Códigos Binarios No Lineales en MAGMA

Memoria del proyecto de final de carrera correspondiente a los estudios de Ingeniería Superior en Informática presentado por Víctor Ovalle Arce y dirigido por Mercè Villanueva Gay.

Bellaterra, Junio de 2008

La firmante, Mercè Villanueva Gay , profesora del Departament d'Enginyeria de la Informació i de les Comunicacions de la Universidad Autónoma de Barcelona

CERTIFICA:

Que la presente memoria ha sido realizada bajo su dirección por Víctor Ovalle Arce

Bellaterra, Junio de 2008

Firmado: Mercè Villanueva Gay

A mis padres, por brindarme esta oportunidad

A mi hermana, por su apoyo

A mi niña, por existir

Agradecimientos

En primer lugar, he de decir que este proyecto no habría llegado a ningún lado sin la inestimable ayuda de mi directora de proyecto, Mercè Villanueva. Gracias a su ayuda y a su guía, hoy estoy escribiendo esta memoria.

También quiero dar las gracias a Bernat Gastón (mi manual de MAGMA personal), por su inalcanzable conocimiento de MAGMA. Él sabe estas cosas, y muchas más.

A parte, a toda la gente de proyectistas (sendistas g**s en su mayoría). Peña, Xavi, Álex, Vila, Gerard, Juan...muy buenos momentos, de verdad, difíciles de olvidar, como vosotros.

Por último, a todos los que han estado conmigo hasta el final de este camino, que se ha hecho duro, pero ha sido una de las mejores experiencias de mi vida.

A todos vosotros, mi más sincero agradecimiento.

Índice general

Índice de figuras

Capítulo 1

Introducción

Dependiendo de las propiedades algebraicas de las palabras código de un código corrector de errores, podemos clasificarlo como lineal o no lineal. Este proyecto se centra en el caso de los códigos correctores de errores construidos sobre el alfabeto binario, es decir, cuyas palabras se componen de ceros y unos.

Como su nombre indica, los códigos lineales cumplen la propiedad de linealidad, con lo cual podemos expresarlos mediante un subespacio vectorial. Este hecho es muy útil a la hora de trabajar con ellos en un ordenador, ya que podríamos representar un código de una cardinalidad considerable con una base de vectores que generan todas las palabras del código. Por otro lado, los códigos no lineales presentan la desventaja de la no linealidad. Esto hace que su tratamiento en un ordenador sea muy complicado, ya que si tenemos muchas palabras código, forzosamente tendremos que guardarlas todas en memoria, ya que no disponemos de una representación lineal con la que disminuir el tamaño utilizado.

Aquí empieza este proyecto. La idea es conseguir representar códigos no lineales de una manera similar a la que se utiliza en los códigos lineales, es decir, buscar una manera de minimizar el espacio ocupado en memoria. Para esto, nos centramos en la idea del núcleo de un código, al que llamaremos *kernel*. El kernel es una parte del código que, independientemente del tipo

del código (lineal o no lineal), siempre es lineal. Cuando encontramos el kernel del código, particionamos el resto del código en clases, y de cada clase escogemos un representante. A partir de estos representantes y del kernel, podemos generar todo el código, aunque sea no lineal. En el peor de los casos, en el que el kernel esté compuesto sólo de una palabra, tendríamos que guardar todas las palabras del código, pero conforme va creciendo el número de palabras del kernel, vamos mejorando el espacio ocupado. Además, hay que tener en cuenta que el kernel sí es lineal, así que se puede guardar simplemente una base de vectores que lo generen.

Este proyecto se centra, inicialmente y como principal objetivo, en ser capaces de representar códigos binarios no lineales de manera eficiente en un ordenador, utilizando el intérprete de MAGMA. Para ello, nos basamos en la idea del super dual, que es una generalización del dual para códigos lineales. Para construir el super dual, utilizaremos el kernel y los representantes de las clases anteriormente citados.

1.1. Objetivos

A grandes rasgos, los objetivos que hemos ido realizando a lo largo de la vida de este proyecto han sido los siguientes:

1. Estudiar el cálculo del kernel e implementar una función en MAGMA que lo calcule.

2. Estudiar el super dual e implementar en MAGMA una función que lo calcule a partir del kernel y los líderes de las clases.

3. Pensar cómo representar en MAGMA los códigos binarios no necesariamente lineales utilizando el super dual. Implementar constructores de códigos binarios no necesariamente lineales.

4. Implementar funciones que calculen las invariantes de un código, funciones para trabajar con las palabras de un código y funciones boleanas

que nos den más datos sobre un código. Implementar un constructor de códigos binarios aleatorio.

5. Testear todas las funciones implementadas y comprobar su correcto funcionamiento. Realizar ejemplos de funcionamiento de nuestro paquete.

6. Escribir la documentación del capítulo de ayuda del libro de MAGMA con ejemplos.

7. Escribir la memoria del proyecto.

1.2. Contenido de la memoria

Esta memoria se divide en varios capítulos. Veamos una pequeña descripción de lo que se va a tratar en cada uno de ellos.

- Bases teóricas: Veremos las bases teóricas de este proyecto. Teoría general de códigos binarios, así como teoría sobre el kernel y el super dual de un código. También se describirán algunas propiedades del super dual, que permitirán desarrollar funciones para trabajar con códigos no lineales.

- Planificación del proyecto: Veremos los objetivos del proyecto en profundidad, así como las diferencias entre la planificación temporal inicial y la real del proyecto. Desglosaremos el proyecto en tareas y veremos cuánto tiempo nos ha llevado cada una de ellas.

- Desarrollo del proyecto: Explicaremos con detalle qué ha sido necesario implementar para el paquete de MAGMA, así como el entorno en el que hemos trabajado. Veremos cómo hemos resuelto los problemas que han ido surgiendo.

- Handbook of MAGMA functions: Capítulo del manual de ayuda de MAGMA. Veremos de manera explícita qué funciones se han imple-

mentado, así como una breve explicación y ejemplos de cada una. Este capítulo está escrito en inglés debido a que se tiene que integrar en el libro de ayuda de MAGMA.

- Conclusiones y resultados: Explicaremos de manera resumida los resultados del proyecto. También, posibles mejoras que se pueden implementar en futuros proyectos.
- Bibliografía.
- Apéndice A: CD con el código fuente, tests, ejemplos y manual.

Capítulo 2

Bases Teóricas

En este capítulo veremos cuál es la base teórica sobre la que nos hemos apoyado para la correcta realización de nuestro paquete de software. Nos hemos basado en teorías desarrolladas en varios artículos, [4, 5], así como en libros de teoría básica de códigos, [6, 8].

2.1. Conceptos básicos

Para empezar, definiremos qué es un *código corrector de errores*. En las comunicaciones digitales, se pueden producir errores. Un código corrector de errores consiste en un método basado en la idea de introducir redundancia en los datos a transmitir para que, en el caso de que se produzca un error, el receptor sea capaz de corregirlo.

En nuestro caso, nos centraremos en los códigos binarios. Sea $\mathbb{Z}_2^n$ el espacio vectorial de dimensión n sobre el cuerpo finito $\mathbb{Z}_2 = GF(2)$. La *distancia de Hamming* entre dos vectores $x, y \in \mathbb{Z}_2^n$, $d(x, y)$, es el número de coordenadas en las cuales x e y difieren. La distancia de Hamming cumple la propiedad de simetría, es decir, el resultado de $d(x, y)$ y $d(y, x)$ es el mismo. Definiremos un *código binario* como un subconjunto de $\mathbb{Z}_2^n$. Los elementos de este subconjunto se denominan *palabras código*. La *distancia mínima de Hamming* de un código es la distancia de Hamming más pequeña entre todas las parejas de palabras

código diferentes. Así, definiremos un *código binario* $C(n, M, d)$ como un subconjunto de $\mathbb{Z}_2^n$ con cardinalidad M y distancia mínima de Hamming d.

Un código binario puede ser lineal o no lineal. Los códigos binarios lineales son aquellos que están definidos sobre el cuerpo finito $\mathbb{Z}_2$ y que, además de ser un subconjunto de $\mathbb{Z}_2^n$, son un subespacio vectorial. Así, podemos asegurar que la palabra cero (palabra en la que todas sus coordenadas son cero) pertenece al código. Por el contrario, si un código binario es no lineal, entonces no se cumple que sea un subespacio vectorial y, por lo tanto, la palabra cero no tiene porqué pertenecer al código.

Dos códigos binarios C_1 y C_2 de longitud n son *isomorfos* si existe una permutación π tal que $C_2 = \pi(C_1) = \{\pi(c) \mid c \in C_1\}$. Dos códigos binarios C_1 y C_2 de longitud n son *equivalentes* si existe un vector $a \in \mathbb{Z}_2^n$ y una permutación de coordenadas π tal que $C_2 = \{a + \pi(c) \mid c \in C_1\}$.

Sea C un código binario no lineal. Si la palabra cero no está en C, considerando un nuevo código binario C', tal que $C' = C + c$ para cualquier $c \in C$, podemos asegurar que la palabra cero pertenece al código binario C', que es equivalente a C.

Dos propiedades estructurales de los códigos binarios son el *rango* y la dimensión del *kernel*. El rango de un código binario C, $r = rank(C)$, es simplemente la dimensión de la expansión lineal, $\langle C \rangle$, de C. El kernel de un código binario C se define como $K(C) = \{x \in \mathbb{Z}_2^n \mid x + C = C\}$. Si la palabra cero pertenece a C, entonces $K(C)$ es un subespacio lineal contenido en C. Denominaremos la dimensión del kernel de C como $k = dimension(K(C))$. Estos parámetros r y k se pueden usar como distinción entre códigos binarios no equivalentes, ya que los equivalentes tienen los mismos parámetros r y k. Además, r y k nos permiten decidir si un código binario es lineal o no, ya que si el código binario no es lineal, se cumple que $K(C) \subset C \subset \langle C \rangle$. Los tres conjuntos son iguales únicamente cuando el código binario es lineal.

Definiremos también los conceptos de *matriz generadora* y *matriz de control* de un código. Dado un código C, su matriz generadora, G, es una matriz cuyas filas forman una base de C, es decir, es una matriz que genera todas

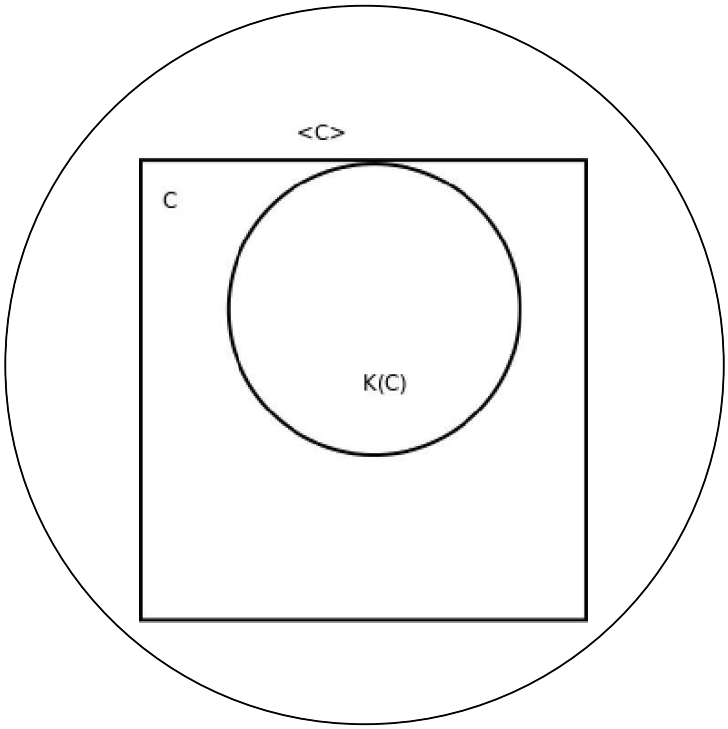

Figura 2.1: La relación entre $K(C)$, C y $\langle C \rangle$ en un código no lineal es $K(C) \subset C \subset \langle C \rangle$

las palabras código de C. Antes de definir qué es una matriz de control, es importante definir el concepto de *código dual*. Dado un código lineal C, su código dual es $C^{\perp} = \{x \in \mathbb{Z}_2^n \mid x \cdot y = 0, \forall\ y \in C\}$, donde $x \cdot y$ es el producto escalar de x e y . Si C es lineal, $C^{\perp}$ también es lineal. También es fácil ver que $C^{\perp\perp} = C$. Así, la matriz generadora del código $C^{\perp}$ es la *matriz de control* de C. La matriz de control nos permite saber si una palabra es del código o no, ya que las palabras código operadas con la matriz de control, dan cero como resultado.

Son importantes también los conceptos de *clases* y *líderes de las clases*. Sea C un código binario y x una palabra código de C, llamaremos clase al subconjunto $x + C = \{x + y \mid \forall\ y, x + y \in C\}$ de C. Es decir, la clase $x + C$ está formada por las palabras que resultan de sumar todas las palabras del código C a x, siempre y cuando la palabra resultado pertenezca a C. Además,

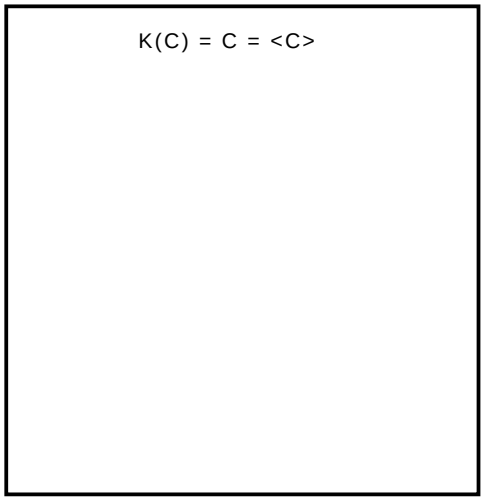

Figura 2.2: En un código lineal el kernel es igual al código, es decir, $K(C) = C = \langle C \rangle$

en nuestro caso, x es el líder de la clase $x + C$.

Para acabar, veremos qué es un *código perfecto* y un *código perfecto extendido*. Un código C de longitud n es perfecto si para algún entero $r \geq 0$, cada elemento de $\mathbb{Z}_2^n$ está a distancia menor o igual que r de, exactamente, una palabra código de C. Sea C un código perfecto, si añadimos un dígito de paridad a todas sus palabras código, obtenemos el código C^*. Podemos decir que C^* es un código perfecto extendido de C.

2.2. Representación de códigos binarios

Un código binario no lineal C se puede escribir como la unión de su kernel $K(C)$ más los trasladados del kernel, es decir,

$$C = \bigcup_{i=0}^{t} \Big(K(C) + c_i \Big),$$

donde $c_0 = \vec{0}$. Lo que hacemos es subdividir el código C en unas cuantas clases, $[c_1, c_2, \ldots, c_t]$, que son traslados del kernel. El número de clases se

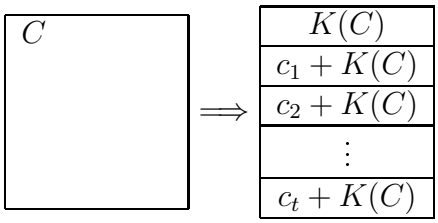

Figura 2.3: Ejemplo de división de un código binario en $K(C)$ y las clases de C

calcula en función de la dimensión del kernel, k, y la cardinalidad del código, M, es decir, $t+1 = M/2^k$, donde t es el número de trasladados del kernel. Así, tendremos el código binario C dividido en el kernel y sus t trasladados, es decir, tendremos $t+1$ clases. De ahora en adelante, nos referiremos a $[c_1, c_2, \ldots, c_t]$ como líderes de las clases. En la figura 2.3 se ve gráficamente la división de un código binario no lineal en el kernel y las clases.

Sea C un código binario de longitud n tal que la palabra cero pertenece a C, con kernel $K(C)$ de dimensión k y líderes de las clases $S = [c_1, c_2, \ldots, c_t]$. Es fácil ver que podemos representar el código binario C como el kernel más los líderes de las clases, es decir, C puede ser representado por la matriz $\binom{H}{S}$, donde H es una matriz generadora de $K(C)$. Como $K(C)$ es un subespacio lineal, podemos representarlo de manera compacta como un código binario lineal, aprovechando su matriz generadora binaria de tamaño $k \times n$. El kernel ocupa un espacio en memoria del orden de $O(nk)$. Además, el kernel más los t líderes de las clases c_i, donde $i \in \{1, \ldots, t\}$, ocupa un espacio de orden $O(n(k+t))$.

Para el caso $t = 0$, es decir, cuando el código binario C es lineal, tenemos que $C = K(C)$ y que el código puede ser representado por su matriz generadora binaria, osea que el espacio en memoria es de orden $O(nk)$. Cuando $t+1 = M$, quiere decir que el kernel $K(C)$ sólo está formado por una palabra, que es la palabra cero. Por consiguiente, el resto de clases estarán formadas de una sola palabra. Por lo tanto, esta solución es tan mala como representar el código como un conjunto de todas sus palabras código, osea que su orden

sería $O(nM)$ donde $M >> k$.

La representación expuesta es una representación compacta del código C pero no es recomendable para la mayoría de situaciones ya que no es independiente de cualquier matriz generadora particular H, es decir, esta representación es demasiado poco flexible a la hora hacer transformaciones básicas de filas. Por ejemplo, si permutamos una de las filas de S con una de las filas de H, la matriz ya no representa el mismo código. Por esto, utilizaremos la siguiente representación.

El *sistema de paridad* del código binario C es una matriz binaria de dimensión $(n-k) \times (n+t)$, $(G|S)$, donde G es la matriz generadora del dual del kernel, $K(C)^{\perp}$, de dimensión $(n-k) \times n$, y $S = (G \cdot c_1\ G \cdot c_2 \ldots\ G \cdot c_t)$, de dimensión $(n-k) \times t$. El *super dual* del código binario C es el código binario lineal generado por el sistema de paridad $(G|S)$. Notar que si C es un código binario lineal, el super dual es el código dual $C^{\perp}$. Por lo tanto, el sistema de paridad de un código binario es una generalización de la matriz de paridad para códigos binarios lineales.

2.3. Propiedades del super dual

Ha sido necesario estudiar cómo podemos decidir si dos códigos binarios C_1, con con matriz de paridad generalizada $(G_1|S_1)$, y C_2, con matriz de paridad generalizada $(G_2|S_2)$ son iguales, o bien si C_1 está contenido en C_2. También se ha estudiado cómo podemos saber si una palabra código x pertenece a C, y cómo calcular algunos parámetros del código, adaptándonos al super dual.

Para saber si $C_1 = C_2$, teniendo en cuenta la representación que nos proporciona el super dual, el número de filas de las matrices $(G_1|S_1)$ y $(G_2|S_2)$ tiene que ser el mismo. Además, el kernel de C_1 tiene que ser igual al kernel de C_2, o equivalentemente, el espacio generado por G_1 tiene que ser igual al generado por G_2. Así tendremos también que S_1 y S_2 tienen el mismo número de filas. Finalmente, hay que comprobar que los líderes de las clases de C_2

son palabras código de C_1 o viceversa.

Para ver si $C_1 \subseteq C_2$ se ha tenido que hacer un estudio más en profundidad, ya que el problema no es trivial. Primero de todo, hay que comprobar que las longitudes de ambos códigos son iguales. Para que C_1 esté contenido en C_2, la cardinalidad de C_1 tiene que ser menor o igual a la cardinalidad de C_2. En este momento, tenemos dos caminos a seguir. El primero es que se cumpla que $K(C_1) \subseteq K(C_2)$. En este caso, es suficiente con comprobar que todos los líderes de las clases de C_1 forman parte de C_2. Sin embargo, si el kernel de C_1 no forma parte del kernel de C_2, no podemos asegurar que $C_1 \nsubseteq C_2$.

Si el segundo kernel no contiene al primero, tenemos que trabajar con la porción de kernel en que los dos kernels coinciden, es decir, necesitamos $\Bbbk = K(C_1) \bigcup K(C_2)$. Una vez calculada la unión de los dos kernels, tenemos que extender una base de $\Bbbk$ sobre el kernel de C_2. Ahora, para que $C_1 \subseteq C_2$ se tiene que cumplir que todas las nuevas palabras código que hay en la base extendida pertenecen a C_2. Si esto se cumple, ya sólo nos queda mirar que los líderes de las clases de C_1 pertenecen a C_2 y que el resultado de sumar las nuevas palabras de la base extendida a los líderes de las clases también pertenece a C_2. En este momento, ya podemos saber si C_1 está contenido en C_2.

Si queremos saber si una palabra código, x forma parte del código binario C, con matriz de paridad generalizada $(G|S)$, hay que plantearse dos casos. El primero, que es el más sencillo, es el caso en que C es lineal, osea que la matriz de paridad generalizada es G, que es la matriz de control de C. En este caso, el procedimiento habitual es suficiente, es decir, multiplicamos la palabra x por la matriz de control de C y, si su resultado es cero, es decir, si $G \times x^\top = 0$, podemos asegurar que x pertenece a C. Hay que tener en cuenta que, debido a la representación a partir del super dual, tenemos que mirar si la palabra x pertenece al kernel, es decir, hay que comprobar que $x \in K(C)$. Si C es un código no lineal, en el caso que $G \times x^\top \neq 0$, no podemos asegurar que x no pertenece a C. Hay que comprobar si el vector $G \times x^\top$ es una de las columnas de la matriz S. Es decir, x pertenece a C si $G \times x^\top = 0$ o si

$G \times x^{\top}$ es una columna de la matriz S.

Para el cálculo del rango y la dimensión del kernel, también se tiene que tener en cuenta que nuestro código se representa con la matriz $(G|S)$, osea que tendremos que amoldarnos a esto para realizar los cálculos. Para calcular la dimensión del kernel, k de C, sabiendo que G es la matriz generadora de $K(C)^{\perp}$, tendremos que k es la resta de la dimensión de G menos la longitud del código C. Para calcular el rango de un código binario C cuya matriz de paridad generalizada es $(G|S)$, tenemos que sumar la dimensión del kernel al rango de S. También hemos tenido que estudiar cómo saber si un código binario ha sido construido a partir de un código cuaternario, o un código $\mathbb{Z}_2\mathbb{Z}_4$-aditivo. Este problema tampoco es nada trivial, o por lo menos, no es tan trivial como esperábamos.

Los códigos cuaternarios son aquellos que se construyen sobre el alfabeto $\mathbb{Z}_4 = \{0, 1, 2, 3\}$. Los códigos $\mathbb{Z}_2\mathbb{Z}_4$-aditivos son códigos cuyas palabras tienen coordenadas en $\mathbb{Z}_2$ y en $\mathbb{Z}_4$. Para pasar de $\mathbb{Z}_4$ a $\mathbb{Z}_2$, utilizamos la notación en código de Gray. El código de Gray consiste en ordenar 2^n números binarios de tal forma que cada número sólo tenga un dígito binario diferente a su predecesor. Así, un mapa de Gray podría ser:

$0 \leftrightarrow 00$
$1 \leftrightarrow 01$
$2 \leftrightarrow 11$
$3 \leftrightarrow 10$

Para pasar de códigos cuaternarios a códigos binarios se utiliza este mapa de Gray. De esta forma, cada palabra que tiene longitud n en $\mathbb{Z}_4$, se convierte en una palabra de longitud $2n$ en $\mathbb{Z}_2$.

Para pasar de códigos $\mathbb{Z}_2\mathbb{Z}_4$-aditivos a códigos binarios, la idea es la misma. En el proyecto [3], se han desarrollado funciones que realizan esta conversión de códigos $\mathbb{Z}_2\mathbb{Z}_4$-aditivos a códigos binarios utilizando también el mapa de Gray.

En ambos casos, la decisión de si un código binario está construido a partir de un código cuaternario o $\mathbb{Z}_2\mathbb{Z}_4$-aditivo, pasa por comprobar que el espacio

generado por el código binario es la antiimagen de un código cuaternario o $\mathbb{Z}_2\mathbb{Z}_4$-aditivo bajo el mapa de Gray.

Las funciones implementadas en este nuevo paquete son aplicables a códigos binarios no necesariamente lineales.

Capítulo 3

Planificación del proyecto

Ya hemos comentado los objetivos en la introducción, pero en este capítulo se verán más a fondo. También veremos la planificación que teníamos inicialmente, y en qué se ha convertido al final.

3.1. Objetivos

1. Estudiar el cálculo del kernel. Es difícil, computacionalmente hablando, calcular el kernel de un código binario de una manera eficiente. Intentamos mejorar la función desarrollada en [7] y [9] , `KernelZ2(C)`, que ya se intentó mejorar en un proyecto del año pasado [2].

2. Implementar una función en MAGMA que calcule el kernel teniendo en cuenta las mejoras implementadas el año pasado en [2]. Nuestra implementación debe ser lo más eficiente posible, tanto en complejidad de cálculo, como en tamaño ocupado en memoria. Decidimos que retornaremos el kernel como un código binario lineal y los líderes de las clases como una lista de palabras código.

3. Estudiar el `super dual`. Hemos de decidir si es interesante aplicarla a nuestro problema para representar códigos binarios no necesariamente lineales. El hecho de poder representar un código no lineal a partir de

una matriz nos puede ser muy útil.

4. Implementar en MAGMA las funciones necesarias para calcular el super dual de un código binario C. De esta manera, representamos un código no necesariamente lineal como el super dual calculado a partir del kernel de C y los líderes de las clases.

5. Comenzar a realizar el paquete de MAGMA para códigos binarios no necesariamente lineales. Hay que pensar cómo vamos a representar estos códigos. Podemos basarnos en la estructura que MAGMA utiliza para códigos, o crear una representación totalmente nueva. Nos inclinamos por la opción de reutilizar la estructura que MAGMA utiliza para códigos, ya que así será más fácil integrar nuestro paquete en el intérprete.

6. Decidir qué atributos vamos a añadir a la estructura de datos de MAGMA para representar nuestro código de forma única. Consideramos necesario guardar la longitud, el kernel y los líderes de las clases, así como un boleano que nos diga si el código es lineal o no.

7. Implementar los constructores de códigos binarios no necesariamente lineales. Decidimos a partir de qué tipos de datos (secuencias de vectores, subespacios lineales, matrices, etc) vamos a permitir construir códigos binarios en MAGMA con esta estructura.

8. Implementar funciones que calculen las invariantes de un código (longitud, cardinal, distancia mínima de Hamming, rango y dimensión del kernel, etc).

9. Implementar funciones para trabajar con las palabras código. Estas funciones nos permitirán saber si una palabra u pertenece al código binario C, o nos retornarán una palabra código aleatoria del código binario.

10. Implementar funciones boleanas que nos pueden dar más datos sobre el código con el que estemos trabajando, como por ejemplo, saber si un

código es perfecto, o si es binario lineal, si está construido a partir de un código $\mathbb{Z}4$ o un código $\mathbb{Z}_2\mathbb{Z}_4$-aditivo, etc.

11. Implementar un constructor de códigos binarios aleatorio, es decir, un constructor que tan sólo recibe la longitud y la cardinalidad del código y lo genera de manera aleatoria. También damos la opción de introducir la dimensión del kernel a este constructor de códigos binarios aleatorio.

12. Testear todas las funciones implementadas y comprobar su correcto funcionamiento. A partir de unos datos de entrada, ver si obtenemos la salida esperada.

13. Realizar ejemplos de funcionamiento de nuestro paquete para incluirlos en la carpeta de ejemplos del paquete y escribir el capítulo de ayuda del libro de MAGMA en inglés.

14. Escribir la memoria del proyecto.

3.2. Planificación Temporal

En su planteamiento inicial, este proyecto tenía, como condición indispensable, calcular el kernel de un código binario no lineal, y calcularlo de manera eficiente. He de decir que, aunque mi directora de proyecto confiaba en que sobrara el tiempo suficiente como para realizar también un paquete para MAGMA, yo no estaba muy seguro ya que el cálculo del kernel me parecía complejo y difícil de hacer de manera eficiente. No obstante, la idea era implementar esta función del kernel, realizar un paquete para MAGMA para el tratamiento de códigos binarios no necesariamente lineales, y realizar algunas funciones de construcción de familias de códigos particulares.

Aun así, al final no se han podido realizar las funciones de construcción de familias de códigos particulares, debido a que algunas de las tareas anteriores resultaron más complicadas de lo que creíamos y, por tanto, se han alargado más de lo previsto, como se muestra en la figura 3.1.

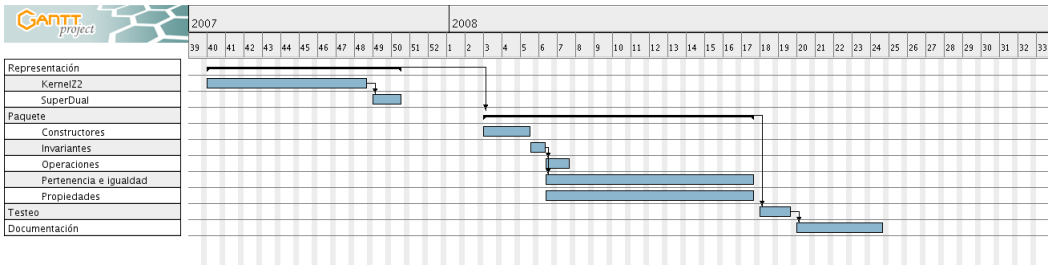

Figura 3.1: Planificación temporal real

Hablando de fechas, este proyecto ha tenido una duración aproximada de 8 meses, que comprenden el período de octubre de 2007 a mayo de 2008. Si lo desglosamos, teniendo en cuenta los objetivos y las tareas que se ven en la figura:

- **KernelZ2**: La implementación de esta función nos ha llevado un total de dos meses. Se inició en octubre de 2007 y la dimos por correcta a finales de noviembre.

- **SuperDual**: Una vez que sabemos calcular el kernel y los líderes de las clases, los utilizamos para implementar el super dual. Para esto, tardamos menos, y en unos 10 días implementamos y testeamos la función de cálculo del super dual. Así, nos fuimos de vacaciones en invierno con una parte muy importante del proyecto hecha.

- **Constructores**: Cuando volvimos en enero de 2008, nos quedaba implementar el paquete, así que entre estudiar qué queríamos exactamente e implementar constructores para los códigos, estuvimos un par de semanas. Hacia el final del proyecto, nos dimos cuenta de que teníamos algunos pequeños errores que hubo que corregir, pero no fue nada dramático, y por eso no se muestra en el diagrama de Gantt.

- **Invariantes**: Esta ha sido, quizá, la parte más sencilla de realizar. El cálculo de estas propiedades de los códigos es sencillo y rápido, y sólo invertimos en esta parte una semana.

- **Operaciones**: Estas funciones tampoco fueron muy complicadas. No fueron tan triviales como las anteriores, pero tan sólo requirieron diez días de trabajo, en paralelo con las funciones de pertenencia y las de propiedades.

- **Pertenencia e igualdad** y **Propiedades**: Sin duda, estas dos partes han sido las más duras en cuanto a trabajo se refiere, y por ello las pongo en el mismo punto. La dificultad teórica de las funciones aquí implementadas ha hecho que hayamos tenido que hacer nuevas versiones continuamente.

Capítulo 4

Desarrollo del proyecto

En este capítulo veremos los detalles de implementación del proyecto, así como una breve explicación del entorno que se nos ha proporcionado para trabajar.

4.1. Entorno de desarrollo

Para realizar este proyecto, el *Departament d'Enginyeria de la Informació i de les Comunicacions (dEIC)*, me proporcionó un ordenador en la sala de proyectistas del departamento, así como acceso a una cuenta en un servidor del *Grupo de Combinatoria y Codificación (CCG)*.

El ordenador de proyectistas es un Intel Pentium 4 a 2.00 GHz, con 512 MB de RAM. El sistema operativo es una Fedora 5. Disponemos de una copia de Magma en un servidor llamado Macwilliams, que tiene un procesador de doble núcleo, 2GB de RAM, y conexión por ssh habilitada.

Magma es un paquete de software privativo de alta calidad desarrollado en Australia. Tiene una gran cantidad de funcionalidades en teoría de números, geometría algebraica, teoría de grafos y teoría de códigos. Trabaja con un lenguaje especializado para matemáticos. A continuación, vamos a describir algunas de sus características más importantes:

- Magma tiene una interfaz por linea de comandos pero no tiene interfaz

gráfica. En nuestro proyecto, esto no ha sido una desventaja, ya que no necesitamos interfaz gráfica para nada.

- MAGMA tiene un soporte limitado para computación numérica e integración simbólica. No es posible para el usuario final definir sus propios tipos de datos, pero los desarrolladores de MAGMA suelen escuchar las peticiones de los usuarios de añadir nuevos tipos de datos al núcleo escrito en C de MAGMA para poder ofrecerlas en la siguiente versión del programa.

- MAGMA da grandes facilidades a la hora de guardar todo lo que se ha hecho durante una sesión, aunque, por ejemplo, no existe ninguna manera de guardar en disco un objeto independiente que se haya creado.

- MAGMA tiene una librería grande y muy optimizada. Es muy potente y útil para la investigación en ciertas áreas, como la teoría de códigos. La gran mayoría de MAGMA está escrito en C, osea que se pueden utilizar las funcionalidades de algunas librerías de C ya existentes. No obstante, este linkado sólo se puede realizar recompilando MAGMA, y, por tanto, los usuarios finales no pueden hacerlo, ya que carecen del código fuente para recompilarlo. Además, debido a que MAGMA es privativo, algunas librerías libres no se pueden linkar debido a restricciones de licencia.

Actualmente MAGMA da soporte a funcionalidades básicas para códigos sobre cuerpos finitos y códigos sobre anillos de enteros y anillos de Galois. Todos estos tipos de códigos son lineales. Además, MAGMA nos provee de funciones para el caso especial de códigos binarios lineales, que son los que se construyen sobre el cuerpo finito $GF(2)$, o de manera equivalente, sobre el anillo $\mathbb{Z}_2$.

4.2. Implementación en MAGMA

4.2.1. Cálculo del kernel en MAGMA

A la hora de implementar el paquete para trabajar con códigos binarios ya sean lineales o no lineales, teníamos que seguir un orden más o menos estricto, es decir, lo primero de todo es conseguir calcular el kernel de un código binario C de manera eficiente. En proyectos de años anteriores ya se había intentado hacer esta función.

En [7] y [9], simplemente se hizo una implementación de la definición matemática de kernel en MAGMA. Es decir, la implementación de la fórmula

$$K(C) = \{x \in \mathbb{Z}_2^n \mid x + C = C\}$$

Esta primera versión, evidentemente, daba resultados correctos. El problema es que para códigos de relativamente pocas palabras, la función empezaba a tardar muchísimo en calcular el kernel. En la figura 4.1 se ve la diferencia entre la primera versión del cálculo del kernel, y la versión implementada en el proyecto actual. Por ejemplo, para un código de 1024 palabras, que no son muchas, la implementación anterior comienza a ser poco eficiente. Por lo tanto, comparándolo con la primera versión, mejoramos muchísimo el cálculo del kernel.

En [2] se intentaron mejorar los resultados de la función implementada en [7] y [9] introduciendo la idea del subespacio lineal. Ya que el kernel es un subespacio lineal que se puede generar con una base de vectores, y dado que MAGMA trabaja de manera muy eficiente con espacios vectoriales, ¿Por qué no representar el kernel como un subespacio vectorial? Partiendo de esta base, el proyectista implementó la segunda versión del cálculo del kernel, que mejoraba bastante el funcionamiento de la anterior, como se muestra en la memoria de su proyecto. Aun así, como se ve en la figura 4.2, conseguimos mejorarla. Esta segunda versión calculaba el kernel más rápido que la primera versión, pero aun así, cuando el número de palabras crecía, su tiempo de

Cardinalidad del código	Cardinalidad del kernel	Tiempo Versión 1	Tiempo Nuevo	Mejora
16	4	0.000	0.000	0.000
32	8	0.010	0.000	0.010
64	8	0.000	0.010	-0.010
128	16	0.010	0.010	0.000
256	32	0.070	0.010	0.060
512	64	0.460	0.020	0.440
1024	128	3.280	0.070	3.210
2048	128	12.910	0.230	12.680
4096	1024	386.550	0.650	385.900
8192	512	776.450	2.410	774.040

Figura 4.1: Comparación con la primera versión

ejecución se incrementaba de manera espectacular, como se ve en el último caso, en el que el tiempo se multiplica por más de diez.

Cardinalidad del código	Cardinalidad del kernel	Tiempo Versión 2	Tiempo Nuevo	Mejora
16	4	0.000	0.000	0.000
32	8	0.000	0.000	0.000
64	8	0.000	0.000	0.000
128	16	0.000	0.010	-0.010
256	16	0.010	0.020	-0.010
512	32	0.060	0.030	0.030
1024	128	0.250	0.070	0.180
2048	256	1.120	0.200	0.920
4096	128	5.300	0.760	4.540
8192	128	16.010	2.800	13.210
16384	2048	180.960	10.010	170.950

Figura 4.2: Comparación con la segunda versión

Siguiendo con la idea de utilizar espacios vectoriales, pero dándole un enfoque totalmente nuevo, comenzamos a implementar nuestra versión de la función, utilizando como base un algoritmo implementado en C por el *Grupo de Combinatoria y Codificación (CCG)*, y adaptándolo a Magma. El algoritmo es el siguiente:

```
funct KernelLeaders(C : array [1 . . M ] of CodeWord, M : integer)
 const
   KERNEL = 0
   NOT PROCESSED = -1
   LEADER = -2
 endconst
 var
   kernelSet : array [1 . . M ] of integer
   leadersSet : array [1 . . M ] of integer
   kernelBound : integer
   count, i, j : integer
 endvar
 begin
   QSort(C)
   count <- 1
   C[1].state <- KERNEL
   C[i].state <- NOT PROCESSED for all i > 1
   kernelSet <- {1}
   leaderSet <- {}
   if Weight(M) = 1
     then
       kernelBound <- M/4
     else
       kernelBound <- M
   fi
   i<-2
   while count <= M || #kernelSet <= kernelBound do
     if C[i].state = NOT PROCESSED
       then
         v <- C[i].v
         if C + v = C
           then
             for u in kernelSet do
               j <- BinarySearch(C, v + u)
               C[j].state <- KERNEL
               kernelSet <- kernelSet union {v + u}
               count <- count + 1
               for c in leaderSet do
```

```
                j <- BinarySearch(C, v + u + c)
                if C[j].state = NOT PROCESSED
                  then
                    Union(C, c, C[j])
                    count <- count + 1
                  else
                    Find(C, c, C[j])
                    Union(C, c, C[j])
                fi
              od
            od
          else
            C[i].state <- LEADER
            count <- count + 1
            leaderSet <- leaderSet union {v}
            for u in kernelSet do
              j <- BinarySearch(C, v + u)
              Union(C, v, C[j])
            od
        fi
    fi
    i<-i+1
  od
  if #kernelSet > kernelBound
    then
      kernelSet = C
      leaderSet = {}
  fi
  return(kernelSet, leaderSet)
end
```

Aunque el núcleo de MAGMA está escrito en C, no podemos utilizar este código directamente. El problema es que MAGMA es un programa privativo, y no somos desarrolladores para ellos. Por otro lado, tampoco es posible dar uso en MAGMA a la salida del programa hecho en C.

La idea del algoritmo es calcular el kernel y los líderes de las clases (ver capítulo 2), pero sin procesar una vez todas las palabras, es decir, procesamos

una palabra, decidimos si pertenece al kernel o no, y, en caso afirmativo, añadimos al kernel la palabra código y todas las que se generan sumándole las palabras que ya forman parte del kernel.

En MAGMA, esta idea mejora mucho más debido a que, como ya hemos comentado, MAGMA es muy eficiente en el tratamiento de espacios vectoriales. Así, si resulta que la palabra código que estamos procesando pertenece al kernel, añadimos al subespacio del kernel esta palabra y todo el subespacio generado por ella.

Este algoritmo ha sufrido bastantes cambios en su adaptación a MAGMA. Para empezar, en el algoritmo se guardan las palabras código en una estructura que contiene la palabra código, su estado y su peso. En MAGMA, no necesitamos el peso, así que no lo guardamos. En el algoritmo, el peso se utiliza para optimizar las búsquedas binarias que se hacen. Dado que MAGMA nos provee de otras funcionalidades, no es necesario que hagamos ninguna búsqueda de este estilo.

Otro cambio muy importante es la manera de retornar el kernel. En el algoritmo, se retorna el kernel como un conjunto de todas sus palabras. En un principio se pensó en retornar el kernel como un espacio vectorial, ya que es muchísimo más óptimo tanto en memoria ocupada como en tiempo de cálculo. Pero en un momento de la vida del proyecto, se decidió que era más útil devolverlo como un código binario lineal, ya que al utilizar la representación del super dual, nos interesaba más que el kernel fuera directamente un código. Dado que para el cálculo del super dual se trabaja con la matriz generadora del código generado por el kernel, si seguíamos devolviendo un subespacio vectorial, tendríamos que calcular el código generado por el kernel, para saber su matriz generadora y así calcular el super dual. Como MAGMA representa sus códigos con su matriz generadora y algunos parámetros más, decidimos que guardar el kernel como un código era mucho más práctico, y más teniendo en cuenta que la representación que utiliza MAGMA para los códigos y para los espacios vectoriales, son equivalentes tanto en tiempo de cálculo como en memoria ocupada.

La implementación de esta función ha sido uno de los puntos que más han pesado en el proyecto. Era imperativo calcular el kernel de una manera más óptima que la implementada en proyectos anteriores, si queríamos llegar a hacer un paquete de software eficiente. Por ello, nos hemos volcado en la realización de esta función desde el inicio de la vida del proyecto. Así, a finales de noviembre, obtuvimos una buena solución. La función se llama `KernelZ2(C)`.

El siguiente paso era implementar el super dual en MAGMA. Como vimos en el capítulo 2, una vez que tenemos el kernel y los líderes de las clases, es sencillo calcular la matriz de paridad generalizada que se utiliza en el sistema del super dual. Si recordamos, esta matriz es de la forma $(G|S)$, donde G es la matriz generadora del dual del código generado por el kernel y S se construye multiplicando la matriz G por los líderes de las clases.

A diferencia de la función `KernelZ2`, y para nuestra sorpresa, esto no nos llevó mucho tiempo. Estuvimos unos diez días entre implementación y pruebas de la función.

4.2.2. Representación de códigos binarios en MAGMA

Una vez implementado el super dual, debíamos diseñar e implementar el paquete para trabajar con códigos binarios. En este momento teníamos una manera robusta de representar códigos binarios no necesariamente lineales en MAGMA, y la intención era basar todo el paquete en el kernel, los líderes de las clases y el super dual. Este ha sido el grueso del proyecto, que nos ha ocupado desde la vuelta al trabajo en enero, hasta bien entrado abril.

Lo primero que debíamos pensar era cómo íbamos a estructurar los códigos creados con nuestro paquete. Teníamos dos opciones:

1. Crear una estructura de datos en la que guardar la matriz de paridad generalizada y los parámetros del código.

2. Utilizar el tipo de datos `Code` existente en MAGMA añadiendo los campos que hicieran falta.

Después de pensarlo, decidimos que la segunda opción era, en nuestro caso, mejor. Como la idea es representar códigos a partir de otros códigos, este tipo de datos nos iba bien, y aunque debíamos añadirle algunos campos, la solución es muy elegante y se amolda mejor a MAGMA.

Primero de todo, decidimos que un código no necesariamente lineal se representaría con un código lineal más unos cuantos atributos. Así, en la variable que representa al código, guardaríamos el super dual generado a partir del kernel y los líderes de las clases, es decir, el código con matriz de paridad generalizada $(G|S)$, y algunos atributos más.

También nos planteamos qué necesitábamos añadir al tipo `Code` para identificar cada código de forma única. Después de un par de reuniones, decidimos añadir los siguientes atributos:

- **Longitud**. MAGMA ya implementa una función que retorna la longitud de un código, pero debido a que nuestra representación no es exactamente la misma que utiliza MAGMA, la función implementada por el intérprete no se puede utilizar, ya que retornaría la longitud total del código super dual, que es la longitud del código más el número de clases. Este atributo y la matriz de paridad generalizada $(G|S)$ son indispensables a la hora de trabajar con códigos representados con el super dual, ya que, la longitud nos permite separar G y S.

- **Kernel**. Aunque podríamos obtener este atributo a partir de la longitud y la matriz de paridad generalizada, consideramos que sería interesante tenerlo previamente calculado, ya que se utiliza continuamente a la hora de trabajar con los códigos representados por el super dual.

- **Líderes de las clases**. Es una lista de palabras código. Con cada palabra representamos toda una subclase del código. Este es otro atributo que no es estrictamente necesario guardar pero, de manera similar al kernel, se utiliza continuamente a la hora de trabajar con códigos representados por el super dual.

- **Distancia mínima**. Como la distancia mínima es algo complejo de

calcular, al guardarla como una parte del código, nos aseguramos de que sólo se calcula una vez.

- **Condición de linealidad**. Este es un centinela que nos dice si el código es lineal o no lo es. Durante la ejecución de las funciones desarrolladas en el paquete, es interesante saber si el código es lineal o no para dar un tratamiento u otro.

Una vez decididos los nuevos atributos, empezamos a pensar en los constructores. Había que definir los datos de entrada del constructor, es decir, las estructuras de datos a partir de las cuales podemos generar un código binario lineal o no lineal. Siguiendo la linea de otro proyecto del departamento [3], decidimos construir nuestros códigos a partir de:

- Una lista de vectores sobre $V = \mathbb{Z}_2^n$,
- o una matriz $m \times n$ sobre el anillo $\mathbb{Z}_2$,
- o un subespacio vectorial de $V = \mathbb{Z}_2^n$,
- o un código binario lineal de MAGMA,
- o un código cuaternario lineal de MAGMA,
- o un código $\mathbb{Z}_2\mathbb{Z}_4$-aditivo [3].

Si estamos en el primer caso, simplemente enviamos la lista de vectores a la función `KernelZ2`, que calcula el kernel. Con el kernel y los líderes, ya tenemos suficiente para decidir si el código resultante será lineal o no. Es fácil ver que, si la lista de líderes de las clases está vacía, el código es lineal, ya que todas las palabras código pertenecen al kernel, y el kernel, como ya se ha dicho, es lineal. Por lo tanto, mirando la lista de líderes, actuamos de una manera u otra en función de si está vacía o no. Si el código no es lineal, calculamos el super dual con el kernel y los líderes de las clases y rellenamos los campos nuevos del tipo de datos:

- La longitud es el número de coordenadas de las palabras código.

- El kernel y los líderes son exactamente los que retorna la función `KernelZ2`
- La distancia mínima, por defecto, no se calcula en los códigos no lineales, a no ser que el usuario haga una llamada a la función de cálculo, en cuyo caso se calcula y se añade en este atributo.
- El centinela se pone a `false`.

Si por el contrario el código es lineal, para seguir siendo fieles a la idea del super dual, el código que guardamos tiene que ser su código dual, y los nuevos campos:

- La longitud es, de la misma manera que en los no lineales, el número de coordenadas de las palabras código.
- El kernel y los líderes son exactamente los que retorna la función `KernelZ2`, pero en este caso, la lista de líderes está vacía.
- La distancia mínima, en este caso sí que se calcula, ya que MAGMA implementa funciones eficientes de cálculo de distancia mínima para códigos lineales.
- El centinela se pone a `true`.

El segundo caso, en el que la entrada es una matriz $m \times n$, convertimos las filas de la matriz en una lista para tratarla exactamente de la misma forma que en el caso anterior, y una vez tenemos la lista de filas, generamos el código binario como en el primer caso.

El tercer caso, también es sencillo, ya que, al ser la entrada un subespacio vectorial, podemos afirmar que el código resultante será lineal y, por lo tanto, no es necesario calcular el kernel. Lo que hacemos es generar un código binario lineal a partir del subespacio, y rellenar los campos de la estructura de manera conveniente. Podríamos ver este caso como una conversión de un subespacio vectorial a nuestra estructura especial.

Hasta este caso, las construcciones eran más o menos fáciles de hacer, ya que apenas teníamos que contemplar casos especiales. Realmente, el camino a seguir en estos casos estaba muy bien definido. Pero cuando llegamos a los códigos como entrada del constructor, nos dimos cuenta de que los casos no eran tan triviales como habíamos pensado en un principio.

Podemos tener como entrada un código binario lineal, un código cuaternario lineal o un código $\mathbb{Z}_2\mathbb{Z}_4$-aditivo. Los dos primeros casos se pueden unir, ya que ambos son del tipo `Code` de MAGMA. Si el código es binario lineal, no tenemos muchos problemas, ya que simplemente miramos si el código tiene nuestra estructura, en cuyo caso, simplemente retornamos la entrada, o, por el contrario, es un código construido directamente con los constructores que nos ofrece MAGMA, lo cual quiere decir que el código es lineal y la construcción del código se reduce a realizar una conversión del tipo de datos `Code` a nuestra estructura.

Si el código es cuaternario, lo primero que tenemos que hacer es convertir las palabras código del alfabeto cuaternario al alfabeto binario a través del mapa de Gray (ver capítulo 2). Al hacer esta conversión, podemos encontrarnos con que un código que en cuaternario es lineal, al convertirlo a binario es no lineal. Para saber si la conversión va a resultar no lineal, disponemos de la función `HasLinearGrayMapImage`. Esta función nos dice si un código cuaternario tiene una imagen binaria lineal, y si la tiene, retorna una base de vectores que generan la imagen binaria lineal. En caso contrario, hay que llamar a la función `GrayMapImage`, que nos devolverá una lista de vectores. Estos vectores son todas las palabras código del código cuaternario, convertidas al alfabeto binario. Con esta lista, ya tenemos todo lo que necesitamos para llamar a la función de cálculo del kernel y empezar el proceso de construcción de un código binario (no lineal, en este caso) a partir de uno cuaternario.

Por último, el caso de los $\mathbb{Z}_2\mathbb{Z}_4$-aditivos, es un caso especial. Una librería para este tipo de códigos ha sido desarrollada en el *Grupo de Combinatoria y Codificación (CCG)* como otro proyecto del departamento [3]. Las

palabras que forman estos códigos tienen coordenadas en $\mathbb{Z}_2$ y en $\mathbb{Z}_4$. Su representación en MAGMA, en este caso, se hace mediante una estructura de datos que contiene todos los parámetros que se necesitan para tratar un código de este tipo. Uno de los campos de la estructura es el código $\mathbb{Z}_2\mathbb{Z}_4$-aditivo. Como tenemos coordenadas cuaternarias, tenemos que hacer también una conversión, como en el caso anterior. En este caso, contamos con funciones desarrolladas en [3]. El resultado de las funciones es similar a las que utilizamos en el caso de los códigos cuaternarios. Las funciones son `HasZ2Z4LinearGrayMapImage` y `Z2Z4GrayMapImage`. Estas dos funciones son una versión de las funciones anteriormente citadas `HasLinearGrayMapImage` y `GrayMapImage` adaptadas para trabajar con códigos $\mathbb{Z}_2\mathbb{Z}_4$-aditivos. Mediante la función `HasZ2Z4LinearGrayMapImage`, sabremos si existe una imagen binaria lineal a través del mapa de Gray y, si existe, obtendremos la base de vectores que la generan. Si no existe, tenemos que utilizar la función `Z2Z4GrayMapImage` para que nos retorne una lista con todas las palabras código del código $\mathbb{Z}_2\mathbb{Z}_4$-aditivo convertidas al alfabeto binario, y generamos nuestro código a partir de esta lista de vectores.

4.2.3. Paquete de códigos binarios en MAGMA

Una vez terminados los constructores, y siguiendo la estructura de MAGMA, teníamos que implementar funciones para trabajar con nuestros códigos. En esta línea, implementamos funciones clasificadas en cuatro tipos:

1. Invariantes de un código binario.
2. Operaciones sobre las palabras código.
3. Pertenencia e igualdad.
4. Propiedades de los códigos binarios.

Las invariantes de un código binario son parámetros del código que se crean al definirlo y no cambian aunque sometamos el código a transformaciones isomorfas. Para cada invariante, definimos una función que lo calcula:

- `BinaryLength(C)` nos retorna la longitud del código C.
- `BinaryCardinal(C)` retorna la cantidad de palabras que pertenecen a C.
- `BinaryMinimumDistance(C)` retorna la distancia mínima del código C. Si ya se ha calculado anteriormente, retornamos el valor del atributo que guarda la distancia mínima de C. Si no se ha calculado, se calcula y se guarda en el campo correspondiente.
- `BinaryRank(C)` retorna la dimensión del rango del código C.
- `BinaryKernelDimension(C)` retorna la dimensión del kernel del código C.
- `BinaryKernel(C)` retorna el kernel y los líderes de las clases de C.

Además de esto, la función `BinaryParameters(C)` nos devuelve una lista en la que encontramos la longitud, la cardinalidad y la distancia mínima de C.

En particular, la función `BinaryMinimumDistance(C)` que hay ahora en el paquete, se podría decir que es una segunda versión. La primera implementación miraba la distancia entre todas las palabras del código.

Esta segunda versión calcula la distancia mínima del kernel, aprovechando que es un código lineal. Una vez calculada esta distancia, se centra en los líderes de las clases. De esta forma, calculamos la distancia entre todas las palabras de una clase y todas las palabras de las siguientes clases, tratando el kernel como si fuera una clase más, es decir, como si fuera la primera clase. Así, nos ahorramos el cálculo manual de la distancia en el kernel y no repetimos cálculo de diferencias, ya que aplicamos $d(u, v) = d(v, u)$ (ver capítulo 2).

Las funciones del tipo operaciones sobre las palabras código, son funciones que o bien retornan una palabra código, o bien utilizan las palabras código como parámetro. Así tenemos:

- `BinaryRandom(C)` nos devuelve una palabra al azar del código C.

- `IsInBinaryCode(C,u)/IsNotInBinaryCode(C,u)` son dos funciones que nos dicen si la palabra u pertenece a C o no.

Estas funciones no nos costaron mucho. No fueron triviales, pero, la verdad es que el planteamiento que se hizo inicialmente fue muy acertado, así que las implementamos con bastante rapidez.

Los dos siguientes tipos son funciones cuyo resultado es `true` o `false`. En el caso de las funciones de pertenencia e igualdad, se utilizan para ver la relación que hay entre dos códigos de entrada. Las funciones son:

- `IsBinarySubset(C,D)/IsBinaryNotSubset(C,D)` nos dice si el código C está contenido en el código D.

- `IsBinaryEqual(C,D)/IsBinaryNotEqual(C,D)` nos dice si los códigos C y D son iguales.

Estas funciones han supuesto todo un reto a la hora de hacerlas correctamente y además de manera eficiente. Lo que en un principio nos pareció trivial, al final se complicó sobremanera, sobre todo en el caso de `IsBinarySubset(C)`(ver capítulo 2). Puesto que MAGMA dispone de funciones para decidir si un código es parte de otro, pensamos que sería algo muy fácil de implementar, pero nos dimos cuenta de que los resultados obtenidos al utilizar las funciones que nos ofrece el intérprete, no podían ser correctos, ya que nuestros códigos se construyen a partir del kernel y los líderes, y ésto es lo que tenemos que utilizar para estas funciones de decisión.

Por último, nos quedan las funciones contenidas en el tipo propiedades de los códigos binarios. Mediante estas funciones podemos saber otras características del código que no hemos calculado aún. Son las siguientes:

- `IsBinaryCode(C)` nos dice si un código está construido sobre el alfabeto $\mathbb{Z}_2 = \{0, 1\}$.

- `IsBinaryLinearCode(C)/IsZ2LinearCode(C)` nos dice si un código está construido sobre el alfabeto $\mathbb{Z}_2 = \{0, 1\}$ y si, además, es lineal.

- `IsBinaryPerfectCode(C)` nos dice si un código binario es perfecto.
- `IsBinaryExtendedPerfectCode(C)` nos dice si un código binario es perfecto extendido.
- `IsZ4LinearCode(C,p)` nos dice si un código binario C está construido a partir de un código cuaternario al que se le aplica una permutación p.
- `IsZ2Z4LinearCode(C,alpha,p)` nos dice si un código binario C tiene como base un código $\mathbb{Z}_2\mathbb{Z}_4$-aditivo, a partir del código C, una permutación p y $alpha$, que es un parámetro propio de los códigos $\mathbb{Z}_2\mathbb{Z}_4$-aditivos que nos dice cuantas coordenadas de la palabra código pertenecen al alfabeto $\mathbb{Z}_2$.

Exceptuando los dos primeros casos, estas funciones también nos han sorprendido por su complejidad (ver capítulo 2). Igual que nos pasó anteriormente, esperábamos implementarlas en mucho menos tiempo e invirtiendo menos esfuerzo del que al final fue necesario.

Así llegamos al final de la implementación del paquete software que nos habíamos planteado. Debido al poco tiempo que quedaba y teniendo en cuenta que había que testear este paquete y escribir la documentación, decidimos que dejaríamos en el tintero las funciones de creación de familias de códigos binarios particulares.

4.3. Testeo

Una vez creado el paquete, hay que testear su correcto funcionamiento. Hemos implementado un test para la función del kernel, y otro test para el paquete de MAGMA. A parte, también hemos hecho una pequeña función que muestra las mejoras de la función del kernel respecto de las anteriores versiones.

Para efectuar el test del kernel se genera un código binario no lineal a partir de un código $\mathbb{Z}_2\mathbb{Z}_4$-aditivo. El conjunto de palabras que forman el

código se da como entrada de las funciones de cálculo de kernel. Así, tenemos dos kernels calculados con dos funciones distintas cuyos resultados deberían ser iguales o, por lo menos, equivalentes, ya que el espacio generado por los dos kernels tiene que ser el mismo. Comprobamos la igualdad de los espacios generados y retornamos si ha habido éxito o no. Para comparar, utilizamos los resultados de la función desarrollada en [2], ya que calcula más rápido que la función implementada en [7] y [9], y los resultados son correctos.

Para testear el paquete, hemos hecho un test que pasa por todas las líneas de código implementado. En este caso, ha habido mucho trabajo, ya que había que contemplar muchísimos casos. La idea es darle a cada función una entrada con la que sabemos qué resultado vamos a obtener, es decir, para cada entrada, sabemos cuál va a ser su salida. De esta manera, se comprueba el correcto funcionamiento de todas las funciones del paquete en todos los casos.

Como a la hora de realizar el paquete dividimos las funciones en diferentes tipos, el testeo se hace respetando esos tipos. Es decir hacemos un test para los constructores, otro para las invariantes, etc. Así, la función de test recibe un parámetro que nos dice qué queremos testear. También está la posibilidad de testear todo el paquete. Por ejemplo, en el testeo de las propiedades de los códigos binarios, que es el test más sencillo, la función es la siguiente:

```
testProperties:=procedure()
  Q:=RandomLinearCode(IntegerRing(4),5,3);
  B:=LinearCode(VectorSpace(GF(2),5));
  //generamos un código binario lineal
  GM:=false;
  while (GM eq false) do
    BCL:=BinaryRandomCode(5,8);
    GM:=BCL`IsLinear;
  end while;
  //generamos un código binario no lineal
  GM:=true;
```

```
  while (GM) do
    BCNL:=BinaryRandomCode(5,8);
    GM:=BCNL‘IsLinear;
  end while;
  print "Result must be false-->",IsBinaryCode(Q);
  print "Result must be true-->",IsBinaryCode(B);
  print "Result must be true-->",IsBinaryCode(BCL);
  print "Result must be true-->",IsBinaryLinearCode(B);
  print "Result must be false-->",IsBinaryLinearCode(BCNL);
  print "Result must be true-->",IsZ2LinearCode(B);
  print "Result must be true-->",IsZ2LinearCode(BCL);
end procedure;
```

y la salida del test:

```
Testing Properties of Binary Codes
Result must be false--> false
Result must be true--> true
Result must be true--> true
Result must be true--> true
Result must be false--> false
Result must be true--> true
Result must be true--> true
```

Capítulo 5

Handbook of MAGMA functions

5.1. Introduction

MAGMA currently supports the basic facilities for codes over finite fields and codes over integer residue rings and galois rings, all that are linear codes (see [1, Chapters 127-130]). Therefore, MAGMA provides functions for the special case of binary linear codes, that is when the finite field is $GF(2)$, or equivalently the finite ring is $\mathbb{Z}_2$. This chapter describes functions which are applicable to binary codes not necessarily linear. A (n, M, d) *binary code* C is a subset of $\mathbb{Z}_2^n$ with cardinality M and minimum Hamming distance d. If C is not linear, then the zero word does not need to belong to C. If the zero word is not in C, considering a new binary code $C' = C + c$ for any $c \in C$, we can assure that the zero word is in the binary code C', which is equivalent to C.

In this chapter, the term "code" will refer to a binary code not necessarily linear such that it contains the zero word, unless otherwise specified.

Two structural properties of binary codes are the rank and kernel. The *rank* of a binary code C, $r = rank(C)$, is simply the dimension of the linear span, $\langle C \rangle$, of C. The *kernel* of a binary code C is defined as $K(C) = \{x \in \mathbb{Z}_2^n \mid x + C = C\}$. If the zero word is in C, then $K(C)$ is a linear subspace of C. We will denote the dimension of the kernel of C by $k = ker(C)$. In

general, C can be written as the union of cosets of $K(C)$:

$$C = \bigcup_{i=0}^{t} \Big(K(C) + c_i \Big),$$

where $c_0 = \mathbf{0}$ and $t+1 = M/2^k$. These parameters can be used to distinguish between non-equivalent binary codes, since equivalent ones have the same parameters r and k.

Let C be a binary code of length n such that the zero word belongs to C, with kernel $K(C)$ of dimension k and coset leaders $[c_1, \ldots, c_t]$. The *parity check system* of the binary code C is a $(n-k) \times (n+t)$ binary matrix $(G|S)$, where G is a generator matrix of the dual code $K(C)^{\perp}$ and $S = (G \cdot c_1 \; G \cdot c_2 \; \ldots \; G \cdot c_t)$. The *super dual* of the binary code C is the binary linear code generated by the parity check system $(G|S)$. Note that if C is a binary linear code, the super dual is the dual code $C^{\perp}$. From these definitions, we can establish the following properties (see [5]):

- Let $col(S)$ denote the set of columns of the matrix S.
  Then, $c \in C$ if and only if $G \cdot c = \mathbf{0}$ or $G \cdot c \in col(S)$.

- Let $r = rank(C)$ and $k = ker(C)$.
  Then, $r = n - dim(G) + dim(S)$ and $k = n - dim(G)$.

5.2. Construction of Binary Codes

`BinaryCode(L)`

Creates a binary code C given by its super dual, which is a binary linear code of length $n+t$ generated by the parity check system $(G|S)$ for the binary code C. As it is explained above, the parity check system $(G|S)$ is constructed using the kernel $K(C)$ and the coset leaders $[c_1, \ldots, c_t]$ of the binary code obtained by L, where:

1. L is a sequence of elements of $V = \mathbb{Z}_2^n$,
2. or, L is a subspace of $V = \mathbb{Z}_2^n$,
3. or, L is a $m \times n$ matrix A over the ring $\mathbb{Z}_2$,
4. or, L is a binary linear code,
5. or, L is a quaternary linear code,
6. or, L is a $\mathbb{Z}_2\mathbb{Z}_4$-additive code.

Note that in general, depending on the cardinality M of the binary code C, this function could take some time to compute $K(C)$ and $[c_1, \ldots, c_t]$, in order to return the binary code given by its super dual.

If L is a quaternary linear code or a $\mathbb{Z}_2\mathbb{Z}_4$-additive code, then the binary code corresponds to the image of L under the Gray map. If the zero word is not in L, then L is substituted by $L + c$, where c is the first element in L.

This constructor appends five attributes to the code category:

- `Length`: The length n of the binary code.
- `Kernel`: The kernel of the binary code as a binary linear subcode.
- `CosetLeaders`: The sequence of coset leaders $[c_1, \ldots, c_t]$.
- `MinimumDistance`: The minimum (Hamming) distance.
- `IsLinear`: It is `true` if and only if C is a binary linear code.

If L is a subspace of $V = \mathbb{Z}_2^n$ or a binary linear code, this function returns the dual code. In this case, the `Kernel` is identical to the binary linear code L defined by the function `LinearCode()`, the `Cosets` attribute is the empty sequence, the `Length` attribute is n and the `IsLinear` attribute is `true`.

`BinaryRandomCode(`n`,`M`)`

Given two positive integers n and M, return a random binary code of length n and cardinality M.

`BinaryRandomCode(`n`,`M`,`k`)`

Given three positive integers n, M and k, return a random binary code of length n, cardinality M and with a kernel of minimum dimension k.

Example H5E1

We can define a binary code by giving a sequence of elements of a vector space $V = \mathbb{Z}_2^n$, or a matrix over $\mathbb{Z}_2$. Note that C is the union of the kernel and the kernel added to the coset leaders.

```
> V := VectorSpace(GF(2),4);
> L := [V!0,V![1,0,0,0],V![0,1,0,1],V![1,1,1,1]];
> C1 := BinaryCode(L);
> C1;
[7, 4, 2] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 1 0 1]
[0 1 0 0 0 1 1]
[0 0 1 0 0 0 1]
[0 0 0 1 0 1 1]
> IsBinaryLinearCode(C1);
false

> A := Matrix(L);
> C2 := BinaryCode(A);
> C2;
[7, 4, 2] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 1 0 1]
[0 1 0 0 0 1 1]
[0 0 1 0 0 0 1]
[0 0 0 1 0 1 1]
> IsBinaryEqual(C1, C2);
true

> C3 := BinaryCode(sub<V|[[0,0,0,0],[0,0,1,1],[1,0,1,1]]>);
```

```
> C3;
[4, 2, 1] Linear Code over GF(2)
Generator matrix:
[0 1 0 0]
[0 0 1 1]
> IsBinaryLinearCode(C3);
true
```

Example H5E2

We can also construct random binary codes with a given length, number of codewords and, optionally, qith a given minimum kernel dimension.

```
> C1 := BinaryRandomCode(8,56);
> C1;
[14, 5, 4] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 1 0 1 0 1 0]
[0 1 0 0 0 0 1 0 1 0 1 0 0 0]
[0 0 1 0 0 0 1 1 1 0 1 0 0 1]
[0 0 0 1 0 1 1 1 0 1 1 1 1 1]
[0 0 0 0 1 0 0 1 0 0 0 1 1 1]
> (BinaryLength(C1) eq 8) and (BinaryCardinal(C1) eq 56);
true

> C2 := BinaryRandomCode(8,48,2);
> C2;
[19, 6, 6] Linear Code over GF(2)
Generator matrix:
[1 0 0 1 0 0 0 0 1 1 0 1 0 0 0 1 0 1 0]
[0 1 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0 0 1]
[0 0 1 1 0 0 0 0 0 1 0 0 0 1 0 0 0 1 1]
[0 0 0 0 1 0 0 0 0 0 1 1 1 1 0 1 1 1 1]
[0 0 0 0 0 1 0 1 0 0 0 0 1 1 0 0 1 1 1]
[0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 1 1 1]
> BinaryKernelDimension(C2) ge 2;
true
```

5.3. Invariants of a Binary Code

`BinaryLength(C)`

Given a binary code C, return the length of the code.

`BinaryCardinal(C)`

Given a binary code C, return the number of words belonging to C.

`BinaryMinimumDistance(C)`

Given a binary code C, return the minimum (Hamming) distance of the words belonging to the code C.

`BinaryParameters(C)`

Given a binary code C, return a list with the parameters $[n, M, d]$, where n is the length of the code, M the number of codewords and d the minimum (Hamming) distance.

`BinaryRank(C)`

Given a binary code C of length n, return its rank. The rank of a binary code C is the dimension of the linear span of C over $\mathbb{Z}_2$.

`BinaryKernel(C)`

Given a binary code C of length n, return its kernel as a binary linear code, and the leaders of the cosets as a list of binary vectors of length n. The kernel of a binary code C is the set of codewords c such that $c + C = C$.

`BinaryKernelDimension(C)`

Given a binary code C of length n, return the dimension of its kernel.

Example H5E3

Given a binary code C, we compute its length, number of codewords, minimum distance, rank and kernel dimension.

```
> C := BinaryRandomCode(10,64);
> C;
[17, 7, 4] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 1 0 0 0 1 1 1 1 0 1 1]
[0 1 0 0 0 0 0 0 1 1 0 1 1 0 1 0 0]
[0 0 1 0 0 0 1 0 1 1 1 1 0 0 1 1 0]
[0 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 1]
[0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 1 1]
[0 0 0 0 0 1 1 0 1 0 0 1 0 1 1 1 1]
[0 0 0 0 0 0 0 1 0 1 0 0 1 1 1 1 1]
> BinaryLength(C);
10
> BinaryCardinal(C);
64
> BinaryMinimumDistance(C);
1
> BinaryParameters(C);
[ 10, 64, 1 ]
> r := BinaryRank(C);
> r;
10
> k := BinaryKernelDimension(C);
> k;
3
> kernel,CosetLeaders := BinaryKernel(C);
> ((2^k)*(#CosetLeaders+1)) eq BinaryCardinal(C);
true
```

5.4. Operations on Codewords

`BinaryRandom(C)`

A random codeword of the binary code C.

`IsInBinaryCode(C, u)`

Return `true` if and only if the vector u of $V = \mathbb{Z}_2^n$ belongs to the binary code C of length n.

`IsNotInBinaryCode(C, u)`

Return `true` if and only if the vector u of $V = \mathbb{Z}_2^n$ does not belong to the binary code C of length n.

Example H5E4

```
> V := RSpace(IntegerRing(4), 5);
> C1 := Z2Z4AdditiveCode([V![2,2,1,1,3],V![0,2,1,2,1],V![2,2,2,2,2],V![2,0,1,1,1]] : Alpha
> C := BinaryCode(C1);
> C;
[8, 2, 4] Quasicyclic of degree 4 Linear Code over GF(2)
Generator matrix:
[1 1 0 1 0 0 1 0]
[0 0 1 1 0 0 1 1]
> c:=BinaryRandom(C);
> IsInBinaryCode(C,c);
true
```

5.5. Membership and equality

`IsBinarySubset(C, D)`

Return `true` if and only if the binary code C is a subcode of the binary code D.

`IsBinaryNotSubset(C, D)`

Return `true` if and only if the binary code C is not a subcode of the binary code D.

`IsBinaryEqual(C, D)`

Return `true` if and only if the binary codes C and D are equal.

`IsBinaryNotEqual(C, D)`

Return `true` if and only if the binary codes C and D are not equal.

Example H5E5

```
> V := VectorSpace(GF(2),5);
> C1 := BinaryCode([V![1,1,1,1,1],V![0,0,0,0,0],V![1,1,0,0,0],V![1,0,1,1,1]]);
> C2 := BinaryCode(Matrix([V![1,1,1,1,1],V![0,0,0,0,0],V![1,1,0,0,0],V![1,0,1,1,1]]));
> IsBinaryEqual(C1,C2);
true

> C3 := BinaryRandomCode(8,64,2);
> C4 := BinaryRandomCode(8,32);
> IsBinarySubset(C3,C4);
false

> C3 := BinaryCode(C4`Kernel);
> IsBinarySubset(C3,C4);
true
```

5.6. Properties of Binary Codes

`IsBinaryCode(C)`

Return `true` if and only if C is a binary code.

`IsBinaryLinearCode(C)`
`IsZ2LinearCode(C)`

Return `true` if and only if C is a binary linear code.

`IsZ4LinearCode(C,p)`

Given a binary code C of length n and a permutation p, return `true` if and only if the binary code $p(C) = \{p(c) : c \in C\}$ is the anti-image under the Gray map of a code over $\mathbb{Z}_4$, or equivalently, a quaternary linear code.

`IsZ2Z4LinearCode(C,`α`,p)`

Given a binary code C of length n, a positive integer α such that $\alpha \leq n$, and a permutation p, return `true` if and only if the binary code $p(C) = \{p(c) : c \in C\}$ is the anti-image under the Gray map of a $\mathbb{Z}_2\mathbb{Z}_4$-additive code over $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$, where $\beta = (n - \alpha)/2$.

`IsBinaryPerfectCode(C)`

Return `true` if and only if C is a binary perfect code.

`IsBinaryExtendedPerfectCode(C)`

Return `true` if and only if C is a binary extended perfect code.

Example H5E6

```
> V := VectorSpace(GF(2),3);
> C1 := BinaryCode([V!0,V![0,1,0],V![0,0,1]]);
> IsBinaryCode(C1);
true
> IsBinaryLinearCode(C1);
false
> C1`IsLinear;
false

> C2 := BinaryCode([V!0,V![1,1,1]]);
> BinaryMinimumDistance(C2);
3
> IsBinaryPerfectCode(C2);
true
> C3:=BinaryRandomCode(4,2);
> IsBinaryExtendedPerfectCode(C3);
true

> V := RSpace(IntegerRing(4),4);
> C := Z2Z4AdditiveCode([V![2,2,1,1],V![0,2,1,2],V![2,2,2,2],V![2,0,1,1]] : Alpha:=2);
> Cb := BinaryCode(C);
> Cb;
[6, 0, 6] Cyclic Linear Code over GF(2)
> S1:=Sym(Cb`Length);
> p1:=S1!(1,2);
> Q := RandomLinearCode(IntegerRing(4),5,2);
> Qb:=BinaryCode(Q);
> Qb;
[13, 8, 2] Linear Code over GF(2)
```

```
Generator matrix:
[1 0 0 0 0 0 0 1 0 0 1 0 1]
[0 1 0 0 0 0 0 1 0 0 0 0 0]
[0 0 1 0 0 0 0 1 0 1 0 0 1]
[0 0 0 1 0 0 0 1 0 1 1 1 1]
[0 0 0 0 1 0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0 0 1 0 1 1]
[0 0 0 0 0 0 1 1 0 0 1 0 1]
[0 0 0 0 0 0 0 0 1 1 0 1 1]
> S2:=Sym(Qb`Length);
> p2:=S2!(1,2);
> IsZ2Z4LinearCode(Cb,2,p1) and IsZ4LinearCode(Qb,p2);
true
```

Capítulo 6

Conclusiones y Resultados

Ahora veremos los resultados y conclusiones de este proyecto, así como qué sería interesante hacer, siguiendo en la linea que hemos marcado.

6.1. Conclusiones

En este proyecto hemos realizado los siguientes objetivos:

1. Hemos estudiado el cálculo del kernel.
2. Hemos implementado una función en MAGMA que calcula el kernel de un código binario.
3. Hemos estudiado el super dual.
4. Hemos implementado en MAGMA la función para calcular el super dual.
5. Hemos estudiado qué nos ofrece MAGMA a la hora de realizar nuestro paquete, y hemos decidido qué representación utilizar.
6. Hemos implementado en MAGMA un paquete completo de trabajo con códigos binarios no necesariamente lineales.
7. Hemos escrito la memoria del proyecto.

6.2. Resultados

Hemos implementado un paquete de software para MAGMA que sirve para trabajar con códigos no necesariamente lineales. MAGMA es un intérprete muy útil para trabajar en teoría de códigos, pero carece de tratamiento para cualquier tipo de códigos no lineales. Nuestro paquete permite al usuario trabajar con el caso concreto de los códigos binarios no lineales. Como ejemplo genérico, veamos un código binario no lineal construido con nuestro paquete:

```
> C := BinaryRandomCode(7,60);
> C;
[21, 5, 8] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 1 1 1 0 1 0 1 0 1 0 1 0 0 1 1]
[0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 1 0 1 1 1]
[0 0 1 0 0 0 1 0 0 1 1 0 1 0 0 1 1 0 0 0 1]
[0 0 0 1 0 1 0 0 0 0 0 1 1 0 0 0 0 1 1 1 1]
[0 0 0 0 1 0 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
> C‘IsLinear;
false
> C‘Kernel;
[7, 2, 2] Linear Code over GF(2)
Generator matrix:
[1 0 1 0 1 0 1]
[0 0 0 1 0 1 0]
> C‘CosetLeaders;
[
    (1 0 0 0 0 0 0),
    (1 1 0 0 0 0 0),
    (0 0 1 0 0 0 0),
    (1 0 1 0 0 0 0),
    (0 1 0 1 0 0 0),
```

```
    (1 0 1 1 0 0 0),
    (0 0 0 0 1 0 0),
    (1 1 0 0 1 0 0),
    (0 0 1 0 1 0 0),
    (1 1 1 0 1 0 0),
    (0 0 0 1 1 0 0),
    (0 1 0 1 1 0 0),
    (1 1 0 1 1 0 0),
    (1 1 1 1 1 0 0)
]
> BinaryCardinal(C);
60
> BinaryMinimumDistance(C);
1
> BinaryRandom(C);
(0 0 0 1 1 1 0)
> IsBinaryCode(C);
true
> IsZ2LinearCode(C);
false
```

6.3. Lineas futuras

Para continuar con este proyecto, hay varios aspectos que se podrían modificar para ampliar la funcionalidad que ofrece el paquete en este momento.

- Mejorar la función de cálculo del kernel. Como hemos visto en el capítulo 4, si seguimos el algoritmo en el que nos hemos basado para la función de cálculo del kernel, veremos que se implementan búsquedas binarias. Si cambiamos la estructura de datos que se utiliza en nuestra versión de la función de cálculo del kernel, podríamos mejorar su

eficiencia.

- Utilizar la función de cálculo del kernel para códigos $\mathbb{Z}_2\mathbb{Z}_4$-aditivos, desarrollada en [3]. En nuestro constructor de códigos binarios, cuando la entrada es un código $\mathbb{Z}_2\mathbb{Z}_4$-aditivo, el kernel se calcula con nuestra versión genérica. La versión específica para $\mathbb{Z}_2\mathbb{Z}_4$-aditivos es mucho más rápida para este tipo de códigos. No se ha unificado ya porque los dos proyectos se han realizado en paralelo.

- Calcular el kernel de un código cuaternario lineal con una función específica para este tipo de códigos. La función de cálculo del kernel para códigos cuaternarios está en desarrollo y, por este motivo, no se ha podido realizar esta mejora en nuestro proyecto.

- Implementar familias de códigos binarios no lineales, como los códigos Vasilev. En un principio, estos códigos se iban a implementar como parte del proyecto, pero debido a la dificultad sorprendente de algunas funciones, lo hemos dejado en el tintero.

- Mejorar la función de cálculo de la distancia mínima de un código binario representado a partir del super dual, aprovechando la división del código en clases.

- Aumentar los parámetros de la función que genera códigos aleatorios. En este momento, se necesitan la longitud, la cardinalidad y, opcionalmente, la dimensión del kernel. Sería interesante poder añadir el rango como parámetro opcional de la función. En este proyecto se intentó, pero el problema era muy complejo y no teníamos tiempo suficiente.

- Ampliar las funcionalidades del paquete. Existen funcionalidades secundarias que MAGMA ofrece para códigos binarios lineales, como la concatenación de códigos, la suma directa de dos códigos, la unión e intersección de dos códigos, etc. Se pueden añadir funciones al paquete que realicen estos cálculos para códigos binarios no necesariamente lineales teniendo en cuenta la estructura desarrollada en el proyecto.

Bibliografía

[1] J. J. Cannon and W. Bosma (Eds.),*Handbook of* MAGMA *Functions*,Edition 2.13, 4350 pages, 2006. (`http://magma.maths.usyd.edu.au/magma/`)

[2] F. Díez, *Invariants de matrius Hadamard en* MAGMA, Universidad Autónoma de Barcelona, 2007.

[3] B. Gastón, *Codis $\mathbb{Z}_2\mathbb{Z}_4$-additius en* MAGMA, Universidad Autónoma de Barcelona, 2008.

[4] B.Gastón, M. Villanueva, J.Pujol, "Development of algorithmic methods for binary non-linear codes in MAGMA.", en los Proceedings de las V Jornadas de Matemática discreta y Algorítmica, Lleida, Julio 2008.

[5] O. Heden, "On perfect p-ary codes of length $p + 1$.", *Designs, Codes and Cryptography*, vol.46, no. 1, pp. 45-56, 2008.

[6] F.J. Macwilliams, N.J.A. Sloane *The theory of Error-Correcting Codes.* North-Holland, New York, 1977.

[7] M.Pujol, *Rang i nucli de matrius Hadamard de mida $n = 2^t$ en* MAGMA, Universidad Autónoma de Barcelona, 2006.

[8] J.Rifà, LL. Huguet *Comunicación Digital: Teoría Matemática de la Información, Codificación Algebraica, Criptología.* Masson, Barcelona, 1991.

[9] L.Rodríguez, *Invariants de matrius Hadamard de mida* $n = 2^t \cdot s$ *en* MAGMA, Universidad Autónoma de Barcelona, 2006.

[10] M. Villanueva *On rank and kernel of perfect codes*, phD Thesis, Universidad Autónoma de Barcelona, 2001.

Firmado: Víctor Ovalle Arce

Bellaterra, Junio de 2008

Resum

La finalitat d'aquest projecte és aconseguir representar codis binaris no lineals de manera eficient en un ordinador. Per fer-ho, hem desenvolupat funcions per representar un codi binari a partir del super dual. Hem millorat la funció de càlcul del kernel d'un codi binari, implementada en projectes d'anys anteriors. També hem desenvolupat un paquet software per l'intèrpret MAGMA. Aquest paquet ens proveeix d'eines per al tractament de codis binaris no necessàriament lineals.

Resumen

La finalidad de este proyecto es conseguir representar códigos binarios no lineales de manera eficiente en un ordenador. Para ello, se han desarrollado funciones para representar un código binario a partir del super dual. Se ha mejorado la función de cálculo del kernel de un código binario, implementada en proyectos de años anteriores. También se ha desarrollado un paquete de software para el intérprete MAGMA. Este paquete nos provee de herramientas para el tratamiento de códigos binarios no necesariamente lineales.

Abstract

The purpose of this project is to be able to represent nonlinear binary codes in an efficient way in a computer. In order to do that, functions to represent a binary code from the super dual have been developed. The function that computes the kernel of a binari code, implemented in previous projects, has been improved. Moreover, a software package for the interpreter MAGMA has been developed. This package provides us with tools for the treatment of binary codes not necessarily linear.