevo programa que añada el ruido a las solicitudes.

El programa tendrá dos outputs. En primer lugar las respuestas de los servidores con parte de ellas modificadas por errores. En segundo lugar, na matriz de ceros y unos representando los borrones. Un uno en la fila i y columna j se interpreta como un borrón en la respuesta del servidor j a la sub-solicitud i.

Los inputs del programa serán similares a los del caso sin ruido:

- Base de datos: $CodedDB$
- Lista de solicitudes: $query$
- Número máximo de errores permitidos por sub-solicitud: Z
- Número máximo de borrones permitidos por sub-solicitud: R
- Lista de indices de los servidores a los que dirigimos las solicitudes: $servers = -1$

En este programa empezamos usando la función sin ruido para calcular las respuestas y luego añadimos los borrones y errores. Para ello simplemente usamos la clase *Channel* de Sage, que nos proporciona métodos para crear los errores y borrones.

Código completo del programa:

```
def PIRQueryToServerNoise(CodedDB,Query,Z,R,servers=-1):
    Response=PIRQueryToServer(CodedDB,Query,servers);

    rho=Response[0].nrows()
    N=len(Response)
    F=Response[0][0,0].parent()

    Grouped=Matrix(F,rho,0)
    for i in range(N):
        Grouped=Grouped.augment(Response[i])

    Erasures=Matrix(GF(2),0,N)

    Channel=channels.ErrorErasureChannel(VectorSpace(F,N),Z,R)

    for i in range(rho):
        x=Channel(Grouped.row(i))
        Grouped[i,:]=x[0]
        Erasures=Erasures.stack(x[1])

    Response=[]
    for i in range(N):
        Response.append(Grouped[:,i])


    return [Response,Erasures]
```

Ahora si podemos empezar a describir el protocolo que nos permitirá recuperar el archivo mientras corregimos errores y borrones. Ya hemos visto en el segundo capítulo que podemos corregirlos en palabras de un código lineal. El algoritmo que explicaremos a continuación se basa en la corrección que poseen los códigos lineales para corregir los mensajes alterados por ruido.

6.2. Protocolo inicial

En este protocolo descargaremos una banda por sub-solicitud, así que no necesitamos suponer que los archivos tienen un número determinado de bandas. Al igual que casos anteriores supondremos que la base de datos está codificada usando un código lineal sobre el que no pondremos ninguna condición.

En cuanto al ruido, supondremos que queremos que el protocolo proteja contra una cantidad determinada de borrones y errores en una misma sub-solicitud. Denominaremos la cantidad de borrones o servidores no responsivos como r y la cantidad de errores como z.

Al igual que antes, tendremos que elegir un código para crear solicitudes D del que dependerán los servidores cooperantes contra los que el protocolo protege. Será capaz de recuperar el archivo en presencia de como máximo $b = d_{D^\perp} - 1$ servidores cooperantes.

El cPoP que conseguiremos con este protocolo es de $(n-r)/k$ ya que en cada sub-solicitud recibimos $n - r$ respuestas (hay r que no responden), para recuperar k fragmentos del archivo original.

Una vez explicado estos conceptos podemos empezar con la primera parte del protocolo.

6.2.1. Creación de las solicitudes

Como hemos dicho, el protocolo de esta sección está basado en una versión del anterior protocolo y por ello el proceso de creación de las solicitudes es muy similar. Supondremos que la base de datos tiene m archivos de α bandas y deseamos obtener el archivo f. Iteraremos por cada banda, es decir, en cada sub-solicitud descargaremos una de ellas.

Elegiremos un código D a partir del cual crearemos las solicitudes. La innovación ocurre en la elección de un tercer código E que usaremos para corregir errores y borrones. La elección de E sin embargo no puede ser arbitraria. Debe ser un código de dimensión 1 que en conjunto con los otros dos códigos C y D cumpla las siguientes propiedades.

- La intersección de los códigos $C \star D$ y $C \star E$ es nula
- $C \star E$ tiene dimensión k
- La distancia mínima $d_\star$ del código $C^{\star D}_{\star E} = C \star D + C \star E$ cumple la siguiente desigualdad: $d_\star - 1 \geq 2z + r$

Como E es un vector de dimensión 1, está generado por un único vector que denominaremos v_E.

Una vez encontrado un código E con dichas características generaremos cada sub-solicitud de forma simple. Igual que antes generamos αm palabras aleatorias de D y a partir de ellas creamos n vectores d_j tomando las cordenadas j-ésimas de cada uno de los vectores. La sub-solicitud la obtendremos a partir de estos vectores y del código E mediante:

$$q_{l,1} = d_l + v_{E,l} e_{\alpha(f-1)+1}$$

donde $v_{E,l}$ es la l-ésima componente de v_E y $e_{\alpha(f-1)+1}$ es el vector canónico con un 1 en la posición $\alpha(f-1)+1$.

El resto de sub-solicitudes las generaremos tomando unos vectores d_j distintos, hasta realizar un total de α sub-solicitudes, una por cada banda del archivo.

6.2.2. Recuperación del archivo

Una vez realizadas las solicitudes recibiremos una respuesta de cada servidor, sin embargo, la respuesta que recibimos ha podido ser alterada por ruido. Supondremos que el mensaje ha sido afectado, como dijimos anteriormente, por como máximo z errores y r borrones.

La primera labor que debemos hacer es corregir estos errores para poder extraer los fragmentos del archivo. Para ello nos apoyaremos en el siguiente resultado.

Proposición 6.1. *La agrupación de las respuestas recibidas por los servidores a una misma sub-solicitud, forman una palabra de $C_{\star E}^{\star D} = C\star D + C\star E$ suponiendo que todas estas respuestas son correctas, es decir, sin considerar servidores bizantinos o no responsivos.*

En definitiva, la agrupación de las respuestas a una sub-solicitud que hemos recibido es una palabra del código $C_{\star E}^{\star D}$ al que se le han aplicado los borrones y errores. Al ser una palabra de un código lineal si la cantidad de borrones y errores es suficientemente pequeña podremos corregirlos. Por las suposiciones que hicimos en 6.2.1, la distancia mínima del código es $2z + r + 1$. Esto quiere decir que dicho código puede corregir hasta z errores y r borrones. Por tanto usando algoritmos de decodificación de errores del código podemos recomponer la palabra que ha sido enviada por los servidores.

Proposición 6.2. *La matriz generatriz de $C_{\star E}^{\star D} = C \star D + C \star E$ está formada por la concatenación vertical de las matrices generatrices de $C \star D$ y $C \star E$*

$$M = G_{C_{\star E}^{\star D}} = \begin{pmatrix} G_{C\star D} \\ G_{C\star E} \end{pmatrix}$$

Proposición 6.3. *Sea $C_{\star E}^{\star D} = C\star D + C\star E$ con matriz generatriz M según la proposición anterior. Sea r la palabra respuesta recibida a una misma sub-solicitud con errores y borrones corregidos. Sea m la palabra fuente que genera r: $r = mG$. Entonces las k últimas coordenadas de m (correspondientes a los generadores de $C \star D$) forman una banda del archivo que desamos sin codificar.*

Estos dos resultados anteriores nos dan el método para recuperar todas las bandas. En cada sub-solicitud recuperaremos una de ellas. Tras corregir los errores, creamos la matriz generatriz de $C_{\star E}^{\star D}$ y usamos los algoritmos de decodificación del código para obtener la palabra fuente. Una vez realizado las últimas k coordenadas del resultado son la banda del archivo.

6.2.3. Funcionamiento en códigos GRS

Al igual que en el protocolo anterior, el producto estrella puede hacer muy impredecible una buena elección de códigos D y E. De hecho, las condiciones exigidas en 6.2.1 hacen

que para códigos arbitrarios sea realmente difícil encontrar códigos que satisfagan las tres condiciones.

En el caso de los códigos GRS, dado un código C, existe un método para elegir códigos D y E de forma óptima para aplicar el protocolo. De hecho, este protocolo a pesar de funcionar con un código arbitrario fue diseñado pensando en estos códigos específicamente.

Supongamos que la base de datos ha sido codificada usando un código $C = GRS_k(\alpha, v)$ para dos vectores $\alpha = (\alpha_1, \ldots, \alpha_n)$ y v. Supondremos que queremos conseguir PIR contra b servidores cooperantes en presencia de como máximo r servidores no responsivos y z servidores bizantinos (servidores que pueden crear un error).

En primer lugar, el código D que elegiremos será el mismo que en el caso anterior $D = GRS_b(\alpha, w)$, de forma que $C \star D = GRS_{k+b-1}(\alpha, v \star w)$.

Ahora, en vez de elegir E un código GRS, lo haremos de forma que el código que sea GRS sea $C^{\star D}_{\star E}$. El código E que debemos usar para lograrlo es el código generado por la palabra

$$v_E = (\alpha_1^{b+k-1}, \ldots, \alpha_n^{b+k-1}) \star w$$

Mediante la elección de este código conseguiremos que $C^{\star D}_{\star E}$ sea un código GRS de parámetros $C^{\star D}_{\star E} = GRS_{b+2k-1}(\alpha, v \star w)$. Es fácil comprobar que todas las condiciones de 6.2.1 se cumplen para estos códigos.

Usando estos códigos, el protocolo desarrollado a partir de ellos protegerá contra b servidores cooperantes, sin embargo, la cantidad de errores y borrones que podemos corregir puede ser superior a los b errores y r borrones que inicialmente queríamos corregir. Es por ello que si no deseamos corregir tantos errores y borrones debería poder existir una adaptación de este protocolo que obtenga un mayor cPoP. Dicha adaptación se presentará en la siguiente sección.

6.2.4. Implementación del algoritmo de creación de solicitudes

El programa que implementaremos únicamente funcionará con códigos GRS. La complejidad que genera el producto estrella hace que sea muy complicado encontrar códigos D y E para un código arbitrario C y por tanto, en la práctica la utilidad que puede tener el protocolo es con este tipo de códigos.

El output del programa será una lista de matrices solicitud. Como input usaremos los siguientes parámetros:

- Código GRS con el que se ha codificado la base de datos: C
- Número de servidores cooperantes contra los que se desea proteger: b
- Número de bandas que forman cada archivo: $alpha$
- Número de archivos que forman la base de datos: M
- Índice del archivo que se desea recuperar: f

Inicialmente recuperaremos valores útiles a partir de los parámetros como siempre hacemos. Además de los habituales, al tratarse C de un código GRS, obtendremos *alphaa*, los puntos de evaluación del código. Posteriormente construimos el código D y E tal y como hemos explicado anteriormente.

```
def PIRBizantineQueryMatrices(C,b,alfa,M,f):
    N=C.length()
    K=C.dimension()
    rho=alfa
    F=C.base_field()

    alphaa=C.evaluation_points()
    D=codes.GeneralizedReedSolomonCode(alpha,b,C.column_multipliers())

    E=vector(F,N)
```

El siguiente paso es inicializar las matrices de solicitudes en la lista *queries* y comenzar rellenándolas. Iteraremos sobre el número de sub-solicitudes, igual al número de bandas. En cada una de ellas generamos $M \cdot alpha$ palabras aleatorias y las guardamos en una matriz U, cada una en una columna.

Solo tenemos ya que sumar la posición correspondiente de E en cada una de las filas de U y volcar cada fila en una matriz de *queries* para finalizar el proceso.

```
    for i in range(N):
        E[i]=alpha[i]^(b+K-1)*C.column_multipliers()[i]

    queries=[]
    for i in range(N):
        queries.append(Matrix(F,rho,alfa*M))
    for r in range(rho):
        #Generamos las palabras aleatorias
        U=Matrix(F,N,0)
        for i in range(M):
            for j in range(alfa):
                U=U.augment(D.random_element().column())

        for i in range(N):
            queries[i][r,:]=U[i,:]
            queries[i][r,(f-1)*alfa+r]+=E[i]

    return queries
```

6.2.5. Implementación del algoritmo de recuperación del archivo

En el proceso de recuperación el output será el propio archivo en su forma original. Como input usaremos los siguientes parámetros:

- Código GRS con el que se ha codificado la base de datos: C
- Número de servidores cooperantes contra los que se desea proteger: b
- Número de bandas que forman cada archivo: *alpha*
- Lista de vectores respuestas recibidas de los servidores: R
- Matriz de unos y ceros representando los borrones en las respuestas : *Erasures*. Cada columna se corresponde con uno de los vectores de respuesta. Si en la fila i columna j hay un uno, se interpretará la posición i de la repusta del servidor j como un borrón.

El programa comienza obteniendo valores útiles de los parámetros. En este caso también será necesario recuperar los puntos de evaluación de C, *evalp* y los multiplicadores *mult*. Suponiendo que el código D usado es de la forma estándar para códigos GRS creamos el código $C^{\star D}_{\star E}$ al que llamaremos *Star*. Los puntos de evaluación serán *evalp*, los multiplicadores serán $mult \star mult$ y será de dimensión $2K + b - 1$. Por último extraemos de este nuevo código un decodificador que nos ayudará con el proceso de decodificación, el decodificador de errores y borrones definido en la clase de los códigos GRS.

```
def PIRBizantineDecodeFromResponse(C,b,alfa,R,Erasures):

    N=C.length()
    K=C.dimension()
    k=K+b-1

    F=C.base_field()
    FF=R[0][0,0].parent()

    evalp=vector(FF,C.evaluation_points())
    mult=C.column_multipliers()
    Star=codes.GeneralizedReedSolomonCode(
          evalp,2*K+b-1,mult.pairwise_product(mult))
    decoder=codes.decoders.GRSErrorErasureDecoder(Star)
```

A continuación agrupamos todas las respuestas en una misma matriz *Response* para poder tratar con ellas de forma más cómoda.

```
    rho=alfa

    File=Matrix(FF,alfa,K)

    Response=Matrix(FF,rho,0)
    for i in range(N):
        Response=Response.augment(R[i])
```

Por último iteramos sobre el número de bandas o sub-solicitudes. En cada una de ellas usamos el decodificador para recuperar la palabra fuente que genera la agrupación de las respuestas a una misma sub-solicitud. Tomamos las k últimas posiciones del resultado y las añadimos a una matriz $File$ ya que serán una banda del archivo sin codificar.

Por último, una vez recuperado el archivo sin codificar, usamos la función $DecodeToFile()$ para obtener la forma original del archivo.

```
        for i in range(rho):

            word=decoder.decode_to_message([Response.row(i), Erasures.row(i)])
            if len(word)!=K:
                      zero = zero_vector(FF, K-len(word))
                      word = vector(list(word)+list(zero))
            File[i,:]=word[k:len(word)]

        file=DecodeToFile(File,F)
        return file
```

Código completo del programa

```
    def PIRBizantineDecodeFromResponse(C,b,alfa,R,Erasures):

        N=C.length()
        K=C.dimension()
        k=K+b-1

        F=C.base_field()
        FF=R[0][0,0].parent()

        evalp=vector(FF,C.evaluation_points())
        mult=C.column_multipliers()
        Star=codes.GeneralizedReedSolomonCode(
            evalp,2*K+b-1,mult.pairwise_product(mult))
        decoder=codes.decoders.GRSErrorErasureDecoder(Star)


        rho=alfa

        File=Matrix(FF,alfa,K)

        Response=Matrix(FF,rho,0)
        for i in range(N):
            Response=Response.augment(R[i])

        for i in range(rho):

            word=decoder.decode_to_message([Response.row(i), Erasures.row(i)])
            if len(word)!=K:
```

```
                zero = zero_vector(FF, K-len(word))
                word = vector(list(word)+list(zero))
        File[i,:]=word[k:len(word)]
    file=DecodeToFile(File,F)
    return file
```

6.3. Mejora del protocolo

En esta sección nos centraremos en el caso de que C sea un código GRS y trataremos de mejorar el protocolo de la sección anterior. Hemos visto que si $C = GRS_k(p, v)$, el código D que debemos usar se trata también de un código GRS $D = GRS_b(p, w)$. Por último E sería el código generado por un único vector $E = (p_1^{b+k-1}, \ldots, p_n^{b+k-1})$. Mediante esta elección conseguíamos que $C_{\star E}^{\star D} = GRS_{2k+b-1}(p, v \star w)$.

Si analizamos la capacidad correctora de este protocolo, por las condiciones que teníamos impuestas sobre el código E, tenemos que la distancia mínima $d_\star$ del código $C_{\star E}^{\star D}$ tiene que cumplir la siguiente desigualdad: $d_\star - 1 \geq 2z + r$. Al ser un código MDS, esta desigualdad se traduce a la siguiente:

$$n - (2k + b - 1) \geq 2z + r$$

Esta desigualdad nos da el número de errores y borrones que seremos capaces de corregir. Sin embargo, si conocemos una cota menor sobre el número de errores y borrones, entonces mediante la utilización de este protocolo, tendríamos una capacidad correctora mayor de la requerida. Esto en principio no es ningún problema, pero deberíamos ser capaces de lograr una adaptación de este protocolo reduciendo la capacidad correctora a la que necesitamos, y aumentando la cantidad de fragmentos que recuperamos por solicitud.

El protocolo que crearemos en esta sección logrará este resultado. Al ser similar al anterior, la cantidad de servidores cooperantes contra los que protege será la misma $b = d_{D_\perp} - 1$. En cambio el cPoP será algo mejor: $(n - r)/c$.

6.3.1. Creación de las solicitudes

Lograr esto es posible, realizando una pequeña variación en el protocolo anterior. En esta versión usaremos códigos E_i distintos en cada sub-solicitud i, que serán de dimensión mayor que 1. Supongamos por tanto que nos encontramos en la misma situación que en la sección anterior. Tenemos una base de datos codificada usando un código $C = GRS_k(p, v)$ formada por m archivos. Deseamos obtener el archivo f, protegiendo contra b servidores cooperantes y corrigiendo z errores y r borrones. En primer lugar, definimos la cantidad de fragmentos que descargaremos por sub-solicitud como $c = n - k - b - 2z - r + 1$. El protocolo funciona de forma óptima para el siguiente número de bandas y sub-solicitudes. Durante la elaboración del protocolo supondremos que estos son los parámetros con los que tratamos:

$$\alpha = \frac{mcm(c, k)}{k} \qquad \rho = \frac{mcm(c, k)}{c}$$

El procedimiento de craci\'on de las solicitudes es muy similar al caso anterior. Usaremos el mismo código $D = GRS_b(p, w)$ que en el caso anterior, pero un código E distinto. Iteraremos el mismo proceso para generar cada una de las sub-solicitudes. Inicialmente generaremos $m\alpha$ palabras aleatorias de D, y para cada una de ellas formamos las palabras $d_l \quad l \leq n$ tomando la coordenada l de cada una de los vectores aleatorios. La diferencia aparece en la elección de E. Usaremos un distinto código E_i para cada sub-solicitud i. De hecho la dimensión de estos códigos variará en cada sub-solicitud. Definimos la dimensión de E_i como:

$$dim(E_i) = \lceil ic/k \rceil$$

De esta forma, para la última sub-solicitud $i = \rho$, $dim(E_i) = \alpha$. La matriz generatriz de cada código E_i será la siguiente:

$$G_i = \begin{pmatrix} p_1^{\phi(dim(E_i),i)} & \dots & p_n^{\phi(dim(E_i),i)} \\ p_1^{\phi(dim(E_i)-1,i)} & \dots & p_n^{\phi(dim(E_i)-1,i)} \\ \vdots & \ddots & \vdots \\ p_1^{\phi(2,i)} & \dots & p_n^{\phi(2,i)} \\ p_1^{\phi(1,i)} & \dots & p_n^{\phi(1,i)} \end{pmatrix} \cdot diag(w)$$

donde ϕ es la función $\phi(l, i) = ic - lk + k + b - 1$. Por la forma de ϕ sabemos que todos estos exponentes son mayores que b. Además los exponente de dos filas contiguas distan k unidades entre ellos.

Cada fila de G_i es un generador del código E_i. Denominamos $g_l^{E_i}$ a el generador correspondiente a la potencia $\phi(l, i)$ de p. Entonces a partir de estos generadores podemos obtener cada una de las sub-solicitudes de la siguiente forma:

$$q_{l,i} = d_l + \sum_{a=1}^{\lceil c/k \rceil} e_{\alpha(f-1)+a} g_a^{E_i}(l)$$

Esta fórmula es la misma que en el caso anterior pero dado que E_i tiene una mayor dimensión, debemos sumar cada uno de sus generadores, en vez de su único generador como anteriormente.

6.3.2. Recuperación del archivo

Para recuperar el archivo en el caso anterior nos aprovechábamos de que los códigos $C \star D$ y $C \star E$ eran disjuntos y por tanto podíamos hacer una clara diferenciación entre las primeras $b + k - 1$ coordenadas de la palabra fuente y las k últimas. En este caso, en cambio, ambos códigos no son disjuntos y por tanto no podemos realizar exactamente el mismo proceso, pero sí podemos realizar uno similar.

Dado que además la dimensión de E_i va creciendo a medida que i aumenta, empezaremos explicando cómo decodificar la primera sub-solicitud y como podemos a partir de ella decodificar el resto.

Proposición 6.4. *La matriz del código $C^{\star D}_{\star E_1} = C \star D + C \star E_i$ es la siguiente:*

$$G_\star = \begin{pmatrix} 1 & \dots & 1 \\ p_1 & \dots & p_n \\ \vdots & \ddots & \vdots \\ p_1^{b-1} & \dots & p_n^{b-1} \\ p_1^{b} & \dots & p_n^{b} \\ \vdots & \ddots & \vdots \\ p_1^{b+k-2} & \dots & p_n^{b+k-2} \\ \vdots & \ddots & \vdots \\ p_1^{\phi(1,i)} & \dots & p_n^{\phi(1,i)} \\ \vdots & \ddots & \vdots \\ p_1^{\phi(1,i)+k-1} & \dots & p_n^{\phi(1,i)+k-1} \end{pmatrix} \cdot diag(v \star w)$$

Denominaremos a cada uno de las filas de esta matriz (generadores de $C^{\star D}_{\star E_i}$) como $g^\star_t$ donde t es el exponente de la potencia de p correspondiente al generador.

Proposición 6.5. *Podemos dividir los generadores de $C^{\star D}_{\star E_1}$ en tres grupos distintos dependiendo de la procedencia de los mismos. Estos tres grupos serán los siguientes:*

1. *Las primeras $\phi(dim(E_1), 1)$ filas: $\{g^\star_0 \dots g^\star_{\phi(\lceil c/k \rceil),1)-1}\}$ Son los generadores que solo están en $C \star D$*
2. *$\{g^\star_{\phi(\lceil c/k \rceil,1)} \dots g^\star_{b+k-2}\}$ Son los generadores comunes a ambos*
3. *Los generadores a partir de la fila $b+k$: $\{g^\star_{b+k-1} \dots g^\star_{\phi(1,1)+k-1}\}$ Son los generadores que solo están en $C \star E_1$*

En este caso entonces, ambos códigos no son totalmente disjuntos, pero existe una parte que sí lo es. Intentaremos aprovecharnos de esta parte (los generadores correspondientes al 3º grupo) para recuperar los fragmentos del archivo de la misma forma a como lo hicimos en el protocolo anterior.

Antes de ver cómo descargaremos los fragmentos, daremos un orden sobre los distintos fragmentos del archivo que indicarán el orden en el que los iremos extrayendo. En este orden los fragmentos en las primeras bandas serán los primeros, sin embargo, los símbolos al final de la banda irán delante de los del principio.

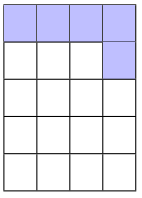

Figura 6.1: Primeros cinco símbolos de un archivo con 5 bandas de dimensión 4

Proposición 6.6. *Sea r la palabra recibida por los servidores como respuesta a la primera sub-solicitud. Dicha palabra tendrá a lo sumo z errores y r borrones. Podemos usar los*

algoritmos de decodificación del código $C^{\star D}_{\star E_1}$ para corregir los errores y borrones y obtener una palabra fuente m tal que $mC = r'$ donde r' es la palabra resultado de corregir los errores y borrones en r. Entonces las últimas c coordenadas de m se corresponden con los primeros c fragmentos del archivo que deseamos.

Usando esta proposición podemos entonces recuperar los primeros c fragmentos del archivo en la primera sub-solicitud. Ahora solamente tenemos que ver como realizar el mismo proceso en el resto de las sub-solicitudes.

Para el caso de una sub-solicitud i supondremos que hemos recuperado los primeros $(i-1)c$ fragmentos del archivo en sub-solicitudes anterior y veremos como descargar los c siguientes.

Para este caso, también podemos dividir los generadores en los tres grupos, al igual que antes. El problema es que nada nos garantiza que todos ellos sean linealmente independientes, es decir, estos generadores no forman una base del código. La respuesta de los servidores será la siguiente palabra, es decir, la suma de una palabra de $C \star D$ y productos entre fragmentos del archivo y generadores de $C \star E$.

$$r_i = (d_l w_l)_{l \in [n]} + \sum_{a=1}^{\lceil ic/k \rceil} \sum_{t=1}^{k} x^f_{a,t} g^{\star}_{\phi(a,i)+t}$$

Sin embargo en sub-solicitudes anteriores ya hemos calculado antes $(i-1)c$ de estos fragmentos. Si restamos cada uno de estos fragmentos multiplicado por el generador correspondiente, nos quedará una palabra del código $C^{\star D}_{\star E_1}$, encontraremos en el mismo caso de la sub-solicitud primera. De la misma forma podemos extraer otros c símbolos a partir de la palabra fuente.

6.3.3. Implementación del algoritmo de creación de solicitudes

El output del algoritmo será una lista de matrices de solictudes. Los input que necesitaremos serán los siguientes:

1. Código con el que se ha codificado la base de datos, debe ser un código GRS: C
2. Número de servidores cooperantes contra los que se protege: b
3. Cantidad de errores que se desea corregir: z
4. Cantidad de borronoes que se desea corregir: r
5. Número de bandas en las que se divide el archivo: $alfa$
6. Número de archivos que forman la base de datos: M
7. Índice del archivo que deseamos obtener: f

El primer paso en el programa, como es habitual, es recuperar parámetros y realizar comprobaciones de que las condiciones que requiere el protocolo se cumplen. En este caso

comprobaremos que la cantidad c de símbolos que recuperamos sea mayor que 0, y que el número de bandas sea el indicado por el protocolo. Después inicializamos las matrices de solicitud para posteriormente rellenarlas.

```
def PIRBetterBizantineQueryMatrices(C,b,z,r,alfa,M,f):
    N=C.length()
    K=C.dimension()
    c=N-K-b-2*z-r+1
    if c<1:
        raise ValueError('No se puede realizar PIR para esos parámetos')

    mcm=lcm(c,K)
    a=mcm/K
    rho=mcm/c

    if a!=alfa:
        print("Number of stripes does not match the required number")
        return -1;

    alpha=C.evaluation_points()


    mult=C.column_multipliers()
    D=codes.GeneralizedReedSolomonCode(alpha,b,mult)

    queries=[]
    for i in range(N):
        queries.append(Matrix(F,rho,alfa*M))
```

Comenzamos iterando sobre el número necesario de sub-solicitudes. Para cada una de ellas empezamos generando una matriz de vectores aleatorios de D, de la misma forma que hemos hecho en protocolos anteriores.

```
        G=Matrix(F,0,N)
    v=vector(F,N)

    for r in range(rho):
        U=Matrix(F,N,0)
        for i in range(M):
            for j in range(alfa):
                U=U.augment(D.random_element().column())
```

El siguiente paso es crear el código E_i y sumar a U cada uno de los generadores. En primer lugar calculamos la dimensión de este subespacio vectorial. E_i tendrá tantos generados como dimensión tenga, así que iteramos para cada uno de ellos. Calculamos la potencia de *alpha* correspondiente al generador y a partir de ella obtenemos dicho vector. Si se trata de el generador j-ésimo lo sumamos a la fila j-ésima de U.

```
        dim=ceil((r+1)*c/K)

        for i in range(dim):
            pot=(r+1)*c-(i+1)*K+K+b-1
            for j in range(N):
                 v[j]=alpha[j]^pot*mult[j]
            U[:,(f-1)*alfa+i]+=v.column()
```

Por último solo nos falta introducir cada una de las filas de U como una sub-solicitud en cada una de las matrices de *queries*. Tras realizar el proceso con todas las sub-solicitudes devolvemos la lista de solicitudes *queries*.

```
        for i in range(N):
            queries[i][r,:]=U[i,:]
    return queries
```

Código completo del programa:

```
def PIRBetterBizantineQueryMatrices(C,b,z,r,alfa,M,f):


    N=C.length()
    K=C.dimension()
    c=N-K-b-2*z-r+1
    if c<1:
        raise ValueError('No se puede realizar PIR para esos parámetos')

    mcm=lcm(c,K)
    a=mcm/K
    rho=mcm/c

    if a!=alfa:
        print("Number of stripes does not match the required number")
        return -1;

    alpha=C.evaluation_points()


    mult=C.column_multipliers()
    D=codes.GeneralizedReedSolomonCode(alpha,b,mult)

    queries=[]
    for i in range(N):                          #Initialize queries
       queries.append(Matrix(F,rho,alfa*M))

    G=Matrix(F,0,N)
    v=vector(F,N)
```

```
    for r in range(rho):
        U=Matrix(F,N,0)
       for i in range(M):
            for j in range(alfa):
                U=U.augment(D.random_element().column())


        dim=ceil((r+1)*c/K)

        for i in range(dim):
            pot=(r+1)*c-(i+1)*K+K+b-1
            for j in range(N):
                v[j]=alpha[j]^pot*mult[j]
            U[:,(f-1)*alfa+i]+=v.column()


        for i in range(N):
            queries[i][r,:]=U[i,:]
    return queries
```

6.3.4. Implementación del algoritmo de recuperación del archivo

En el proceso de recuperación el output será el propio archivo en su forma original. Como input usaremos los siguientes parámetros:

- Código con el que se ha codificado la base de datos, debe ser un código GRS: C
- Número de servidores cooperantes contra los que se protege: b
- Cantidad de errores que se desea corregir: z
- Cantidad de borronoes que se desea corregir: r
- Número de bandas en las que se divide el archivo: $alfa$
- Lista de vectores respuestas recibidas de los servidores: R
- Matriz de unos y ceros representando los borrones en las respuestas : $Erasures$. Cada columna se corresponde con uno de los vectores de respuesta. Si en la fila i columna j hay un uno, se interpretará la posición i de la repusta del servidor j como un borrón.

En primer lugar realizamos las mismas comprobaciones que en el caso anterior e inicializamos las variables necesarias, de forma similar a otros protocolos. También crearemos los códigos D y $Star$, ambos códigos GRS con los parámetros explicados anteriormente.

```
def PIRBetterBizantineDecodeFromResponses(C,b,z,r,alfa,R,Erasures):
```

```
N=C.length()
K=C.dimension()
c=N-K-b-2*z-r+1
if c<1:
    raise ValueError('No se puede realizar PIR para esos parámetos')
```

```
mcm=lcm(c,K)
a=mcm/K
rho=mcm/c

if a!=alfa:
    print("Number of stripes does not match the required number")
    return -1;

F=C.base_field()
FF=R[0][0,0].parent()

alpha=vector(FF,C.evaluation_points())
mult=C.column_multipliers()
prod=mult.pairwise_product(mult)

dim=c+K+b-1
Star=codes.GeneralizedReedSolomonCode(alpha,dim,prod)
decoder=codes.decoders.GRSErrorErasureDecoder(Star)

v=Matrix(FF,1,N)
File=Matrix(FF,alfa,K)

Response=Matrix(FF,rho,0)
for i in range(N):
    Response=Response.augment(R[i])
```

Ahora iteraremos cada una de las sub-respuestas recibidas para extraer c fragmentos de ellas. Lo primero que debemos hacer es restar los fragmentos que ya conocemos multiplicados por el generador de $C^{\star D}_{\star E_i}$ correspondiente. De forma similar a antes, calculamos la potencia del generador correspondiente, y sustraemos de las respuestas el producto del generador y el fragmento. Se realizará este proceso para cada uno de los $c * r$ fragmentos que ya hemos obtenido y que están almacenados en $File$.

```
for r in range(rho):
    pot=(r+1)*c+K+b-2

    #Delete known values
    for i in range(c*r):
```

```
            banda=i//K
            column=i%K
            for j in range(N):
                v[0,j]=alpha[j]^(pot-i)*prod[j]
            Response[r,:]-=v*File[banda,K-1-column]
```

El resultado es una palabra de $C^{\star D}_{\star E_i}$. Usamos los métodos de generación de errores para obtener la palabra fuente que genera dicha palabra código. Las últimas c coordenadas de esta palabra fuente son los c fragmentos siguientes del archivo que deseamos. Simplemente necesitamos añadirlos a File. Una vez obtenidos todos los fragmentos usamos el método $DecodeToFile(File, F)$ para recuperar el archivo en su forma original. Finalmente lo devolvemos.

```
        #Decode the rest
        word=decoder.decode_to_message([Response.row(r), Erasures.row(r)])
        if len(word)!=dim:
             zero = zero_vector(FF, dim-len(word))
             word = vector(list(word)+list(zero))

        pos=c*r
        for i in range(c):

            banda=(pos+i)//K
            column=(pos+i)%K

            File[banda,K-1-column]=word[c+K+b-2-i]


    #Decode file
    file=DecodeToFile(File,F)
    return file
```

Código completo del programa

```
def PIRBetterBizantineDecodeFromResponses(C,b,z,r,alfa,R,Erasures):


    N=C.length()
    K=C.dimension()
    c=N-K-b-2*z-r+1
    if c<1:
        raise ValueError('No se puede realizar PIR para esos parámetos')

    mcm=lcm(c,K)
    a=mcm/K
    rho=mcm/c
```

```
if a!=alfa:
    print("Number of stripes does not match the required number")
    return -1;

F=C.base_field()
FF=R[0][0,0].parent()

alpha=vector(FF,C.evaluation_points())
mult=C.column_multipliers()
prod=mult.pairwise_product(mult)

dim=c+K+b-1
Star=codes.GeneralizedReedSolomonCode(alpha,dim,prod)
decoder=codes.decoders.GRSErrorErasureDecoder(Star)

v=Matrix(FF,1,N)
File=Matrix(FF,alfa,K)

Response=Matrix(FF,rho,0)
for i in range(N):
    Response=Response.augment(R[i])

for r in range(rho):
    pot=(r+1)*c+K+b-2

    #Delete known values
    for i in range(c*r):
        banda=i//K
        column=i%K
        for j in range(N):
            v[0,j]=alpha[j]^(pot-i)*prod[j]
        Response[r,:]-=v*File[banda,K-1-column]



    #Decode the rest
    word=decoder.decode_to_message([Response.row(r), Erasures.row(r)])
    if len(word)!=dim:
         zero = zero_vector(FF, dim-len(word))
         word = vector(list(word)+list(zero))

    pos=c*r
    for i in range(c):

        banda=(pos+i)//K
        column=(pos+i)%K

        File[banda,K-1-column]=word[c+K+b-2-i]
```

```
#Decode file
file=DecodeToFile(File,F)
return file
```

Capítulo 7

Pruebas y escalabilidad

En este capítulo incluimos parte de las pruebas que se han realizado para verificar el correcto funcionamiento de los programas, y además medir los tiempos de ejecución para comprobar la eficiencia y la posible escalabilidad.

Para ello supondremos una base de datos formada por 9 archivos de texto. El tamaño aproximado de estos archivos será de 500 Bytes, con el mayor archivo teniendo un tamaño exacto de 519 Bytes.

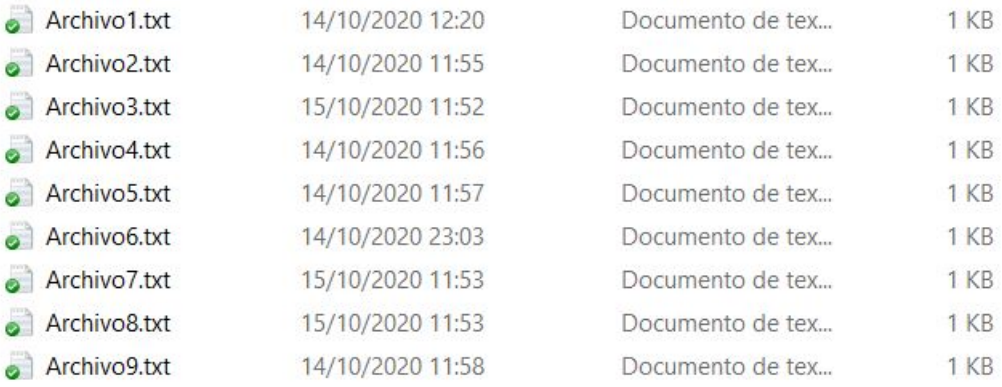

Figura 7.1: Archivos de texto

Recuperaremos uno de los archivos de forma privada usando el protocolo numero 4.

En primer lugar debemos cargar las funciones

```
In [1]: load("../PIR.sage")
```

Importamos time para medir los tiempos de ejecucion

```
In [2]: import time
```

Leemos los archivos de texto y los almacenamos en una matriz

```
In [3]: F=GF(2^8)
        Files = Matrix(F,519,9)

        elementos =F.list()


        for files in range(1,10):
            filename = "./Ejemplo/Archivo"+str(files)+".txt"
            f = open(filename, "r")
            a = bytearray(f.read())
            f.close()
            for pos in range(len(a)):
                Files[pos,files-1]=elementos[a[pos]]
```

Creamos el código con el que codificaremos y realizamos la codificación

```
In [4]: C=codes.GeneralizedReedSolomonCode(elementos[26:46],
             9,elementos[1:21])
```

```
In [5]: start=time.time()
        CodedDB = PIREncodeToServers(Files,10,C)
        end=time.time()
        end - start
```

```
Out[5]: 1.405876874923706
```

Cada elemento de la base de datos es un elemento de este cuerpo

```
In [6]: CodedDB[0,0].parent()
```

```
Out[6]: Finite Field in z48 of size 2^48
```

Obtenemos el código D necesario para proteger contra 2 servidores cooperantes

```
In [7]: D=codes.GeneralizedReedSolomonCode(elementos[26:46],
             2,elementos[76:96])
```

Empezamos con el protocolo calculando las matrices solicitud. Como solo se usa el cuerpo F_8 el tiempo de ejecución es muy rápido

```
In [8]: start=time.time()
        Queries = PIRStarQueryMatrices(C,D,10,9,1,11)
        end=time.time()
        end - start
```

```
Out[8]: 0.4572570323944092
```

Calculamos las respuestas. El tiempo resultante es muy superior a el tiempo que se tardaría realmente. En este caso nuestra computadora está calculando todas las solicitudes. Sin embargo, en una situación real cada servidor calcularía la suya, dividiendo la carga entre cada uno de los 20 servidores

```
In [9]: start=time.time()
        Responses = PIRQueryToServer(CodedDB,Queries)
        end=time.time()
        end - start
```

```
Out[9]: 1.583104133605957
```

Calculamos el archivo que deseamos. Como en este caso tenemos que usar la extensión $F_{2^{48}}$ el tiempo es algo superior, aunque en estas situaciones aún muy bajo.

```
In [10]: prod = vector(elementos[76:96]).
             pairwise_product(vector(elementos[1:21]))
         Star = codes.GeneralizedReedSolomonCode(elementos[26:46],10,prod)
```

```
In [11]: start=time.time()
         File = PIRStarDecodeFromResponses(C,Star,10,Responses,11)
         end=time.time()
         end - start
```

```
Out[11]: 0.532181978225708
```

Transformando los bytes a texto y eliminando los ceros del final recuperamos el texto

```
In [12]: for i in range(len(File)):
             File[i] = elementos.index(File[i])
         File=bytearray(File)

         l = len(File)-1
         while File[l]==0:
             l=l-1
         File=File[0:l+1]

         filename = "./Ejemplo/textorecuperado.txt"
         f = open(filename, "w")
         f.write(File)
         f.close()

         File.decode("utf-8")
```

```
Out[12]: u'Este archivo contiene cosas relacionadas con 1\r\nLorem ipsum
         dolor sit amet, consectetur adipiscing elit. Vivamus quis
         scelerisque nisl. Donec vitae lectus non orci gravida consequat.
         Sed fringilla velit lectus, vel ornare sapien scelerisque ut.
         Nullam ultricies faucibus tellus, sed tempus felis ornare nec.
         Donec et felis odio. Vivamus posuere leo eu suscipit bibendum.
         Nullam lorem dolor, venenatis sit amet turpis a, imperdiet
         sagittis est. Donec hendrerit, ante a sodales fermentum, ligula
         sapien blandit dolor.\r\n\r\n'
```

De hecho podemos comprobar que se ha creado un nuevo archivo *textorecuperado.txt* con el mismo contenido que el original.

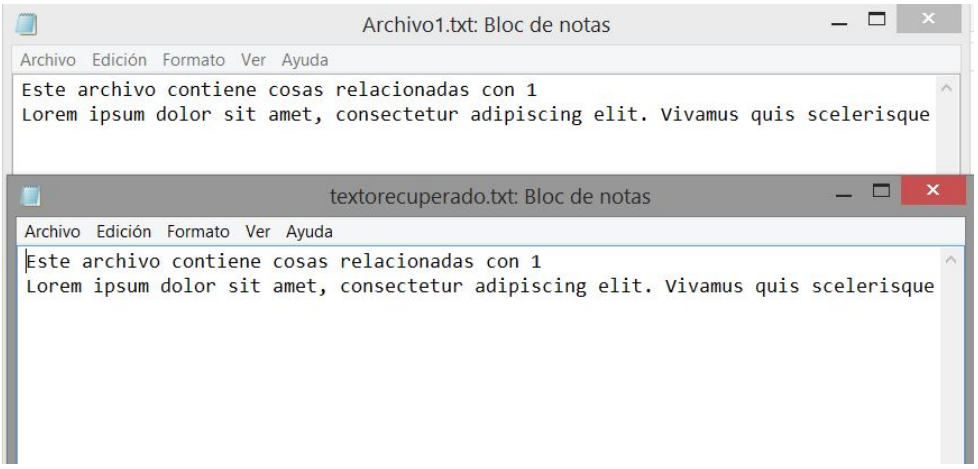

Figura 7.2: Archivos "*texto (1).txt*" y "*textorecuperado.txt*
"

Analizando los tiempo de ejecución que todos son realmente pequeños , siempre en torno a un único segundo. Podemos concluir entonces que la efectividad de estos protocolos para bases de datos formadas por pequeños archivos como pequeños textos es muy buena, ya que todas las operaciones se han realizado en un corto periodo de tiempo.

Queremos también probar el funcionamiento con archivos de tamaño más grande. El principal problema de esta prueba es que la implementación en *Sage* no es muy buena para estos casos, ya que al tratar con cuerpos muy grandes y archivos muy grandes nos encontramos con el problema de falta de memoria para realizar el cálculo de la base de datos. Sin embargo el ejemplo que aparece a continuación es el mayor que he conseguido realizar sin obtener un fallo de falta de memoria.

Para este ejemplo usaremos fotos, de un tamaño aproximado de 10-20 KB cada una de ellas. El tamaño máximo de una foto es 23476 Bytes.

```
In [1]: load("../PIR.sage")
```

```
In [2]: import time
```

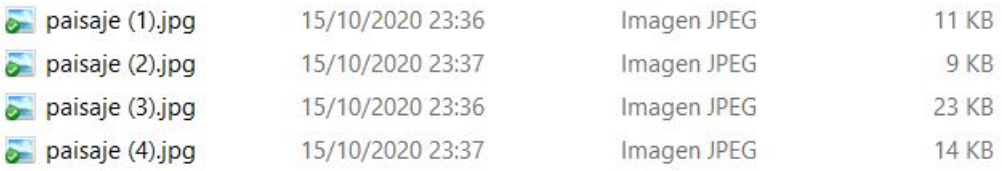

Figura 7.3: Archivos de las fotos de paisajes
”

```
In [3]: F=GF(2^8)
        Files = Matrix(F,23476,4)

        elementos =F.list()


        for files in range(1,5):
            filename = "./Ejemplo/paisaje ("+str(files)+").jpg"
            f = open(filename, "r")
            a = bytearray(f.read())
            f.close()
            for pos in range(len(a)):
                Files[pos,files-1]=elementos[a[pos]]
```

En este caso usaremos un código GRS de dimensión 11 y dividiremos el archivo en 28 bandas, con el objetivo de dividir el archivo en la mayor cantidad de fragmentos posibles

```
In [4]: C=codes.GeneralizedReedSolomonCode(elementos[26:51],11,elementos[1:26])
```

```
In [5]: start=time.time()
        CodedDB = PIREncodeToServers(Files,28,C)
        end=time.time()
        end - start
```

```
Out[5]: 46.520009994506836
```

En este caso el tamaño del cuerpo finito es realmente grande y por ello los cálculos serán más lentos

```
In [6]: CodedDB[0,0].parent()
```

```
Out[6]: Finite Field in z616 of size 2^616
```

Comenzamo con el proceso de recuperar el archivo

```
In [7]: start=time.time()
        Queries = PIRBizantineQueryMatrices(C,2,28,4,1)
        end=time.time()
        end - start

Out[7]: 1.4635679721832275

In [8]: start=time.time()
        [Responses,Erasures] = PIRQueryToServerNoise(CodedDB,Queries,1,0)
        end=time.time()
        end - start

Out[8]: 8.738785028457642

In [9]: start=time.time()
        File = PIRBizantineDecodeFromResponses(C,2,28,Responses,Erasures)
        end=time.time()
        end - start

Out[9]: 37.26402401924133

In [10]: for i in range(len(File)):
             File[i] = elementos.index(File[i])
         File=bytearray(File)

         print(File[23576:23660])

         l = len(File)-1
         while File[l]==0:
             l=l-1
         File=File[0:l+1]

         filename = "./Ejemplo/fotonueva.jpg"
         f = open(filename, "w")
         f.write(File)
         f.close()
```

En primer lugar vemos que el cuerpo que hemos usado para cada uno de los fragmentos es de 2^{616} elementos. El tiempo de ejecución dependerá de como de grande sea este cuerpo, además que la memoria también se acabará antes cuanto mayor sea. Para cualquier valor superior a 2^{650} la memoria se acaba y la ejecución no se puede realizar. Sin embargo realizando un implementación en otro lenguaje, con este problema en mente se podría conseguir realizar con cuerpos mayores.

Analizando los tiempos de ejecución vemos en primer lugar que el tiempo necesario para codificar la base de datos a aumentado en gran medida, hasta un total de 46 segundos. Dado que este proceso se realizará una única vez, antes de "publicar"la base de datos,

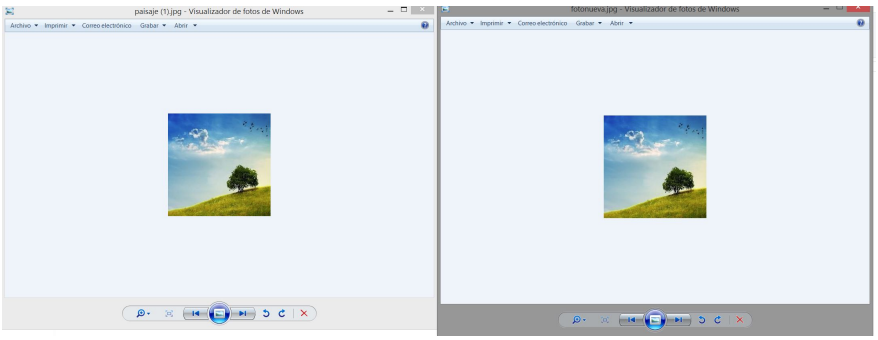

Figura 7.4: Archivos "*paisaje (1).jpg*" y "*fotonueva.jpg*
"

no supone un gran problema que el tiempo sea muy alto, y por tanto para un caso con archivos mucho mayores se seguirían consiguiendo tiempos razonables.

En cambió, si que estamos interesados que el método de generación de las solicitudes sea lo más rápido posible, ya que es un proceso que se realizará un gran número de veces. Como podemos observar, este método sigue siendo realmente rápido, con no más de dos segundos de tiempo de ejecución. Esto se debe a que en este método únicamente tratamos con el cuerpo base original, que en este caso estamos usando el código de 2^8 elementos (bytes).

El cálculo de las solicitudes también se realiza de forma muy rápida, ya que los casi 9 segundos que tarda en ejecutarse, en la realidad estarían divididos entre los 25 servidores, haciendo un tiempo de $9/25 = 0{,}36$ segundos por servidor. Es un tiempo realmente pequeño pero hay que tener en cuenta que si la cantidad de peticiones que reciben los servidores es muy alta, es posible que para casos con archivos de mayor tamaño no sea suficiente.

Por último, el cálculo del archivo a partir de la respuesta, si que es en mi opinión demasiado lento, con un tiempo casi del orden del cálculo de la base de datos. Al igual que en dicho caso, estamos tratando con elementos del cuerpo grande, por lo que el tiempo de computación es muy alto.

En conclusión, la rapidez de estos protocolos los hace aceptables para aplicarlos a una base de datos real en el caso de archivos de un tamaño no demasiado grande. En el caso de un tamaño mayor, tendríamos la opción de dividirlos en la mayor cantidad de fragmentos posibles para tratar con cuerpos más pequeños. En caso de no poder ser así, el tiempo de recuperación del archivo podría suponer un problema.

Capítulo 8

Conclusión

En este proyecto hemos realizado un estudio teórico, además de realizar una implementación, de los protocolos PIR más importantes. Para todos ellos hemos visto en qué consiste su funcionamiento, además de ver ejemplos prácticos de su uso. Debido a que este tema es bastante novedoso, no existen ahora muchas fuentes que traten el tema y además aún quedan muchas otras líneas de investigación.

Durante el desarrollo de este proyecto he podido poner en práctica muchos de los conceptos aprendidos durante la carrera como la teoría de códigos y la elaboración de diversos programas. Gracias a este proyecto además he aprendido el lenguaje de programación *Python* y *Sage*, además de perfeccionar mis conocimientos en la elaboración de un documento usando LaTeX. Además he aprendido mucho en relación a la importancia de mantener la privacidad en las transacciones que realizamos en internet.

Fijándonos en los objetivos que fijamos en la introducción, podemos ver que los hemos alcanzado en su totalidad.

Inicialmente estudiamos la teoría de códigos lineales más básica, como la definición de código y su matriz generatriz y de control. Además también vimos los métodos y estructuras correspondientes ya implementados en *Sage*. También realizamos el mismo procedimiento con los códigos de Reed-Solomon y Reed-Solomon Generalizados, centrándonos en su implentación en *Sage*.

En relación a los protocolos PIR, realizamos la lectura de los cuatro artículos de la bibliografía [5], [6], [7] y [8]. En cada uno de ellos realizamos una descripción de su funcionamiento. También hicimos un análisis de la eficiencia y la cantidad de servidores cooperantes que conseguía cada uno de ellos.

Por último, también realizamos la implementación en *Sage* de los métodos de codificación de bases de datos en los servidores, y los métodos tanto generales como específicos a cada uno de los protocolos PIR, logrando simular el proceso de solicitudes y respuestas a los distintos servidores.

8.1. Trabajo futuro

Como se ha dicho antes, este tema actualmente es muy novedoso y popular. Por ello aún existen numerosas líneas de investigación que se pueden seguir en relación a este tema. En esta sección daremos un conjunto de posibles ideas con las que completar este proyecto:

- Estudiar los protocolos posibles que podrían surgir cuando se usan otro tipo de códigos para codificar la base de datos. En este proyecto únicamente nos hemos fijado en el caso en el que el código empleado es un código GRS. Esto se debe a que este tipo de códigos tienen propiedades muy interesantes como su distancia mínima o su buen comportamiento frente al producto estrella, que los hace muy buenos para este tema. Sin embargo tienen un gran inconveniente. La longitud de un código GRS nunca puede ser mayor a la cantidad de elementos en su cuerpo base. Es por ello, que utilizar otro tipo de códigos con propiedades similares pero sin ese inconveniente puede resultar interesante. Un ejemplo de estos códigos son los Matrix Product Codes.

- Investigar la forma de implementar estos protocolos en una base de datos real. Todo el trabajo realizado en este proyecto no ha sido en una base de datos real. Siempre hemos utilizado un modelo de base de datos en forma de matriz. En cada uno de estos servidores veíamos los datos como un vector de elementos de un cuerpo, pese que un servidor SQL ve los datos como una tabla, al final, estos datos están como un archivo secuencial de bits. Sin embargo, SQL no soporta el tipo de solicitudes que enviamos, por lo que se debería añadir una capa para interpretar estas solicitudes.

- Realizar un análisis de la seguridad de este tipo de base de datos. Dado que al realizar solicitudes pedimos una combinación lineal de todas las entradas de la base de datos, deberíamos ser capaces de recuperar cada una de ellas. Sin embargo, por razones de seguridad es posible que parte de esta información esté restringida a ciertos usuarios, o cierto rol. Se debería estudiar que ocurriría en este supuesto y en caso de suponer un problema que posibles soluciones pueden implementarse.

- Debido a que todos los archivos deben tener la misma longitud, se debería estudiar una forma de implementar un padding para añadir al archivo antes de la codificación para que coincida con el tamaño del mayor archivo. De esta forma se conseguiría que la longitud de todos sea la misma. Tras recuperar el archivo original este padding debe ser fácil de identificar para eliminarlo.

- Implementación de los métodos en un lenguaje de más bajo nivel, teniendo en cuenta la administración de la memoria de la forma más eficaz, para poder aplicar estos protocolos en bases de datos formadas por archivos de un tamaño mayor.

Apéndice A

Manual de Usuario

En este capítulo veremos cuáles son las funciones implementadas y cómo usar cada una de ellas. Además también incluiremos un ejemplo de su uso. Sin embargo, antes de empezar con las funciones, veremos cómo instalar y usar *Sage*.

Para comenzar con la descarga del programa accedemos a la página web `https://www.sagemath.org/download.html`. Allí tenemos que elegir la opción del servidor más cercano, en nuestro caso, el servidor español *RedIRIS Reserarch Network*. Tras seleccionar esta opción seremos dirigidos a una página web donde nos pedirán seleccionar el sistema operativo donde queremos instalarlo. Tras seleccionar la opción adecuada elegimos la última versión disponible y deberíamos descargar un fichero ejecutable. La versión usada en el desarrollo de estos programas es la 8.9, pero versiones posteriores deberían ser también compatibles. Tras abrir este fichero debería empezar el proceso de instalación.

Tras finalizar la instalación, habremos obtenido tres programas distintos. En primer lugar, el programa *Sage* nos abrirá una ventana donde podemos escribir y ejecutar comandos *Sage* uno a uno. En segundo lugar, *SageShell* nos abre una ventana de comandos donde podemos crear variables para abrir posteriormente *Sage* con una configuración determinada. La que usaremos es la tercera de estas opciones, *SageNotebook*. Tras abrir este programa, se abrirá una consola que actúa como servidor y también se abrirá el entorno de trabajo *Jupyter* en el navegador web que usemos por defecto. Desde este entorno podemos navegar por las carpetas de nuestro equipo para abrir un archivo con la extensión *.ipynb* en el editor, o podemos también crear uno nuevo.

Una vez creado un nuevo archivo, para poder usar la funcionalidad implementada en este documento, primero debemos cargar todas estas funciones. Para ello tenemos que cargar el archivo $PIR.sage$, donde está toda esta funcionalidad almacenada. Para ello debemos introducir el comando $load(PIR.sage)$. Este comando ejecuta todas las líneas de código en el archivo introducido, lo que da como resultado la carga de todas las funciones almacenadas.

En cada una de las funciones implementadas se ha incluido un pequeño resumen de la utilidad de la función, además de los inputs y outputs que acepta. Para acceder a esta información tenemos que usar el comando *help* de python. Tras ejecutar la sentencia $help(PIREncodeToServers)$ obtendremos la información correspondiente a la función $PIREncodeToServers$. Además de está ayuda, incluiremos a continuación un pequeño manual donde damos una breve descripción de cada una de las funciones, además de

incluir ejemplos de su funcionalidad.

A.1. Manual de usuario

PIREncodeToServer

- **Descripción:** Codifica un conjunto de archivos en forma de vector a su forma matricial usando un código. La cantidad de servidores en los que se codifican corresponde con la longitud del código elegido. Los archivos se dividirán en el número indicado de filas (bandas).
- **Inputs:**
  - f: Matriz de archivos donde cada columna es uno de ellos.
  - *alpha*: Numero de bandas en las que estará dividido cada archivo.
  - C: Código lineal que se usará para codificar la base de datos.
  - $systematic = false$: Booleano que indica si queremos usar o no la matriz sistemática de C en la codificación. En caso de no incluirse se asumirá que no deseamos usar la forma sistemática.
- **Output:** Matriz conteniendo la forma matricial codificada de la base de datos. Cada una de las columnas representa los datos almacenados por uno de los servidores.
- **Ejemplos:**

  Creamos un cuerpo y una matriz formada por dos archivos de 8 elementos de F.

```
F=GF(11)
f=Matrix(F,[[2,4,6,7,8,1,9,4],[5,4,2,1,2,10,9,2]])
f=f.transpose()
C=codes.random_linear_code(F,6,4)
PIREncodeToServers(f,2,C)
```

```
[ 6  0  7  5  4  0 ]
[ 1  0  1  9 10  2 ]
[10  7  7  5  7  7 ]
[10  4  9  1  9  7 ]
```

```
PIREncodeToServers(f,2,C,true)
```

```
[ 2  4  6  7  0  0 ]
[ 8  1  9  4  4  8 ]
[ 5  4  2  1 10  6 ]
[ 2 10  9  2  0  4 ]
```

  Como vemos cuando usamos la codificación sistemática los fragmentos en los cuatro primeros servidores son los propios fragmentos de los archivos

```
PIREncodeToServers(f,1,C,true)

        [ 4*z2 + 2  7*z2 + 6    z2 + 8  4*z2 + 9  8*z2 + 6 10*z2 + 5]
        [ 4*z2 + 5    z2 + 2 10*z2 + 2  2*z2 + 9  2*z2 + 8  9*z2 + 3]
```

En este casose han combinado fragmentos del archivo en elementos de una extensión de F, de forma que cada archivo ocupe una única banda.

DecodeToFile

- **Descripción:** Devuelve a un archivo en su forma matricial sin codificar a su forma original como vector. Esta función generalmente no pretende ser usada por el usuario, sino en el desarrollo del resto de funciones.
- **Inputs:**
  - $File$: Archivo en forma matricial a decodificar.
  - F: Cuerpo al que pertenecían los diferentes fragmentos del archivo original.
- **Output:** Vector sobre F conteniendo la forma original del archivo $File$.
- **Ejemplos:**

  La forma matricial del primer archivo del ejemplo anterior son las cuatro primeras columnas de la primera fila del resultado de PIREncodeToServers():

```
F=GF(11)
FF=GF(11^2)
z2=FF.gen()
File=Matrix(FF,[ 4*z2 + 2,  7*z2 + 6,    z2 + 8,  4*z2 + 9 ])
DecodeToFile(File,F)
        [2, 4, 6, 7, 8, 1, 9, 4]
```

  Que coincide con el archivo original que definimos en el ejemplo anterior.

PIRQueryToServer

- **Descripción:** Simula el proceso de envío de solicitudes a los servidores de una base de datos, calcula las respuestas correspondientes y las envía de vuelta.
- **Inputs:**
  - $CodedDB$: Base de datos codificada en forma matricial, obtenida normalmente mediante el método $PIREncodeToServers$.
  - $query$: Lista de matrices. Cada una de las matrices de la lista es una de las matrices solicitud dirigida a uno de los servidores. Estas matrices deben tener tantas columnas como filas tiene $CodedDB$.
  - $servers = -1$: Lista de indices de la misma longitud que $query$. Indican los índices de los servidores a los que van dirigidos cada una de las solicitudes. El valor por defecto es -1 que posteriormente pasa a ser los servidores iniciales.

- **Output:** Lista de vectores. Cada uno de los vectores es la respuesta de una de las matrices solicitud al servidor correspondiente.
- **Ejemplos:**

  Supongamos la primera de las bases de datos creadas a partir del ejemplo de *PIREncodeToServers*, guardada en la variable $CodedDB$.

```
F=GF(11)
Query1=Matrix(F,[[2,3,1,2],[4,4,2,6]])
Query2=Matrix(F,[[1,0,0,0],[0,1,0,0]])
Queries=[Query1,Query2]

PIRQueryToServer(CodedDB,[Query1,Query2])

        [
        [4]  [6]
        [2], [5]
        ]


PIRQueryToServer(CodedDB,Queries,[0,5])

        [
        [4]  [6]
        [2], [6]
        ]
```

  En el primero de los casos las solicitudes se han enviado al primero y segundo servidor. Como cada matriz tenía dos filas recibimos dos respuestas a cada una de ellas. En el segundo de los casos hemos elegido los servidores a los que enviamos las solicitudes. Son el 1^{o} y el 6^{o}.

PIRQueryToServerNoise

- **Descripción:** Simula el proceso de envío de solicitudes a los servidores de una base de datos en un canal con ruido, o con servidores maliciosos. Calcula las respuestas correspondientes y las envía de vuelta tras añadir la cantidad indicada de errores y borrones a cada sub-solicitud.
- **Inputs:**
  - $CodedDB$: Base de datos codificada en forma matricial, obtenida normalmente mediante el método $PIREncodeToServers$.
  - *query*: Lista de matrices. Cada una de las matrices de la lista es una de las matrices solicitud dirigida a uno de los servidores. Estas matrices deben tener tantas columnas como filas tiene $CodedDB$.
  - z: Número de errores que crear por sub-solicitud
  - r: Número de borrones que añadir por sub-solicitud.

- *servers* $= -1$: Lista de indices de la misma longitud que *query*. Indican los índices de los servidores a los que van dirigidos cada una de las solicitudes. El valor por defecto es -1 que posteriormente pasa a ser los servidores iniciales.

- **Output:** El programa devuelve una lista con una lista de vectores y una matriz. Cada uno de los vectores de la primera lista es la respuesta de una de las matrices solicitud al servidor correspondiente. Cada una de las columnas de la matriz devuelta es un vector de unos y ceros que indica con unos las posiciones de las respuestas que deben interpretarse como borrones.

- **Ejemplos:**

  En el ejemplo realizaremos las mismas solicitudes que en la función anterior pero añadiremos errores y borrones.

```
F=GF(11)
Query1=Matrix(F,[[2,3,1,2],[4,4,2,6]])
Query2=Matrix(F,[[1,0,0,0],[0,1,0,0]])
Queries=[Query1,Query2]

PIRQueryToServerNoise(CodedDB,[Query1,Query2],1,0)

        [
        [ [3], [6]      [0 0]
          [9], [5] ] ,  [0 0]

        ]


PIRQueryToServerNoise(CodedDB,[Query1,Query2],0,1)

        [
        [ [4], [0]      [0 1]
          [2], [0] ] ,  [0 1]

        ]
```

  En el primero de los casos se han generado un error en cada fila y ningún borrón. En el segundo de ellos se ha generado un borrón en cada fila, las posiciones afectadas aparecen con un cero como respuesta y un uno en la matriz de borrones.

StarProduct

- **Descripción:** Calcula el producto estrella entre dos códigos lineales.

- **Inputs:**

  - C: Primer factor del producto.
  - D: Segundo factor del producto

- **Output:** Código lineal resultado del producto estrella entre C y D.

- **Ejemplos:**

```
F=GF(17)
C=codes.random_linear_code(F,6,2)
D=codes.random_linear_code(F,6,2)

StarProduct(C,D)

        [6, 4] linear code over GF(17)

C=codes.LinearCode(Matrix(F,[[2,2,2,2,2]]))
D=codes.LinearCode(Matrix(F,[[1,2,3,4,5],[2,4,6,8,0]]))

StarProduct(C,D).generator_matrix()

        [ 2  4  6  8 10]
        [ 4  8 12 16  0]
```

PIRQueryMatrices

- **Descripción:** Calcula las matrices de solicitud necesarias para realizar los protocolos número 1,2 y 3.
- **Inputs:**
  - C: Código usado para codificar la base de datos. Es necesario que sea un código MDS, en caso contrario el programa no funcionará.
  - *alpha*: Número de bandas en las que está dividida cada archivo en la base de datos. Debe coincidir con la cantidad necesaria dependiente del protocolo.
  - M: Número de archivos en la base de datos.
  - b: Cantidad de servidores cooperantes contra los que se desea proteger. Si $b = 1$ se usará el protocolo 1, en caso contrario el 2 o 3 dependiendo de otros parámetros.
  - f: Índice del archivo que se desea recuperar, empezando en 1.
  - $eff = false$: Booleano indicando si se desea realizar la versión más eficiente (protocolo 3), cuando $b > 1$. En caso de no introducirlo se supone el valor false.
- **Output:** Lista de matrices solicitudes para dirigir a los servidores. La dimensión de estas matrices es de $\rho \times m\alpha$ dependiendo del protocolo usado.
- **Ejemplos:**

  Mostraremos un ejemplo conjunto en la siguiente sección correspondiente a PIRDecodeFromResponses.

PIRDecodeFromResponses

- **Descripción:** Recupera el archivo deseado a partir de las respuestas de los servidores a las solicitudes creadas por PIRQueryMatrices.

- **Inputs:**
  - C: Código usado para codificar la base de datos. Es necesario que sea un código MDS, en caso contrario el programa no funcionará.
  - *alpha*: Número de bandas en las que está dividida cada archivo en la base de datos. Debe coincidir con la cantidad necesaria dependiente del protocolo.
  - b: Cantidad de servidores cooperantes contra los que se desea proteger. Si $b = 1$ se usará el protocolo 1, en caso contrario el 2 o 3 dependiendo de otros parámetros.
  - R: Lista de respuestas, generalmente obtenidas a partir del método PIRQueryToServers.
  - $eff = false$: Booleano indicando si se desea realizar la versión más eficiente (protocolo 3), cuando $b > 1$. En caso de no introducirlo se supone el valor false.
- **Output:** Vector conteniendo el archivo que se deseaba recuperar
- **Ejemplos:**

  En primer lugar creamos una base de datos formada por dos archivos.

```
F=GF(13)
x=vector(F,[2,3,4,1,5,6])
m=vector(F,[1,1,1,1,1,1])
C=codes.GeneralizedReedSolomonCode(x,4,m)

alpha=6-4

X1=vector(F,[1,2,3,4,5,6,7])
X2=vector(F,[8,9,10,11,12,1,2])
X=X1.column().augment(X2.column())
files=2
deseado=1
```

  La codificamos de acuerdo al primer protocolo y usamos los métodos correspondientes.

```
CodedDB=PIREncodeToServers(X,alpha,C,true)

        [ 1  2  3  4  0  2]
        [ 5  6  7  0 12 12]
        [ 8  9 10 11  7  9]
        [12  1  2  0 11 11]

queries=PIRQueryMatrices(C,alpha,files,1,deseado)

        [
        [ 7  2  2  6]  [ 6  2  2  6]  [ 6  2  2  6]  [ 6  3  2  6]
        [ 4  2 10  8]  [ 5  1 10  8]  [ 4  1 10  8]  [ 4  1 10  8]
        [ 1  4  6  4]  [ 1  5  6  4]  [ 2  4  6  4]  [ 1  4  6  4]
        [ 7  7 10  1], [ 7  7 10  1], [ 7  8 10  1], [ 8  7 10  1],
```

```
        [ 6  2  2  6]  [ 6  2  2  6]
        [ 4  1 10  8]  [ 4  1 10  8]
        [ 1  4  6  4]  [ 1  4  6  4]
        [ 7  7 10  1], [ 7  7 10  1]
        ]

Response = PIRQueryToServer(CodedDB,queries)

        [
        [1]  [ 9]  [12]  [ 7]  [0]  [3]
        [8]  [10]  [ 5]  [ 9]  [1]  [3]
        [0]  [12]  [11]  [ 5]  [4]  [5]
        [4], [ 4], [10], [12], [9], [4]
        ]

PIRDecodeFromResponses(C,alpha,1,Response)

        [1, 2, 3, 4, 5, 6, 7, 0]
```

Codificando la base de datos de acuerdo al segundo protocolo podemos aplicarlo para $b = 2$

```
CodedDB=PIREncodeToServers(X,1,C,true)

        [  2*z2 + 1   4*z2 + 3   6*z2 + 5    7  8*z2 + 12  10*z2 + 3]
        [  9*z2 + 8 11*z2 + 10    z2 + 12    2          5   3*z2 + 6]

queries=PIRQueryMatrices(C,1,files,2,deseado)

        [
        [ 5  6]  [ 8  3]  [ 2  8]  [ 8  9]  [ 7 10]  [10  3]
        [ 9 11]  [ 3  0]  [ 4  4]  [ 4  6]  [ 8  9]  [12  9]
        [ 3  3]  [11  7]  [12  6]  [12  2]  [ 3 10]  [12 11]
        [ 2 11], [ 1  7], [ 7  3], [ 4  4], [12  0], [ 1  1]
        ]

Response = PIRQueryToServer(CodedDB,queries)

[
[ 12*z2 + 1]  [        2]  [7*z2 + 2]  [ 9]  [  4*z2 + 4]  [  5*z2 + 9]
[         6]  [12*z2 + 9]  [2*z2 + 3]  [ 1]  [12*z2 + 11]  [ 4*z2 + 12]
[  7*z2 + 1]  [4*z2 + 12]  [       2]  [10]  [ 11*z2 + 8]  [10*z2 + 11]
[12*z2 + 12], [ 3*z2 + 8], [6*z2 + 6], [10], [  5*z2 + 1], [         9]
]

PIRDecodeFromResponses(C,1,2,Response)
```

```
[1, 2, 3, 4, 5, 6, 7, 0]
```

Por último también podemos usar una base de datos para realizar el protocolo numero 3.

```
C=codes.GeneralizedReedSolomonCode(x,2,m)

CodedDB=PIREncodeToServers(X,alpha,C,true)

        [   4*z4^3 + 3*z4^2 + 2*z4 + 1     7*z4^2 + 6*z4 + 5
           9*z4^3 + 11*z4^2 + 10*z4 + 9     8*z4^3 + 12*z4^2 + 11*z4 + 10,
           5*z4^3 + 2*z4^2 + z4     z4^3 + 6*z4^2 + 5*z4 + 4]
        [ 11*z4^3 + 10*z4^2 + 9*z4 + 8     2*z4^2 + z4 + 12
           2*z4^3 + 7*z4^2 + 6*z4 + 3     9*z4^3 + 5*z4^2 + 4*z4 + 4
           4*z4^3 + 12*z4^2 + 11*z4 + 7     6*z4^3 + 4*z4^2 + 3*z4 + 11]

queries=PIRQueryMatrices(C,1,files,2,deseado,true)

        [[ 1 10], [12  8], [9 0], [4 5], [3 1], [11  7]]

Response = PIRQueryToServer(CodedDB,queries)

        [
        [10*z4^3 + 12*z4^2 + z4 + 3], [9*z4^2 + 2*z4],
        [3*z4^3 + 8*z4^2 + 12*z4 + 3], [12*z4^3 + 8*z4^2 + 12*z4 + 8],
        [6*z4^3 + 5*z4^2 + z4 + 7], [z4^3 + 3*z4^2 + 11*z4 + 4]
        ]

PIRDecodeFromResponses(C,1,2,Response,true)

        [1, 2, 3, 4, 5, 6, 7, 0]
```

PIRStarQueryMatrices

- **Descripción:** Calcula las matrices de solicitud necesarias para realizar el protocolo número 4.
- **Inputs:**
  - C: Código usado para codificar la base de datos.
  - D: Código que se desea usar para generar las sub-solicitudes.
  - *alpha*: Número de bandas en las que está dividida cada archivo en la base de datos. Debe coincidir con el número de bandas requerido.
  - M: Número de archivos en la base de datos.
  - f: Índice del archivo que se desea recuperar, empezando en 1.
  - $d = -1$: Valor de la distancia mínima del código $C \star D$. Si no se indica se calcula aumentando significativamente el coste computacional.

- **Output:** Lista de matrices solicitudes para dirigir a los servidores. La dimensión de estas matrices es de $\rho \times m\alpha$.
- **Ejemplos:**

  Mostraremos un ejemplo conjunto en la siguiente sección correspondiente a PIRStarDecodeFromResponses.

GetGRSQueryCode

- **Descripción:** Calcula la elección que debemos hacer de D para usar el protocolo 4 cuando C es un código GRS
- **Inputs:**
  - C: Código usado para codificar la base de datos.
  - b: Cantidad de servidores cooperantes contra los que se desea proteger.
- **Output:** Código D GRS, óptimo para generar las solicitudes del protocolo 4, y código $Star$, producto estrella entre C y D.
- **Ejemplos:**

```
F=GF(17)
x=vector(F,[2,3,4,1,5,6,14,8,16,15])
m=vector(F,[1,1,1,1,1,1,1,1,1,1])
C=codes.GeneralizedReedSolomonCode(x,7,m)

[10, 7, 4] Reed-Solomon Code over GF(17)

GetGRSQueryCode(C,3)

([10, 3, 8] Reed-Solomon Code over GF(17),
 [10, 9, 2] Reed-Solomon Code over GF(17))

GetGRSQueryCode(C,4)

ValueError: PIR cannot be achieve against this amount of colluding
servers
```

PIRStarDecodeFromResponses

- **Descripción:** Recupera el archivo deseado a partir de las respuestas de los servidores a las solicitudes creadas por PIRStarQueryMatrices.
- **Inputs:**
  - C: Código usado para codificar la base de datos.
  - $Star$: Código producto estrella entre C y el código D usado al generar las solicitudes.

- *alpha*: Número de bandas en las que está dividida cada archivo en la base de datos. Debe coincidir con el número de bandas requerido.
- R: Lista de respuestas, generalmente obtenidas a partir del método PIRStarQueryToServers.
- $d = -1$: Valor de la distancia mínima del código $C \star D$. Si no se indica se calcula aumentando significativamente el coste computacional.

- **Output:** Vector conteniendo el archivo que se deseaba recuperar
- **Ejemplos:**

  Necesitamos una base de datos codificada usando un código C

```
F=GF(17)
x=vector(F,[10,4,6,14,2,11,16,9,1,5])
m=vector(F,[1,1,1,1,1,1,1,1,1,1])
C=codes.GeneralizedReedSolomonCode(x,5,m)

alpha=2

X1=vector(F,[1,2,3,4,5,6,7,8,9])
X2=vector(F,[2,4,6,8,10,12,14,16,1])
X=X1.column().augment(X2.column())
files=2
deseado=2

CodedDB=PIREncodeToServers(X,alpha,C)

        [ 6 12  2 13 10  1  3 11 15  5]
        [16  7  2  1  5  8 15  2 13  6]
        [12  7  4  9  3  2  6  5 13 10]
        [15 14  4  2 10 16 13  4  9 12]

cooperantes=4

[D,Star]=GetStarQueryMatrices(C,cooperantes)

        [[10, 4, 7] Reed-Solomon Code over GF(17),
         [10, 8, 3] Reed-Solomon Code over GF(17)]

queries=PIRStarQueryMatrices(C,D,2,files,deseado,Star.minimum_distance())

        [
        [ 2 16 11  6]  [12  2  3 16]  [ 2 13  2 11]  [11  6 13 12]
        [10 12  1  9]  [ 3  2  3 15]  [ 2  5  1 13]  [ 4  4 15  2]
        [ 1  0 16 13]  [ 3  1 11  3]  [12 15  3 10]  [ 2 11 14 16]
        [ 0 16  0  4]  [11 16 10  5]  [13 10  0  5]  [ 3  3 14  4]
        [ 1  1  2  8], [ 2  4  4  6], [ 1 14  2 13], [11 11  2  8],

        [ 5  6 11  5]  [13 16 15  8]  [ 8  5 10  9]  [16 12 10 10]
```

```
        [ 6  6 15  7]  [ 1  5  0  8]  [11  1  6 11]  [16  0  5 14]
        [15  7  5 10]  [ 2 15  7  1]  [ 9 15  2  8]  [ 5 10  7  9]
        [16  6  9 12]  [ 7  3  4  3]  [ 8  3 15  1]  [11 14  0  4]
        [ 9  7  6  6], [ 7 16 15  9], [14  3  0 11], [ 2 14  6 15],

        [ 1  8  6  0]  [12 13  1  9]
        [ 9  9  9 12]  [16  4  0  9]
        [ 1 12 14 12]  [ 1  1  0  0]
        [14  1  9  1]  [ 8  6  9  8]
        [16  2 16  2], [ 1  0  7  9]
        ]

Response = PIRQueryToServer(CodedDB,queries)

        [
        [14]  [12]  [14]  [1]  [10]  [10]  [ 4]  [ 1]  [10]  [ 1]
        [ 8]  [ 9]  [ 2]  [8]  [ 1]  [16]  [ 6]  [ 2]  [ 1]  [ 8]
        [ 2]  [ 9]  [ 4]  [8]  [11]  [16]  [11]  [10]  [ 2]  [11]
        [10]  [10]  [15]  [6]  [14]  [ 2]  [ 2]  [12]  [ 9]  [ 7]
        [13], [11], [ 5], [1], [16], [ 3], [ 9], [ 4], [16], [13]
        ]

PIRStarDecodeFromResponses(C,Star,3,Response,Star.minimum_distance())

        [2, 4, 6, 8, 10, 12, 14, 16, 1, 0]
```

PIRBizantineQueryMatrices

- **Descripción:** Calcula las matrices de solicitud necesarias para realizar el protocolo número 5.
- **Inputs:**
  - C: Código usado para codificar la base de datos. Debe ser un código GRS.
  - b: Cantidad de servidores cooperantes contra los que queremos proteger.
  - *alpha*: Número de bandas en las que está dividida cada archivo en la base de datos. Debe coincidir con el número de bandas requerido.
  - M: Número de archivos en la base de datos.
  - f: Índice del archivo que se desea recuperar, empezando en 1.
- **Output:** Lista de matrices solicitudes para dirigir a los servidores. La dimensión de estas matrices es de $\alpha \times m\alpha$.
- **Ejemplos:**

  Mostraremos un ejemplo conjunto en la siguiente sección correspondiente a PIRBizantineDecodeFromResponses.

PIRBizantineDecodeFromResponses

- **Descripción:** Recupera el archivo deseado a partir de las respuestas de los servidores a las solicitudes creadas por PIRBizantineQueryMatrices.
- **Inputs:**
  - C: Código usado para codificar la base de datos.
  - b: Cantidad de servidores cooperantes contra los que queremos proteger.
  - *alpha*: Número de bandas en las que está dividida cada archivo en la base de datos. Debe coincidir con el número de bandas requerido.
  - R: Lista de respuestas, generalmente obtenidas a partir del método PIRStarQueryToServers.
  - *Erasures*: Matriz de unos y ceros haciendo referencia a los borrones obtenidos en la transmisión. Se puede obtener como salida del método PIRQueryToServerNoise.
- **Output:** Vector conteniendo el archivo que se deseaba recuperar
- **Ejemplos:**

  Necesitamos una base de datos codificada usando un código C

```
F=GF(17)
x=vector(F,[10,4,6,14,2,11,16,9,1,5])
m=vector(F,[1,1,1,1,1,1,1,1,1,1])
C=codes.GeneralizedReedSolomonCode(x,3,m)

alpha=2

X1=vector(F,[1,2,3,4,5,6])
X2=vector(F,[2,4,6,8,10,12])
X=X1.column().augment(X2.column())
files=2
deseado=2

CodedDB=PIREncodeToServers(X,alpha,C)
```

```
[15  6  2  5  0 12  2  7  6  1  9  7]
[ 8  1 12  9  4  3  5  8 15  9 10 12]
[13 12  4 10  0  7  4 14 12  2  1 14]
[16  2  7  1  8  6 10 16 13  1  3  7]
```

```
cooperantes = 2
queries=PIRStarQueryMatrices(C,cooperantes,alpha,files,deseado)
```

```
[
[ 1 14 13  4]  [ 3  7  2 16]  [ 8 15  2 12]  [11 13 16 13]
[ 0  1  0 11], [16  3 14  4], [ 5  8 15 14], [12 11  2  0],
```

```
[15 16  3  3]  [12  1  3  2]  [16  4  1  9]  [ 7 10  1  6]
[10 15 13 12], [ 3 12  9  6], [ 1 16  3 12], [14  7  8 11],

[ 4 12 15  5]  [14 11  4 14]  [ 2  2  9 10]  [ 0  9 14 15]
[ 7  4  4  2], [ 2 14  6 11], [ 8  2  7  9], [ 9  0 10 10]
```

Seremos capaces de corregir 1 error y dos borrones. Crearemos las respuestas con esa cantidad.

```
[Response,Erasures] = PIRQueryToServerNoise(CodedDB,queries)

 [
 [
 [ 3]  [13]  [0]  [ 5]  [3]  [10]  [10]  [1]  [7]  [ 0]  [11]   [1]
 [14], [ 3], [0], [13], [3], [ 1], [10], [0], [6], [15], [ 0],  [1]
 ],
 [0 0 1 0 0 0 0 0 0 1 0 0]
 [0 0 1 0 0 0 0 0 0 0 1 0]
 ]

PIRBizantineDecodeFromResponses(C,cooperantes,alpha,Response,Erasures)

        [2, 4, 6, 8, 10, 12]
```

PIRBetterBizantineQueryMatrices

- **Descripción:** Calcula las matrices de solicitud necesarias para realizar el protocolo número 6.
- **Inputs:**
  - C: Código usado para codificar la base de datos. Debe ser un código GRS.
  - b: Cantidad de servidores cooperantes contra los que queremos proteger.
  - z: Cantidad de errores que deseamos corregir.
  - r: Cantidad de borrones que deseamos corregir.
  - *alpha*: Número de bandas en las que está dividida cada archivo en la base de datos. Debe coincidir con el número de bandas requerido.
  - M: Número de archivos en la base de datos.
  - f: Índice del archivo que se desea recuperar, empezando en 1.
- **Output:** Lista de matrices solicitudes para dirigir a los servidores. La dimensión de estas matrices es de $\rho \times m\alpha$.
- **Ejemplos:**

  Mostraremos un ejemplo conjunto en la siguiente sección correspondiente a PIRBetterBizantineDecodeFromResponses.

PIRBetterBizantineDecodeFromResponses

- **Descripción:** Recupera el archivo deseado a partir de las respuestas de los servidores a las solicitudes creadas por PIRBizantineQueryMatrices.
- **Inputs:**
  - C: Código usado para codificar la base de datos.
  - b: Cantidad de servidores cooperantes contra los que queremos proteger.
  - z: Cantidad de errores que deseamos corregir.
  - r: Cantidad de borrones que deseamos corregir.
  - *alpha*: Número de bandas en las que está dividida cada archivo en la base de datos. Debe coincidir con el número de bandas requerido.
  - R: Lista de respuestas, generalmente obtenidas a partir del método PIRStarQueryToServers.
  - *Erasures*: Matriz de unos y ceros haciendo referencia a los borrones obtenidos en la transmisión. Se puede obtener como salida del método PIRQueryToServerNoise.
- **Output:** Vector conteniendo el archivo que se deseaba recuperar
- **Ejemplos:**

  Necesitamos una base de datos codificada usando un código C. En el ejemplo del protocolo 5 corregimos 1 error y 2 borrones. Este protocolo posee una mayor capacidad correctora. Corregiremos un error más en este caso.

```
F=GF(17)
x=vector(F,[10,4,6,14,2,11,16,9,1,5])
m=vector(F,[1,1,1,1,1,1,1,1,1,1])
C=codes.GeneralizedReedSolomonCode(x,3,m)

alpha=2

X1=vector(F,[1,2,3,4,5,6])
X2=vector(F,[2,4,6,8,10,12])
X=X1.column().augment(X2.column())
files=2
deseado=2

CodedDB=PIREncodeToServers(X,alpha,C)

        [15  6  2  5  0 12  2  7  6  1  9  7]
        [ 8  1 12  9  4  3  5  8 15  9 10 12]
        [13 12  4 10  0  7  4 14 12  2  1 14]
        [16  2  7  1  8  6 10 16 13  1  3  7]

queries=PIRBetterBizantineQueryMatrices(C,cooperantes,2,2,alpha,
        files,deseado)
```

```
        [
        [14  5  0  4]  [ 6  0 12  6]  [ 3 13  1 11]  [ 8 14  7 14]
        [ 2  2  4 16]  [ 6 14 14 12]  [16 10 13 11]  [ 5 11  2  2]
        [ 7  3 13 14], [ 9  5 11  6], [14 10  4  5], [ 0 13 15 15],

        [ 9  4  0  1]  [ 4  3  3 15]  [ 5 10  6  2]  [ 7  7  6 10]
        [13  1 12  4]  [ 7  0  6  1]  [15  7  2  7]  [14  4  8 16]
        [ 4  0 15  8], [ 1 14  7 12], [ 5  1  0 16], [13  9 14 11],

        [ 2  6 15  7]  [13 15  0  0]  [10 11  4  5]  [ 1 16  9  3]
        [ 8  3  0  3]  [11 12  5  2]  [ 4  8 15  5]  [ 0 13  5 11]
        [10  6 11 12], [ 3 16  4 16], [ 8  4 15  3], [ 6  2  0  5]
        ]

[Response,Erasures] = PIRQueryToServerNoise(CodedDB,queries,2,2)

  [
  [
  [ 8]  [ 0]  [ 5]  [ 0]  [ 7]  [15]  [8]  [9]  [16]  [12]  [ 7]  [ 6]
  [14]  [13]  [ 9]  [ 0]  [ 2]  [13]  [7]  [5]  [ 0]  [12]  [10]  [ 1]
  [10], [16], [12], [13], [13], [ 0], [5], [0], [13], [ 1], [ 0], [16]
  ],
   [0 1 0 1 0 0 0 0 0 0 0 0]
   [0 0 0 1 0 0 0 0 1 0 0 0]
   [0 0 0 0 0 1 0 1 0 0 0 0]
  ]

PIRBetterBizantineDecodeFromResponses(C,cooperantes,2,2,alpha,Response,
        Erasures)

        [2, 4, 6, 8, 10, 12]
```

Apéndice B

Contenido del repositorio

Junto a este documento se adjunta un link a un repositorio que contiene todo el código implementado y pruebas realizadas. En este apéndice explicaremos los contenidos de este repositorio.

En el momento de entrega del TFG dicho repositorio se encuentra en el siguiente link

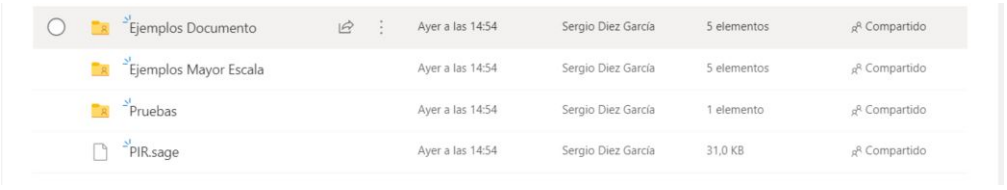

Figura B.1: Contenido del repositorio

El contenido del repositorio está dividido en tres carpetas y un archivo con extensión *.sage*. En primer lugar, el archivo $PIR.sage$ contiene la implementación de todos los métodos explicados a lo largo del documento. Para cargarlo en una nueva hoja de cálculo, únicamente tenemos que introducir el comando $load(./PIR.sage)$ con la ruta de dicho archivo.

En la carpeta *"Ejemplos Documento"* encontraremos las hojas de cálculo donde se han realizado los distintos ejemplos a lo largo del documento. Se ha incluido el código para los 5 ejemplos realizados.

Figura B.2: Contenido de Ejemplos Documento

En la carpeta *Ejemplos mayor escala* se incluyen los dos ejemplos introducidos en el capítulo 7. Además de los dos especificados en dicho capítulo se han añadido otras pruebas similares para otros protocolos. Además de los programas se incluye en la carpeta *Ejemplos* un conjunto de archivos de texto y fotos que se usan en las pruebas.

Figura B.3: Contenido de Ejemplos mayor escala

Por último, en la carpeta *Pruebas* podemos encontrar otro conjunto de pruebas a las que no se ha hecho referencia en el documento, pero que se han realizado para comprobar el correcto funcionamiento de los programas.

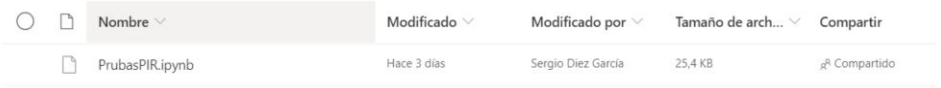

Figura B.4: Contenido de Pruebas

Bibliografía

[1] Sergio Díez García, *PIR: Recuperación de Información de forma Privada*, TFG de matemáticas, Universidad de Valladolid, 2020

[2] Ruud Pelikan, Xin-Wen Wu, Stanislav Bulygin, Relinde Jurrius. *Codes, Criptology and Curves with Computer Algebra*, Cambridge University Press, 2017

[3] Carlos Munuera Gomez, Juan Gabriel Tena Ayuso. *Codificación de la Información*, Universidad de Valladolid, 1997

[4] Jørn Justesen, Tom Hohøldt. *A course in Error-correcting Codes*. European Mathematical Society, 2004

[5] R. Tajeddine, O.W. Gnilke, S. El Roayheb, *Private Information Retrieval From MDS Coded Data in Distributed Storage Systems*, IEEE Transactions on Information Theory 64 (2018) 7081-7093.

[6] R. Freij-Hollanti, O. Gnilke, C. Hollanti, D. Karpuk, *Private Information Retrieval from Coded Databases with Colluding Servers*, SIAM J. Appl. Algebra Geometry 1 (2017), 647–664

[7] R. Tajeddine, O.W. Gnilke, D. Karpuk, R. Freij-Hollanti, C. Hollanti, *Robust Private Information Retrieval from Coded Systems with Byzantine and Colluding Servers*, arXiv 1802.03731

[8] R. Tajeddine, O.W. Gnilke, D. Karpuk, R. Freij-Hollanti, , *Private Information Retrieval from Coded Storage Systems with Colluding, Byzantine, and Unresponsive Servers*, IEEE Transactions on Information Theory 65 (2019) 3898 - 3906

[9] SageMath, the Sage Mathematics Software System (Version 9.1), The Sage Developers, 2020, https://www.sagemath.org

[10] Gregory V. Bard: University of Winsconsin-Stout, Menomonie, WI, *Sage for Undergraduates*, AMS

[11] Van Rossum, G., & Drake, F. L. (2009), *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace.