Universidad de Valladolid

Escuela Técnica Superior
de Ingenieros de Telecomunicación

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Simulación en GPU de secuencias de resonancia magnética

Autor:
Cristina Mayo Sarmiento

Tutor:
Carlos Alberola López

Valladolid, 9 de Julio de 2018

Título:	**Simulación en GPU de secuencias de resonancia magnética**
Autor:	**Cristina Mayo Sarmiento**
Tutor:	**Carlos Alberola López**
Departamento:	**TSCIT**

TRIBUNAL

Presidente:	**Carlos Alberola López**
Secretario:	**Federico Simmross Wattenberg**
Vocal:	**Marcos Martín Fernández**
Suplente:	**Juan Pablo Casaseca de la Higurera**
Suplente:	**Rodrigo de Luis García**

Fecha:	**9 de Julio de 2018**
Calificación:	

RESUMEN

La Imagen de Resonancia Magnética (MRI) es una herramienta de diagnóstico fundamental en el campo de la medicina. Debido a su popularidad, en los últimos años se han desarrollado múltiples investigaciones para mejorar su rendimiento. En este contexto surgen numerosas técnicas conocidas como imagen en paralelo (PI, *Parallel Imaging*) que utilizan múltiples bobinas como receptores y permiten la reconstrucción de las imágenes a partir de un número menor de líneas de codificación de fase, haciendo uso de la información espacial de las bobinas. La principal ventaja que se consigue mediante el uso de este tipo de técnicas es la reducción de los tiempos de adquisición respecto a un sistema con una única bobina.

En este TFG se plantea la implementación de un simulador de síntesis de imágenes de resonancia magnética, con la finalidad de reconstruir imágenes utilizando el algoritmo de adquisición en paralelo SENSE (*SENSitiviy Encoding*). Para ello es necesario reproducir de la manera más realista posible el comportamiento de varias bobinas receptoras, así como los mapas de sensibilidad asociados a cada una de ellas. Como punto de partida de este proyecto se tomó un simulador de MRI con una única bobina, programado en C++ y OpenCL, de forma que se ha dotado de mayor funcionalidad a esta aplicación y se ha realizado una adaptación de los datos existentes, procedentes de la simulación de una bobina individual, para simular los que se obtendrían en el nuevo escenario con múltiples receptores.

Antes de comenzar el desarrollo de la aplicación es importante estudiar las características que tiene la simulación de imágenes. El problema más importante se encuentra en los elevados tiempos de ejecución, como consecuencia del elevado coste computacional que supone realizar múltiples operaciones para sintetizar y procesar cada uno de los puntos de las imágenes. Sin embargo, es sencillo aplicar el concepto de paralelismo para resolver estos inconvenientes.

En relación con lo anterior, la implementación que en el presente proyecto se propone, incorpora programación paralela, principalmente en GPU (Unidad de Procesamiento Gráfico) mediante la API (Interfaz de Programación de Aplicaciones) y el lenguaje de OpenCL, aunque también se ha utilizado paralelismo multihilo a nivel de CPU (Unidad de Procesamiento Central). Por otro lado, con el objetivo de obtener una aplicación cerrada, se ha incorporado una interfaz gráfica de usuario en MATLAB, de modo que facilite el manejo de la misma a usuarios de diversa índole.

Por último, respecto a los resultados obtenidos, se han validado a nivel cualitativo en forma de imágenes y cuantitativo con medidas de errores relativos. A su vez, con la intención de conseguir un rendimiento óptimo del simulador, se han comparado distintas implementaciones y algoritmos, demostrando la importancia de la computación heterogénea.

PALABRAS CLAVE

MRI, GPU, OpenCL, imagen en paralelo, bobina, mapa de sensibilidad, SENSE

ABSTRACT

Magnetic Resonance Imaging (MRI) is an essential medical diagnostic tool. Due to its popularity, in recent years extended research has been developed to improve its performance. In this context there are many techniques known as parallel image (PI, *Parallel Imaging*) by means of multiple coils as receivers. Image reconstruction is possible from a smaller number of phase encoding steps, making use of spatial coil information. The main advantage of using this type of techniques is the reduction of scan time versus single coil case.

A simulator of magnetic resonance imaging is proposed in this work in order to achieve image reconstruction using SENSE method. This will require simulate multiple receiver coils sensitivities and their behaviour as realistic as possible. As a starting point for this project, a single coil MRI simulator programmed in C++ and OpenCL was taken, so that, this application design to provide additional functionality and were modified to simulate new scenarios with multiple receiver, as well as their associated data.

First, before beginning development of application programm, a study of MRI simulation features is necessary. Worst problem lies in high run times as a result the multiple operations to compute and batch each voxel in the image to synthesize. However, parallelism concept may be used to easily solve this disadvantage .

Furthermore, code implementation proposed in this text includes parallel computing, mainly in GPU (Graphics Processing Unit) through the OpenCL API (Application Programming Interface) and programming language, although CPU (Central Processing Unit) multithreading programming is used too. Moreover, in order to provide a end-to-end application, a graphical user interface that supply the simulator with an user-friendly environment, was developed in MATLAB.

Finally, simulated results were validated both a qualitative and a quantitative stage by resulting images and relative error measurements. At the same time, different implementations and algorithms have been compared with the aim of achieving optimum performance of the simulator and showing the importance of heterogeneous computing.

KEYWORDS

MRI, GPU, OpenCL, parallel imaging, coil, sensitivity map, SENSE

Agradecimientos

En primer lugar, quiero expresar mi gratitud a todas aquellas personas de la Universidad de Valladolid y concretamente de la Escuela Técnica Superior de Ingenieros de Telecomunicación que me han ayudado en la realización de este Trabajo Fin de Grado y a lo largo de toda mi estancia en la Universidad. Concretamente me gustaría agradecer a mi tutor Carlos Alberola López por darme la posibilidad de realizar este Trabajo Fin de Grado.

Así mismo, agradecer al Laboratorio de Procesado de Imagen su amabilidad desde el primer momento. Muy especialmente me gustaría mencionar a Elisa Moya por su apoyo y ayuda diaria, ya que sin ella no habría conseguido realizar este trabajo.

Por último, dar las gracias a mis compañeros de Everis por su inestimable ayuda, a mis amigos y a mi familia, por su apoyo, consejos y la confianza depositada en mí para hacer posible este momento.

Índice general

Capítulo **1**

Introducción

La Imagen de Resonancia Magnética (MRI, *Magnetic Resonance Imaging*), es una técnica de imagen tomográfica que produce imágenes de las características físicas y químicas internas de un cuerpo u objeto bajo estudio, a partir de las señales de resonancia magnética nuclear (NMR, *Nuclear Magnetic Resonance*) medidas en el exterior del mismo. La formación de la imagen utilizando señales NMR es posible debido a los principios de codificación espacial de información. A pesar de que los fenómenos físicos en los que se basa fueron descubiertos durante la primera mitad del siglo XX, no se demostró hasta la década de 1970 la posibilidad de crear una imagen usando el fenómeno de resonancia magnética y fue unos años más tarde cuando comenzó a aplicarse al campo de la medicina [1, 2].

Debido a las propiedades y ventajas que presenta, la MRI se ha popularizado desde su origen hasta la actualidad, dando lugar a varias áreas de investigación en torno a la misma. En los últimos años, gran parte de la investigación en este ámbito se ha centrado en la reducción de los tiempos de adquisición. Dentro de este contexto, se han desarrollado numerosas técnicas conocidas como imagen en paralelo (PI, *Parallel Imaging*). Emplean sistemas compuestos por un conjunto de bobinas de radiofrecuencia dispuestas en un *array*. Estas estructuras permiten la reconstrucción de la imagen final a partir de una cantidad de datos menor, junto con la información espacial asociada a los distintos perfiles de sensibilidad espacial de los elementos receptores [3].

Existen numerosos métodos de adquisición en paralelo, centrándose este documento en SENSE (*SENSitivity Encoding*). Es un método que trabaja en el dominio de la imagen. Utiliza el conocimiento previo de las sensibilidades individuales de las bobinas y, junto con las imágenes con un campo de visión (FOV, *Field of view*) reducido recuperadas en cada bobina receptora, combina todo ello para reconstruir la imagen final. Por tanto, se puede reducir en un determinado factor el número de codificaciones de fase o líneas del espacio k que se adquieren. Haciendo uso de álgebra lineal, es posible deshacer el solapamiento o *aliasing* presente en las imágenes solapadas o *folded* recuperadas en las bobinas.

La principal ventaja de las técnicas de adquisición de la imagen en paralelo en general y de SENSE en particular, es la reducción de los tiempos de adquisición, y en contrapartida, destacar la degradación de la relación señal a ruido (SNR, *Signal to Noise Ratio*) de la imagen [4].

1.1 Motivación

La simulación de imágenes de resonancia magnética consiste en crear imágenes sintéticas, que mimeticen a las que se obtendrían en un aparato de resonancia real, a partir de los parámetros relevantes de los tejidos (véase capítulo 2) definidos mediante los correspondientes modelos anatómicos. Esto conlleva unas ventajas importantes en varios campos, entre los que se puede destacar la investigación (como una herramienta de ayuda para físicos e ingenieros) o con fines educativos.

En el ámbito de la investigación, el principal beneficio de los simuladores es la versatilidad que proporcionan y los distintos grados de complejidad que pueden añadirse al modelo utilizado para simplificarlo o intentar aproximarlo a la realidad. Además, permiten considerar distintos escenarios de reconstrucción, técnicas de adquisición de imagen en paralelo, como SENSE, o aplicar determinados efectos sobre las imágenes, tales como la presencia de artefactos, las inhomogeneidades del campo magnético o el impacto del ruido. Todo ello es posible con las mismas herramientas, añadiendo funcionalidad al simulador. Al crear estos fenómenos de forma simulada es fácil distinguir los efectos que cada uno de ellos provoca en las imágenes, lo cual, tiene su complejidad en la práctica. Por otro lado, permite probar nuevas secuencias de pulsos y algoritmos de reconstrucción, proporcionando una aproximación inicial de su comportamiento en la práctica.

Por otra parte, los equipos de resonancia magnética tienen un elevado coste y las listas de espera son un problema relevante en los sistemas nacionales de salud de los países avanzados. Así pues, estos equipos tienen una alta tasa de uso asistencial, de forma que el tiempo destinado con fines exclusivamente docente es prácticamente nulo. Debido a esto, otro campo muy distinto en el que los simuladores de estas imágenes tienen su utilidad, es como herramienta educativa para la formación de técnicos radiólogos y personal clínico.

1.1.1 Objetivos

El objetivo perseguido con la elaboración de este trabajo es la ampliación de un simulador de síntesis de imágenes de resonancia magnética con una única bobina [5]. Se pretende simular varias bobinas receptoras, sus mapas de sensibilidad asociados y la técnica de adquisición en paralelo SENSE. Partiendo del simulador inicial y de varios scripts en MATLAB, se emplearán técnicas de computación paralela, principalmente en GPU, haciendo uso del lenguaje de programación OpenCL. Adicionalmente se ha considerado el uso de paralelismo multihilo a nivel de CPU con OpenMP para la implementación de una función para la inversión de matrices complejas, como se detallará en el capítulo 4.

Este objetivo global puede verse desglosado en los siguientes subobjetivos:

- **Simulación de los mapas de sensibilidad de las bobinas receptoras con técnicas de computación paralela**, necesarios para aplicar el algoritmo de reconstrucción. Se pretenden reducir los tiempos de cómputo debido al paralelismo que presentan los cálculos asociados a cada bobina.
- **Simulación de las imágenes que se obtendrían en cada bobina a partir de un**

único espacio k aplicando, en su caso, los factores de submuestreo que proceda y empleando técnicas de computación paralela. Estas imágenes serán el punto de partida para la aplicación de SENSE, junto con los mapas de sensibilidad.

- **Implementación del algoritmo de reconstrucción SENSE mediante técnicas de computación heterogénea**, para obtener la imagen reconstruida, mediante la resolución y el planteamiento de múltiples sistemas lineales en paralelo, optimizando el tiempo de ejecución.
- **Ampliación y modificación de una GUI (*Graphical User Interface*) para la ejecución del simulador desde MATLAB**, mejorando la interacción del usuario con el simulador. Debe permitir seleccionar los modelos anatómicos de entrada, modificar parámetros de simulación, seleccionar el factor de aceleración y la disposición y configuración de las bobinas, entre otras funcionalidades. Además, se debe habilitar una región para visualizar la imagen y el espacio k simulados como punto de partida, varias imágenes con efectos de submuestreo en las bobinas y la imagen reconstruida final mediante SENSE. Para ello, previamente se ejecuta el código programado en C++ y OpenCL.

1.1.2 Fases y Métodos

Para alcanzar los objetivos planteados, las labores a desarrollar se clasifican en dos grandes etapas:

I. **Etapa de formación**, comprende el estudio de los conceptos básicos en varias áreas necesarias para la elaboración de este trabajo. Se pueden distinguir las siguientes fases:

   a) Estudio de los fundamentos de la imagen de resonancia magnética, así como de los fundamentos físicos en los que se basa y los principios de codificación de la información para la obtención de imágenes.

   b) Estudio de los principios básicos generales de las técnicas PI. Posteriormente, es necesario profundizar en el método de reconstrucción SENSE.

   c) Estudio de los principios de la computación paralela en GPU, concretamente del lenguaje de programación OpenCL.

   d) Análisis del simulador inicial programado en C++ y OpenCL al que se va a añadir funcionalidad.

   e) Estudio de los scripts en MATLAB para el cálculo de mapas de sensibilidad y reconstrucción SENSE, tomados como punto de partida.

II. **Etapa de diseño, implementación y validación**, compuesta por las siguientes fases:

   a) Programación y validación de los métodos necesarios para la simulación de las bobinas y los mapas de sensibilidad asociados a cada una de ellas.

   b) Implementación del algoritmo de reconstrucción SENSE utilizando librerías de C/C++ y OpenCL con el objetivo de priorizar los tiempos de ejecución mínimos.

   c) Análisis del problema y diseño del diagrama de clases utilizando el paradigma de programación orientada a objetos.

d) Integración de las nuevas funcionalidades en el simulador de partida, conociendo las limitaciones presentes e intentando solventarlas.

e) Ampliación de la interfaz gráfica (GUI, Matlab) de la que disponía el simulador inicial. Para lograr este propósito será necesario modificar sus vistas y añadir nuevas funciones. Integración con el código propio del nuevo simulador.

1.1.3 Medios

Será necesario, para la realización de este proyecto, el acceso a las herramientas *software* y *hardware* que se detallan a continuación:

Software

- MATLAB R2017b [6]: lenguaje de programación técnico de alto nivel y entorno de desarrollo integrado para el desarrollo de algoritmos, visualización y análisis de datos, y computación numérica.
- Eclipse MARS.2 [7]: entorno integrado de desarrollo de *software*.
- LaTeX[8]: sistema de composición de textos, orientado a la creación de documentos escritos.
- Astah Professional 7.2.0[9]: herramienta de diseño de sistemas que soporta UML, Diagrama de Relación de Entidades, diagramas de flujo, CRUD, Diagrama de flujo de datos, Tabla de Requisiciones y Mapas Mentales.

Hardware

- PC portátil con las siguientes características:
  - Procesador Intel® Core™ i7-3557U @ 2.00 GHz.
  - 4 GB de memoria RAM.
  - Disco duro de 500 GB de capacidad.
- PC de sobremesa con las siguientes características:
  - Procesador Intel® Core™ i7-7700 CPU @ 3.60GHz
  - 16 GB de memoria RAM.
- A partir de los equipos anteriores, se realiza una conexión remota a los servidores de cálculo disponibles en el Laboratorio de Procesado de Imagen (LPI) de la Universidad de Valladolid (UVA). Se ha utilizado el siguiente servidor:
  - Tebas, con las características:
    - Procesador Intel® Xeon® CPU E5-2697 v4 @ 2.30GHz
    - 500 GB de memoria RAM
    - GPU NVIDIA CUDA GeForce GTX 1070 con 8 GB de RAM

- Device Version OpenCL 1.2 CUDA
- Driver Version 384.111

En cuanto a los modelos anatómicos que se utilizarán como punto de partida para la simulación de las imágenes de resonancia magnética, se hará uso de los disponibles dentro el grupo en el que se realiza el trabajo, el LPI.

1.2 Estructura del documento

El documento se encuentra estructurado en 6 capítulos, entre los que se incluye el presente capítulo. Su contenido se describe brevemente a continuación:

Capítulo 1 - Introducción. Se expone la temática del documento, así como la motivación, los objetivos perseguidos, las fases y medios necesarios para alcanzarlos y la estructura del presente documento.

Capítulo 2 - Fundamentos teóricos previos. Se plantean los principios básicos de la adquisición de imágenes por resonancia magnética y los componentes fundamentales del escáner de MRI. Este capítulo se centra además en las técnicas de adquisición de imagen en paralelo (PI) y profundiza en el método conocido como SENSE, presentando con detalle la teoría subyacente al mismo y su aplicación mediante un ejemplo práctico. Finalmente, se explica la base teórica de la simulación de los mapas de sensibilidad de las bobinas que se ha empleado.

Capítulo 3 - Computación heterogénea CPU-GPU. Introduce el término de programación paralela y computación en GPU. El objetivo es aplicar estas técnicas para paralelizar los cálculos y algoritmos para lograr una disminución en los tiempos de computación. Se comienza describiendo los fundamentos del paralelismo y la computación heterogénea y en GPU, seguido de las ventajas e inconvenientes presentes y remarcando las diferencias existentes entre CPU y GPU. Finalmente se describe la API y el lenguaje de programación de OpenCL, incluyendo un ejemplo de la estructura de un programa en OpenCL.

Capítulo 4 - Diseño e implementación. Se presenta una visión global del diseño y la implementación realizados para la obtención del *software* resultante. Para ello, se detalla el diseño funcional y el diagrama de flujo del algoritmo. Se introduce el paradigma de programación orientada a objetos y se ilustra el diseño realizado para el simulador mediante su diagrama de clases, concretando más adelante la implementación realizada. Por último, se incorpora el diseño de la interfaz de usuario que permite la integración del núcleo de la simulación en una aplicación cerrada.

Capítulo 5 - Resultados. Se muestran los resultados obtenidos. En una primera sección, en cuanto a las imágenes simuladas, donde se realizará una comparativa de los efectos producidos en estas imágenes al variar algunos parámetros de la simulación tales como la configuración del *array* de bobinas y el factor de aceleración. Y en una segunda sección,

la comparativa de los tiempos de ejecución entre distintos métodos de resolución de los sistemas lineales para deshacer el solapamiento de las imágenes de SENSE.

Capítulo 6 - Conclusiones y líneas futuras. Este capítulo final recoge las principales conclusiones extraídas con la elaboración de este Trabajo Fin de Grado y se proponen posibles líneas de trabajo futuro manifiestas a partir de su realización.

Además de un capítulo en el que se recogen las fuentes bibliográficas y referencias consultadas para la redacción del presente documento, se añaden unos apéndices cuyo contenido se expone en las líneas siguientes:

Apéndice A - *Coordenadas cilíndricas*. Introduce el sistema de coordenadas cilíndricas y su relación con el sistema de coordenadas cartesiano.

Apéndice B - *Rotaciones básicas en 3 dimensiones*. Incluye las matrices de rotación que representan una rotación en el espacio tridimensional.

Apéndice C - *Diagonalización e inversión de matrices hermíticas*. Formula el desarrollo matemático para la inversión de una matriz hermítica a partir de la diagonalización de la misma mediante sus autovalores y autovectores.

Apéndice D - *Estructura del código del simulador*. Incluye una descripción detallada de los ficheros de los que se compone la aplicación, así como de su contenido.

Capítulo 2

Fundamentos teóricos previos

Este capítulo se divide en tres partes, la primera de ellas enuncia los fundamentos de la MRI, explica el principio físico en el que se basa, así como el proceso de codificación de la información para la formación de las imágenes. Por otro lado, se describen los dispositivos que hacen posible la generación de imágenes y sus componentes. Finalmente, en la última sección se abordan las técnicas de imagen en paralelo y se profundiza en el método de reconstrucción SENSE.

2.1 Fundamentos de la MRI

La MRI se engloba dentro de la tomografía, una importante área en auge en el campo de la imagen médica. Se centra en la obtención de imágenes de cortes o secciones de algún objeto. La tomografía comprende varios tipos de imágenes, a parte de la MRI, como los rayos X-CT (*Computer Tomography*), PET (*Positron Emission Tomography*), SPECT(*Single Photon Emission Computed Tomography*), MEG (Magnetoencefalograma), SAR o sistemas de imagen acústicos. Aunque estas técnicas usan distintos principios físicos para la generación y detección de señales, los principios de procesamiento de las señales para la formación de las imágenes son, a grandes rasgos, los mismos.

Se pueden destacar importantes ventajas de la MRI sobre otras técnicas, como PET o SPECT, que también utilizan señales NMR provenientes del objeto (emisión tomográfica) para la formación de las imágenes. Principalmente se puede señalar que en este caso no se necesita inyección de isótopos en el objeto para generar la señal de MRI. Además, opera en el rango de la radiofrecuencia (RF), de forma que no utiliza radiación ionizante asociada a efectos dañinos sobre el paciente. Otros puntos positivos son la calidad y versatilidad de las imágenes obtenidas, ya que debido al exclusivo esquema de imágenes utilizado, la resolución espacial resultante de la MRI no está limitada por el rango de frecuencia de sondeo, proporcionando como resultado, imágenes extremadamente ricas en contenido e información, con excelente contraste para tejidos blandos. Asimismo, es posible obtener imágenes del mismo sitio anatómico muy diferentes, en función del protocolo de adquisición de datos. Sin duda, la flexibilidad en la adquisición de datos y los múltiples mecanismos para el contraste han hecho que esta técnica sea cada vez más popular en multitud de diagnósticos.

En general, la imagen MRI puede ser un mapa espacial de la densidad de espines estacionarios, espines en movimiento, tiempos de relajación o coeficientes de difusión del agua [1, 10].

2.1.1 Principios físicos

El momento angular de las partículas y subpartículas atómicas se conoce como espín. El número de protones de un átomo viene dado por su número atómico. Si dicho número es par, los espines se cancelan; pero en caso contrario, el núcleo del átomo distingue por su momento angular $\vec{J}$. Un núcleo cuyo espín no es nulo, va a crear un campo magnético caracterizado por un momento magnético $\vec{\mu}$ y, como enuncian las leyes físicas, un campo magnético variable induce una corriente eléctrica, hecho que será relevante y se detallará posteriormente. Aunque existen varios núcleos atómicos con esta característica, los núcleos de hidrógeno (^1H), compuestos por un único protón, tienen especial interés desde el punto de vista de la NMR, ya que están presentes en abundancia en forma de H_2O en los tejidos biológicos.

Si se tiene un conjunto de núcleos del mismo tipo que forman un sistema de espines, en ausencia de campo magnético externo, los momentos de los átomos están orientados aleatoriamente y no aparece campo magnético neto. Sin embargo, si se aplica un campo magnético estático $\vec{B_0}$, los espines se alinean en la dirección del campo y aparece una magnetización $\vec{M}$ paralela al mismo. Este efecto aparece representado en la figura 2.1.

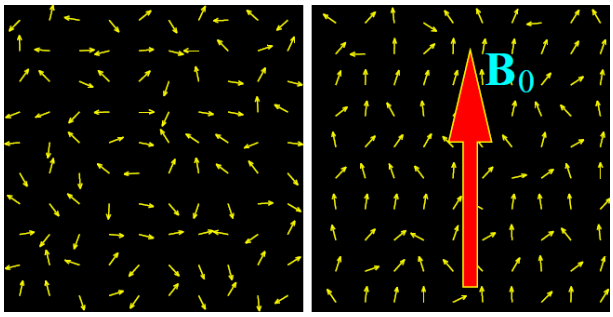


Figura 2.1: Alineamiento de los espines bajo un campo magnético $\vec{B_0}$

Además, como se puede observar en la figura 2.2, el momento magnético $\vec{\mu}$ de cada espín realiza un movimiento de precesión alrededor de $\vec{M}$, a la frecuencia de resonancia o frecuencia de Larmor w_0, donde γ es la constante giromagnética y su valor es una propiedad del núcleo.

$$w_0 = -\gamma \cdot \vec{B_0} \tag{2.1}$$

Hay que tener en cuenta las dos posibles orientaciones de los núcleos de hidrógeno sometidos a un campo magnético estático y, por tanto, de sus momentos magnéticos. La orientación paralela conlleva una menor energía que la antiparalela, por lo que habrá un mayor número de núcleos rotando con esta orientación. Como consecuencia, el vector de magnetización neta del sistema de espines ya no será nulo.

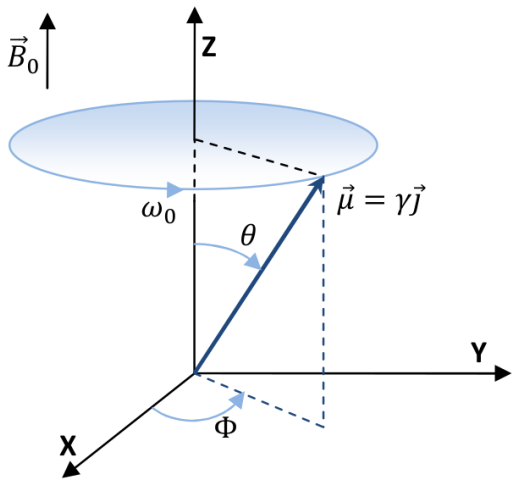


Figura 2.2: Movimiento de precesión de un espín sometido a un campo magnético estático $\vec{B_0}$

Por simplicidad, consideraremos el campo $\vec{B_0}$ orientado en la dirección del eje z. Como se ha descrito anteriormente, el vector de magnetización $\vec{M}$ es paralelo al campo magnético, de forma que solo tendrá componente longitudinal M_z. La razón subyacente es que la precesión de los momentos magnéticos tienen fases aleatorias.

Si en esta situación se aplica un segundo campo magnético $\vec{B_1(t)}$, en la dirección transversal XY, oscilando a la frecuencia de Larmor, los momentos magnéticos pasan del estado de baja energía a alta energía y se reduce la componente longitudinal del vector de magnetización, M_z. Adicionalmente, este campo produce la coherencia en fase entre los espines, apareciendo así una componente transversal M_{xy} rotando en torno al eje z a la frecuencia de Larmor. La frecuencia de precesión de los núcleos de hidrógeno bajo la presencia de un campo magnético estático de $B_0 = 3T$ es de aproximadamente 128 MHz, dentro del rango de la radiofrecuencia (RF), por lo que este campo se conoce normalmente como pulso de RF.

La aplicación de este segundo campo provocará que el vector de magnetización neta del sistema de espines $\vec{M}$ se mueva respecto a la posición de equilibro, paralela a $\vec{B_0}$, siguiendo una trayectoria espiral. Aparecerá un campo magnético $\vec{B_{eff}}$ alineado con el plano XY. Los espines se alinearán con él y formarán un ángulo α (configurable según la secuencia de pulsos a utilizar) con $\vec{B_0}$. En el momento en el que desaparece $\vec{B_1(t)}$, los espines liberan la energía absorbida, en un proceso conocido como relajación, y vuelven a colocarse paralelos al campo magnético estático $\vec{B_0}$. Esta energía liberada es captada por las bobinas receptoras en forma de voltaje.

El tiempo que tarda la componente M_z en volver a su estado original se llama tiempo de relajación longitudinal T_1. De la misma forma, el tiempo de relajación de la componente transversal M_{xy}, se denota por T_2. Estos parámetros, junto con la densidad de protones, son propios de cada tejido y el valor de los píxeles de la imagen de resonancia magnética depende

fundamentalmente de ellos [1, 11, 12].

2.1.2 Adquisición de la imagen de MR

Una vez se ha especificado cómo se puede generar la señal de MR de un objeto, se va a exponer el procesamiento de esta.

En primer lugar, para formar la imagen son necesarias las siguientes etapas:

1. Generación de señal (descrito en el apartado 2.1.1)
2. Detección de señal
3. Manipulación de la señal una vez se ha detectado.
4. Procesamiento

Esquemáticamente el proceso es el siguiente:

$$\vec{\mu} \rightarrow \vec{M} \rightarrow \vec{M_{xy}} \rightarrow S(t) \rightarrow S(\vec{k}) \rightarrow I(\vec{x}), \tag{2.2}$$

donde $\vec{\mu}$ son los momentos magnéticos microscópicos en un objeto, $\vec{M}$ el vector de magnetización neta, $\vec{M_{xy}}$ la magnetización transversal, $S(t)$ la señal eléctrica en las bobinas receptoras, $S(\vec{k})$ la señal del espacio k y finalmente, $I(\vec{x})$ la imagen deseada.

Los dos primeros pasos del proceso hacen referencia a la generación de la señal de MR. El primer paso, $\vec{\mu} \rightarrow \vec{M}$, se logra exponiendo el objeto al campo $\vec{B_0}$ y el segundo con las excitaciones de RF, como se ha explicado en detalle en el apartado 2.1.1. El tercero se corresponde con la detección de señal y se basa en la conocida Ley de inducción de Faraday. El cuarto es el núcleo de la MRI, el cual involucra el uso de los gradientes para la codificación espacial de la información de las respuestas transitorias de los espines, debido a las excitaciones RF. Finalmente, el último ataña el problema de reconstrucción de la imagen, común a muchas otras técnicas de imagen tomográfica.

Sobre las ideas expuestas, para poder reconstruir una imagen es necesario localizar el origen espacial de cada señal, para lo que se utilizan diferentes gradientes de campo magnético. Primero es necesario seleccionar un *slice* del objeto. Para ello se utiliza un gradiente que modifique la frecuencia de resonancia de los espines en función de la posición. Posteriormente, se utilizan dos métodos de codificación espacial:

- **Codificación de frecuencia:** se aplican gradientes para que la frecuencia de precesión de los espines del *slice* excitado dependa linealmente de la codificación espacial. De esta forma se codifica la información del eje x.
- **Codificación de fase:** respecto al eje y, se consigue que el vector de magnetización posea un desfase dependiente de la posición, aplicando un gradiente durante un periodo corto de tiempo en esta dirección [1, 12].

2.2 Escáner MRI

La generación de imágenes de MRI se lleva a cabo con un escáner. Las salidas de la mayoría de los dispositivos de este tipo y del escáner de MRI, es un array de datos multidimensional (o imagen). Sin embargo, presenta una importante diferencia frente a otros dispositivos de imagen tomográfica, ya que en este caso se pueden generar imágenes bidimensionales de una sección en cualquier orientación, imágenes tridimensionales o en cuatro dimensiones representando distribuciones espaciales-espectrales, sin ajustes mecánicos en el escáner.

Está formado por tres componentes *hardware* principales:

- **Imán principal**: genera un fuerte campo magnético estático y uniforme conocido como $\vec{B_0}$. Se utiliza para la polarización de los espines (ver apartado 2.1.1). Cuanto mayor sea la intensidad del campo mejor es la relación señal a ruido y la resolución espectral, pero, en contrapartida, ocurren problemas relacionados con la penetración de RF y supone mayor coste. Los valores típicos para un sistema clínico de cuerpo entero oscilan entre 0.5 y 2 T. Además de la intensidad, es importante la homogeneidad del campo generado para obtener buena calidad de imagen. En la práctica se usan bobinas de compensación para tener la homogeneidad deseada.

- **Sistema de gradientes:** normalmente son tres bobinas de gradientes ortogonales. Estas bobinas están diseñadas para producir campos magnéticos que varían con el tiempo, controlados por una no uniformidad espacial. Es esencial para la localización de señales. Típicamente un tiempo de subida de 1.0 ms para 10 mT/m son valores adecuados para métodos de adquisición de imagen convencionales.

- **Sistema de RF:** se compone de una bobina transmisora capaz de generar un campo magnético de rotación, llamado $\vec{B_1}$, para la excitación de los espines; y una bobina receptora que convierte una magnetización de precesión en una señal eléctrica. Se puede utilizar una única bobina como transmisora y receptora, llamada bobina transceptora. Ambas bobinas, tanto transmisora como receptora, se suelen llamar bobinas de RF porque resuenan en este rango de frecuencias, como requiere la excitación de los espines y la detección de señales. Se desea que los componentes RF proporcionen un campo $\vec{B_1}$ uniforme y una alta sensibilidad de detección. Para ello, un sistema MR está equipado con bobinas RF de varias formas y tamaños para distintas aplicaciones [1].

2.3 Imagen en paralelo (PI)

La formación de imágenes de MRI se basa en recorrer el espacio k en dos o tres dimensiones de la manera determinada por la secuencia de pulsos. Aunque la adquisición de datos en la dirección de codificación de frecuencia suele ser rápida y del orden de varios milisegundos, se necesita un eco, con un valor ligeramente diferente al del gradiente de codificación de fase aplicado para muestrear cada valor de k_y a lo largo del eje de codificación de fase. Por lo tanto, el número de líneas del espacio k representa la mayor parte del tiempo de adquisición en la mayoría de las adquisiciones de imágenes de MR. Dentro de este contexto junto con la introducción de sistemas con múltiples receptores, conocidos como bobinas en *array* o *phased-array*, surgen las técnicas

de *Parallel Imaging* (PI), con las que se ha conseguido una reducción significativa en los tiempos de adquisición.

Fundamentalmente, las técnicas PI persiguen el objetivo de reducir el tiempo de adquisición mediante el submuestreo del espacio k y la obtención simultánea de imágenes a través de múltiples bobinas. El submuestreo reduce los tiempos, mientras que el uso de las bobinas permite la reconstrucción de la imagen final. Se basan en el hecho de que el tiempo de adquisición es proporcional al número de líneas de codificación de fase o líneas del espacio k en una adquisición Cartesiana. Generalmente, el número de líneas de codificación de fase muestreadas se reduce mediante un factor de aceleración o reducción R, acortando así el tiempo de adquisición por el mismo factor. En contrapartida, se puede comprobar que la SNR se reducirá en un factor $\sqrt{R}$. Por ejemplo, para $R = 2$, se muestrea solo la mitad del espacio k. Como resultado, el tiempo de adquisición se acorta en un factor de dos. En el caso de $R = 3$, solo se muestrea un tercio del espacio k y el tiempo de adquisición se reduce al triple. Como se ha comentado anteriormente, hay que tener en cuenta que la información omitida en el proceso de codificación es capaz de obtenerse gracias a la dependencia espacial de los elementos del *array* de bobinas.

A continuación, se exponen dos ejemplos de adquisición parcial del espacio k para comprender las consecuencias que tienen en las imágenes. En la figura 2.3 a), se representa el espacio k completamente muestreado y la imagen asociada al mismo, con FOV completo y alta resolución, que sirve de base para analizar las siguientes. En la figura adyacente, 2.3 b), se observa qué ocurre si el espaciado entre líneas del espacio k se mantiene, pero disminuyen las frecuencias espaciales máximas. En este caso, se obtiene una imagen con FOV completo, pero con una resolución menor. Por último, en la figura 2.3 c), se refleja cómo al incrementar la distancia entre líneas del espacio k, pero manteniendo las frecuencias espaciales máximas fijas, se reduce el FOV de la imagen manteniendo su resolución. Este caso se va a desarrollar más adelante con detalle, pues es la base de SENSE.

Por otro lado, cabe destacar la existencia de numerosas técnicas dentro de PI, que se pueden clasificar de acuerdo con dos métodos según el dominio en el que se realice la reconstrucción:

- **Dominio espectral:** las líneas del espacio k no adquiridas se reconstruyen antes de realizar la conversión al espacio imagen mediante la transformada de Fourier (TF). Las técnicas más populares son: SMASH (*Simultaneous acquisition of spatial harmonics*) y GRAPPA(*Generalized autocalibrating partially parallel acquisition*)

- **Dominio imagen:** el *aliasing* se elimina en el espacio imagen, después de haber realizado la TF. Dentro de este dominio se encuentran técnicas como SENSE (*SENSitivity Encoding*).

Al mismo tiempo, algunos de estos métodos necesitan un conocimiento de los perfiles de sensibilidad de las bobinas, pero otros, los denominados métodos autocalibrados, estiman dichos perfiles mediante la adquisición de líneas espectrales adicionales. Destacan SENSE, mSENSE (versión de SENSE autocalibrada) y GRAPPA como los más usados [3, 13, 14].

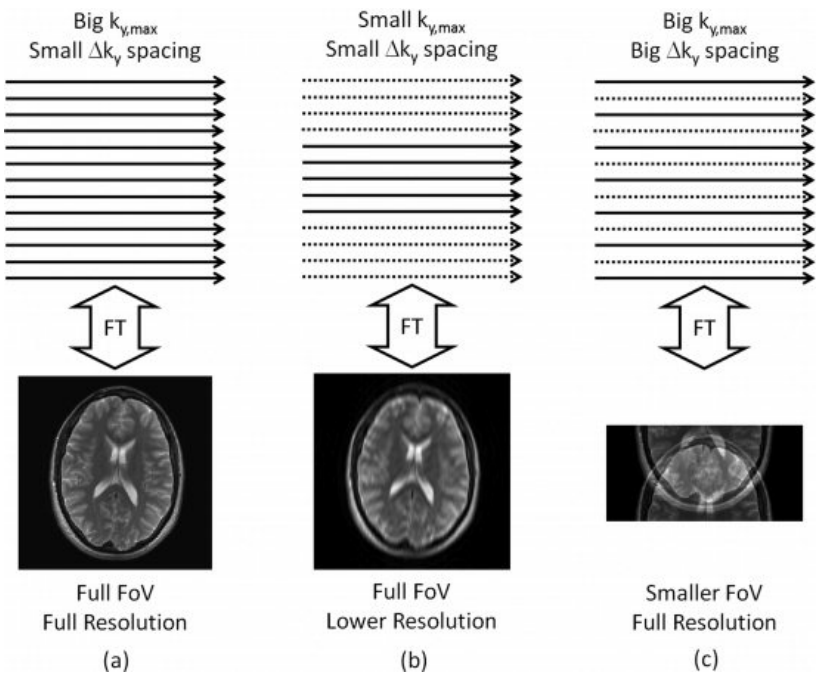


Figura 2.3: Diferentes comportamientos del espacio k y la imagen en función de la frecuencia espacial máxima y el espaciado entre líneas del espacio k

2.3.1 SENSitivity Enconding (SENSE)

SENSE es un método PI basado en el dominio de la imagen y su objetivo es la reconstrucción de una imagen final a partir de las imágenes con efectos de submuestreo (*folded*) recuperadas en cada bobina y la información contenida en los mapas de sensibilidad.

Dos parámetros son de particular interés cuando se implementa SENSE. En primer lugar, el rango de valores muestreados a lo largo del eje de codificación de fase k_y, siendo el máximo $k_{y_{max}}$, que determinará la resolución espacial a lo largo de la dirección de codificación de fase (ver figura 2.3). A medida que aumenta el número total de líneas muestreadas, N_y, se adquieren frecuencias espaciales del espacio k más altas, lo que da como resultado una resolución espacial mejorada a lo largo del eje de codificación de fase. En segundo lugar, la separación entre líneas sucesivas a lo largo del eje k_y, Δk_y. Este parámetro va a determinar el FOV correspondiente, FOV_y, en el dominio de la imagen. Cuando se aplica el algoritmo SENSE para reducir el tiempo de adquisición, el número de líneas de codificación de fase muestreadas N_y se reduce en R, teniendo N_y/R líneas en el espacio k, mientras que los valores máximos muestreados de k_y ($k_{y_{max}}$) no cambian, como se representó en la figura 2.3 c). Este enfoque permite que la imagen adquirida por SENSE tenga la misma resolución espacial que la adquisición de referencia, pero a expensas de un aumento en Δk_y. Debido a la relación inversa entre FOV_y y Δk_y, el campo de visión resultante a lo largo del eje de codificación de fase en la adquisición SENSE también se reduce por R:

$$\Delta k_y = 1/FOV_y \tag{2.3}$$

La reducción consiguiente en la FOV_y con la técnica SENSE típicamente introduce artefactos de

aliasing o *wraparound* en las imágenes que se obtienen al calcular la TF de los datos del espacio k adquiridos por cada bobina, como se muestra en la figura 2.3 c). El algoritmo de reconstrucción SENSE soluciona este problema (múltiples imágenes *folded*, una por cada bobina), utilizando información espacial conocida de las bobinas (mapas de sensibilidad), para obtener una imagen con el FOV original. La intensidad de la señal en un punto particular varía en función de la distancia desde la bobina receptora. En teoría, si se puede obtener el perfil de sensibilidad de cada elemento del *array*, entonces las diferencias de intensidad de señal entre cada elemento en cualquier píxel dado en una imagen pueden estar relacionadas con la ubicación espacial de ese punto. Esta relación, a su vez, se usa en la técnica SENSE para desplegar y eliminar los artefactos de las imágenes adquiridas con SENSE.

El esquema principal de reconstrucción de SENSE está representado en la figura 2.4. Se ha tomado como ejemplo 4 bobinas receptoras y un factor de aceleración 2. Como se puede observar, cada bobina de manera individual tiene sus datos del espacio k, de los que solo se han adquirido la mitad de las líneas de codificación de fase, debido a que en este ejemplo $R = 2$. Aplicando la TF a cada conjunto de datos de manera individual, se tienen tantas imágenes como bobinas (4 en este caso). Como se ha submuestreado el espacio k, estas imágenes presentan *aliasing* y tienen un FOV reducido en el eje y. Posteriormente, gracias a los mapas de sensibilidad de las bobinas previamente calibrados, es posible combinar las imágenes anteriores con las ponderaciones adecuadas para llevar a cabo la reconstrucción y obtener la imagen sin efectos de submuestreo.

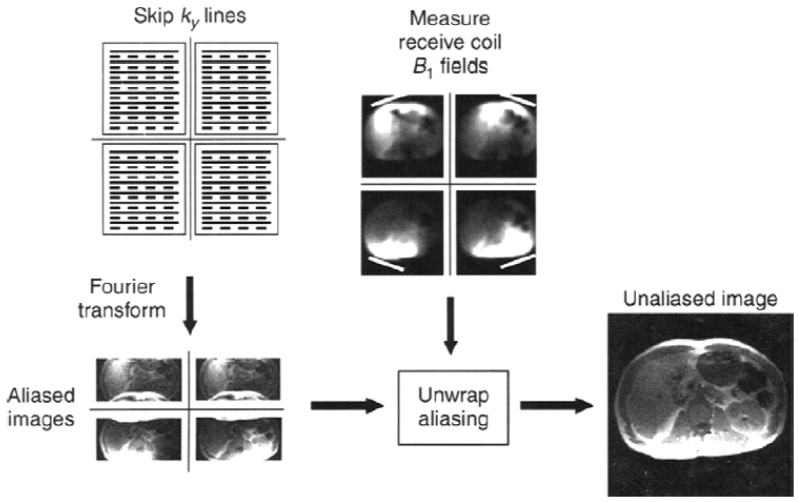


Figura 2.4: Representación esquemática de reconstrucción SENSE para $n_C = 4$ y $R = 2$

De acuerdo con el algoritmo SENSE, el factor de aceleración (R) puede alcanzar valores, teóricamente, hasta la cantidad de elementos empleados en el *array* de receptores (n_C), es decir, se debe cumplir que:

$$R \leq n_c, \tag{2.4}$$

En la práctica, sin embargo, la mayoría de los proveedores de imágenes de MR usan valores de

R que van de 1.5 a 3, mientras que la cantidad de bobinas varía de 4 a 12 [3].

2.3.1.1 Teoría y métodos

En la teoría general del algoritmo de reconstrucción de SENSE no hay restricciones en la configuración de las bobinas ni en el patrón de muestreo del espacio k. Además, se discuten dos estrategias de reconstrucción:

- **Reconstrucción fuerte o *strong reconstruction***: busca una forma de vóxel óptima.
- **Reconstrucción débil o *weak reconstruction***: el criterio de la forma del vóxel es más débil, a favor de la SNR.

Con ambas estrategias, el algoritmo de reconstrucción es numéricamente exigente en el caso general. Esto se debe principalmente a que, con la codificación híbrida, la mayor parte del trabajo de reconstrucción generalmente no se puede hacer mediante la transformada rápida de Fourier (FFT). Aunque se puede demostrar que en la reconstrucción débil, la FFT se puede aplicar al espacio k si se muestrea de manera regular Cartesiana. Por esta razón, la codificación de la sensibilidad con muestreo cartesiano es particularmente factible. Además, el mecanismo de reconstrucción es relativamente sencillo para este caso y es el que se va a implementar en este TFG.

Una vez explicado en qué consiste SENSE, se va a desarrollar matemáticamente. Como se ha enunciado anteriormente, la reconstrucción propiamente dicha consiste en crear una imagen con FOV completo a partir del conjunto de imágenes intermedias. Para lograr esto, se debe deshacer la superposición de las señales subyacente al efecto de plegado. Es decir, para cada píxel en el FOV reducido, las contribuciones de señal de varias posiciones en el FOV completo deben separarse. La clave para la separación de la señal radica en el hecho de que, para cada bobina, la superposición que se da en la señal de imagen ocurre con diferentes pesos, de acuerdo con las sensibilidades de la bobina local. Por tanto, el valor de un píxel en la imagen con FOV reducido en la bobina j viene dado por:

$$F_j = \sum_{l=0}^{R-1} C_{jl} I_l, \qquad j = 0, ..., n_C \tag{2.5}$$

siendo C_j los valores de la sensibilidad de la bobina j en las R coordenadas de los R píxeles equiespaciados e I los valores de los R píxeles en la imagen con FOV completo, que van a superponerse en un único píxel en la imagen con *aliasing*. Se puede reescribir el sistema anterior en forma matricial con las dimensiones de cada elemento:

$$F_{(n_C x 1)} = C_{(n_C x R)} \cdot I_{(Rx1)} \tag{2.6}$$

Este sistema se puede resolver si se cumple la condición de la ecuación 2.4, con SNR óptima, aplicando mínimos cuadrados:

$$\hat{I} = [(C^H \Psi^{-1} C)^{-1} C^H \Psi^{-1}] F, \tag{2.7}$$

donde el superíndice H indica transpuesto conjugado y Ψ es la matriz de correlación ruido de las bobinas. Si se repite este proceso para todos los píxeles del FOV reducido se obtiene la imagen

de FOV completo sin *aliasing*. En caso de que se requiera optimizar la SNR, hay que considerar los niveles de ruido y la correlación de las bobinas. Si no se desea realizar esta optimización basta con sustituir Ψ por la matriz identidad. La reconstrucción es posible de igual manera, pero con SNR no óptima.

De este modo, es posible definir la matriz de *unfolding* o reconstrucción que permite deshacer el plegado de las imágenes intermedias como:

$$U = (C^H \Psi^{-1} C)^{-1} C^H \Psi^{-1}, \tag{2.8}$$

Haciendo uso de la matriz anterior se simplifica la notación de la reconstrucción:

$$\hat{I} = U \cdot F \tag{2.9}$$

En cuanto a la relación señal a ruido en SENSE, se ve afectada en primer lugar por una disminución en un factor $\sqrt{R}$, pero además, también va a depender de la configuración específica del sistema de bobinas, su número, disposición... Estos aspectos se modelan mediante un parámetro o factor de geometría g, de manera que la SNR será:

$$SNR_{SENSE} = \frac{SNR_{full}}{\sqrt{R} \cdot g} \tag{2.10}$$

La principal desventaja de este método es que se necesita un conocimiento preciso de la sensibilidad de las bobinas. Por ello, en situaciones de SNR baja o movilidad del objeto bajo estudio esta técnica no será la más adecuada [4, 13].

2.3.1.2 Ejemplo básico de reconstrucción

Para clarificar el desarrollo previo, se va a explicar el siguiente ejemplo de reconstrucción con SENSE basado en la figura 2.5. En la parte izquierda de la imagen están las imágenes con efectos de submuestreo (FOV reducido) obtenidas en las bobinas receptoras. En el lado derecho, se observa la imagen reconstruida una vez se ha aplicado el algoritmo SENSE.

El valor del píxel en la posición $\vec{x_A}$ de la imagen recuperada en la bobina j viene dada por la ecuación 2.5. Como se puede observar, es la superposición de los píxeles I_A e I_B (con diferentes pesos en cada caso, dados por los mapas de sensibilidad) de la imagen con FOV completo, debido al plegado que sufre con factor de aceleración $R = 2$.

$$F_j(\vec{x_A}) = I(\vec{x_A}) C_j(\vec{x_A}) + I(\vec{x_B}) C_j(\vec{x_B}) \tag{2.11}$$

Los valores de los píxeles marcados en verde en las imágenes *folded* de cada bobina de la figura 2.5 se obtienen particularizando para cada bobina en la ecuación anterior:

$$\left.\begin{aligned} F_1(\vec{x_A}) &= I(\vec{x_A}) C_1(\vec{x_A}) + I(\vec{x_B}) C_1(\vec{x_B}) \\ F_2(\vec{x_A}) &= I(\vec{x_A}) C_2(\vec{x_A}) + I(\vec{x_B}) C_2(\vec{x_B}) \\ F_3(\vec{x_A}) &= I(\vec{x_A}) C_3(\vec{x_A}) + I(\vec{x_B}) C_3(\vec{x_B}) \\ F_1(\vec{x_A}) &= I(\vec{x_A}) C_4(\vec{x_A}) + I(\vec{x_B}) C_4(\vec{x_B}) \end{aligned}\right\}$$

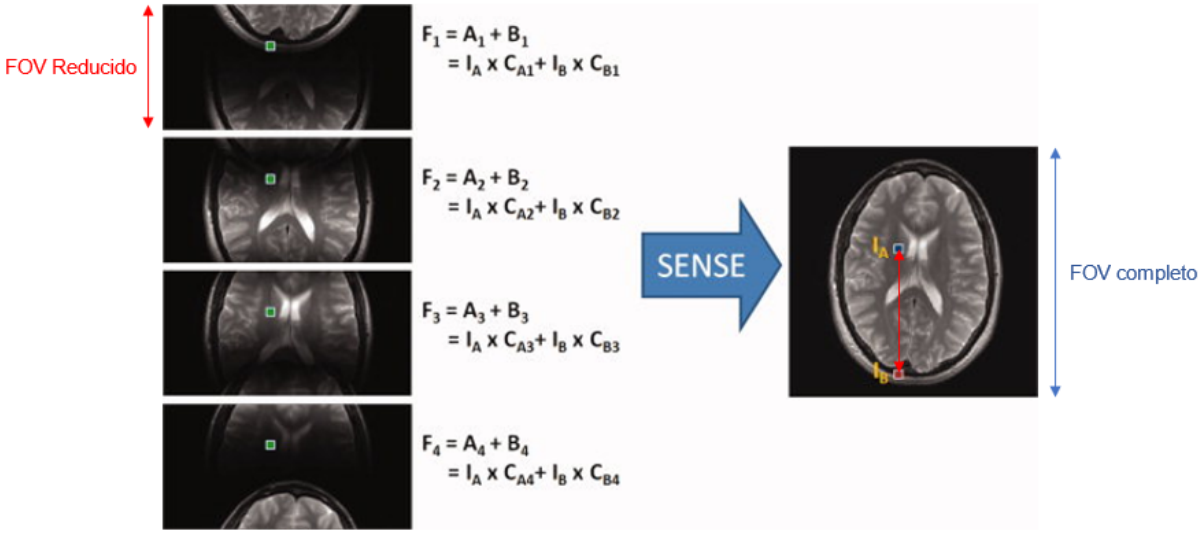


Figura 2.5: Ejemplo básico de reconstrucción con SENSE para $R = 2$ y $n_C = 4$

Como lo que se desea es conocer el valor de los píxeles I_A e I_B de la imagen reconstruida, se expresa el sistema anterior en formato matricial:

$$\begin{pmatrix} C_1(\vec{x_A}) & C_1(\vec{x_B}) \\ C_2(\vec{x_A}) & C_2(\vec{x_B}) \\ C_3(\vec{x_A}) & C_3(\vec{x_B}) \\ C_4(\vec{x_A}) & C_4(\vec{x_B}) \end{pmatrix} \begin{pmatrix} I(\vec{x_A}) \\ I(\vec{x_B}) \end{pmatrix} = \begin{pmatrix} F_1(\vec{x_A}) \\ F_2(\vec{x_A}) \\ F_3(\vec{x_A}) \\ F_4(\vec{x_A}) \end{pmatrix} \tag{2.12}$$

Este sistema se puede resolver ya que se tienen 4 bobinas y el factor de aceleración es 2, por lo que se cumple la ecuación 2.4 y se tiene que $C^H C$ es invertible, siendo C la matriz de sensibilidades de las bobinas. La solución se obtiene particularizando la ecuación 2.7:

$$\begin{pmatrix} I(\vec{x_A}) \\ I(\vec{x_B}) \end{pmatrix} = [C^H C]^{-1} C^H \begin{pmatrix} F_1(\vec{x_A}) \\ F_2(\vec{x_A}) \\ F_3(\vec{x_A}) \\ F_4(\vec{x_A}) \end{pmatrix} \tag{2.13}$$

Con la resolución de este sistema se han obtenido los valores de 2 píxeles de la imagen reconstruida. Para obtener la imagen final completa basta con repetir este proceso para todos los píxeles de las imágenes con FOV reducido.

2.3.2 Mapas de sensibilidad

A lo largo de esta sección se ha destacado la importancia de la información espacial inherente a los sistemas receptores de bobinas en *array* como parte fundamental en las técnicas de PI. Como se ha comentado anteriormente, existen métodos denominados autocalibrados en los

que la sensibilidad de las bobinas se estima, sin embargo en otros, como SENSE, es necesario un conocimiento preciso de esta información para lograr la reconstrucción. En el ámbito de los simuladores, la sensibilidad de las bobinas será también simulada. El presente apartado tiene como objetivo describir los fundamentos en los que se basa la simulación de los mapas de sensibilidad de las bobinas utilizados.

El modelado de las bobinas propuesto proviene del software XCAT (*extended Cardiac-Torso*), el cual simula matemáticamente distintos *phantom* [15]. De acuerdo con esto, es posible simular las sensibilidades de las bobinas en 3 dimensiones (3D), utilizando la Ley de Biot-Savart:

$$\vec{B}(\vec{r}) = \frac{\mu_0}{4\pi} \oint_{coil} \frac{i\vec{dl} \times (\vec{r} - \vec{r'})}{\mid \vec{r} - \vec{r'} \mid^3}, \tag{2.14}$$

discretizando la integral y calculando el campo magnético efectivo en las posiciones del vóxel.

Se consideran las bobinas como espiras circulares y se sitúan en circunferencias llamadas anillos entorno al *phantom* (en una situación real, se correspondería con la parte del paciente de la que se quiere adquirir la imagen). El esquema sería el que se muestra en la figura 2.6, variando en función del número de bobinas y de anillos que se desee. En verde se representan los distintos anillos en los cuales se distribuyen las bobinas, dibujadas en color rojo. En un sistema real, en el interior del cilindro se colocaría el paciente.

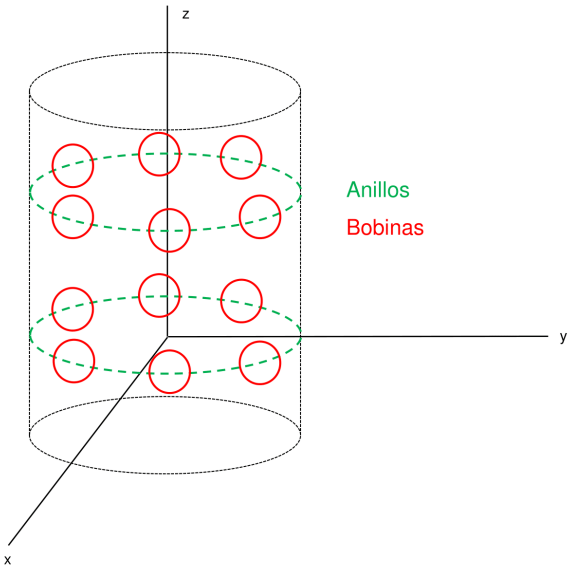
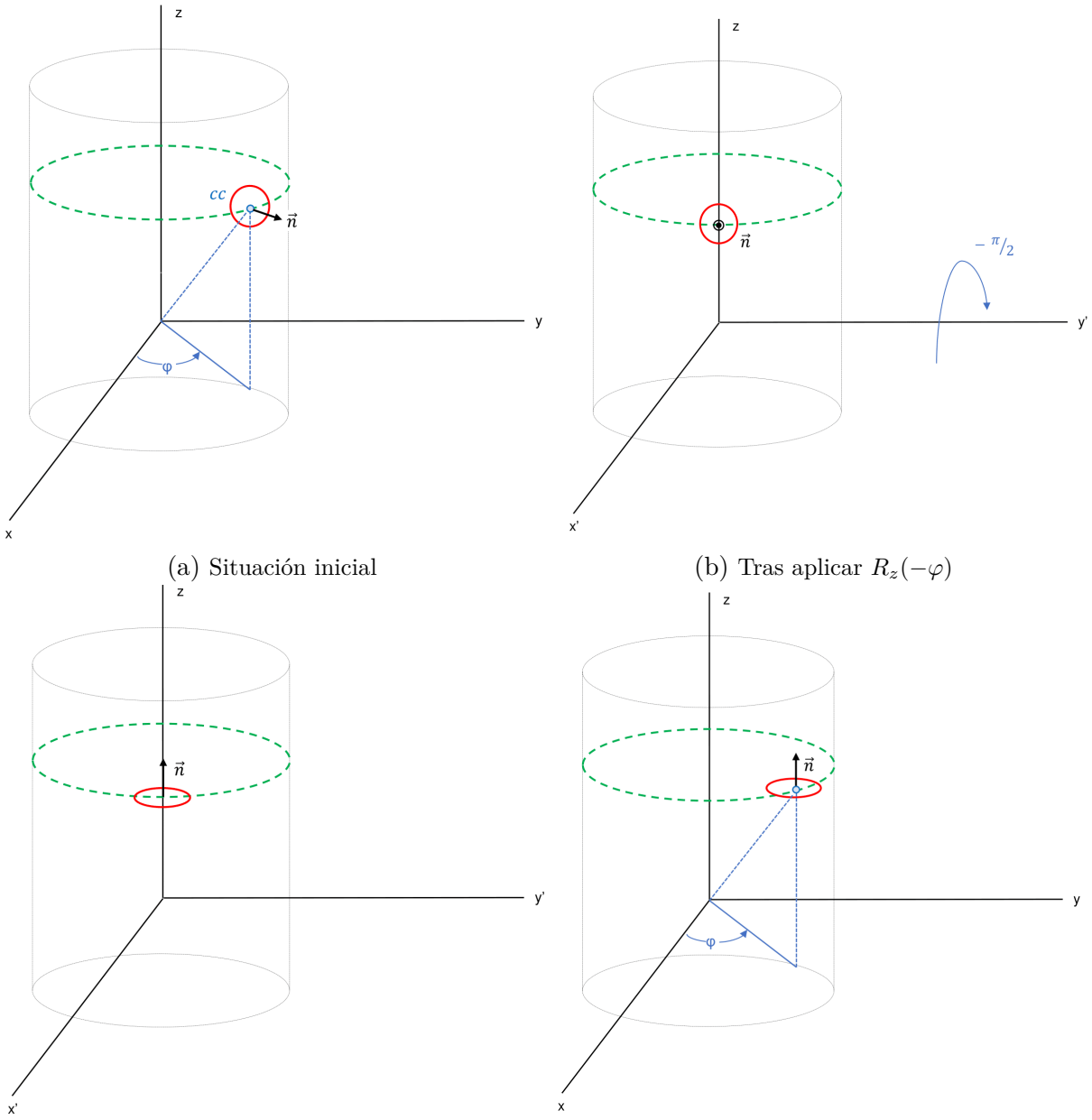


Figura 2.6: Representación de las bobinas. Ejemplo para 12 bobinas, 6 por anillo

Como anotación, es importante tener en cuenta que la ley de Biot-Savart solo es válida en la aproximación de campo cercano. En el caso que nos ocupa se considera una hipótesis válida, ya que se cumple para las intensidades de campo magnético habituales en MR, menores de 4 T.

En el mismo orden de ideas, para simular los mapas de sensibilidad, se calcula el campo magnético producido por cada bobina en los distintos puntos del vóxel de la imagen. Cada bobina, por tanto, tendrá su propio mapa de sensibilidad que será una imagen 2D o 3D según corresponda, que representa la intensidad de campo magnético en cada punto.

De este modo, para poder aplicar las expresiones conocidas de cálculo de un campo magnético producido por una espira circular en un punto P, siguiendo las expresiones de [16], es necesario realizar una serie de transformaciones al sistema de partida de la figura 2.6. En la figura 2.7a, se presenta un ejemplo de la situación de una bobina en el *array*. Su centro está situado en las coordenadas cartesianas *cc* y en coordenadas cilíndricas esto corresponde con un ángulo φ (ver Apéndice A). Para poder aplicar las expresiones definidas en [16] es necesario que la bobina esté en el plano XY y que su vector normal apunte en la dirección z. En primer lugar, es necesario realizar una rotación de $-\varphi$ radianes en torno al eje z, de forma que obtendríamos el sistema de la figura 2.7b. A continuación como se muestra en la figura 2.7c, una rotación de $-\pi/2$ radianes alrededor del eje y' es necesaria para que el vector normal $\vec{n}$ apunte en la dirección z. Finalmente, una vez posicionada la bobina en el plano adecuado es necesario volver a su posición inicial aplicando una última rotación de $-\varphi$ radianes en torno al eje z, dando lugar a la situación de la figura 2.7d. Todas estas rotaciones del sistema de coordenadas son posibles de realizar de manera sencilla utilizando las matrices de rotación definidas en el Apéndice B, [15, 17].



(a) Situación inicial

(b) Tras aplicar $R_z(-\varphi)$

(c) Tras aplicar $R_{y'}(-\frac{\pi}{2})$

(d) Tras aplicar $R_{z'}(\varphi)$

Figura 2.7: Esquema ilustrativo de las rotaciones necesarias a aplicar al sistema de bobinas simulado para utilizar las ecuaciones de cálculo de $\vec{B}$ producido por una espira circular [16] y calcular los mapas de sensibilidad [15]

Capítulo 3

Computación heterogénea CPU-GPU

En este capítulo se describe en qué consiste el término de computación heterogénea, además de proporcionar una visión general de los principios básicos en los que se basa la computación paralela, particularizando en la programación en GPU. Por último, se centra en el lenguaje de programación OpenCL, dado que ha sido el utilizado para llevar a cabo la implementación que en este trabajo se plantea.

3.1 Introducción a la programación paralela

Actualmente, los entornos de computación se presentan cada vez más heterogéneos. Este término da lugar al concepto de computación heterogénea, que se refiere a sistemas que usan más de un tipo procesador. Son sistemas que ganan en rendimiento no por añadir el mismo tipo de procesadores, sino por añadir procesadores distintos, ya que normalmente incorporan capacidades de procesado especializadas para realizar tareas particulares. Estas arquitecturas de ordenadores presentan frecuentemente microprocesadores *multi-core*, unidades de procesamiento central (CPUs), procesadores digitales de señal (DSPs), dispositivos lógicos programables (FPGAs) y unidades de procesamiento de gráfico (GPUs). Debido a este nivel de heterogeneidad, el proceso de desarrollo de *software* eficiente para una gama tan amplia de arquitecturas plantea una serie de desafíos a la comunidad de programación, siendo el reto de los programadores encontrar la concurrencia en su problema y expresarla a nivel de *software* de forma que el programa resultante tenga el rendimiento deseado.

Cabe desatacar que las aplicaciones poseen una serie de comportamientos de carga de trabajo, que van desde el control intensivo (por ejemplo, búsqueda, clasificación y análisis sintáctico) hasta el uso intensivo de datos (por ejemplo, procesamiento de imágenes, simulación y modelado, y extracción de datos). Algunas de estas aplicaciones se pueden caracterizar como intensivas en cómputo, como por ejemplo los métodos iterativos y los métodos numéricos, donde el rendimiento general de la aplicación depende en gran medida de la eficacia computacional del hardware subyacente. Así pues, es importante conocer que cada una de estas clases de carga de trabajo normalmente se ejecuta de la manera más eficiente en un estilo específico de arquitectura de *hardware*. Ahora bien, no hay una arquitectura única que sea la mejor para ejecutar todas las clases de cargas de trabajo y la mayoría de las aplicaciones poseen una combinación de las características de la carga de trabajo. Para que el caso que nos ocupa, nótese que las aplicaciones intensivas en datos tienden a ejecutarse rápidamente en arquitecturas vectoriales, donde la mis-

ma operación se aplica a múltiples datos al mismo tiempo [18].

Surge en este contexto la computación paralela, como forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente, operando sobre el principio de que problemas grandes, a menudo se pueden dividir en otros más pequeños, que pueden ser resueltos simultáneamente (en paralelo). Para comprender este concepto de paralelismo es necesario conocer primero el concepto más general de concurrencia. Un sistema de *software* es concurrente cuando consta de más de una secuencia de operaciones que están activas y pueden progresar a la vez. La concurrencia es fundamental en cualquier sistema operativo moderno. Maximiza la utilización de recursos al permitir que otras secuencias de operaciones (subprocesos) progresen mientras que otras se detienen esperando a algún recurso. La computación paralela sucede cuando se ejecuta un *software* concurrente en un ordenador con varios elementos de procesamiento, de forma que los hilos o *threads* se ejecuten simultáneamente. La concurrencia habilitada por *hardware*, es decir mediante el uso sistema *multi-core*, es paralelismo [19].

Hay varias formas diferentes de computación paralela: paralelismo a nivel de bit, paralelismo a nivel de instrucción, paralelismo de datos y paralelismo de tareas. Dentro de este paradigma se encuentra la computación acelerada por GPU, que hace uso de una GPU junto a una CPU para acelerar el funcionamiento de las aplicaciones con elevada carga computacional como el procesado de imagen y la simulación [20].

3.1.1 Computación en GPU

La computación acelerada por GPU permite asignar a la GPU el trabajo de los aspectos de la aplicación donde la computación es más intensiva, mientras que el resto del código se ejecuta en la CPU, como se representa en el esquema de la figura 3.1. Desde la perspectiva del usuario, las aplicaciones se ejecutan de forma mucho más rápida.

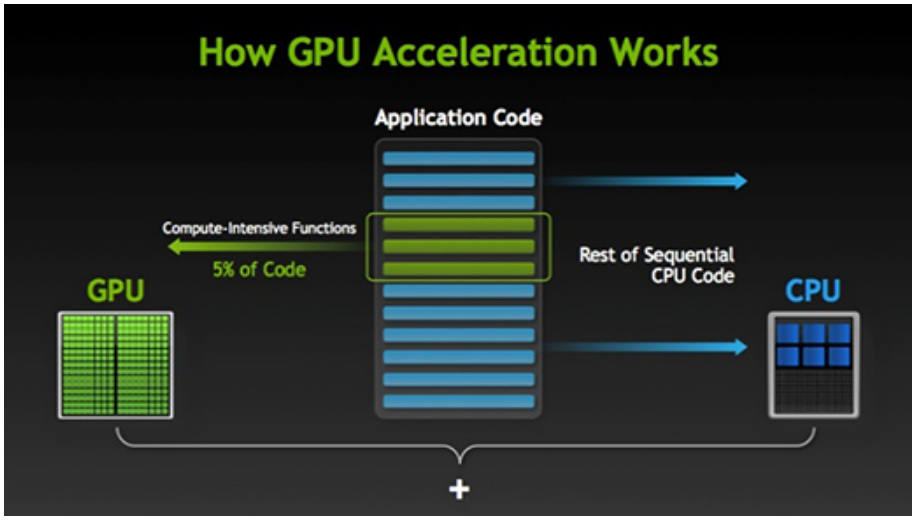


Figura 3.1: Funcionamiento de la aceleración del *software* mediante el uso de GPU

Para comprender la diferencia entre una GPU y una CPU se puede comparar cómo procesa las tareas cada tipo de dispositivo. En la figura 3.2 se representa de forma ilustrativa la arquitectura de cada dispositivo respectivamente. Una CPU tiene unos cuantos núcleos optimizados para el procesamiento secuencial, mientras que una GPU cuenta con una arquitectura en paralelo enorme que consiste en miles de núcleos más pequeños y eficaces, diseñados para resolver varias tareas al mismo tiempo (paralelismo)[20].

Del mismo modo, otras diferencias importantes entre ambos dispositivos son la latencia y el *throughput*. Estas medidas generalmente se utilizan como base para el rendimiento de la instrucción en un microprocesador y son específicas de una instrucción individual. La latencia es el número de ciclos de reloj que requiere una instrucción para que sus datos estén disponibles para que los use otra instrucción. Por otra parte, el *throughput* es el número de ciclos de reloj necesarios para ejecutar una instrucción o realizar sus cálculos. Entonces, mientras que en la CPU cualquier operación toma típicamente del orden de 20 ciclos entre entrar y salir del *pipeline*, en la GPU una operación puede tardar miles de ciclos de principio a fin. Como consecuencia, la latencia es menor en la CPU, pero en la GPU el paralelismo proporciona un mayor *throughput*. Es por tanto crucial, conocer las ventajas e inconvenientes de los dispositivos, para diseñar *software* de calidad y que se adapte a los requisitos deseados [21].

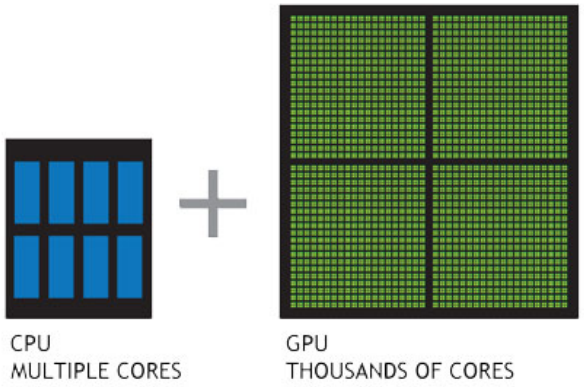


Figura 3.2: CPU vs GPU

3.1.2 Lenguajes de programación en GPU

En este ámbito, se introduce la programación en GPU como una solución eficiente para mejorar el desarrollo de aplicaciones que supongan una elevada carga computacional. Surgen por tanto numerosos lenguajes de programación, necesarios para codificar las instrucciones y crear los programas, cada uno de los cuales está enfocado al desarrollo de distintos tipos de aplicaciones. Por un lado, cabe mencionar DirectX, propietario de Microsoft, y OpenGL y Vulkan, estándares abiertos de Khronos Group, los cuales están orientados al procesamiento y la programación de gráficos. Por otro lado, existen otros lenguajes para programación de GPU con propósito general, tales como CUDA, propietario de NVIDIA, y OpenCL, estándar abierto de Khronos Group. Dado que en este TFG interesan las aplicaciones de este tipo, se va a centrar

únicamente en estos últimos.

La principal diferencia entre CUDA y OpenCL radica en el hecho de que el primero es propietario, por lo que solo está disponible en tarjetas gráficas de NVIDIA. Como consecuencia, este lenguaje ofrece mejores prestaciones cuando se implementa sobre GPUs de esta marca que las ofrecidas por OpenCL sobre la misma marca. No obstante, OpenCL presenta una ventaja importante con respecto a CUDA y otros lenguajes de programación de GPUs propietarios, ya que al ser un estándar abierto se pueden eliminar las restricciones de *hardware* pudiendo ser utilizado en un entorno multiplataforma [22].

3.2 OpenCL

OpenCL (*Open Computing Language*) es un *framework* estándar industrial de computación heterogénea gestionado por el consorcio tecnológico Khronos Group. Permite el desarrollo de aplicaciones con distintos niveles de paralelismo que aprovechen el potencial de las plataformas de procesamiento heterogéneas como CPUs, GPUs y otros procesadores. Para ello, cuenta con una API para coordinar la computación paralela entre procesadores heterogéneos, un lenguaje de programación con un entorno de computación bien especificado, librerías y un sistema de tiempo de ejecución (*runtime system*) para dar soporte al desarrollo de *software*.

3.2.1 Arquitectura

La especificación de OpenCL define la siguiente jerarquía de modelos [19]:

- **Modelo de plataforma (o *Platform Model*):** una descripción a alto nivel del sistema heterogéneo.
- **Modelo de ejecución (o *Execution Model*):** una representación abstracta de cómo los flujos de instrucciones se ejecutan en la plataforma heterogénea.
- **Modelo de memoria (o *Memory Model*):** la colección de regiones de memoria dentro de OpenCL y cómo interactúan durante la computación.
- **Modelos de programación (o *Programming Models*):** las abstracciones de alto nivel que utiliza el programador cuando diseña algoritmos para implementar una aplicación.

3.2.1.1 Modelo de plataforma

El modelo de la plataforma de OpenCL se ilustra en la figura 3.3. Incluye siempre un único *host* (típicamente la CPU) encargado de gobernar todo el sistema, conectado a uno o más dispositivos de computación de OpenCL (*OpenCL devices*). El dispositivo es donde los flujos de instrucciones (o *kernels*) se ejecutan. A menudo, se denominan dispositivos de cómputo y pueden ser una CPU, una GPU, un DSP o cualquier procesador compatible con OpenCL. Un dispositivo de computación de OpenCL está dividido en una o más unidades de computación (*Compute Units*, CUs), que se dividen en uno o varios elementos de procesamiento (*Processing Elements*, PEs). Los cálculos en un dispositivo se producen dentro de los PEs. Además, hay

que tener en cuenta que una CU es un conjunto de PEs que comparten un banco de memoria local, mientras que un PE es una abstracción de una ALU (*Arithmetic Logic Unit*) con su propia memoria privada [19, 23, 22] .

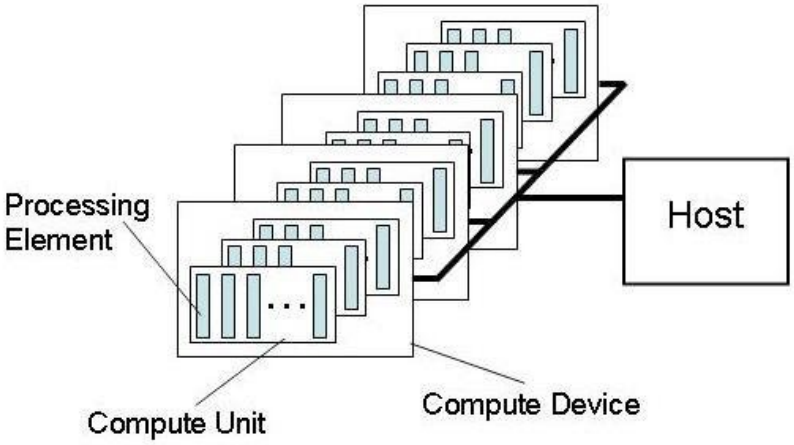


Figura 3.3: Modelo de plataforma de OpenCL con un *host* y uno o más dispositivos OpenCL

3.2.1.2 Modelo de ejecución

La ejecución de un programa de OpenCL se lleva a cabo en dos partes: los *kernels* que se ejecutan en uno o más dispositivos OpenCL y el programa de *host* que, como su nombre indica, se ejecuta en el *host*. Este define el contexto para los *kernels* y gestiona su ejecución.

El núcleo del modelo de ejecución de OpenCL se define por cómo se ejecutan los *kernels*. Para ello es necesario explicar cómo un *kernel* se ejecuta en un dispositivo OpenCL, cómo el *host* define el contexto de ejecución para el *kernel* y cómo los *kernels* se ponen en colas para su ejecución.

En primer lugar, un *kernel* es una función declarada en un programa[1] de OpenCL con el calificador `__kernel` y ejecutada en un dispositivo de OpenCL. Es el programa de *host* el que emite un comando para enviar el *kernel* a ejecutarse en el dispositivo OpenCL y en este momento, el sistema de ejecución de OpenCL crea un espacio de índices enteros. Una instancia del *kernel* se ejecuta para cada punto en este espacio de índices. A cada instancia de un *kernel* en ejecución se denomina elemento de trabajo o *work-item* y se identifica por sus coordenadas en el espacio de índices definido. Estas coordenadas son el identificador global (*global ID*) para dicho *work-item*.

El comando que envía el *kernel* a ejecutarse, crea una colección de *work-items*, de forma que cada uno de ellos tiene la misma secuencia de instrucciones definida por un único *kernel*. Aunque la secuencia de instrucciones es la misma, el comportamiento de cada *work-item* puede

[1]Conjunto de *kernels*, funciones auxiliares y constantes

variar debido a sentencias condicionales en función del ID global. A su vez, los *work-items* se organizan en *work-groups*. A los *work-groups* se les asigna un único identificador de *work-group* (*work-group ID*), con la misma dimensionalidad que el espacio de índices que se utiliza para los *work-items*. A los *work-items* se les asigna un identificador único local (*local ID*) dentro de un *work-group*, para que un único *work-item* pueda ser identificado de manera unívoca por su ID global o por una combinación de su ID local y del ID de *work-group*.

Los *work-items* de un *work-group* dado se ejecutan de manera concurrente en los PEs de una única CU. Esto en un punto crítico en la concurrencia de OpenCL, ya que solo garantiza que los *work-items* dentro de un *work-group* se ejecutan simultáneamente Por lo tanto, nunca se puede asumir que los *work-groups* o las invocaciones de un *kernel* se ejecuten simultáneamente. De hecho, a menudo se ejecutan al mismo tiempo, pero el diseño del algoritmo no puede depender de esto.

En cuanto al espacio de índices, se extiende en un rango de valores N-dimensional y se llama *NDRange*, donde N es 1, 2 ó 3. Dentro de un programa de OpenCL, un *NDRange* se define por un *array* de enteros de longitud N especificando el tamaño del espacio de índices en cada dimensión. Cada *work-item* tiene un ID global y un ID local que son tuplas N-dimensionales.

En la figura 3.4 se expone un ejemplo de un espacio de índices bidimensional. Definimos el espacio de índice de entrada para los *work-items* (G_x, G_y), el tamaño de cada *work-group* (S_x, S_y) y el ID de desplazamiento u *offset* global (F_x, F_y). Los índices globales se definen en un espacio de índices de G_x por G_y, donde el número total de *work-items* es el producto de G_x por G_y. Los índices locales se definen en un espacio de índices de S_x por S_y, donde el número de *work-items* de un *work-group* es $S_x \cdot S_y$. Dado el tamaño de cada *work-group* y el número total de *work-items* se puede calcular el número de *work-groups*. Se utiliza un espacio de dos dimensiones para identificar un *work- group*. Cada *work-item* se identifica por su identificador global (g_x, g_y) o por la combinación de la identificación de *work-group* (w_x, w_y) , el tamaño de cada *work-group* y el ID local (s_x, s_y) dentro del *work-group* de tal manera:

$$(g_x, g_y) = (w_x \cdot S_x + s_x + F_x, w_y \cdot S_y + s_y + F_y)$$

Como se ha explicado anteriormente, es en los dispositivos OpenCL donde el trabajo computacional de una aplicación OpenCL tiene lugar, sin embargo el *host* también juega un papel muy importante, ya que es donde se definen los *kernels*, además de definir el *NDRange* y las colas que controlan cómo y cuándo se ejecutan los mismos. Por tanto, una vez expuesto cómo se ejecutan los *kernels*, es necesario comprender el concepto de contexto. El *host* define el contexto (creado y manipulado con funciones de la API de OpenCL) para la ejecución de los *kernels* e incluye los siguientes recursos [19], [23]:

- **Dispositivos o *Devices***: colección de dispositivos OpenCL que pueden ser usados por el *host* y ejecutan los *kernels*.
- ***Kernels***: funciones escritas en lenguaje OpenCL que se ejecutan en los *devices*.
- **Colas de comandos o *Command Queues***: estructuras de datos para coordinar la ejecución de los *kernels* en cada *device*. Permiten la interacción entre el *host* y los *devices*,

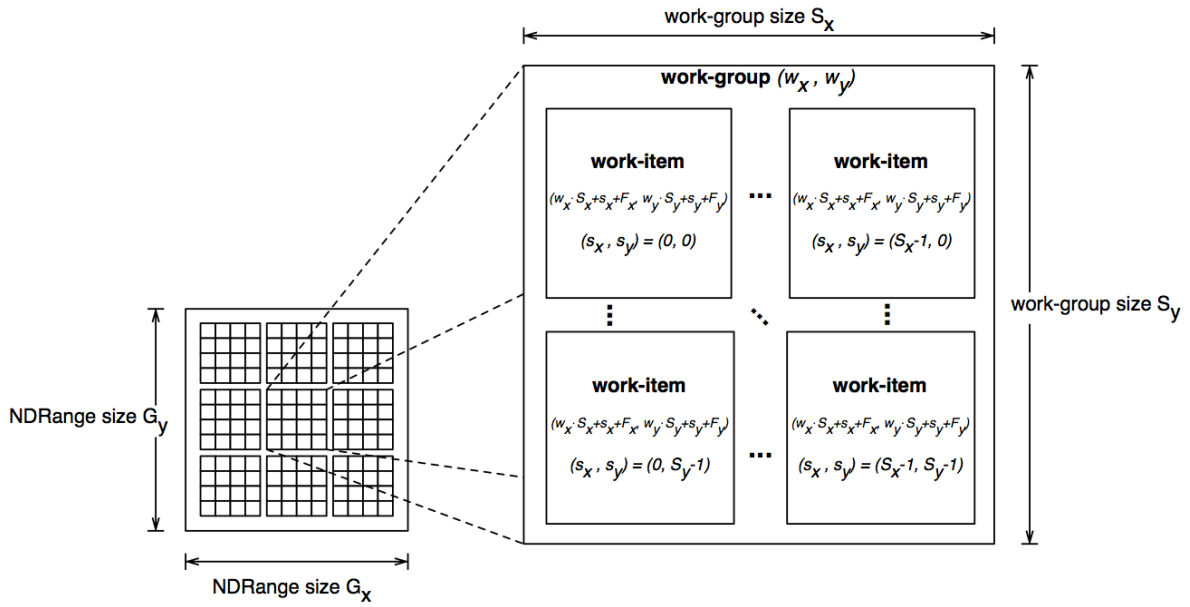


Figura 3.4: Ejemplo de un espacio de índices *NDRange* con sus *work-items*, *work-groups* y el mapeo entre *global IDs* y *local IDs*

a partir de comandos que forman una cola. Es necesario destacar, que deberá existir una *command queue*, creada por el *host*, para cada *device* utilizado. Los comandos se disponen en cola y se programa su ejecución en el *device* asociado. OpenCL soporta tres tipos de comandos: comandos de ejecución de *kernel*, para ejecutar un *kernel* en los PEs del *device*; comandos de memoria, para transferir datos entre el *host* y los objetos de memoria, así como entre objetos de memoria; y comandos de sincronización, para restringir el orden de ejecución de los comandos.

- **Objetos de programa o *Program Objects***: código fuente del programa y ejecutables que implementan los *kernels* de un mismo fichero fuente.
- **Objetos de memoria o *Memory Objects***: conjunto de objetos en memoria que contienen los datos que pueden ser operados por instancias de un *kernel*. Son visibles tanto por el *host* como por los *devices*. En OpenCL los objetos de memoria se pueden definir como *buffers* (unidimensionales) o imágenes (bidimensionales o tridimensionales).

Así pues, se puede mapear un amplio rango de modelos de programación en este modelo de ejecución, pero OpenCL solo considera explícitamente dos que se comentarán en el apartado correspondiente a modelos de programación [23].

3.2.1.3 Modelo de memoria

Los *work-items* que ejecutan instancias de un *kernel* tienen acceso a cuatro regiones de memoria, como se muestra en la figura 3.5, [23]:

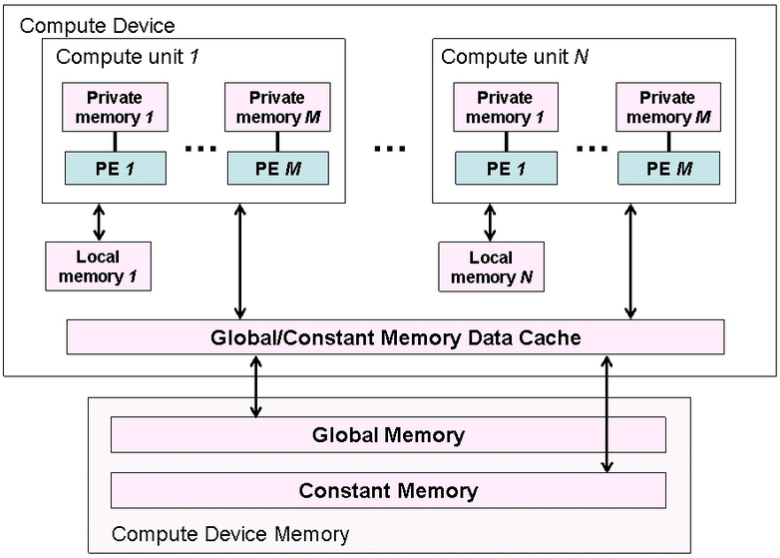


Figura 3.5: Resumen del modelo de memoria de OpenCL y cómo las distintas regiones interactúan con el modelo de la plataforma

- **Memoria global:** unión de todas las memorias RAM de los *devices* (CPU o GPU) seleccionados para trabajar. Permite el acceso de lectura y escritura a todos los *work-items* de todos los *work-group*.
- **Memoria constante:** región de la memoria global declarada como de solo lectura durante la ejecución de un *kernel*. Es necesario destacar que algunos *devices* pueden tener una memoria dedicada a este fin.
- **Memoria local:** región de memoria compartida por todos los *work-items* de un único *work-group*.
- **Memoria privada:** región de memoria accesible para un único *work-item*.

La aplicación que se ejecuta en el *host* utiliza la API de OpenCL para crear objetos de memoria en la memoria global e introducir los comandos de memoria en la cola para operar con estos objetos de memoria.

Los modelos de memoria del *host* y los dispositivos de OpenCL son, en general, independientes, aunque en ocasiones interactúan. Estas interacciones se pueden dar de dos formas, con la copia explícita de datos y con la asignación/desasignación de regiones del espacio de direcciones del *host* a un objeto de memoria.

Finalmente, otro punto destacable es que OpenCL usa un modelo de memoria con consistencia relajada entre los *work-items*. Esto quiere decir que la memoria local es consistente para los *work-items* de un *work-group* tras una barrera de grupo y lo mismo sucede para la memoria global, pero no se garantiza consistencia entre *work-groups* diferente [23].

3.2.1.4 Modelo de programación

El modelo de ejecución OpenCL soporta los modelos de programación de paralelismo de datos y de tareas, así como híbridos de estos dos modelos, siendo el modelo principal que impulsa el diseño de OpenCL el paralelismo de datos. Además, será este modelo el usado principalmente para la implementación del simulador de este trabajo [23].

- **Paralelismo de datos:** define la computación en términos de una secuencia de instrucciones que se aplica a varios elementos de un objeto de memoria. Además, el espacio de índices asociado al modelo de ejecución de OpenCL define los *work-items* y cómo se mapean los datos en los *work-items*. En un modelo estrictamente paralelo de datos, existe una asignación uno a uno entre el *work-item* y el elemento en un objeto de memoria sobre el cual un *kernel* puede ser ejecutado en paralelo. Sin embargo, OpenCL implementa una versión relajada del modelo de programación paralela en datos donde no se requiere un mapeo estricto uno a uno. Así, el paralelismo de datos es un modelo muy natural donde se replica la ejecución de núcleos sobre un espacio N-dimensional.

- **Paralelismo de tareas:** define una tarea como una única instancia de un *kernel* que se ejecuta como un único *work-item* independientemente del *NDRange* utilizado por otros *kernels* en la aplicación OpenCL. Se utiliza cuando la concurrencia que un programador desea explotar es interna a la tarea. No obstante, este tipo de paralelización presenta algunas restricciones importantes. Por un lado, no todos los *devices* la soportan, y por otro lado, aunque el *device* la soporte, esta paralelización no se podrá realizar dentro de una única CU.

3.2.2 Estructura de un programa en OpenCL

El procedimiento general que se sigue en un programa en OpenCL es el siguiente [22]:

1. Obtener las plataformas[2] existentes y escoger una.
2. Obtener los dispositivos o *devices* de la plataforma y escoger uno o varios si se desea. Estos pueden ser la CPU, la GPU o una mezcla de ambos. Hay que tener en cuenta que todas las CPU se suelen abstraer como una sola con varias CU.
3. Crear un contexto con los dispositivos escogidos.
4. Asociar una cola de comandos o *command queue* a cada dispositivo del contexto.
5. Crear un programa a partir de un fichero .cl que contiene el código fuente del *kernel*
6. Compilar el programa para los dispositivos del contexto. En OpenCL, los *kernels* se compilan al vuelo.
7. Crear un objeto *kernel* a partir del programa compilado.
8. Cargar los datos de entrada en la memoria del *host*.

[2]Unión de todos los *devices* que puede manejar un determinado *driver*

9. Crear los objetos de memoria necesarios en la memoria de los dispositivos y copiar los datos de entrada del *host* a los objetos de memoria previamente creados.
10. Establecer los parámetros del *kernel*, apuntando a los objetos de memoria necesarios.
11. Lanzar la ejecución del *kernel*.
12. Esperar a que finalice la ejecución
13. Copiar los datos de salida de los objetos de memoria a la memoria del *host*.
14. Liberar los recursos previamente reservados: objeto *kernel*, programa, objetos de memoria, cola de comandos, contexto, dispositivos y plataforma.

Adicionalmente, hay que tener en cuenta que el lenguaje de programación utilizado para los *kernels* es OpenCL, mientras que para el programa del *host* se puede utilizar cualquier lenguaje de programación que tenga una API de OpenCL, siendo las oficiales del Khronos Group las de C y C++.

Capítulo 4

Diseño e implementación

Una vez abordados los conceptos teóricos que sirven de base para la comprensión de las ideas fundamentales que enmarcan el problema, así como las soluciones mediante programación en GPU con OpenCL, es necesario hacer uso de la ingeniería de *software* para la construcción de la propia aplicación *software*, que en este caso será un simulador de MRI. Típicamente se distinguen tres fases: análisis, diseño e implementación, aunque en este capítulo solo se tratarán las dos últimas de manera específica, ya que el análisis se ha realizado a lo largo de todo el TFG [24].

4.1 Diseño funcional del algoritmo

En esta sección se plantea una solución para satisfacer los requisitos que debe cumplir el algoritmo de simulación y reconstrucción SENSE de imágenes de resonancia magnética. Hay que tener en cuenta que esta aplicación parte de un simulador de MRI en GPU que lleva a cabo la simulación de la secuencia de pulsos EPI a partir de una secuencia *Spin echo monoeco* (SE-EPI) [5, 25]. Lo que se pretende es dotar de mayor funcionalidad a la aplicación de forma que permita simular la adquisición de imágenes en paralelo mediante múltiples bobinas receptoras. El punto de partida para el nuevo desarrollo *software* son los datos del espacio k obtenidos por una única bobina receptora (*single coil*). El método de reconstrucción PI que se desea aplicar es SENSE, por lo que será necesario simular los mapas de sensibilidad asociados a las bobinas, los datos que adquiere cada una de estas y el propio algoritmo de reconstrucción, con el fin de obtener la imagen de resonancia magnética.

En primer lugar, se van a describir detalladamente los pasos necesarios para llevar a cabo la simulación:

1. Se parte del espacio k de una única bobina receptora, separado en parte real e imaginaria (tras la simulación de la secuencia EPI).

2. Se configuran los parámetros de la simulación: el número de bobinas deseado, el número de bobinas por anillo, las resoluciones deseadas para los mapas de sensibilidad, el ancho del corte o *slice*, la distancia de los centros de las bobinas al centro de la imagen y el factor de aceleración para SENSE.

3. Se simulan los mapas de sensibilidad asociados a cada bobina, con los parámetros configurados anteriormente y el algoritmo desarrollado en [15]. Hay que tener en cuenta que la simulación de los mapas de sensibilidad es completamente independiente del simulador de partida, por lo que son válidos para 3D, sin embargo al integrarse en el simulador MRI solo será posible el uso de 2D.

   3.1. Se definen de coordenadas de vóxel (píxel en caso 2D) en milímetros a partir de las resoluciones y el ancho del corte.

   3.2. Se simula la disposición de las bobinas a partir del cálculo de sus centros de coordenadas en el sistema. Esta viene definida en función del número de bobinas, a partir de distribuciones angulares fijas para un número menor o igual a 6, como se muestra en la figura 4.1. En caso de ser mayor, la distribución se asume uniforme entre 0 y 2π, es decir, alrededor del cuerpo. También se determina el número de anillos necesarios en función del número de bobinas totales y las bobinas por anillo. Además, se calcula el radio de estas con las siguientes expresiones:

   Si el número de bobinas es menor o igual que 6,

$$rcoil = \frac{1}{2} \cdot rbody_{mm} \cdot \frac{50\pi}{180} \tag{4.1}$$

   En caso contrario,

$$rcoil = \frac{1}{2} \cdot rbody_{mm} \cdot \frac{2\pi}{max(nc_{ring})} \tag{4.2}$$

   siendo $rbody_{mm}$ la distancia entre el centro de la imagen y los centros de las bobinas y nc_{ring} un vector que almacena el número de bobinas por anillo.

   3.3. Se simulan los mapas de sensibilidad asociados a cada una de las bobinas siguiendo las expresiones de [16], que permiten el cálculo del campo magnético en los distintos puntos de la imagen producido por una espira circular a través de la cual circula una corriente (asumiendo que las bobinas se comportan como espiras circulares). Las sensibilidades obtenidas se pueden representar como imágenes.

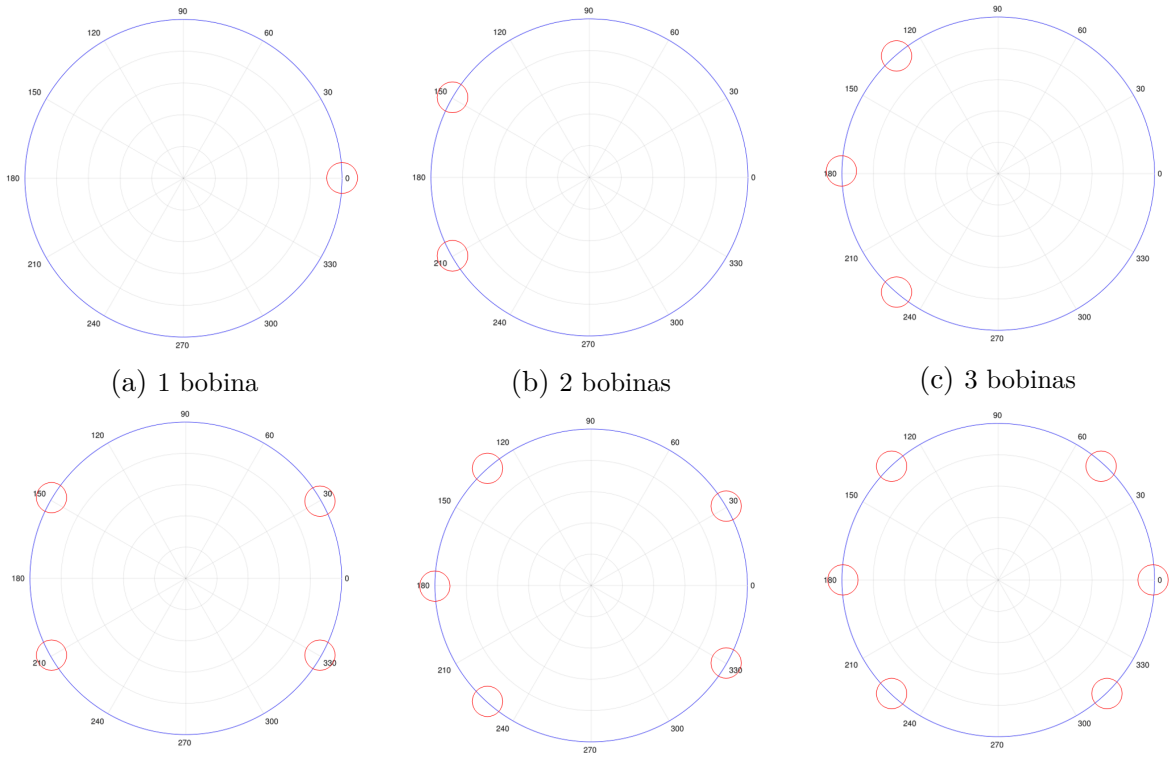
4. Se transforma el espacio k de entrada en un único conjunto de datos complejos.

5. A partir de los datos anteriores, se obtiene el espacio imagen asociado al sistema *single coil* aplicando la Transformada Inversa de Fourier (IFFT, *Inverse Fast Fourier Transform*).

6. Comienza la simulación *multi-coil*, con la simulación de las imágenes con FOV completo para cada bobina, calculadas como el producto de los mapas de sensibilidad por la imagen *single coil* de partida.

7. Se calcula la FFT de las imágenes para trabajar con el espacio k.

8. En SENSE, como se explicó en la subsección 2.3.1, es posible adquirir menos líneas de codificación de fase para reconstruir la imagen final debido a que parte de la información se puede extraer de las sensibilidades de las bobinas. Por tanto, se submuestrea el espacio k asociado a las bobinas que se ha simulado en el paso anterior, obteniendo una de cada R líneas (siendo R el factor de aceleración).



(a) 1 bobina (b) 2 bobinas (c) 3 bobinas

(d) 4 bobinas (e) 5 bobinas (f) 6 bobinas

Figura 4.1: Disposición de bobinas según su cardinal

9. Se simulan las imágenes con FOV reducido que adquiere cada bobina aplicando la FFT al espacio k anterior.

10. Una vez simulados los datos de entrada necesarios y las sensibilidades de las bobinas, comienza la reconstrucción de la imagen con SENSE. Para ello es necesario definir los sistemas de ecuaciones lineales dados por la ecuación 2.6, (tantos como píxeles en una imagen con FOV reducida).

    10.1. Se obtienen las matrices C, extrayendo los puntos de los mapas de sensibilidad que permiten el *unfolding* de cada píxel.

    10.2. Se obtienen los vectores F de píxeles solapados.

11. Se resuelven los sistemas con el método deseado (como se comentará en el capítulo 5, existe la posibilidad de emplear distintos métodos numéricos para solventar estos sistemas lineales sobredeterminados, si el número de bobinas es mayor que el factor de aceleración). Típicamente se utiliza el método de mínimos cuadrados de forma que la solución viene dada por la ecuación 2.7.

12. Finalmente es necesario recolocar de manera adecuada los píxeles una vez se ha llevado a cabo el proceso de *unfolding*, obteniendo así la imagen final con FOV completo.

4.1.1 Diagrama de flujo

El algoritmo explicado a lo largo de esta sección se muestra resumido en el diagrama de la figura 4.2, en el cual se incluyen algunos detalles relacionados con la programación que se va a emplear para el mismo.

4.2 Diagrama de clases

Tras haber comprendido la funcionalidad de la aplicación y sus requisitos es necesario realizar el diseño del *software*. El objetivo del diseño es producir un modelo o una representación genérica previa a la construcción. Se ha utilizado el paradigma de Programación Orientación a Objetos (POO), por lo que en primer lugar, en una etapa de análisis, anterior al diseño, ha sido necesario identificar las clases a modelar. Cabe destacar la existencia de múltiples soluciones distintas de diseño para un mismo problema, siendo unas mejores que otras en diferentes aspectos.

En esta sección se va a presentar el diagrama de clases simplificado utilizando notación UML (*Unified Modeling Language*) que posteriormente servirá de punto de partida para la implementación del simulador. Es importante resaltar la existencia de un simulador de partida, el cual ha sufrido pequeñas modificaciones durante el proceso de integración, pero no en cuanto a su diseño. El desarrollo y diseño de la nueva funcionalidad no se ha basado en el anterior y aunque posteriormente se hayan integrado en una aplicación, estas etapas han sido independientes, por lo que el diseño realizado durante este TFG corresponde solo a las partes añadidas.

En la figura 4.3 se ilustra el diagrama de clases, el cual incluye tanto las clases como las relaciones entre ellas. Por simplicidad, muestra solo clases, sin atributos ni operaciones. Además, no están representadas todas las dependencias, ya que no aportan información relevante al diagrama y añadiría complejidad a la comprensión.

La aplicación está compuesta por 7 clases:

- ***Simulator:*** contiene el simulador de partida en GPU que lleva a cabo la simulación de la secuencia de pulsos EPI a partir de una secuencia *Spin echo monoeco* (SE-EPI) .
- ***Gpu:*** encapsula todo lo relacionado con OpenCL, parámetros (plataformas, dispositivos, contexto, cola de comandos y programas), la inicialización de los mismos y la liberación de los recursos asociados.
- ***Bobinas:*** contiene la configuración de las bobinas receptoras a simular (número total, número de bobinas por anillo, distancia del centro de la imagen a los centros de las bobinas). Se encarga del cálculo de los centros de las bobinas y del radio de estas.
- ***Mapas:*** modela el conjunto de datos asociados a los mapas de sensibilidad de las bobinas y sus operaciones, en concreto la propia simulación de las sensibilidades.
- ***Imagen:*** encapsula los datos (dimensiones y arrays de datos que contienen los valores de las imágenes) y operaciones (cálculo de la FFT2D y simulación de las imágenes en las bobinas) asociados a las imágenes.

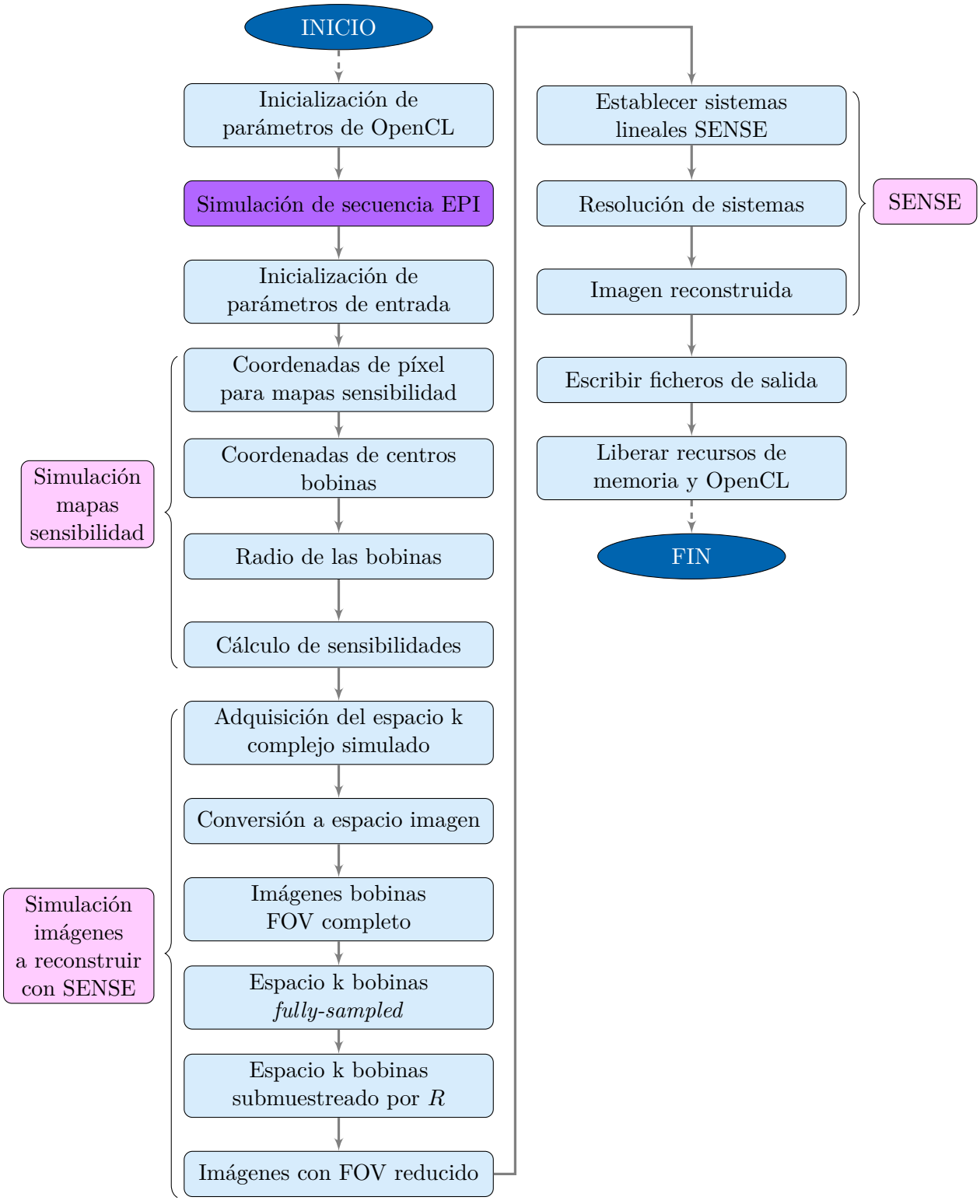


Figura 4.2: Diagrama de flujo del algoritmo

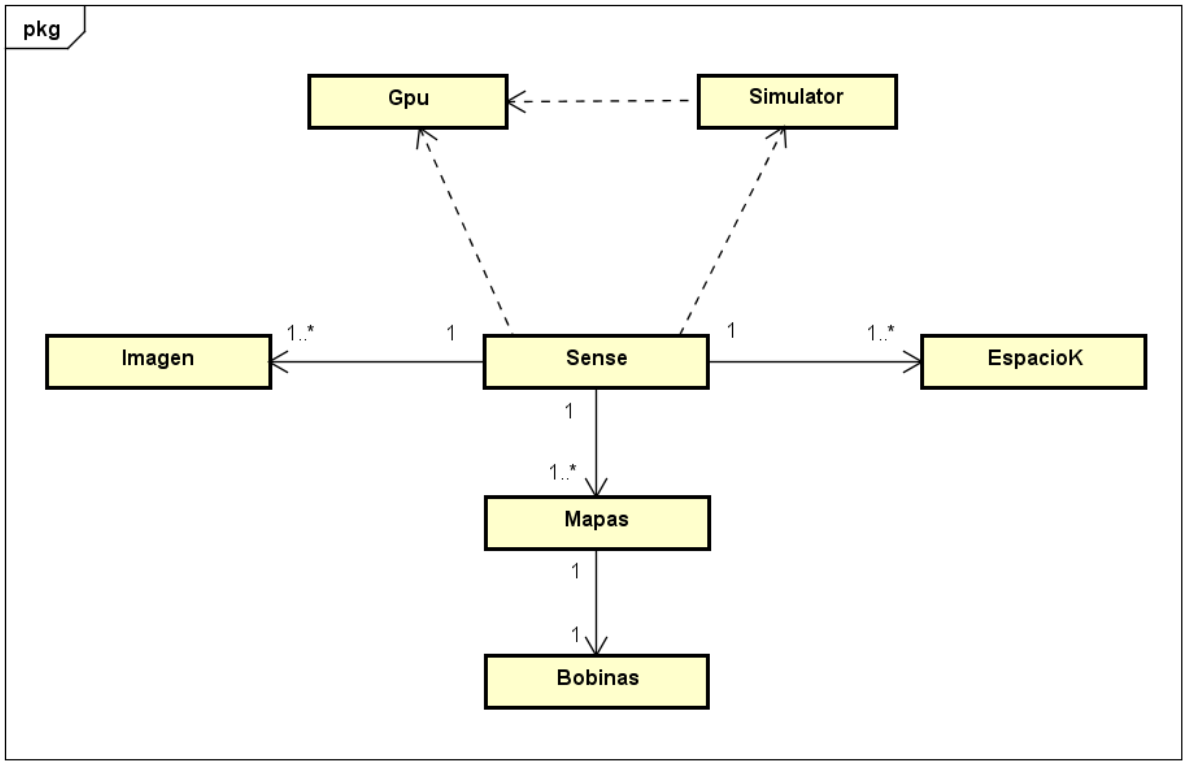


Figura 4.3: Diagrama de clases UML del simulador de MRI con SENSE

- ***EspacioK:*** encapsula los datos (dimensiones y arrays de datos que contienen los valores de los espacios k) y operaciones (cálculo de la IFFT2D y submuestreo del espacio k) asociados.

- ***Sense:*** modela el algoritmo de reconstrucción SENSE. Contiene los datos fundamentales del mismo (factor de aceleración, FOV reducido, datos sobre los que se va a aplicar la reconstrucción y los sistemas lineales a resolver). Las operaciones fundamentales están relacionadas con la inicialización de las entradas y la resolución de los sistemas para obtener la imagen reconstruida. Esta clase también se utilizará para encapsular el funcionamiento de la aplicación y llevar el control de flujo de la mayor parte del sistema de forma que sea transparente para el usuario (desarrollador).

4.3 Implementación con C++ y OpenCL

Una vez diseñado el algoritmo de manera funcional y genérica es necesario implementarlo. La mayor parte del algoritmo es paralelizable, ya que consiste en aplicar las mismas operaciones sobre todas las bobinas, a la hora de calcular sus centros o mapas de sensibilidad, o sobre todos o un gran conjunto de píxeles, en el *unfolding* de las imágenes. Por tanto, como ya se ha comentado, se va a utilizar la potencia que ofrece la programación en GPU, en concreto el paralelismo de datos de OpenCL. Además, el problema se va a enfocar mediante el paradigma de programación orientada a objetos. Dentro de este paradigma existen numerosos lenguajes para llevar a cabo

el desarrollo *software* del programa de *host*, pero debido al uso de OpenCL, se ha optado por utilizar el lenguaje de programación C++ pues, junto con C, son los únicos lenguajes con API de OpenCL oficiales. Con esta implementación se pretende dotar al código de flexibilidad y modularidad, a partir de la creación de distintas clases que puedan ser fácilmente instanciadas, utilizadas e incluso reutilizadas en otras aplicaciones. Es cierto que se podría haber llevado a cabo con una programación orientada a procedimientos pero, dado que un objeto posee un estado propio, la utilización de la programación orientada a objetos permite obtener esta funcionalidad sin necesidad de pasar demasiados argumentos a los métodos ni declarar variables globales. Por otro lado, de esta forma la implementación queda preparada para que en un futuro se pueda utilizar la API de C++ de OpenCL, evitando así los claros inconvenientes que la API de C presenta.

4.3.1 Estructura del código

Antes de explicar el código de la aplicación desarrollada es importante conocer la existencia de los llamados ficheros de cabecera y ficheros fuente. En C++ existen principalmente dos tipos de ficheros, los de cabecera (normalmente `.h` o `.hpp`) y los ficheros fuente (normalmente con extensión `.cpp`). La diferencia entre ellos es puramente conceptual, ya que ambos son ficheros de texto plano y el compilador no distingue entre uno y otro. En general, todo lo que son las definiciones deben ir en los ficheros de cabecera y todo lo que son las implementaciones en los ficheros de código fuente, así que lo normal es tener un archivo `.h` y otro `.cpp` con nombres iguales, de forma que en uno se definen clases, funciones, estructuras, etc. y en el otro se implementan. Los motivos que llevan a dividir el código fuente en ficheros son los siguientes: proporciona una compilación más eficiente de forma que si se hace una pequeña modificación no es necesario recompilar todo el código, solo las dependencias de este; permite mayor organización y facilita la reutilización, entre otras ventajas. Como inconveniente hay que destacar las dependencias entre ficheros que surgen de esta división y los posibles duplicados en definiciones. Sobre el código propio de OpenCL, se debe mencionar que los *kernels* se implementan haciendo uso del propio lenguaje que ofrece el *framework* OpenCL y los ficheros en los que residen tienen extensión `.cl`. Por último, se puede señalar que no existe un criterio para la estructura de los ficheros, de forma que es el programador el que decide.

Por otro lado, respecto al código del proyecto *software* desarrollado a lo largo de este TFG, está dividido en múltiples ficheros de código fuente y cabeceras, con objeto de que sea más legible, modular y mantenible. En primer lugar y más importante, existe un fichero que contiene la declaración de la función principal o `main()` del programa. Su importancia reside en el hecho de que es la invocada cuando el programa se ejecuta. Con relación a los *kernels* de OpenCL, se han dividido en ficheros según su funcionalidad en la parte del simulador de MRI de partida, y en un fichero por clase en la parte que se ha desarrollado a lo largo de este trabajo, de forma que los *kernels* que utiliza cada clase están dentro del mismo fichero `.cl`.

La estructura de ficheros se puede ver en detalle en el Apéndice D, donde además se incluye el contenido de cada uno de ellos. En resumen, existe un fichero `.h`, `.cpp` y otro `.cl` para cada una de las clases definidas (*Gpu*, *Bobinas* (no tiene fichero `.cl`), *Mapas*, *Imagen*, *EspacioK* y *Sense*). A continuación, se describe en forma de pseudocódigo el flujo de instrucciones que sigue

la función `main()`:

Pseudocódigo Función principal (*Simumri+SENSE.cpp*)

1: Establecer los valores de los parámetros de entrada
2: Crear un objeto de tipo *Simulator*
3: Crear un objeto de tipo *Gpu*
4: Inicializar los parámetros de OpenCL con `mygpu.initOpenCL()`
5: Simulación de la secuencia de pulsos EPI
6: Crear un objeto de tipo *Sense*
7: Establecer la disposición de las bobinas, los mapas de sensibilidad y las imágenes con efectos de submuestreo en las bobinas con `mysense.initSENSE()`
8: Definir los sistemas para reconstruir la imagen con `mysense.setLinearSystems()`
9: Invertir las matrices para resolver los sistemas con `mysense.inverseMatrixM()`
10: Realizar la reconstrucción de la imagen con `mysense.applySENSE()`
11: Liberar los recuros de OpenCL con `mygpu.releaseOpenCL()`

Por otra parte, las clases explicadas en la sección 4.2 de este capítulo se van a resumir en las imágenes de la figura 4.4. En el primer compartimento se incluye el nombre de la clase, en el segundo sus atributos y en el tercero sus métodos (para ver en detalle cada uno de ellos se puede consultar el Apéndice D).

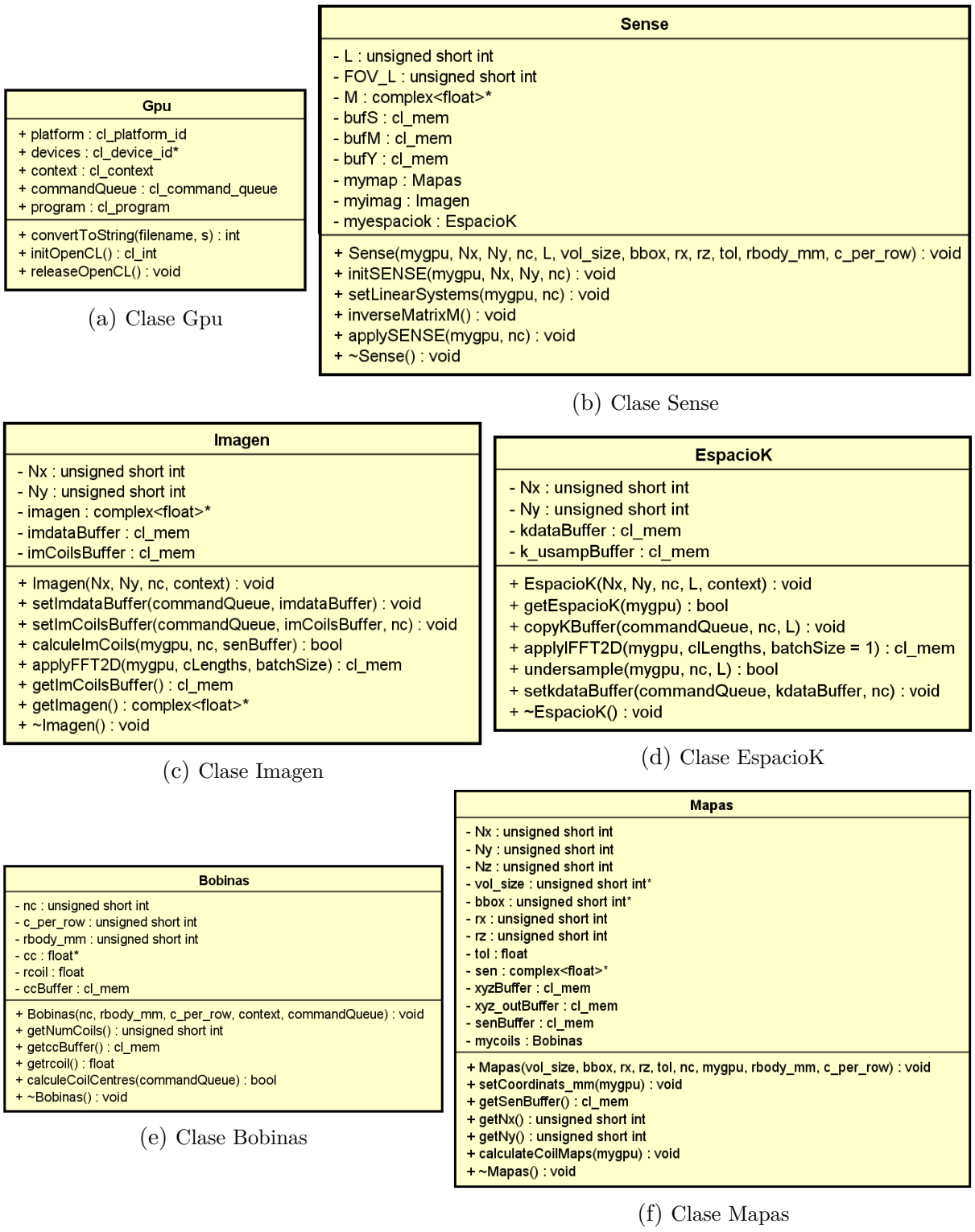
4.4 Interfaz gráfica de usuario

Con el objetivo de integrar el núcleo del simulador, programado en C++ y OpenCL, en una interfaz desde la cual el usuario pueda seleccionar los modelos anatómicos de partida, la disposición y configuración de las bobinas, el factor de aceleración, además de otros parámetros de la simulación y, a su vez, observar la imagen y el espacio k tras la secuencia de pulsos, las imágenes con *aliasing*, así como la imagen final reconstruida con SENSE, se crea una GUI (*Graphical User Interface*) mediante MATLAB. Esta interfaz de usuario se representa en la figura 4.5. Se ha dividido en dos partes para que la visualización en este documento sea más adecuada, teniendo en cuenta que ambas partes son contiguas y dan lugar a la interfaz completa. Como se puede observar, existen dos partes diferenciadas, la zona de configuración, que se ilustra en la figura 4.5a, y el panel de visualización, figura 4.5b.

El funcionamiento interno simplificado de la interfaz implementada se describe a continuación:

1. Cuando el usuario pulsa el botón *Cargar* se leen los tres archivos especificados en los cuadros correspondientes con los modelos anatómicos y sus datos se almacenan en tres variables (*T1Slice*, *T2Slice* y *DPSlice*).

2. Cuando todos los parámetros de la simulación están fijados, se llama automáticamente a la función programada en MATLAB como *crearDeltaBr()*, en la cual según el factor de diezmado se diezman las variables en las que se habían almacenado los datos de los modelos anatómicos y se calculan sus dimensiones. Posteriormente, se calcula la función de inhomogeneidad seleccionada y se representa en la interfaz.



(a) Clase Gpu

(b) Clase Sense

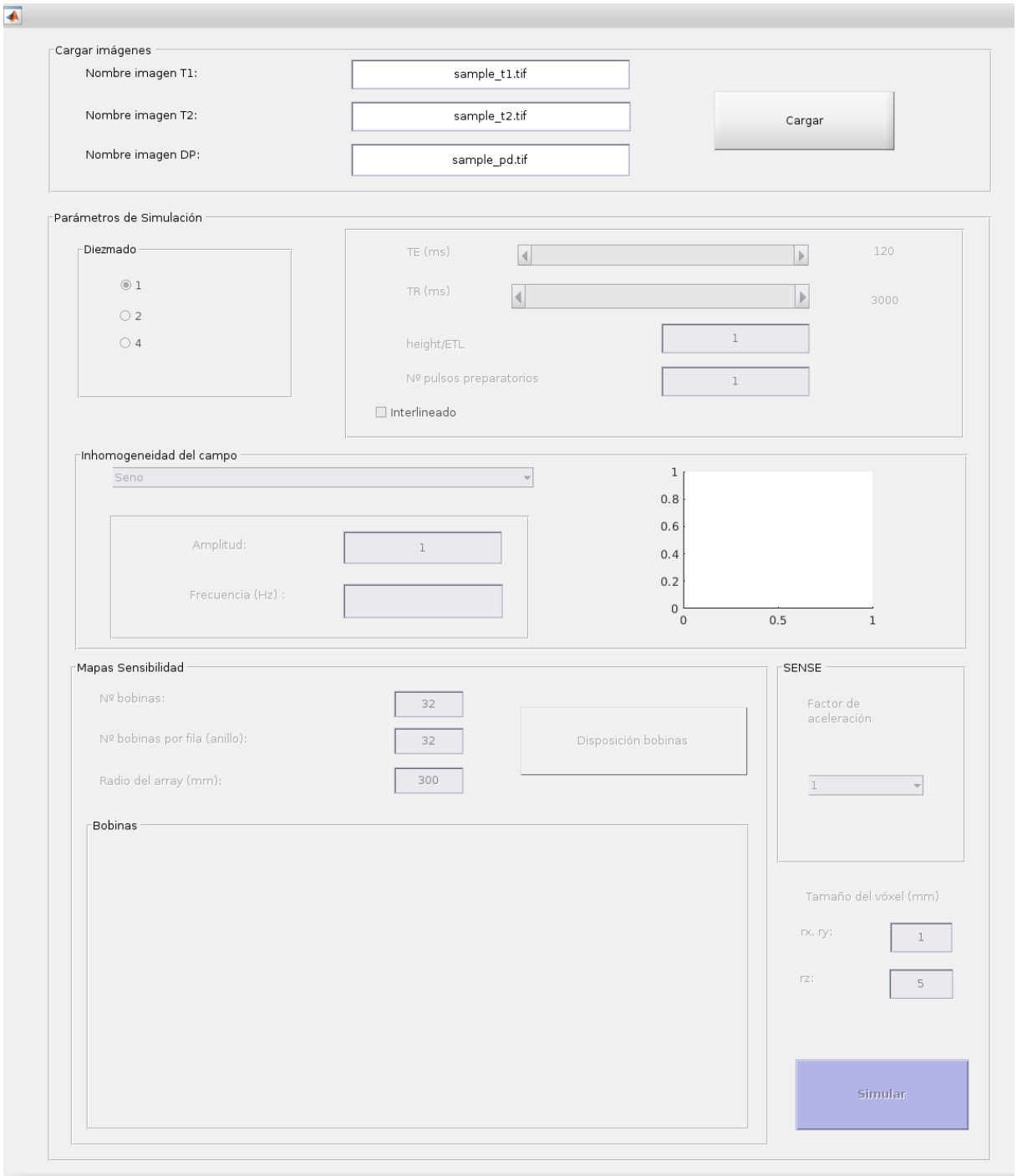
(c) Clase Imagen

(d) Clase EspacioK

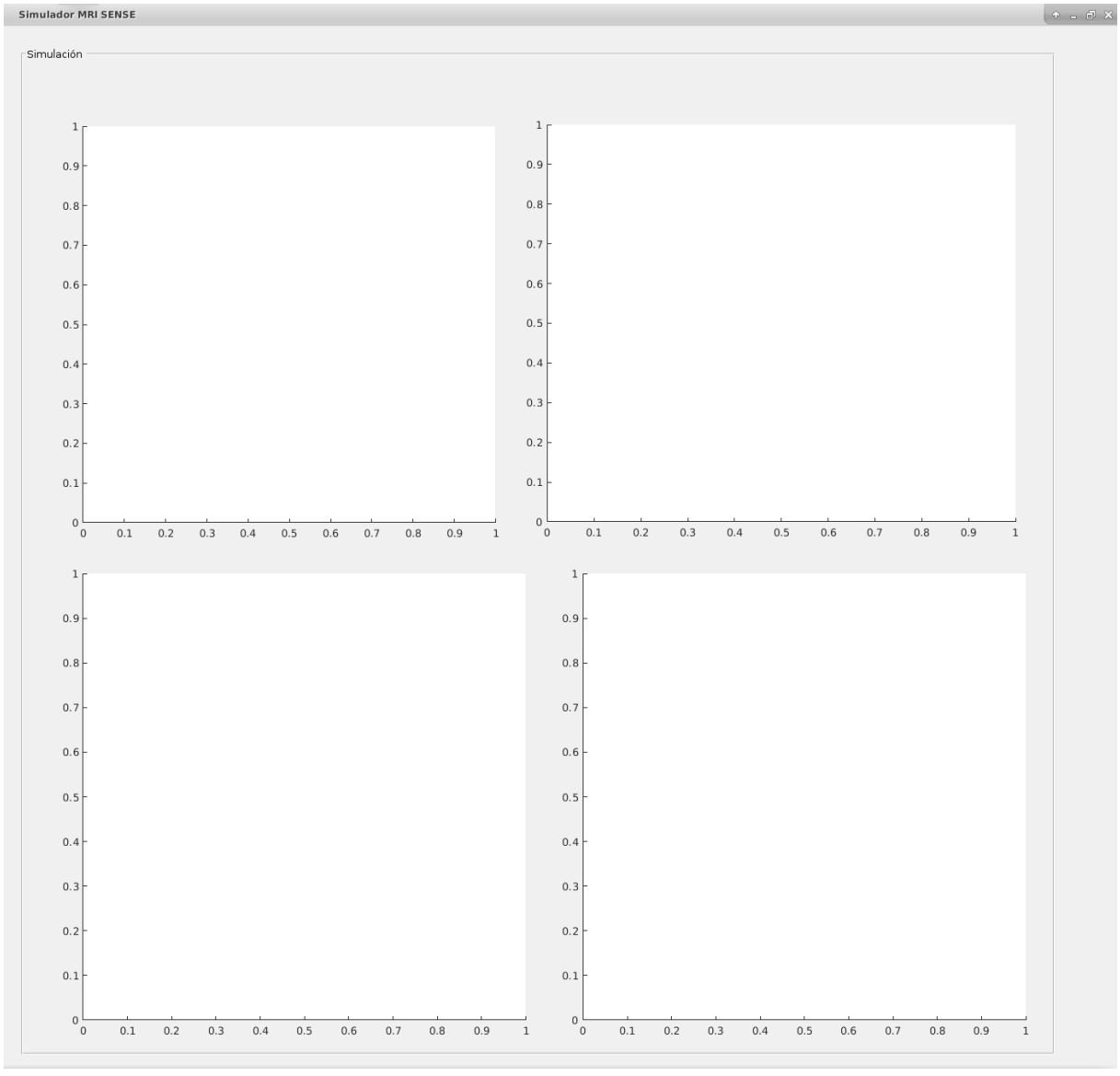
(e) Clase Bobinas

(f) Clase Mapas

Figura 4.4: Clases de diseño en notación UML para la ampliación del simulador



(a) Panel de configuración de la simulación. Parte derecha



(b) Panel de visualización. Parte izquierda

Figura 4.5: Interfaz gráfica de usuario creada con MATLAB

3. La disposición de las bobinas se muestra al usuario en el panel inferior denominado *Bobinas*, en forma de tantos gráficos en coordenadas polares como anillos haya, al pulsar el botón *Disposición Bobinas* con la configuración introducida.

4. Una vez elegida la función de inhomogeneidad es posible seleccionar el factor de aceleración de SENSE mediante un menú de opciones dinámico, que varía en función de la dimensión vertical de la imagen con FOV completo tomada como punto de partida.

5. Finalmente cuando el usuario pulsa el botón *Simular*, se escriben los cuatro archivos en formato *.raw* que contienen los datos del T_1, T_2, PD y el mapa de la inhomogeneidad del campo. Mediante el comando *system()* de MATLAB se ejecuta el ejecutable del código en C++ y OpenCL. A este código se le pasan algunos datos como argumentos de la función `main()`. Concretamente, las dimensiones de las imágenes (ancho y alto), el tiempo de eco (TE), el tiempo de repetición (TR), el número de bloques en los que se quiere escribir el espacio k, el número de pulsos preparatorios que se desean aplicar y un *flag* que permite indicar si la escritura del espacio k se desea hacer secuencial o con interlineado, para la parte de la simulación de la secuencia de pulsos. Por otro lado, para la simulación de múltiples bobinas y SENSE se envían como parámetros el número de bobinas (`nc`), el número de bobinas por anillo (`c_per_row`), el radio del *array* o la distancia del centro de la imagen a las bobinas (`rbody_mm`), la resolución (`rx`) y el ancho del corte (`rz`) en milímetros de los mapas de sensibilidad y el factor de aceleración (`L`). Por último, una vez el código en OpenCL ha terminado de ejecutarse, para el simulador inicial únicamente se leen dos archivos *.raw* que contienen la parte real e imaginaria del espacio k simulado y se realiza la transformada inversa de Fourier para obtener la imagen de resonancia magnética. Ahora bien, debido a la ampliación del mismo, es necesario leer otros dos ficheros *.raw* que contienen hasta 4 imágenes de ejemplo que simulan las adquiridas por las bobinas con FOV reducido y la imagen reconstruida con SENSE. Por último, se representan en los 4 ejes de la GUI el espacio k y la imagen simulados con la secuencia de pulsos EPI (derecha), las imágenes de hasta 4 bobinas en distintos puntos y la imagen reconstruida con SENSE (izquierda).

4.4.1 Guía de usuario

El primer paso en la utilización de la interfaz de usuario es la selección de los modelos anatómicos de partida, lo cual se realiza en la parte de la interfaz representada en la figura 4.6. Es necesario seleccionar un modelo del T_1 (Tiempo de relajación longitudinal), T_2 (Tiempo de relajación transversal) y PD (Densidad de protones) y posteriormente pulsar el botón *Cargar* para leer dichos ficheros.

Una vez los modelos anatómicos han sido cargados, se habilita el panel dónde el usuario puede fijar algunos parámetros de la simulación (representado en la figura 4.7). Por un lado, los parámetros que el usuario puede seleccionar son el factor de diezmado, el tiempo de eco (TE), el tiempo de repetición (TR), el número de bloques en los que se quiere escribir el espacio K (height/ETL), el número de pulsos preparatorios que se desean aplicar y si se quiere que la escritura del espacio k se realice con interlineado o no. También es necesario fijar un mapa de inhomogeneidad del campo magnético. Por otro lado, debe configurar los parámetros siguientes:

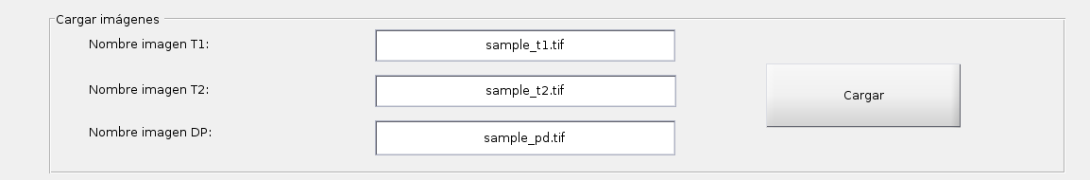


Figura 4.6: Panel de carga de modelos anatómicos en la interfaz

número de bobinas (`nc`), número de bobinas por anillo (`c_per_row`), radio del *array* o la distancia del centro de la imagen a las bobinas (`rbody_mm`), resolución (`rx`) y ancho del corte (`rz`) en milímetros de los mapas de sensibilidad asociados a las bobinas y que se van a simular.

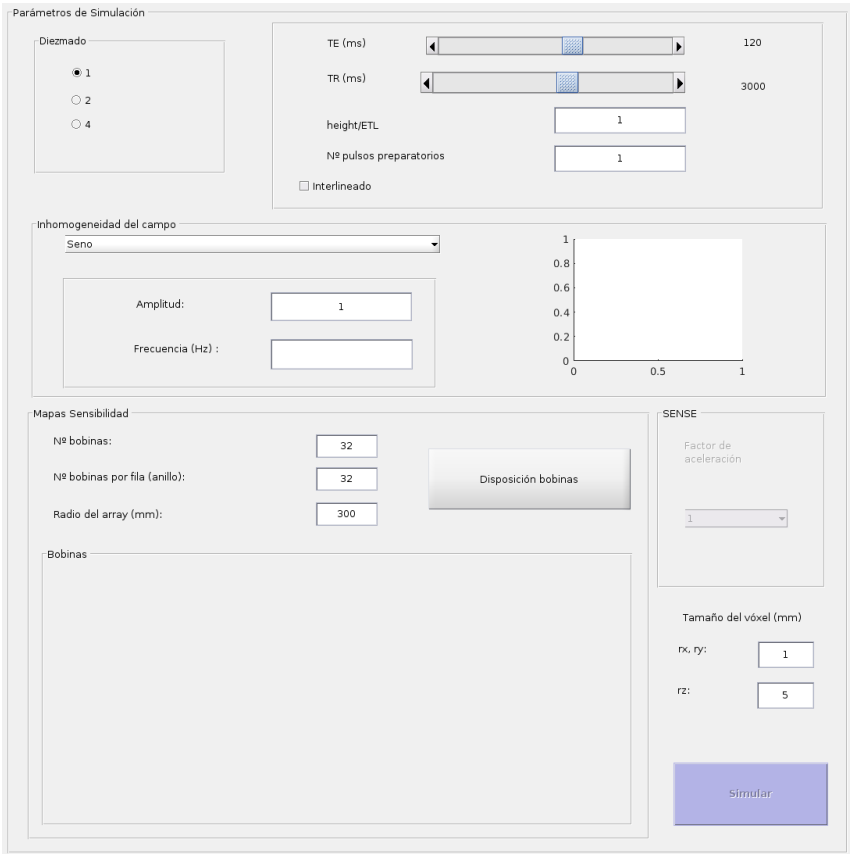
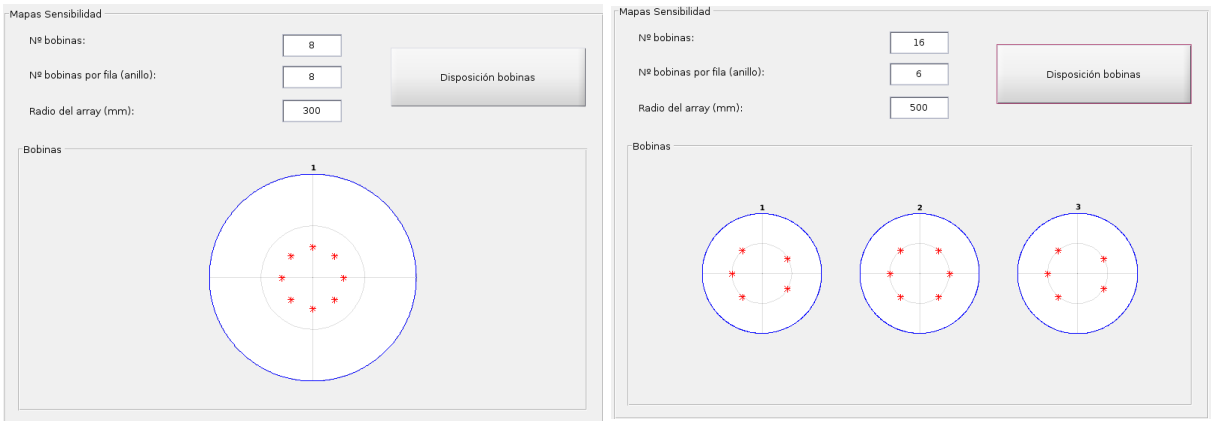


Figura 4.7: Parámetros de configuración en la interfaz

En el momento en que se fija el número de bobinas, el número de bobinas por anillo y el radio del *array* se puede ver un esquema de la colocación de las bobinas pulsando el botón *Disposición bobinas*. En la figura 4.8 se muestran dos ejemplos. Como se puede observar esta representación es en dos dimensiones, dando lugar a una gráfica por anillo, de forma que serían cortes de la figura 3D que representa el *array* de bobinas (ver figura 2.6).



(a) Ejemplo disposición bobinas para `nc = 8`, `c_per_row = 8` y `rbody_mm = 300`

(b) Ejemplo disposición bobinas para `nc = 16`, `c_per_row = 6` y `rbody_mm = 500`

Figura 4.8: Ejemplos de representación de la disposición de las bobinas en la interfaz

Cuando se representa la función de inhomogeneidad del campo, se habilita el panel *SENSE*, representado en la figura 4.9, para seleccionar el factor de aceleración que se desea simular.

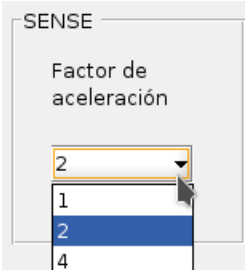


Figura 4.9: Panel de selección del factor de aceleración en la interfaz

Una vez seleccionado este parámetro se activa el botón *Simular* y cuando el usuario lo pulse se podrá observar el espacio k simulado, su correspondiente imagen de resonancia magnética, las imágenes con FOV reducido para hasta 4 bobinas (por ejemplo para 16 bobinas se muestran las imágenes que simuladas para las bobinas número 1, 5, 9 y 13) y la imagen reconstruida con SENSE en el panel de *Simulación* como se muestra en la figura 4.10.

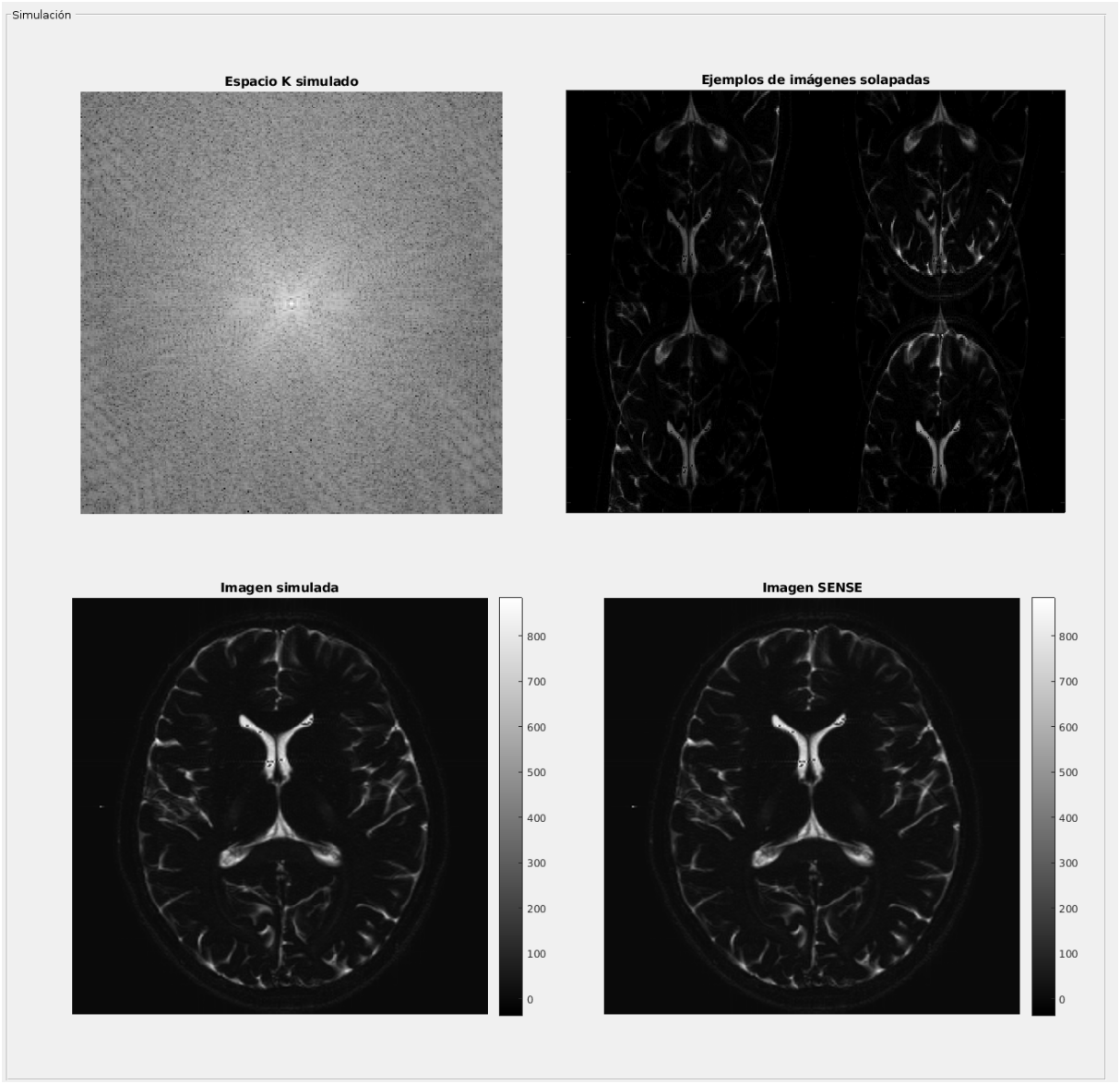


Figura 4.10: Panel de simulación en la interfaz

Capítulo **5**

Resultados

Una vez explicados los métodos utilizados en este estudio, en este capítulo se exponen los resultados obtenidos, tanto a nivel de validación visual de las imágenes de resonancia magnética y los mapas de sensibilidad como a nivel de evaluación de las medidas de rendimiento del algoritmo implementado. La parte más crítica y a la vez la más compleja, ha sido la resolución de los sistemas para la reconstrucción con SENSE, por lo que se va a realizar una comparativa de distintos métodos y alternativas implementadas para dar solución al problema y se expondrán los motivos que han llevado a la elección del método actualmente implementado en el simulador.

5.1 Metodología de evaluación de los resultados

Para evaluar los resultados de las imágenes de resonancia magnética se toma el modelo anatómico de un *slice* del cerebro formado por las imágenes de tiempo de relajación longitudinal (T_1), tiempo de relajación transversal (T_2) y densidad de protones (PD) de los diferentes tejidos del mismo. Como se explicó en el capítulo 2, T_1 es el tiempo necesario para que la componente longitudinal del vector de magnetización del sistema de espines, M_z, regrese a su estado original; T_2 se corresponde con el tiempo de relajación de la componente transversal M_{xy} y PD es la concentración de núcleos de hidrógeno, es decir, de espines en el tejido. Estos datos son obtenidos de la base de datos disponible en el grupo en el que se realiza el trabajo, el LPI.

Por su parte, para validar los mapas de sensibilidad simulados, se comparan los resultados obtenidos con el simulador con los ofrecidos por MATLAB.

La evaluación de los resultados se llevará a cabo desde tres puntos de vista:

I. **A nivel visual:** se comprueban las imágenes de resonancia magnética obtenidas en las bobinas con FOV reducido, por lo que tendrán tamaños de $256 \times 256/L$ píxeles y las imágenes reconstruidas con SENSE, que tendrán un tamaño de 256×256 píxeles. Además, para el caso de los mapas de sensibilidad, una vez esta funcionalidad se ha integrado con el simulador, las dimensiones de estos son iguales que las de las imágenes finales. En un paso anterior, la validación de la simulación de los mapas de sensibilidad se ha realizado con casos 3D, ya que esta opción es posible tal y como está implementado.

II. **A nivel cuantitativo:** se validan los valores de los píxeles de las imágenes o del espacio k con los obtenidos con MATLAB para el mismo algoritmo y se comprueban las dife-

rencias entre la imagen original tras la simulación de la secuencia de pulsos y la imagen reconstruida con SENSE. Para ello, se calculan los histogramas asociados a los errores relativos.

III. **A nivel de rendimiento:** se evalúan los tiempos de ejecución para distintos algoritmos de resolución de sistemas (inversión de matrices con varios métodos y método del gradiente conjugado). Con menor nivel de detalle, se esboza cuál de estas opciones requiere menos memoria al sistema y, a su vez, es más simple.

5.2 Evaluación visual de los resultados

A continuación se analizan de forma visual las imágenes de resonancia magnética obtenidas y los mapas de sensibilidad para distintas configuraciones de parámetros.

En la figura 5.1 se muestra la imagen de resonancia magnética obtenida tras la simulación de la secuencia de pulsos implementada en el simulador del que parte este trabajo.

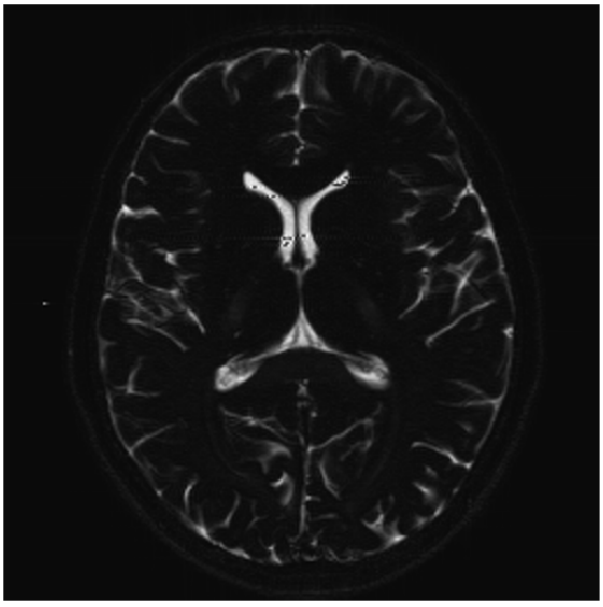
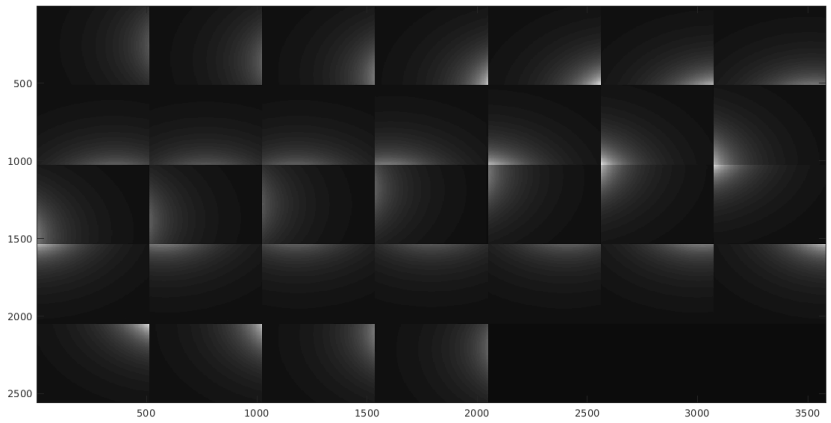


Figura 5.1: Imagen simulada con una única excitación

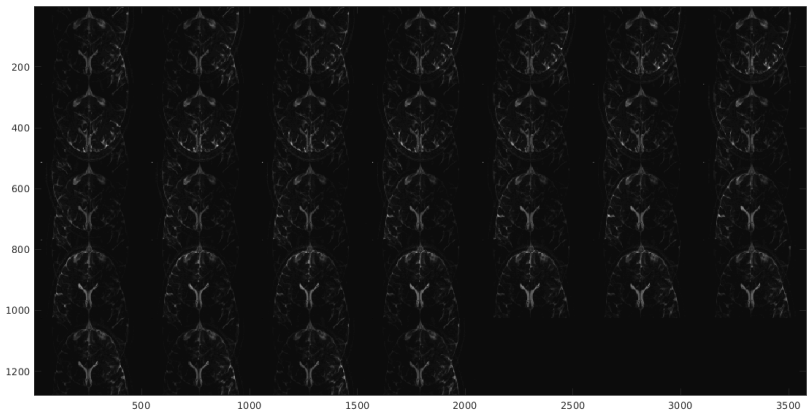
Posteriormente, se ilustran distintas pruebas modificando el número de bobinas, el número de bobinas por anillo y el factor de aceleración.

- En la figura 5.2a, se muestran los mapas de sensibilidad asociados a 32 bobinas, colocando todas ellas en el mismo anillo. Como se puede observar, su distribución es uniforme, coincidiendo la posición de la bobina simulada en el *array* donde la imagen tiene una mayor intensidad de blanco. En este punto el campo magnético es máximo y según la distancia a la bobina crece, el campo magnético disminuye. En la siguiente imagen, figura 5.2b, se

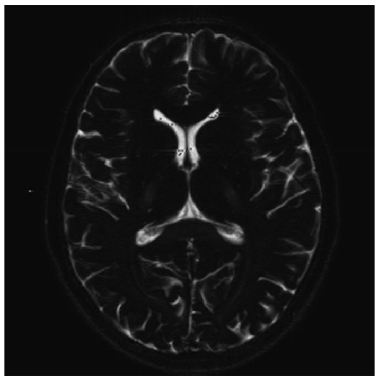
representan las imágenes simuladas para cada bobina aplicando un factor de aceleración $L = 2$. Como se puede ver, estas imágenes sufren los efectos del submuestreo y su dimensión en y es la mitad que en x. Además, su intensidad está condicionada, como era de suponer, por los mapas de sensibilidad de cada bobina, es decir, tienen mayor intensidad en puntos cercanos a la bobina. Finalmente, en la figura 5.2c, se ilustra la imagen reconstruida con SENSE. Como los mapas de sensibilidad simulados no contienen ruido, la reconstrucción es idéntica a la imagen de partida.



(a) Mapas de sensibilidad



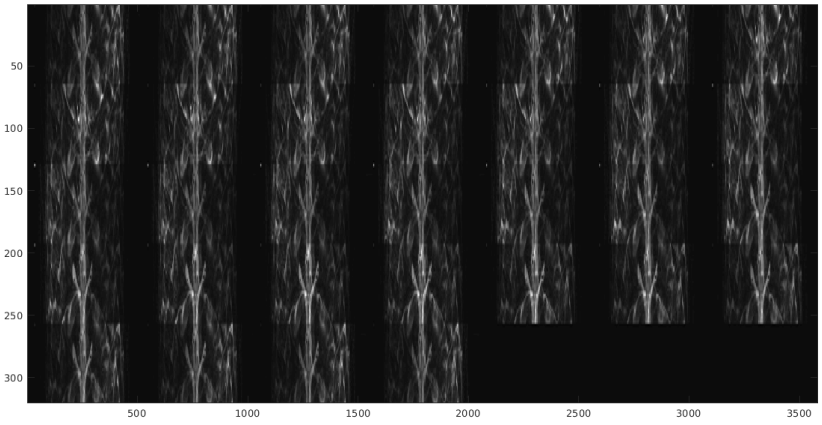
(b) Imágenes simuladas para cada bobina



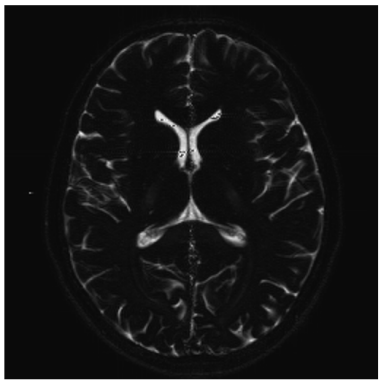
(c) Imagen reconstruida mediante SENSE

Figura 5.2: Prueba 1. Imágenes para los parámetros: `nc=32`, `c_per_row=32`, `rx=1`, `rz=5`, `rbody_mm=300` y `L=2`

- Con la siguiente prueba se va a ilustrar qué ocurre al aumentar el factor de aceleración y se va a comparar con el ejemplo anterior. En este caso, las imágenes en las bobinas (figura 5.3a) tienen dimensiones de 256×32 píxeles y la reconstrucción ya no es perfecta (figura 5.3b), debido a que la cantidad de datos omitida es muy grande y hay errores de precisión numéricos, por ejemplo en los mapas de sensibilidad. Estos errores son arrastrados y aumentan al realizar el cálculo de las matrices inversas para la resolución de los sistemas.



(a) Imágenes con efectos de submuestreo simuladas en las bobinas



(b) Imagen reconstruida mediante SENSE

Figura 5.3: Prueba 2. Imágenes para los parámetros: `nc=32`, `c_per_row=32`, `rx=1`, `rz=5`, `rbody_mm=300` y `L=8`

- En este ejemplo se pretende ilustrar el efecto de la distancia de las bobinas al centro del *array*, mediante el parámetro `rbody_mm`. Si se aumenta la distancia disminuye la intensidad de los mapas de sensibilidad, como se puede observar a partir de las figuras 5.2a y 5.4. Se puede dar el caso en el que los mapas de sensibilidad parezcan más intensos al aumentar la distancia. Este efecto visual se debe a que la intensidad representada en las imágenes depende del rango de valores de las imágenes a representar, tomando como color blanco el valor máximo y como negro el mínimo. Los valores de las sensibilidades ahora son más pequeños que para las dos pruebas anteriores, ya que las bobinas están el doble de lejos.

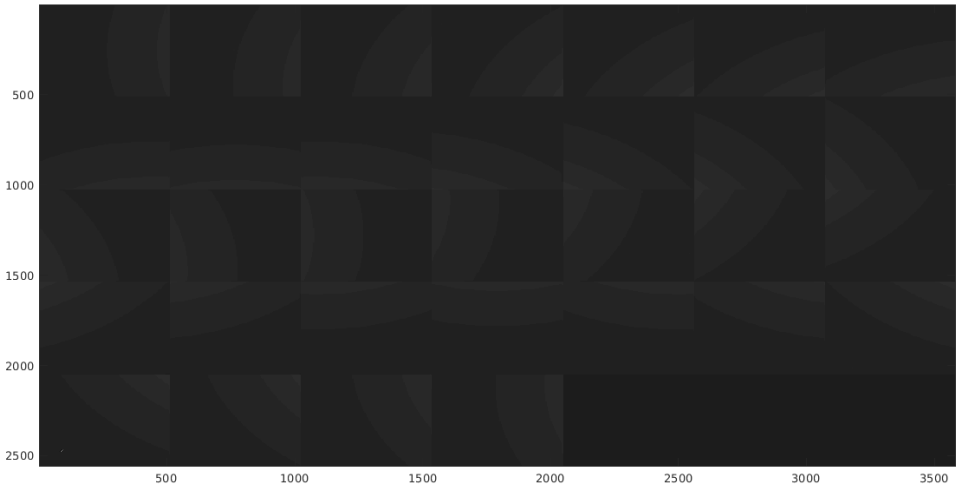
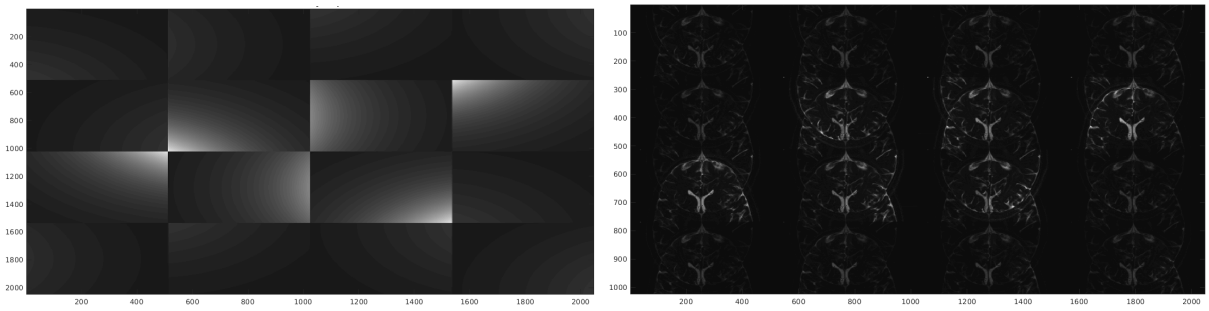


Figura 5.4: Prueba 3. Mapas de sensibilidad simulados con parámetros: `nc=32`, `c_per_row=32`, `rx=1`, `rz=5`, `rbody_mm=600`

- En el caso de que haya varios anillos, las sensibilidades más fuertes son las del anillo central, siendo estas bobinas las que más peso tienen a la hora de llevar a cabo la reconstrucción

de la imagen. A continuación, en las imágenes de la figura 5.5, se muestra un ejemplo en el que hay tres anillos con 5, 6 y 5 bobinas.



(a) Mapas de sensibilidad (b) Imágenes simuladas para cada bobina

Figura 5.5: Prueba 4. Imágenes para los parámetros: `nc=16`, `c_per_row=6`, `rx=1`, `rz=5`, `rbody_mm=300` y `L=2`

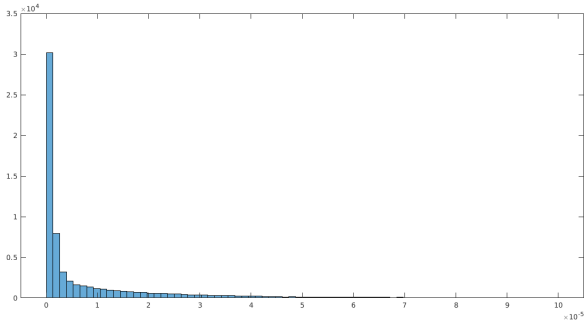
5.3 Medidas de errores en la reconstrucción

Es necesario medir la calidad de la reconstrucción llevada a cabo. Para ello, se puede calcular el error relativo de la imagen final, tomando como valor real la imagen inicial, es decir, las diferencias entre los valores de la imagen reconstruida con SENSE y la imagen obtenida tras la simulación de la secuencia de pulsos. Construyendo el histograma de los errores relativos se puede observar la frecuencia de cada error encontrado. Cuanto menores sean los errores se puede afirmar que mejor es la reconstrucción.

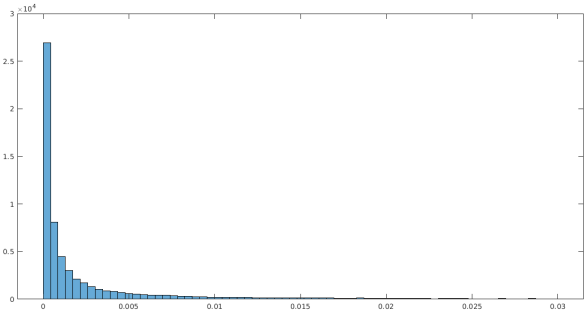
Seguidamente, en la figura 5.6 se muestran los histogramas de los errores relativos para las pruebas realizadas en la sección anterior. Hay que darse cuenta de la diferencia de escalas entre la subfigura 5.6b y el resto. Como se puede observar, los errores más grandes se corresponden a la segunda prueba (figura 5.6b), donde el factor de aceleración toma el valor 8. Este resultado era de esperar ya que es donde se han omitido más datos. Para el resto de experimentos, con factor de aceleración 2, los errores relativos hallados son similares, observándose un comportamiento ligeramente mejor para el caso de la figura 5.6a, ya que al aumentar la distancia de las bobinas al centro del *array* las sensibilidades toman valores menores (figura 5.6c) y al tener menos bobinas dispuestas en varios anillos (figura 5.6d) también disminuye la calidad de la reconstrucción.

5.4 Comparativa de rendimientos

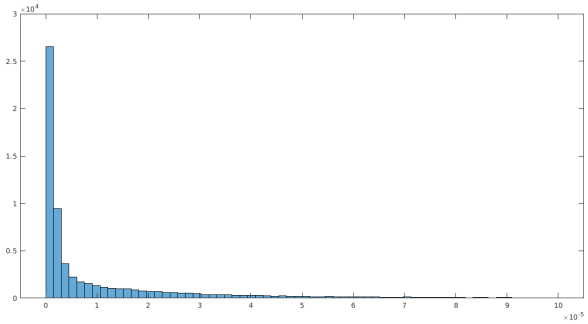
El objetivo de esta sección es realizar una comparativa entre distintos métodos e implementaciones para la resolución de los sistemas lineales que permiten el *unfolding* de los píxeles y la reconstrucción de la imagen con FOV completo. Como se detalla en la función `inverseMatrixM()` del Apéndice D, en el simulador se ha implementado la inversión de matrices hermíticas en



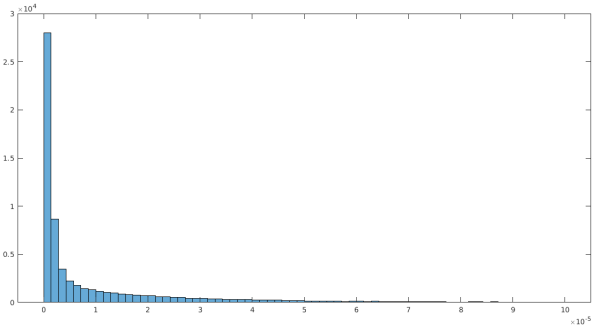
(a) Prueba 1. Simulación con parámetros: `nc=32`, `c_per_row=32`, `rx=1`, `rz=5`, `rbody_mm=300` y `L=2`



(b) Prueba 2. Simulación con parámetros: `nc=32`, `c_per_row=32`, `rx=1`, `rz=5`, `rbody_mm=300` y `L=8`



(c) Prueba 3. Simulación con parámetros: `nc=32`, `c_per_row=32`, `rx=1`, `rz=5`, `rbody_mm=600` y `L=2`



(d) Prueba 4. Simulación con parámetros: `nc=16`, `c_per_row=6`, `rx=1`, `rz=5`, `rbody_mm=300` y `L=2`

Figura 5.6: Histogramas errores relativos de las imágenes reconstruidas con SENSE

CPU para resolver los sistemas mediante mínimos cuadrados haciendo uso de las librerías GSL y OpenMP (ver desarrollo matemático en el Apéndice C), de forma que se utiliza paralelismo multihilo en C++, aunque a lo largo del desarrollo de este TFG se ha buscado resolver los sistemas para la reconstrucción de imágenes de resonancia magnética con la GPU (mediante OpenCL), ya que a priori parece la mejor opción, puesto que la GPU tiene una arquitectura *multicore* óptima para cálculos paralelos (ver capítulo 3).

Debido a la dificultad subyacente a la inversión de matrices complejas en GPU con OpenCL, se estudió la posibilidad de resolver los sistemas haciendo uso de métodos optimizados para la resolución de sistemas como el método del descenso de gradiente, del gradiente conjugado (CG) o del gradiente conjugado precondicionado. Tras realizar pruebas sobre la convergencia de estos métodos para los sistemas del problema, se optó por implementar el método del gradiente conjugado en OpenCL, evitando así esta inversión de matrices para la resolución de los sistemas. Es un método iterativo por lo que cada sistema se resuelve de forma secuencial siguiendo los siguientes pasos para la resolución de un sistema real del tipo $Ax = b$:

1. Inicializar x para la iteración k-ésima (x_k) con valores lo más próximos posibles a la solución.

2. Inicializar los siguientes parámetros antes de comenzar a iterar:

   2.1) $A_p = A^T A$

   2.2) $b = A^T b$

   2.3) $r_k = A_p x_k - b$

   2.4) $p_k = r_k$

3. Iterar hasta la convergencia, es decir, hasta que la norma del vector diferencia de x_k y x_{k+1} sea un cierto número de órdenes de magnitud menor que la norma del vector que contiene la solución de la iteración (x_k), o el número de iteraciones sea suficientemente grande. Repetir los siguientes pasos:

   3.1) $r_k = A_p x_k - b$

   3.2) $\alpha_k = \frac{r_k^T p_k}{p_k^T A_p p_k}$

   3.3) $x_{k+1} = x_k - \alpha_k r_k$

   3.4) $r_{k+1} = A_p x_{k+1} - b$

   3.5) $p_k = r_{k+1} - p_k \frac{r_{k+1}^T A_p p_k}{p_k^T A_p p_k}$

   3.6) $x_k = x_{k+1}$

Aunque el método sea iterativo, es decir, secuencial para cada sistema, se puede paralelizar la resolución de todos los sistemas (recordar que hay tantos sistemas como píxeles solapados en las imágenes con FOV reducido, es decir $Ny \times Nx/L$). El problema de este método es que no todos los sistemas convergen en el mismo número de iteraciones, siendo elevado en algunos casos. Si se quiere paralelizar la resolución de todos ellos es necesario realizar tantas iteraciones como las necesarias para el sistema cuya convergencia sea la más lenta, ya que este sería el

limitante. Además, la implementación de cualquiera de estos métodos supera en complejidad y número de objetos de memoria necesarios al resto de propuestas aquí explicadas, ocupando por tanto más memoria en el equipo en el que se ejecute el simulador. Como ventaja se puede destacar que el uso de la GPU es mayor al ejecutar el simulador que implementa esta solución.

Otra alternativa para la resolución de los sistemas relativamente sencilla, sin cálculo de matrices inversas y realizando el cómputo en GPU, es la descomposición o factorización QR de las matrices utilizando reflexiones de Householder. Esta opción destaca por su estabilidad numérica, muy superior a otros métodos. A partir de una matriz A real, cuadrada o rectangular, se obtienen dos matrices. La primera, Q es ortogonal, de forma que su transpuesta es igual a su inversa, y la segunda, R, es triangular superior. Si se aplican estas propiedades, la resolución de los sistemas es muy sencilla, mediante eliminación gaussiana con sustitución hacia atrás y no requiere invertir matrices:

$$Ax = b \Rightarrow Q \cdot Rx = b \Rightarrow Rx = Q^H b \tag{5.1}$$

Los pasos para hallar las matrices Q y R son los siguientes, denotando las columnas de la matriz A como c_k y las de matriz R como c'_k, siendo k la columna k-ésima.

1. Encontrar el vector u_0 que refleja la columna c_0 en c'_0.
2. Transformar cada columna de A con la reflexión de u_0.
3. Encontrar el vector u_1 que refleja la columna c_1 en c'_1.
4. Transformar cada columna de A con la reflexión de u_1.
5. Construir la matriz de Householder P_k para cada vector u_k.
6. Repetir los pasos del 3 al 5 para cada columna hasta la $k-1$.
7. Multiplicar las matrices de Householder para formar Q.

En cuanto a la implementación con OpenCL [26], se genera un *work-item* para cada columna de la matriz a descomponer. Estos *items* pertenecen al mismo *work-group*, de forma que se puede utilizar una barrera para sincronizar el acceso a la memoria local y global de todos ellos. Por tanto, se está utilizando el modelo de paralelismo de tareas de OpenCL para la implementación de este algoritmo. Posteriormente, como los sistemas a resolver son independientes entre sí, se podría aplicar paralelismo de datos para resolver todos ellos paralelamente. Respecto a los objetos de memoria de tipo *buffer*, son necesarios al menos 4, uno para las matrices a descomponer, para las matrices Q, la matrices P y para los productos de matrices P. Sin embargo, el principal problema de este algoritmo es que los tiempos de ejecución obtenidos para la factorización QR de una única matriz son generalmente superiores a la inversión de todas las matrices, debido a las barreras necesarias para la sincronización. Además falta de añadir el tiempo requerido para la resolución del sistema como tal, una vez factorizada la matriz, mediante algoritmos de sustitución hacia atrás. Entonces, aunque se paralelice el algoritmo para todas las matrices, el tiempo seguirá siendo alto, por lo que este método fue descartado para el simulador de este trabajo.

Respecto a estos métodos alternativos, hay que tener en cuenta que para utilizarlos es necesario reformular el problema con aritmética real, lo que supone cierta cantidad de cómputo a mayores respecto al método elegido.

Nº bobinas	Bobinas por anillo	Factor de aceleración	Inversión de matrices (GSL y OpenMP)	Descomposición QR de una matriz (OpenCL)	Simulador método matriz inversa	Simulador método CG
32	32	2	0.032	0.66	1.17	2.34
32	32	4	0.08	0.3	1.26	1.81
32	32	8	0.13	0.19	1.30	2.49[1]
16	16	2	0.032	0.17	1.13	2.24
16	16	4	0.09	0.11	1.26	1.68
4	4	2	0.042	0.019	1.14	2.12

Tabla 5.1: Tiempos medios de ejecución en segundos para 100 simulaciones

En la tabla 5.1 se incluyen distintos escenarios de pruebas para los cuales se ha calculado el tiempo medio de ejecución para 100 simulaciones, en los que se ha tenido en cuenta únicamente el núcleo de la simulación. Estos tiempos pueden variar en función de la carga del sistema en el que se ejecuten las pruebas, en este caso, se ha ejecutado en el servidor *tebas* del LPI. En los experimentos realizados se evalúan diferentes comportamientos al variar el número de bobinas a simular, las bobinas por anillo y el factor de aceleración. Los resultados que se muestran, de izquierda a derecha, son los tiempos medios de ejecución en segundos para la inversión de $Ny \times Nx/L$ matrices de dimensión $(L \times L)$ en CPU utilizando las librerías GSL y OpenMP, los tiempos para la descomposición QR mediante reflexiones de Householder en OpenCL de una matriz de dimensiones $(2n_C \times 2L)$ (debido a la reformulación en aritmética real) y los tiempos de ejecución de todo el núcleo de simulación empleando el método de la matriz inversa en CPU y el método del gradiente conjugado en OpenCL con un número fijo de 2200 iteraciones para la convergencia del método.

Tras analizar los resultados, se puede concluir que el simulador implementado con el método matriz inversa es el que presenta menores tiempos de ejecución globales para los distintos escenarios, frente al método del gradiente conjugado. Además si se comparan los tiempos requeridos para la inversión de las matrices frente a los necesarios para descomponer una única matriz, son en general menores, excepto en el caso en que las matrices sean muy pequeñas con dimensiones de (8×4), para el caso real (escenario para 4 bobinas y factor de aceleración 2). Por lo tanto, a pesar de que a priori hacer uso de la CPU para realizar este tipo de cálculos podría no parecer lo más adecuado, se ha comprobado su superioridad frente al resto de alternativas aquí presentadas.

[1] No se reconstruye la imagen correctamente con 5000 iteraciones

Capítulo **6**

Conclusiones y líneas futuras

El capítulo final del documento tiene como objetivo resaltar las principales conclusiones extraídas de la realización de este TFG, así como indicar las limitaciones presentes en el simulador desarrollado y las posibles líneas de trabajo futuras a partir del mismo.

6.1 Conclusiones

La simulación de imágenes de resonancia magnética resulta de gran utilidad en el área de la docencia y de la investigación. Esta actividad es posible gracias a unas herramientas conocidas como simuladores. La implementación y desarrollo de éstos requiere de conocimientos técnicos, tanto en el ámbito de la MRI como de la programación.

El principal problema asociado a la simulación de secuencias e imágenes es que se precisan múltiples operaciones para dar lugar a cada uno de los puntos de la imagen a sintetizar. La consecuencia inmediata de este hecho son tiempos de ejecución elevados, debido al elevado coste computacional que implican estas operaciones. Sin embargo, este problema es intrínsecamente paralelo, ya que se realizan las mismas operaciones en cada píxel. Como solución se puede recurrir a las técnicas de computación paralela, aumentando la eficiencia y reduciendo los tiempos de ejecución drásticamente.

A través de la computación paralela anteriormente mencionada, junto con las técnicas de adquisición en paralelo, populares en el ámbito de la investigación de la MRI en los últimos años por su reducción de los tiempos de adquisición, se ha planteado la implementación de un simulador de síntesis de imágenes de resonancia magnética con varias bobinas receptoras y adquisición en paralelo de las imágenes, empleando el algoritmo de reconstrucción SENSE. Principalmente se ha utilizado programación en GPU, mediante el uso de la API y el lenguaje de programación de OpenCL, con el objetivo de reducir los tiempos de ejecución. Para ello se ha tomado como punto de partida un simulador de MRI en esta misma tecnología, que lleva a cabo la simulación de la secuencia de pulsos EPI con una única bobina, además de varios scripts en MATLAB, para la simulación de las bobinas receptoras y la reconstrucción con SENSE.

Con relación a lo anterior, se identifica un doble propósito que se ha completado con éxito. Por un lado, se ha dotado de mayor funcionalidad al simulador descrito, integrando en él los nuevos requisitos que se desean satisfacer. Por otro lado, se ha implementado de forma que los

tiempos de ejecución se reduzcan al mínimo posible para efectuar la simulación. Adicionalmente, se ha mejorado la experiencia de usuario al interaccionar con el simulador, mediante el diseño e implementación de una interfaz gráfica intuitiva, que permite la configuración de los parámetros de la simulación, así como la visualización de los resultados. Por lo tanto, a partir de este desarrollo, en el que se han demostrado las ventajas que proporciona la aplicación de la computación paralela a este tipo de simulaciones, se brinda un amplio abanico de posibilidades para nuevas simulaciones de secuencias y técnicas en esta línea.

6.2 Limitaciones y líneas futuras

6.2.1 Limitaciones

La implementación realizada adolece de ciertas limitaciones como, por ejemplo, que los factores de aceleración posibles deben ser divisores de la dimensión vertical de la imagen. Al mismo tiempo, para factores de aceleración muy grandes, por ejemplo para un factor de 8 para una imagen de 256×256 píxeles, no se asegura la reconstrucción, aunque teóricamente es posible siempre que el número efectivo de bobinas en la dirección de codificación de fase sea mayor que el factor de aceleración. Adicionalmente, otra causa de este suceso son los errores de precisión en los distintos cálculos realizados, principalmente en la simulación de las sensibilidades de las bobinas, ya que tienen órdenes de magnitud muy pequeños (en torno a 10^{-5}), los cuales son acarreados a lo largo de la simulación y producen cierta inestabilidad numérica en la reconstrucción de la imagen final en estos casos. Por otra parte, no se ha tenido en cuenta el ruido inherente a las bobinas ni la correlación espacial entre ellas, por lo que estos efectos no se muestran en las imágenes reconstruidas.

6.2.2 Líneas futuras

A partir del trabajo presentado y las limitaciones expuestas en el apartado anterior, es posible añadir ciertas mejoras a la implementación del simulador realizada, con el fin de permitir simulaciones aún más realistas y versátiles.

En primer lugar, sería interesante otorgar al simulador la capacidad de establecer cualquier factor de aceleración, incluso números no enteros, en vista de en los escáneres reales sucede de esta manera.

Otra línea de investigación está relacionada con la simulación de las bobinas y su disposición en el espacio, ya que actualmente las posiciones son fijas en anillos en torno a un cilindro que simula el array de elementos receptores, en función de su cardinal. En futuros estudios se podrían simular múltiples configuraciones para analizar diferentes comportamientos.

Del mismo modo, para que las imágenes sintéticas sean lo más realistas posible, habría que considerar el ruido de las bobinas receptoras simuladas. Este hecho permitiría ilustrar las perturbaciones sufridas en cada punto de la imagen reconstruida en función de la configuración de las bobinas y la reducción en la SNR causada por estos efectos no deseados, pero presentes

en los entornos reales.

Bibliografía

[1] Z.-P. Liang y P. C. Lauterbur. *Principles of Magnetic Resonance Imaging*. IEEE, 1999.

[2] R. R. Edelman. The history of MR imaging as seen through the pages of Radiology. *Radiology*, 273(2S):S181-S200, noviembre de 2014.

[3] J. F. Glockner, H. H. Hu, D. W. Stanley, L. Angelos y K. King. Parallel MR imaging: a user's guide. *RadioGraphics*, 25(5):1279-1297, septiembre de 2005.

[4] K. P. Pruessmann, M. Weiger, M. B. Scheidegger y P. Boesiger. SENSE: sensitivity encoding for fast MRI. *Magnetic Resonance in Medicine*, 42(5):952-962, noviembre de 1999.

[5] E. M. Sáez. *Síntesis de imagen de resonancia magnética mediante GPU*. Universidad de Valladolid. 2017.

[6] The MathWorks - MATLAB and Simulink for Technical Computing. Website. Último acceso mayo 2018. `http://www.mathworks.com/`.

[7] Open Innovation Community - Eclipse IDE | The Eclipse Foundation. Website. Último acceso mayo 2018. `https://eclipse.org/`.

[8] The LaTeXProject. Website. Último acceso mayo 2018. `https://www.latex-project.org/`.

[9] Astah - Software Design Tools for Agile teams with UML, ER Diagram, Flowchart, Mindmap and More. Website. Último acceso junio 2018. `http://astah.net/`.

[10] M. A. Jacobs, T. S. Ibrahim y R. Ouwerkerk. MR imaging: brief overview and emerging applications. *RadioGraphics*, 27(4):1213-1229, julio de 2007.

[11] R. A. Pooley. Fundamental physics of MR imaging. *RadioGraphics*, 25(4):1087-1099, julio de 2005.

[12] L. Hanson y T. Groth. *Introduction to Magnetic Resonance Imaging Techniques*. 2009.

[13] M. A. Bernstein, K. F. King y X. J. Zhou. *Handbook of MRI Pulse Sequences*. Academic Press, 2004, páginas 522-531.

[14] K. P. Pruessmann. Encoding and reconstruction in parallel MRI. *NMR in Biomedicine*, 19(3):288-299, 2006.

[15] MRXCAT v1.3. Website. Último acceso junio 2018. `http://www.biomed.ee.ethz.ch/mrxcat.html`.

[16] J. C. Simpson, J. E. Lane, C. D. Immer y R. C. Youngquist. Simple Analytic and Expressions for the Magnetic and Field of a Circular and Current Loop. Informe técnico GCN-00-26, NASA Center for Aerospace Information (CASI), 2001. https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20140002333.pdf.

[17] L. Wissmann, C. Santelli, W. P. Segars y S. Kozerke. MRXCAT: realistic numerical phantoms for cardiovascular magnetic resonance. *Journal of Cardiovascular Magnetic Resonance*, 16(1), agosto de 2014.

[18] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry y D. Schaa. *Heterogeneous Computing with OpenCL, Second Edition: Revised OpenCL 1.2 Edition*. Morgan Kaufmann, 2012.

[19] A. Munshi, B. Gaster, T. G. Mattson, J. Fung y D. Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 2011.

[20] ¿Qué es la computación acelerada por GPU?. Website. Último acceso junio 2018. http://la.nvidia.com/object/what-is-gpu-computing-la.html.

[21] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone y J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879-899, mayo de 2008.

[22] F. Simmross. Introducción a la programación heterogénea CPU-GPU en OpenCL. Informe técnico, Universidad de Valladolid, 2015.

[23] Khronos OpenCL Working Group. *The OpenCL Specification*. Editado por A. Munshi. Versión 1.2. https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf. 2012.

[24] R. S. Pressman. *Ingeniería de Software (Spanish Edition)*. McGraw-Hill Interamericana Editores S.A. de C.V., 2010.

[25] E. M. Sáez, D. T. Fernández, R. de Luis García, F. S. Wattenberg y C. A. López. *Implementación Paralela de Secuencias de Resonancia Magnética mediante Programación en GPU*. CASEIB. Bilbao, 2017.

[26] M. Scarpino. *OpenCL in Action: How to Accelerate Graphics and Computations*. Manning Publications, 2011.

[27] L. Råde y B. Westergren. *Mathematics Handbook for Science and Engineering*. Springer Berlin Heidelberg, 2004, páginas 90-117.

[28] GSL - GNU Scientific Library. Website. Último acceso junio 2018. https://www.gnu.org/software/gsl.

[29] OpenMP. Website. Último acceso junio 2018. https://www.openmp.org.

[30] clFFT 2.0 - OpenCL Fast Fourier Transforms (FFTs). Website. Último acceso junio 2018. http://clmathlibraries.github.io/clFFT/.

Apéndice ***A***

Coordenadas cilíndricas

El sistema de coordenadas cilíndricas es muy conveniente en aquellos casos en que se tratan problemas que tienen simetría de tipo cilíndrico o azimutal. Se trata de una versión en tres dimensiones de las coordenadas polares de la geometría analítica plana.

Los rangos de variación de las tres coordenadas son:

$$0 < \rho < \infty \qquad 0 < \varphi < 2\pi \qquad -\infty < z < \infty$$

La relación con el sistema de coordenadas cartesiano es la siguiente:

$$x = \rho \cos \varphi \tag{A.1}$$
$$y = \rho \sin \varphi \tag{A.2}$$
$$z = z \tag{A.3}$$

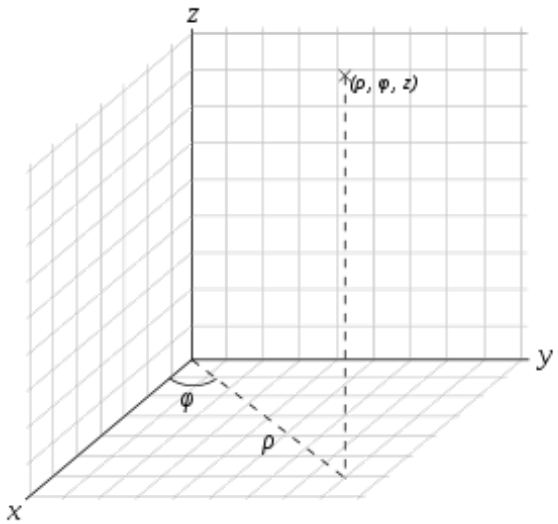


Figura A.1: Coordenadas cilíndricas y ejes cartesianos relacionados.

Apéndice ***B***

Rotaciones básicas en 3 dimensiones

Las siguientes matrices de rotación realizan rotaciones de vectores alrededor de los ejes x, y, o z, en el espacio de tres dimensiones. Cada una de estas tres rotaciones básicas se realiza en sentido antihorario alrededor del eje y considerando un sistema de coordenadas con la regla de la mano derecha.

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin(\theta) & \cos(\theta) \end{pmatrix} \tag{B.1}$$

$$R_y(\theta) = \begin{pmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta & 0 & \cos(\theta) \end{pmatrix} \tag{B.2}$$

$$R_z(\theta) = \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{B.3}$$

Apéndice **C**

Diagonalización e inversión de matrices hermíticas

Una matriz hermítica es una matriz cuadrada de elementos complejos que tiene la característica de ser igual a su propia traspuesta conjugada, es decir $A = A^H$.

En álgebra lineal, el teorema espectral expresa las condiciones bajo las cuales un operador o una matriz pueden ser diagonalizados (es decir, representadas como una matriz diagonal en alguna base). Se puede aplicar a matrices hermíticas dando lugar a la diagonalización de la misma, [27]:

$$A = UDU^H, \tag{C.1}$$

siendo U una matriz unitaria compuesta por los autovectores de A y D una matriz diagonal que contiene los autovalores de A.

Esta propiedad se puede utilizar para el cálculo de la matriz inversa, teniendo en cuenta que U es unitaria y se cumple que $U^{-1} = U^H$:

$$A^{-1} = (UDU^H)^{-1} = (U^H)^{-1}(UD)^{-1} = (U^H)^{-1}D^{-1}U^{-1} = UD^{-1}U^H \tag{C.2}$$

Como D es una matriz diagonal de la forma,

$$D = \begin{pmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ 0 & 0 & \dots & \lambda_n \end{pmatrix} \tag{C.3}$$

su inversa es muy sencilla de calcular, siendo

$$D^{-1} = \begin{pmatrix} \frac{1}{\lambda_1} & 0 & \dots & 0 \\ 0 & \frac{1}{\lambda_2} & \dots & 0 \\ 0 & 0 & \dots & \frac{1}{\lambda_n} \end{pmatrix} \tag{C.4}$$

Apéndice ***D***

Estructura del código del simulador

A continuación se describe detalladamente el contenido de los ficheros pertenecientes al código del simulador implementado:

- *Simumri+SENSE.cpp:* Contiene la función `main()` y en él se declaran los parámetros de entrada que el usuario introduce a través de la interfaz gráfica, o directamente en el código si no se hace uso de esta, y otros parámetros de configuración. La estructura que sigue el flujo de programa es la siguiente: se establecen los valores de los parámetros anteriores, se instancia un objeto de la clase *Simulator* y otro de clase *Gpu*. A continuación se inicializan los parámetros de OpenCL a través del método `initOpenCL()` definido en la clase *Gpu*. Posteriormente se suceden distintas llamadas a los métodos de la clase *Simulator* que permiten la simulación de la secuencia de pulsos EPI. Esta parte no se va a comentar en mayor detalle pues no es el trabajo desarrollado en este TFG. Una vez completa la simulación anterior, se instancia un objeto de la clase *Sense* mediante la llamada a su constructor, que recibe como argumentos de entrada varios parámetros de configuración. Seguidamente se suceden llamadas a los métodos `initSENSE(), setLinearSystems(), inverseMatrixM()` y `applySENSE()` mediante el objeto previamente instanciado para llevar a cabo la simulación de la adquisición de la imagen de resonancia magnética en paralelo y su posterior reconstrucción con el método SENSE. Antes de finalizar la ejecución del programa se liberan los recursos en memoria previamente reservados mediante asignación dinámica de memoria y los recursos de OpenCL haciendo uso del método `releaseOpenCL()` de la clase *Gpu*. No es necesario invocar a los destructores de las clases para liberar la memoria asignada a los objetos por los constructores ya que se llaman automáticamente cuando se abandona el ámbito en el que fueron definidos.
- *Simulator.cpp:* Implementación del constructor de la clase. Debido a que tiene muchos métodos y es muy extenso, la implementación de los métodos de esta clase se ha divido en diversos ficheros .cpp.
- *Simulator.h:* Definición de la clase *Simulator* con sus atributos y métodos.
- *Gpu.cpp:* Implementación de los métodos de la clase y su constructor y destructor.
- *Gpu.h:* Definición de la clase *Gpu*. Los atributos se han considerado públicos por simplicidad y son los parámetros OpenCL. Los métodos realizan las siguientes operaciones:
    - `Gpu()`, constructor. Se utiliza para crear instancias de la clase

- `~Gpu()`, destructor. Elimina los recursos asociados a un objeto de la clase.
- `convertToString()`, convierte los ficheros que contienen los *kernels* en una cadena de caracteres (*string*).
- `initOpenCL()`, implementa los pasos del 1 al 6 descritos en el apartado 3.2.2 utilizando las funciones definidas en la API de OpenCL.
- `releaseOpenCL()`, libera los recursos asociados al objeto de programa, la cola de comandos y el contexto.

- *Sense.cpp*: Implementación de los métodos de la clase y su constructor y destructor.

- *Sense.h*: Definición de la clase *Sense*. En este caso los atributos son privados, es decir, el acceso a los mismos solo podrá darse desde el interior de la clase que los posee. Esta propiedad se conoce como encapsulamiento y es típica de la orientación a objetos [24].

  - Descripción de los atributos:
    - `L`, es el factor de aceleración de SENSE que anteriormente en este documento se ha denotado como R. Según cada autor se denota con una u otra letra.
    - `FOV_L`, es el tamaño en la dimensión vertical y de las imágenes con FOV reducido, es decir, el número de filas si se considera la imagen como una matriz bidimensional donde cada valor de la misma es el valor de un píxel. Se define como el tamaño original de la imagen en esta dirección (número de filas, Ny) dividido por el factor de aceleración. Se utiliza para la reconstrucción.
    - `M`, es un *array* que contiene las matrices resultado de realizar la operación $C^H \cdot C$ de la ecuación 2.7 necesarias para la reconstrucción. Representa todas las matrices de todos los sistemas por lo que las dimensiones vistas como un *array* multidimensional serían $(L \times L \times FOV\_L * Nx)$, siendo cada una de las matrices de tamaño $(L \times L)$. Realmente en el código se considera como un vector de longitud $(L * L * FOV\_L * Nx)$.
    - `bufS`, objeto de memoria de OpenCL de tipo *buffer*. Contiene las matrices C definidas en la teoría de SENSE para todos los sistemas. A pesar de ser unidimensional, la estructura que se consigue accediendo a los índices adecuados es la que se muestra en la figura D.1, siendo nc el número de bobinas. Imaginariamente es como si estuviesen las matrices apiladas una tras otra en la tercera dimensión.

    - `bufM`, objeto de memoria de OpenCL de tipo *buffer*. Contiene los mismos datos que el *array* `M` anterior, sin embargo, es necesario para poder transferir los datos del *host* a la GPU. Es unidimensional al ser un *buffer*, pero el acceso a los índices se ha definido análogamente al `bufS`.
    - `bufY`, objeto de memoria de OpenCL de tipo *buffer*. Su contenido es el de los vectores F de la ecuación 2.6 para todos los sistemas, es decir, los píxeles solapados a partir de los cuales se calculan los píxeles de la imagen con FOV completo. Su estructura es análoga a los *buffers* anteriores, pero en este caso las dimensiones son $(nc \times 1 \times FOV\_L * Nx)$.

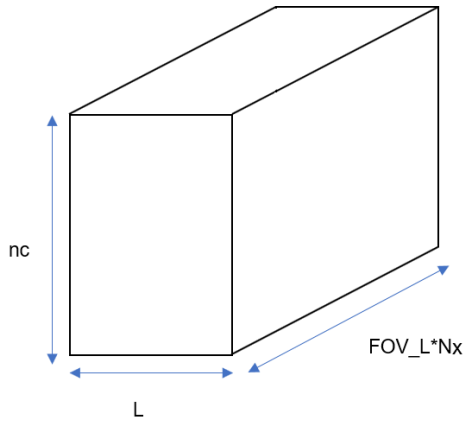


Figura D.1: Estructura de objeto de memoria `bufS`

- `mymap`, objeto de tipo *Mapas*. Tiene la información relacionada con los mapas de sensibilidad que se van a utilizar para llevar a cabo la reconstrucción. Al ser una instancia de una clase propia, tiene asociados los atributos y métodos de la misma.
- `myimag`, objeto de tipo *Imagen*. Engloba lo relativo a la imagen de partida, las imágenes de las bobinas a partir de las cuales se va a reconstruir, así como la imagen reconstruida.
- `myespaciok`, objeto de tipo *EspacioK*. Similar al atributo anterior pero con los datos relativos al espacio transformado de las imágenes, es decir, al espacio k.

- Descripción de los métodos:
  - `Sense()`, constructor de la clase. Recibe como argumentos de entrada un objeto de tipo *Gpu* que permite el uso del contexto de OpenCL definido anteriormente; las dimensiones `Ny` (número de filas) y `Nx` (número de columnas) de la matriz que contiene los valores de los píxeles de la imagen con FOV completo; el número de bobinas, `nc`; el factor de aceleración de SENSE, `L`; y a continuación una serie de parámetros para la simulación de los mapas de sensibilidad de las bobinas: el tamaño de vóxel, `vol_size`; el *array* de *bounding box*, `bbox`; las resoluciones en milímetros tanto en x (igual para y) como en z, `r_x`, `r_z`; una tolerancia usada en los cálculos, `tol`; la distancia del centro de la imagen a los centros de las bobinas `rbody_mm` y el número de bobinas deseado por anillo `c_per_row`. Al instanciar el objeto de tipo *Sense* se inicializan sus atributos `L` y `FOV_L`, se asigna la memoria al *array* `M`, se crean los objetos de memoria de OpenCL `bufS`, `bufM` y `bufY` mediante la función `clCreateBuffer()` de la API de OpenCL y se crean las instancias de las clases correspondientes dando lugar a los atributos `mymap`, `myimag` y `myespaciok` invocando a sus respectivos constructores.
  - `~Sense()`, destructor. Elimina el objeto *Sense* al salir del ámbito donde fue definido. Además invoca a los destructores de las clases *Mapas*, *Imagen* y *EspacioK*.
  - `initSENSE()`, este método se encarga de todas las operaciones previas necesarias a la aplicación del algoritmo SENSE como tal. El atributo `mymap`, objeto de la clase *Mapas*, invoca a los métodos `setCoordinats_mm()` y `calculateCoilMaps()`

para definir las coordenadas de vóxel en milímetros y calcular los mapas de sensibilidad para cada bobina de forma paralela. Se obtiene el espacio k calculado por el simulador de partida separado en dos *buffers*, uno para la parte real y otro para la imaginaria y se combinan en un único objeto de memoria, esto se lleva a cabo invocando al método `getEspacioK()` del objeto `myespaciok`. Se transforma al espacio imagen aplicando la IFFT2D (`myespaciok->applyIFFT2D()`) y se calculan las imágenes en las bobinas(`myimag->calculeImCoils()`) haciendo uso de los mapas de sensibilidad (`mymap->getSenBuffer()`). Posteriormente se realiza la FFT2D (`myimag->applyFFT2D()`) para obtener el espacio k, el cual se submuestrea en razón al factor de aceleración (`myespaciok->undersample()`). Las imágenes en las bobinas con FOV reducido se obtienen aplicando la IFFT2D a los datos anteriores.

- `setLinearSystems()`, establece los sistemas que plantea la teoría de SENSE, ecuación 2.6. Se van a inicializar $FOV\_L * Nx$ matrices C y el mismo número de matrices F. Además como la resolución se va a llevar a cabo con la matriz de *unfolding* de la ecuación 2.8, también es necesario llevar a cabo la operación $C^H C$. Como ya se ha comentado, el objeto de memoria que almacena las matrices C es `bufS`, el que almacena el producto $C^H C$ es `bufM` y F se corresponde con `bufY` en el código. Para realizar esta funcionalidad el método va a crear tres objetos de tipo *kernel* utilizando la función `clCreateKernel()`, apuntará los parámetros de los mismos a los objetos de memoria correspondientes y los pondrán en la cola para su ejecución. Los nombres de las funciones de los *kernels* son: `setMatrixSystems`, `transposeConjugate` y `complexMultiply`, cuya funcionalidad se detallará en el apartado del fichero de los *Sense_Kernel.cl*.
- `inverseMatrixM()`, está función se encarga de invertir las matrices almacenadas en el *array* `M`. Ha sido una de las partes más complejas de este Trabajo de Fin de Grado y se han implementado distintas formas de resolver los sistemas lineales para la reconstrucción SENSE, haciendo uso de distintas herramientas matemáticas y librerías de programación. En el capítulo 5 se comparan las distintas opciones barajadas y se elige esta por ser la que mejor comportamiento ha presentado en tiempos de ejecución.

  Para resolver la inversión de las matrices se ha optado por la descomposición en autovalores y autovectores de las mismas (ver apéndice C). Para ello, se ha utilizado la función `gsl_eigen_hermv()` de la librería numérica GSL (*GNU Scientific Library*) para C++, [28]. Esta librería no es compatible con OpenCL, por lo que para paralelizar el cálculo de $FOV\_L * Nx$ matrices inversas independientes entre sí, se ha optado por el uso de OpenMP (*Open Multi-Processing*), una API (y un conjunto de directivas) que aporta paralelismo multihilo en sistemas de memoria compartida en C/C++, [29]. Con la directiva `#pragma omp parallel for` se especifica que las iteraciones del bucle se distribuyan por los hilos existentes, realizándose el cálculo de varias matrices inversas con GSL de forma simultánea. El fragmento extraído del código es el siguiente:

```
void Sense::inverseMatrixM(){
    omp_set_num_threads(omp_get_max_threads()-5);
#pragma omp parallel for
    for(int i=0;i<FOV_L*mymap->getNy();i++){
        if(matrixinv(M,L,i*L*L)!=0){
            cout<<"matrix_inv KO"<<endl;
            exit(1);
        }
    }
}
```

- `applySENSE()`, una vez calculadas las matrices intermedias es posible obtener la matriz de *unfolding*, y resolver el sistema SENSE dado en la ecuación 2.9 realizando el producto de las matrices correspondientes. Para ello se crea un objeto *kernel* de la función `complexMultiply` que realiza las multiplicaciones entre matrices complejas en OpenCL. Finalmente se crea otro objeto *kernel* de la función `GetImagenSense` para obtener la imagen reconstruida, ya que los valores de los píxeles tras el *unfolding* no están correctamente ordenados. Los datos de la imagen están en GPU, por lo que es necesario copiarlos a la memoria del *host* para escribirlos en un fichero de salida y mostrar la imagen en la interfaz gráfica. La función que permite la trasferencia de datos es `clEnqueueReadBuffer()`.

- *Sense_Kernel.cl:* Contiene la declaración de tres funciones que van a ser *kernels* que se ejecutan en dispositivos OpenCL, en este caso en la GPU (si está disponible).
  - `setMatrixSystems`, este *kernel* se utiliza para extraer los puntos adecuados de los mapas de sensibilidad y de las imágenes con FOV reducido de las bobinas para establecer los sistemas lineales a resolver para la reconstrucción.
  - `transposeConjugate`, este *kernel* realiza el transpuesto conjugado de múltiples matrices simultáneamente.
  - `complexMultiply`, este *kernel* multiplica matrices complejas.

- *Imagen.cpp:* Implementación de los métodos de la clase y su constructor y destructor.

- *Imagen.h:* Definición de la clase *Imagen*, con la declaración de los atributos y métodos definidos a continuación.
  - Descripción de los atributos:
    - `Ny`, número de filas de la matriz que contiene los valores de los píxeles de la imagen.
    - `Nx`, número de columnas.
    - `imagen`, *array* de datos complejos de dimensiones $Ny \times Nx$ que contiene los valores de la imagen reconstruida.
    - `imdataBuffer`, objeto de memoria de OpenCL de tipo *buffer*. Va a contener la imagen de partida, es decir, la que se obtiene tras la simulación de la secuencia de pulsos EPI, por lo que sus dimensiones son $(Ny \times Nx)$.

- `imCoilsBuffer`, objeto de memoria de OpenCL de tipo *buffer*. En él se almacenarán las imágenes de las bobinas antes y después de simular el submuestreo. Sus dimensiones son $(Ny \times Nx \times nc)$.

- Descripción de los métodos:
  - `Imagen()`, constructor de la clase. Inicializa sus atributos mediante los parámetros de entrada que recibe. Reserva memoria de forma dinámica para el *array* `imagen` y crea los dos objetos de memoria `imdataBuffer` y `imCoilsBuffer`.
  - `~Imagen()`, destructor. Elimina el objeto *Imagen* al salir del ámbito donde fue definido. Además elimina los objetos de memoria de OpenCL que tiene como atributos llamando a la función `clReleaseMemObject()` de la API.
  - `setImdataBuffer()`, copia el contenido del objeto *buffer* que recibe como parámetro al atributo `imdataBuffer` mediante la función de la API de OpenCL `clEnqueueCopyBuffer()`.
  - `setImCoilsBuffer()`, copia el contenido del objeto *buffer* que recibe como parámetro al atributo `imCoilsBuffer` mediante la función de la API de OpenCL `clEnqueueCopyBuffer()`.
  - `calculeImCoils()`, simula las imágenes en las bobinas con FOV completo como el producto de los mapas de sensibilidad por la imagen *single coil*. La multiplicación de un mapa por la imagen da lugar a la imagen para esa bobina. Como las bobinas son independientes se puede hacer paralelamente. Se ha implementado el *kernel* `CoilsImages` para llevarlo a cabo.
  - `applyFFT2D()`, este método calcula la FFT en 2 dimensiones de los datos contenidos en `imCoilsBuffer`, utilizando la librería de OpenCL clFFT,[30].
  - `getImCoilsBuffer()`, permite acceder al atributo `imCoilsBuffer` desde fuera de la clase, ya que es privado.
  - `getImagen()`, análogamente al anterior, este método permite acceder al atributo `imagen`.

- *Imagen_Kernel.cl:* Fichero con la declaración de las funciones utilizadas como *kernels* de la clase *Imagen*.
  - `CoilsImages`, este *kernel* calcula las imágenes de las bobinas como producto de la imagen obtenida tras la simulación de la secuencia de pulsos por los mapas de sensibilidad.
  - `GetImagenSense`, este *kernel* ordena los valores de la imagen reconstruida de forma adecuada tras el proceso de *unfolding*.

- *EspacioK.cpp:* Implementación de los métodos de la clase y su constructor y destructor.

- *EspacioK.h:* Definición de la clase *EspacioK*, con la declaración de sus atributos y métodos.
  - Descripción de los atributos:
    - `Ny`, número de filas de la matriz de datos sin procesar del espacio k.
    - `Nx`, número de columnas.

- `kdataBuffer`, objeto de memoria de OpenCL de tipo *buffer*. Su contenido se corresponde con valores del espacio k complejos y puede ser tras la simulación de la secuencia de pulsos, o tras el cálculo de los valores para cada bobina; por lo que sus dimensiones son $(Ny \times Nx \times nc)$.
- `k_usampBuffer`, objeto de memoria de OpenCL de tipo *buffer*. En él se almacenarán los datos del espacio k de cada bobina con submuestreo. Sus dimensiones son $(Ny/L \times Nx \times nc)$.

- Descripción de los métodos:
  - `EspacioK()`, constructor de la clase. Inicializa sus atributos mediante los parámetros de entrada que recibe y crea los dos objetos de memoria `kdataBuffer` y `k_usampBuffer`.
  - `~EspacioK()`, destructor. Elimina el objeto *EspacioK* al salir del ámbito donde fue definido. Además elimina los objetos de memoria de OpenCL que tiene como atributos llamando a la función `clReleaseMemObject()` de la API.
  - `getEspacioK()`, recibe como entradas los objetos de memoria que contienen la parte real e imaginaria del espacio k simulado tras la secuencia de pulsos y combina estos datos en un único objeto de memoria (`kdataBuffer`) utilizando el *kernel* `Kdata`.
  - `copyKBuffer()`, copia el contenido de `k_usampBuffer` a `kdataBuffer` haciendo uso de la función `clEnqueueCopyBuffer()`.
  - `applyIFFT2D`, este método calcula la FFT inversa en 2 dimensiones de los datos contenidos en `kdataBuffer`, utilizando la librería de OpenCL clFFT, [30].
  - `undersample()`, simula que la separación entre líneas sucesivas del espacio k es mayor, es decir, que el número de líneas de codificación de fase se reduce en un número que se corresponde con el factor de aceleración `L` enviado como parámetro al método. Utiliza el *kernel* `EspacioK_L` para realizar el cómputo de los datos de todas las bobinas simultáneamente.
  - `setkdataBuffer()`, copia el contenido del objeto de memoria pasado como parámetro al *buffer* `kdataBuffer` mediante la función `clEnqueueCopyBuffer()`.

- *EspacioK_Kernel.cl:* Fichero con las funciones utilizadas como *kernels* de la clase *EspacioK*.
  - `Kdata`, *kernel* para combinar los *buffers* que contienen las partes reales e imaginarias del espacio k simulado en un único objeto de memoria con datos complejos (*float2*).
  - `EspacioK_L`, *kernel* que submuestrea el espacio k *"fully-sampled"* de cada bobina por un factor L.

- *Mapas.cpp:* Implementación de los métodos de la clase y su constructor y destructor.

- *Mapas.h:* Definición de la clase *Mapas*, con la declaración de los atributos y métodos correspondientes.
  - Descripción de los atributos:
    - `Ny`, número de filas de la matriz (3D) que contiene los valores de los píxeles de los mapas de sensibilidad.

- `Nx`, número de columnas.
- `Nz`, número de matrices, es decir, la dimensión de la tercera dimensión. Típicamente valdrá 1 pues la integración con el simulador de MRI solo permite simulación en 2D.
- `vol_size`, *array* tridimensional que define el tamaño del volumen.
- `bbox`, *array* tridimensional que define los límites del cuadro delimitador o *bounding box*.
- `rx`, resolución en frecuencia y fase en milímetros.
- `rz`, ancho del corte en milímetros.
- `tol`, número usado como tolerancia para el cálculo de las integrales elípticas en la simulación de los mapas de sensibilidad.
- `sen`, *array* de datos que contiene en memoria del *host* los valores de los mapas de sensibilidad. Sus dimensiones son $(Ny \times Nx \times Nz \times nc)$.
- `xyzBuffer`, objeto de memoria de OpenCL de tipo *buffer*. Almacena las coordenadas en milímetros para el sistema de coordenadas. Sus dimensiones son $(Nx + Ny + Nz)$.
- `xyz_outBuffer`, objeto de memoria de OpenCL de tipo *buffer*. Almacena la diferencia de las coordenadas `xyzBuffer` y los centros de las bobinas `ccBuffer` en el sistema de coordenadas cilíndricas (r, φ, z) y tras haber aplicado las rotaciones indicadas en el apartado 2.3.2 para cada bobina. Sus dimensiones son $(Nx \times Ny \times Nz \times 3 * nc)$.
- `senBuffer`, objeto de memoria de OpenCL de tipo *buffer*. Contiene en GPU los mapas de sensibilidad simulados para todas las bobinas. Sus dimensiones son $(Nx \times Ny \times Nz \times nc)$.
- `mycoils`, objeto de tipo *Bobinas*. Contiene los datos relativos a las bobinas de las cuales se van a simular los mapas de sensibilidad.

- Descripción de los métodos:
  - `Mapas()`, constructor de la clase. Inicializa sus atributos mediante los parámetros de entrada que recibe, asigna memoria de forma dinámica a los *arrays* `vol_size`, `bbox` y `sen`, crea los *buffers* `xyzBuffer`, `xyz_outBuffer` y `senBuffer` y además, instancia la clase *Bobinas* mediante su constructor, creando el objeto `mycoils`.
  - `~Mapas()`, destructor. Elimina el objeto *Mapas* al salir del ámbito donde fue definido. Además invoca al destructor de la clase *Bobinas* para eliminar el objeto `mycoils`.
  - `setCoordinats_mm()`, genera las coordenadas de vóxel situando el origen en el centro de la imagen. Posteriormente convierte estas coordenadas a milímetros haciendo uso de las resoluciones, `rx` y `rz`. Utiliza el *kernel* `Vector` para realizar el cómputo de forma paralela.
  - `getSenBuffer()`, permite acceder al atributo `senBuffer` desde fuera de la clase, ya que es privado.
  - `getNx()`, análogo pero para el atributo `Nx`.
  - `getNy()`, análogamente para el atributo `Ny`.

- `calculateCoilMaps()`, es el método más complejo de la clase y su funcionalidad es la de simular los mapas de sensibilidad asociados a las bobinas del atributo `mycoils`. Hace uso de varios *kernels*: `Resta`, `Rot`, `ElipkeBucle`, `Max`, `ElipkeAct` y `Elipke`. En primer lugar, es necesario calcular los centros y los radios de las bobinas (`mycoils->calculeCoilCentres()`), a continuación se accede al atributo del objeto del tipo *Bobinas* que contiene las coordenadas de sus centros (`mycoils->getccBuffer`) para hallar la diferencia de las coordenadas en milímetros almacenadas en `xyzBuffer` y se guarda el resultado en `xyz_outBuffer`. Se transforma el sistema de coordenadas cartesiano a cilíndricas y se aplican las rotaciones necesarias para poder utilizar las expresiones del cálculo de campo magnético producido por una espira circular, para ello es necesario resolver tantas integrales elípticas como bobinas. Finalmente se obtienen las sensibilidades asociadas a cada bobina.

- *Mapas_Kernel.cl:* Fichero con las funciones utilizadas como *kernels* de la clase *Mapas*.
  - `Vector`, *kernel* para calcular las coordenadas del vóxel en milímetros y con origen en el centro de la imagen.
  - `Resta`, *kernel* que resta las coordenadas de los centros de las bobinas a las coordenadas anteriores.
  - `Rot`, este *kernel* convierte el sistema de coordenadas cartesianas a cilíndricas, aplica las rotaciones pertinentes y calcula algunos parámetros necesarios para simular las sensibilidades.
  - `ElipkeBucle`, *kernel* para el cálculo de la integral elíptica. Inicializa los valores que se utilizan en la primera iteración y actualiza otros en función de unos "iniciales" en cada iteración del bucle hasta la convergencia.
  - `ElipkeAct`, *kernel* para el cálculo de la integral elíptica. Actualiza los valores "iniciales" en cada iteración del bucle para el cálculo de la integral elíptica hasta su convergencia.
  - `Max`, *kernel* para obtener el máximo de un vector en cada iteración del bucle del cálculo de la integral elíptica.
  - `Elipke`, este *kernel* calcula el resultado de la integral elíptica a partir de los valores hallados una vez ha convergido el proceso. Además calcula los mapas de sensibilidad para cada bobina.

- *Bobinas.cpp:* Implementación de los métodos de la clase y su constructor y destructor.

- *Bobinas.h:* Definición de la clase *Bobinas*, con la declaración de sus atributos y métodos.
  - Descripción de los atributos:
    - `nc`, número de bobinas a simular para el *array* en fase de bobinas receptoras.
    - `c_per_row`, número de bobinas por fila deseado.
    - `rbody_mm`, distancia entre el centro de la imagen y los centros de las bobinas. Hay que tener en cuenta que la disposición de las bobinas es en anillos, por lo que esta distancia será la misma para todas ellas.

  - `cc`, *array* que contiene las coordenadas de los centros de todas las bobinas. Sus dimensiones son $(3 * nc)$.
  - `rcoil`, radio de las bobinas calculado a partir de las expresiones de la ecuación 4.1 ó 4.2, según corresponda.
  - `ccBuffer`, objeto de memoria de OpenCL de tipo *buffer*. Su contenido es el mismo que el del *array* `cc`.
- Descripción de los métodos:
  - `Bobinas()`, constructor de la clase. Inicializa sus atributos mediante los parámetros de entrada que recibe, asigna memoria de forma dinámica al *array* `cc` y crea el objeto de memoria de OpenCL `ccBuffer`.
  - `~Bobinas()`, destructor. Elimina el objeto *Bobinas* al salir del ámbito donde fue definido.
  - `getNumCoils()`, permite acceder al atributo `nc` desde fuera de la clase, ya que es privado.
  - `getccBuffer()`, análogo al anterior para el atributo `ccBuffer`.
  - `getrcoil()`, análogamente para el atributo `rcoil`.
  - `calculCoilCentres()`, método para simular la disposición de las bobinas, calculando las coordenadas de los centros de cada una de ellas, así como su radio (común para todas ellas) como se explicó en el paso 3.2. del diseño funcional del algoritmo. Esta función podría paralelizarse y realizarse en GPU, pero debido a que el número de datos es pequeño y la latencia es menor en la CPU se ha optado por realizarlo con este dispositivo, ya que se han obtenido tiempos de ejecución menores.