Universidad de Valladolid

E.T.S.I. Telecomunicación

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS ESPECÍFICAS DE TELECOMUNICACIÓN, MENCIÓN EN SISTEMAS DE TELECOMUNICACIÓN

Síntesis de imagen de resonancia magnética mediante GPU

Autor:

Elisa Moya Sáez

Tutores:

Carlos Alberola López y Federico Simmross Wattenberg

Valladolid, 30 de Junio de 2017

Título:	**Síntesis de imagen de resonancia magnética mediante GPU**
Autor:	**Elisa Moya Sáez**
Tutores:	**Carlos Alberola López y Federico Simmross Wattenberg**
Departamento:	**TSCIT**

TRIBUNAL

Presidente:	**Carlos Alberola López**
Secretario:	**Federico Simmross Wattenberg**
Vocal:	**Marcos Martín Fernández**
Suplente:	**Santiago Aja Fernández**
Suplente:	**Rodrigo de Luis García**

Fecha:

Calificación:

RESUMEN DEL PROYECTO

En este Trabajo Fin de Grado se plantea la implementación de un entorno versátil que permita simular la síntesis de imágenes de resonancia magnética a partir de la secuencia de pulsos *Spin Echo-Echo Planar Imaging* (SE-EPI).

Para ello se tomó como punto de partida un código programado en MATLAB en el que estaba implementado la secuencia de pulsos SE-EPI de forma secuencial. El problema está en que, debido al elevado coste computacional que supone la simulación de imágenes de resonancia magnética, pues requiere de múltiples operaciones para dar lugar a cada uno de los puntos de la imagen a sintetizar, la programación secuencial de estos algoritmos se traduce en tiempos de ejecución muy elevados. No obstante, estas operaciones son fácilmente paralelizables, por lo que el uso de GPUs (Unidades de Procesamiento Gráfico) permite aumentar sensiblemente la eficiencia del proceso y reduce los tiempos de ejecución.

Debido a esto, la implementación que en el presente proyecto se plantea, incorpora programación paralela en GPU, concretamente mediante la API (Interfaz de Programación de Aplicaciones) y el lenguaje de programación de OpenCL, orientado a conseguir un paralelismo a nivel de datos. No obstante, siempre se persigue obtener un compromiso entre la reducción de los tiempos de ejecución y la obtención de un código flexible en el que resulte sencillo realizar cambios o incorporar mejoras, tales como simular otros fenómenos físicos inherentes a las imágenes de resonancia magnética o la simulación de otras secuencias de pulsos.

Por otro lado, con el objetivo de obtener una aplicación cerrada de extremo a extremo, se incorpora una interfaz gráfica de usuario mediante MATLAB, que permite aumentar la usabilidad de la aplicación para el usuario final.

Por último, en lo referente a los resultados obtenidos, se validan visualmente las imágenes de resonancia magnética simuladas. A su vez, se evalúa el rendimiento del algoritmo llegando a la conclusión de que, como era de esperar, la implementación mediante programación en GPU supone una reducción considerable en los tiempos de cálculo.

PALABRAS CLAVE

SE-EPI, MRI, GPU, OpenCL, Interfaz gráfica de usuario

ABSTRACT

In this Thesis we present an implementation of a versatile simulator of magnetic resonance imaging (MRI). Namely, it simulates the Spin Echo-Echo Planar Imaging (SE-EPI) sequence.

The Thesis started from MATLAB code which sequentially simulated the SE-EPI sequence. However, the simulation of magnetic resonance images is computationally intensive and time consuming since it requires multiple operations to compute each of the voxels in the image to synthesize. For this reason, the initial sequential structure of the code was extremely inefficient, resulting in high computation times. Nevertheless, several operations present in the simulation are easily parallelized. This is why the usage of GPUs (Graphics Processor Units) increases the efficiency of the overall execution and reduces the execution time.

In this context, the code implementation of this Thesis includes parallel programming in GPU through the OpenCL API (Application Programming Interface) and programming language. This way, we coded the simulator with parallelism at the data level. However, we always tried to achieve balance between the reduction of the computation time and the flexibility of the code. This way, we are able to quickly modify or update the code to take into account more physical effects inherent to MRI or to simulate other pulse sequences.

In addition, in order to provide a end-to-end application, we developed a graphical user interface that provides the simulator with an user-friendly environment.

Finally, the simulated magnetic resonance images were validated by visual inspection. Additionally, we assessed the performance of the algorithm concluding that, as expected, the GPU implementation significantly reduced the overall computation time.

KEYWORDS

SE-EPI, MRI, GPU, OpenCL, Graphical user interface

Agradecimientos

En primer lugar, quiero expresar mi gratitud a mis tutores Carlos Alberola López y Federico Simmross Wattenberg por darme la posiblidad de realizar este Trabajo Fin de Grado y por su esfuerzo y tiempo dedicado a guiarme en el transcurso del mismo.

Así mismo, agradecer al Laboratorio de Procesado de Imagen su amabilidad desde el primer momento. Muy especialmente me gustaria mencionar a Daniel Treceño y Elena Martín por su apoyo, ayuda y paciencia día tras día, ya que sin ellos no habría conseguido realizar este trabajo.

Por último, dar las gracias a mis amigos y familia, que me han apoyado siempre, y sobre todo en los momenontos díficiles.

Índice general

Capítulo 1

Introducción

1.1 Imagen de Resonancia Magnética

La Imagen de Resonancia Magnética (MRI, de las siglas en inglés *Magnetic Resonance Imaging*) es una potente técnica tomográfica de imagen médica que permite obtener imágenes de las características físicas y químicas de un objeto midiendo las señales procedentes de la técnica de Resonancia Magnética Nuclear (NMR, de las siglas en ingles *Nuclear Magnetic Resonance*) [1]. La formación de las imágenes se realiza gracias a los principios de la codificación espacial de la información que permite detectar el objeto a partir de las señales excitadas de resonancia magnética (MR).

El potencial de esta técnica está en que es capaz de obtener imágenes con mucha información sobre los tejidos que componen el objeto de manera no invasiva, ya que, como se discutirá más adelante en el capítulo 2, cada píxel de la imagen depende de unos parámetros intrínsecos del objeto, como son la densidad de protones (PD), el tiempo de relajación espín-red (T_1) o el tiempo de relajación espín-espín (T_2), entre otros. Además, los efectos sobre las imágenes de estos parámetros pueden ser suprimidos o ensalzados mediante otros parámetros seleccionables en cada secuencia de resonancia magnética, como el tiempo de repetición (TR), el el tiempo de eco (TE) o el *flip angle* (α). Como consecuencia, modelando la secuencia de resonancia magnética se pueden obtener imágenes con muy variadas características y contrastes.

Una ventaja importante de la MRI en comparación con otras técnicas tomográficas como PET (Tomografía por Emisión de Positrones) o SPECT (Tomografía Computerizada por Emisión de Fotón Único) es que, en este caso, no es necesario inyectar isótopos radiactivos en el objeto para generar la señal de MR [1]. Por otro lado, dado que la MRI opera en el rango de la radiofrecuencia (RF), no se usa radiación ionizante evitando así los efectos perjudiciales para la salud que esta podría tener sobre el paciente.

A pesar de las múltiples ventajas de la MRI, la optimización de las secuencias de pulsos que permitan

obtener las imágenes con las características deseadas implica un proceso iterativo de modificación de parámetros con el consiguiente consumo de tiempo. Además, en la práctica resulta difícil distinguir los artefactos originados por las distintas fuentes [2].

1.2 Motivaciones

La síntesis de imágenes de resonancia magnética de forma simulada, consiste en obtener la imagen de resonancia magnética final partiendo de unos modelos anatómicos de los tejidos. 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, por un lado, la simulación permite observar determinados efectos sobre las imágenes, tales como la presencia de artefactos, las inhomogeneidades del campo magnético, el efecto del *Chemical Shift* o la imperfección de los gradientes. Además, 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 [3]. Por otro lado, permite probar nuevas secuencias de pulsos y saber si éstas van a funcionar o no, sin necesidad de hacer las pruebas en la máquina física, aligerando notablemente el consumo de tiempo que el diseño de secuencias de pulsos implica [3].

Por otro lado, dado el elevado coste que suponen los equipos de MR, es necesario usarlos el mayor tiempo posible para sacarlos el máximo rendimiento. Como consecuencia, la accesibilidad a los equipos con fines educativos se reduce [4]. 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.

El principal problema de estas simulaciones está en que la síntesis de imagen de resonancia magnética implica un elevado coste computacional, ya que se necesita realizar un gran número de operaciones para obtener todos los puntos del espacio K, es decir, los datos en bruto obtenidos por el equipo de resonancia magnética que permiten obtener la imagen final mediante la transformada inversa de Fourier. Esto se traduce en que, mediante una programación secuencial, los tiempos de ejecución son muy elevados.

No obstante, el problema tiene la ventaja de ser intrínsecamente paralelo, ya que es necesario realizar las mismas operaciones sobre cada uno de los píxeles de la imagen. Debido a esto, la introducción de técnicas de computación paralela, concretamente mediante programación en GPUs (Graphics Processor Units), permite un ahorro relevante en los tiempos de cálculo [2]. Además, con el objetivo de que el simulador se pudiera ejecutar con independencia del fabricante de la GPU se optó por el lenguaje de programación OpenCL (Open Computing Language), ya que, a diferencia de otros lenguajes como CUDA

que es propietario (por lo que solo está disponible para tarjetas gráficas de NVIDIA), OpenCL tiene la ventaja de ser un estándar abierto y libre que no depende del fabricante del hardware subyacente.

1.2.1 Objetivos

El objetivo perseguido con la elaboración de este trabajo es, por lo expuesto en la sección anterior, **la implementación de un simulador de síntesis de imágenes de resonancia magnética mediante técnicas de computación paralela en GPU**, concretamente mediante el lenguaje de programación OpenCL, partiendo de un código programado previamente mediante MATLAB.

El objetivo global se desglosa en los siguientes subobjetivos:

- **Implementación de un algoritmo para la síntesis de imágenes de resonancia magnética mediante la secuencia de pulsos *Spin Echo-Echo Planar Imaging* (SE-EPI) con técnicas de programación paralela**, para así obtener una reducción de los tiempos de ejecución con respecto a la implementación secuencial del mismo.
- **Creación de una GUI (Graphical User Interface) que permita ejecutar el código desde MATLAB**, en la cual se puedan seleccionar los modelos anatómicos de partida, así como modificar de forma dinámica algunos parámetros de la simulación tales como el diezmado, la inhomogeneidad del campo, el TE y TR, el ETL (Echo Train Length), el número de pulsos preparatorios y la forma de escritura del espacio K (secuencial o interlineado). Desde esta interfaz se ejecutará el código programado previamente en OpenCL y se podrán observar las imágenes de salida.

1.2.2 Fases y Métodos

Para alcanzar los subobjetivos planteados, las labores a desarrollar se han llevado a cabo en dos etapas:

I. **Etapa de formación**, realizada de acuerdo a las siguientes fases:

   a) Estudio de los fundamentos de las técnicas de programación paralela en general, y más concretamente en el lenguaje de programación OpenCL.

   b) Estudio de los principios básicos de la técnica de NMR y la obtención de imágenes de resonancia magnética a partir de las señales obtenidas con la técnica anterior.

   c) Comprensión del código de MATLAB, en el que estaba implementado previamente la simulación de imágenes de resonancia magnética mediante la secuencia de pulsos SE-EPI.

II. **Etapa de implementación**, llevada a cabo en las siguientes fases:

a) Programación del algoritmo de simulación de imágenes de resonancia magnética utilizando la API (Interfaz de Programación de Aplicaciones) y el lenguaje de programación OpenCL, orientado a conseguir un paralelismo a nivel de datos.

b) Identificación de las limitaciones del algoritmo e introducción de las mejoras necesarias.

c) Creación de la GUI de MATLAB e integración de esta con el ejecutable generado en OpenCL.

d) Análisis de los efectos obtenidos en las imágenes al variar los diferentes parámetros de la simulación.

e) Análisis de las prestaciones del algoritmo en cuanto a tiempos de ejecución.

1.2.3 Medios

Para la realización de este Trabajo Fin de Grado, será necesario el acceso a las herramientas *software* y *hardware* que se detallan a continuación:

Software

- MATLAB R2015a [5]: 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 [6]: entorno integrado de desarrollo de *software*.
- LaTeX [7] : sistema de composición de textos, orientado a la creación de documentos escritos.

Hardware

- PC de sobremesa con las siguientes características:
  - Procesador 8xIntel® Core™ i7-4790 3.60 GHz.
  - 16 GB de memoria RAM.
  - Disco duro de 500 GB de capacidad.
  - GPU NVIDIA GeForce GTX 970 con 4 GB de memoria RAM
  - *Device* con versión OpenCL C 1.2
  - *Driver* con versión 375.39
- Acceso a servidores de cálculo disponibles en el Laboratorio de Procesamiento de Imagen (LPI) de la Universidad de Valladolid (UVa).

En cuanto a los modelos anatómicos que se utilizaran 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.

Finalmente, el trabajo se realizará en el Laboratorio 25 de la Escuela Técnica Superior de Ingenieros de Telecomunicación (ETSIT) de la UVa.

1.3 Estructura del documento

El resto del documento se estructura según se detalla a continuación:

En el capítulo 2 se plantean los principios básicos de la técnica de NMR. Para ello se empieza explicando las propiedades físicas del fenómeno, y a partir de ahí, se describen las señales obtenidas con está técnica y la introducción de codificación espacial para poder obtener imágenes a partir de las señales.

En el capítulo 3 se describe con mayor detalle el método propuesto para la programación del simulador de síntesis de imágenes de resonancia magnética. En este caso consistirá en la introducción de programación en GPU con el objetivo de paralelizar el problema y conseguir una disminución en los tiempos de computación. En este capítulo se comienza describiendo las ventajas e inconvenientes de la programación heterogénea, remarcando las diferencias entre CPU y GPU. Posteriormente se describe la API y el lenguaje de programación de OpenCL.

En el capítulo 4 se presenta una visión global de la implementación realizada. Para ello, se detallan los pasos que se han seguido en el algoritmo, así como, las variables, objetos de memoria y *kernels* implementados. También se explica la introducción de programación orientada a objetos para conseguir aumentar la flexibilidad del código.

Y por último, la creación de la interfaz de usuario que permite la integración del núcleo de la simulación en una aplicación cerrada.

En el capítulo 5 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 el mapa de inhomogeneidad del campo. Y en una segunda sección, en cuanto a las mejoras obtenidas en los tiempos de ejecución con respecto a la implementación secuencial.

Finalmente, en el capítulo 6 se recogen las principales conclusiones extraídas con la elaboración de este Trabajo Fin de Grado y se plantean las posibles líneas de trabajo futuro surgidas a partir de su realización.

Capítulo 2

Fundamentos de la MRI

2.1 Generación y detección de las señales

Para poder comprender los principios en los que se basa la formación de imágenes de resonancia magnética, en esta sección se va a comenzar con una descripción del fenómeno de la NMR para posteriormente llegar a las expresiones de las señales explicando cómo son y cómo se generan.

2.1.1 Momentos magnéticos nucleares

Los núcleos atómicos que presentan un número atómico o peso atómico impar poseen un momento angular $\boldsymbol{J}$, que se conoce como espín. En el modelo vectorial clásico, esta magnitud es definida como un vector que representa la rotación de los núcleos atómicos alrededor de un determinado eje [1]. Aunque existen varios núcleos atómicos con esta característica, los núcleos de hidrógeno (^{1}H), 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.

Estos núcleos son objetos cargados en rotación, por lo que generarán a su alrededor un campo magnético conocido como momento magnético $\boldsymbol{\mu}$, el cual está relacionado con el espín mediante una constante física característica de cada tipo de núcleo, denominada constante giromagnética (γ) [1].

$$\boldsymbol{\mu} = \gamma \boldsymbol{J} \tag{2.1}$$

Dado que el momento magnético es un vector, su definición se realiza mediante el módulo y la dirección. Según las teorías de la mecánica cuántica, el módulo depende del número cuántico del espín I, también característico de cada tipo de núcleo. Este número tiene que ser no nulo para que las propiedades de la resonancia magnética se puedan manifestar. En el caso de los núcleos de hidrógeno, I=1/2 y la

constante giromagnética toma un valor de 42.58 MHz/T.

Por otro lado, la dirección del momento magnético $\boldsymbol{\mu}$ es aleatoria debido al movimiento térmico aleatorio. Como consecuencia, la magnetización neta del sistema de espines (conjunto de todos los núcleos del mismo tipo presentes en un objeto) será nula por las cancelaciones mutuas producidas entre ellos. No obstante, bajo la aplicación de un campo magnético estático $\boldsymbol{B_0}$ los núcleos de hidrógeno tienden a alinearse con la dirección de este campo, rotando alrededor de dos posibles orientaciones (paralela y antiparalela) [8]. Este efecto está representado en la Figura 2.1

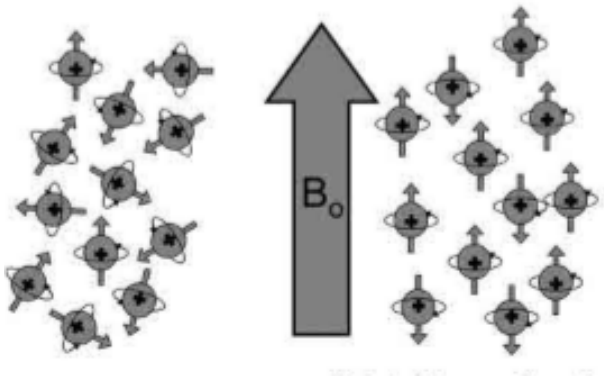


Figura 2.1: Alineamiento de los espines bajo un campo magnético estático [8].

Además, debido al campo magnético estático, los momentos magnéticos de los núcleos precesionan a lo largo de la dirección de $\boldsymbol{B_0}$ (típicamente el eje z) a una frecuencia ω_0 conocida como frecuencia de Larmor y dada por [1]:

$$\omega_0 = -\gamma B_0 \tag{2.2}$$

El movimiento de precesión de un espín sometido a un campo magnético estático se puede ver en la Figura 2.2 [9].

De las dos posibles orientaciones de los núcleos de hidrógeno sometidos a un campo magnético estático, 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 [8].

2.1.2 Magnetización Neta

Para describir el comportamiento conjunto del sistema de espines a nivel macroscópico se utiliza el vector de magnetización neta del sistema, que es la suma de todos los momentos magnéticos microscópicos

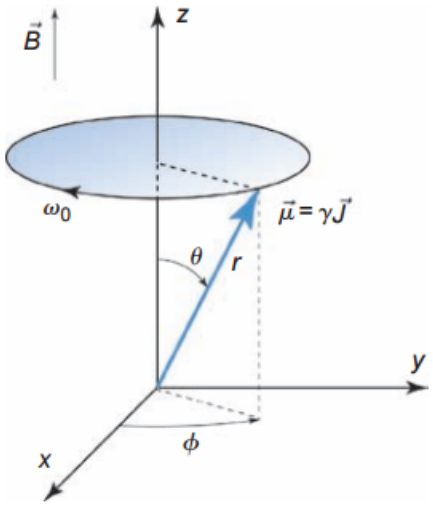


Figura 2.2: Movimiento de precesión de un espín sometido a un campo magnético estático [9].

individuales [1]. Siendo N_s el número total de espines del objeto, el vector de magnetización neta $\boldsymbol{M}$ del sistema viene dado por:

$$\boldsymbol{M} = \sum_{n=1}^{N_s} \boldsymbol{\mu_n} \tag{2.3}$$

Este vector de magnetización neta puede ser descompuesto en sus tres componentes de la forma $\boldsymbol{M} = [M_x, M_y, M_z]$. Bajo la presencia de un campo magnético estático $\boldsymbol{B_0}$ en la dirección longitudinal z, las componentes transversales (M_x^0 y M_y^0) serán nulas, ya que la precesión de los momentos magnéticos tienen fases aleatorias. No obstante, la componente longitudinal será no nula.

$$M_z^0 = \frac{1}{2}(N_\uparrow - N_\downarrow)\gamma\bar{h} \tag{2.4}$$

Donde $N_\uparrow$ es el número de espines rotando en la dirección paralela, $N_\downarrow$ el número de espines rotando en la dirección antiparalela y $\bar{h}$ es la constante de Planck reducida o constante de Dirac, $\bar{h} = \frac{h}{2\pi}$, con $h = 6{,}33 \cdot 10^{-34} J \cdot s$ la constante de Planck [10].

2.1.3 Excitación de RF

Para que las componentes transversales del vector de magnetización neta sean no nulas, se requiere la aplicación de un segundo campo magnético en la dirección transversal que denominaremos $\boldsymbol{B_1}(t)$, nótese

que este segundo campo no será estático sino dependiente del tiempo. Además, es necesario que este campo rote a una frecuencia igual a la frecuencia de Larmor, de forma que los espines entren en resonancia y se pueda dar una transferencia eficiente de energía [8]. La frecuencia de precesión de los núcleos de hidrógeno bajo la presencia de un campo magnético estático de $B_0 = 3\text{T}$ es de aproximadamente 128 MHz, dentro del rango de la radiofrecuencia (RF), por lo que este campo se conoce normalmente como pulso de RF.

El pulso de RF puede ser expresado siguiendo la notación de exponenciales complejas de la siguiente forma:

$$B_1(t) = B_1^e(t)e^{-i(\omega_{rf}t+\phi)} \tag{2.5}$$

Donde $B_1^e(t)$ es la amplitud de la envolvente del pulso de RF, ω_{rf} es la frecuencia de la portadora que deberá coincidir con la frecuencia de Larmor y ϕ es la fase inicial que se supondrá nula.

La aplicación de este segundo campo provocará que el vector de magnetización neta del sistema de espines $\boldsymbol{M}$ se mueva respecto a la posición de equilibro, paralela a $\boldsymbol{B_0}$, siguiendo una trayectoria espiral. La variación con el tiempo de $\boldsymbol{M}$ bajo la aplicación del pulso de RF, es descrita mediante la *ecuación de Bloch* [1]:

$$\frac{d\boldsymbol{M}}{dt} = \gamma \boldsymbol{M} \times \boldsymbol{B} - \frac{M_x\vec{i} + M_y\vec{j}}{T_2} - \frac{(M_z - M_z^0)\vec{k}}{T_1} \tag{2.6}$$

Donde M_z^0 es el valor de equilibrio de $\mathbf{M}$ bajo la presencia únicamente del campo magnético estático $\boldsymbol{B_0}$, mientras que T_1 y T_2 son constantes temporales que caracterizan el proceso de relajación del sistema de espines. Si se asume que la duración del pulso de RF es pequeña comparado con T_1 y T_2 se puede despreciar los dos últimos términos en (2.6) y simplificar la *ecuación de Bloch* como:

$$\frac{d\boldsymbol{M}}{dt} = \gamma \boldsymbol{M} \times \boldsymbol{B} \tag{2.7}$$

Donde:

$$\boldsymbol{B} = \boldsymbol{B_{eff}} = (B_0 - \frac{\omega_{rf}}{\gamma})\vec{k'} + B_1^e(t)\vec{i'} \tag{2.8}$$

Según la condición de excitación *on-resonance*, $\omega_{rf} = \omega_0 = \gamma B_0$, por lo que sustituyendo en (2.10) se obtiene que:

$$\boldsymbol{B} = \boldsymbol{B_{eff}} = B_1^e(t)\vec{i'} \tag{2.9}$$

Bajo estas condiciones ideales, el campo efectivo coincide con el pulso de RF, por lo que se producirá la precesión del vector de magnetización neta sobre el eje del pulso de RF, que por convenio será el eje x. No obstante, en la práctica, efectos como las inohomegeneidades del campo o el *chemical-shift*, dan lugar a una excitación *off-resonance* ya que $\omega_{rf} \neq \omega_0$ [1].

$$\boldsymbol{B} = \boldsymbol{B_{eff}} = (B_0 - \frac{\omega_{rf}}{\gamma})\vec{k'} + B_1^e(t)\vec{i'} = \frac{\Delta\omega_0}{\gamma}\vec{k'} + B_1^e(t)\vec{i'} \tag{2.10}$$

Donde $\Delta\omega_0 = \omega_0 - \omega_{rf}$. Como consecuencia el vector de magnetización neta ya no precesiona sobre el eje x, sino sobre el eje del campo efectivo.

2.1.4 Relajación

Después a de la aplicación del pulso de RF, el vector de magnetización neta del sistema de espines vuelve gradualmente a la posición de equilibrio. Este proceso se caracteriza por tres efectos:

- **Precesión libre:** movimiento de precesión del vector de magnetización neta $\boldsymbol{M}$ alrededor del campo magnético estático $\boldsymbol{B_0}$.
- **Relajación longitudinal o espín-red:** debida a un intercambio de energía entre los espines y el ambiente circundante. Provoca la recuperación de la magnetización longitudinal.
- **Relajación transversal o espín-espín:** debida a la perdida de coherencia de fase de la magnetización del sistema de espines. Da lugar a la perdida de las componentes transversales del vector de magnetización neta.

El proceso de relajación longitudinal queda caracterizado en cada tejido por T_1 (tiempo de relajación longitudinal), mientras que el proceso de relajación transversal queda caracterizado por T_2 (tiempo de relajación transversal). Además, los tiempos de relajación T_1 y T_2 cumplen que $T_1 > T_2$, por lo que la recuperación de la magnetización longitudinal es más lenta que la perdida de la magnetización transversal

[1]. La variación con el tiempo de las componentes de magnetización longitudinal y transversal se puede obtener anulando en (2.6) la contribución del pulso de RF. Resolviendo las ecuaciones se obtiene [1]:

$$\begin{cases} M_z(t) = M_z^0(1 - e^{-t/T_1}) + M_z^{0+}e^{-t/T_1} \\ M_{xy}(t) = M_{xy}^{0+}e^{-t/T_2} \end{cases} \tag{2.11}$$

Donde M_z^0 es la magnetización longitudinal antes del pulso de RF. M_z^{0+} y M_xy^{0+} son la magnetización longitudinal y transversal respectivamente justo después del pulso de RF.

Por otro lado, debido a la inhomogeneidad del campo magnético (ΔB_0), la relajación real de la magnetización transversal es más rápida que T_2 [10]. En la práctica, el tiempo de relajación real observado, denominado T_2^*, es más pequeño que T_2 y viene dado por:

$$T_2^* = \frac{1}{\frac{1}{T_2} + \gamma \Delta B_0} \tag{2.12}$$

2.1.5 Secuencia *Spin echo*

Después de la aplicación de un único pulso de RF sobre un sistema de espines, se generará un campo magnético que puede ser detectado en la misma bobina en la que se aplicó el pulso de RF. Esta señal se irá relajando a razón de T_2^*, y es la que se conoce como FID (*Free induction decay*) [10].

Si se aplican al menos dos pulsos de RF en lugar de uno solo, se generará otra señal de RM que se conoce como eco. Esto es debido a que se produce una realineación de los espines incoherentes gracias al segundo pulso de RF reorientador.

En estas ideas se basa la secuencia de pulsos *Spin echo*, la cual consiste en la aplicación de un par de pulsos de 90º (pulso de saturación) y 180º (pulso de inversión). Esta secuencia está representada en la Figura 2.3 [9]. Tras la aplicación de un primer pulso de 90º, los espines comienzan a desfasarse debido a todos los efectos que contribuyen a la relajación T_2^*. En $TE/2$ (siendo TE un parámetro característico de la secuencia de pulsos que se conoce como tiempo de eco) se aplica un segundo pulso de 180º. Debido a esto, el vector de magnetización es volteado y los espines que se estaban desfasando comienzan a recuperar la fase originando una señal de eco que será máxima en un tiempo TE. Se puede observar que el decaimiento de la señal en un tiempo TE comparado con el comienzo de la secuencia de pulsos, se produce ahora a razón de T_2 [11].

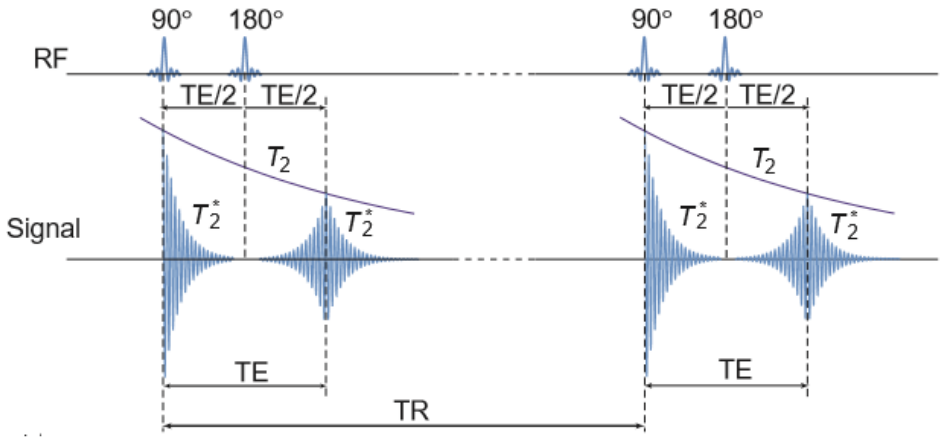


Figura 2.3: Formación de la señal en una secuencia Spin echo [9].

2.2 Formación de la imagen de MR

Una vez visto en la sección anterior como se puede generar la señal de MR de un objeto, en esta sección se van a exponer los principios mediante los cuales dichas señales pueden ser localizadas espacialmente para así poder obtener el correspondiente espacio K de la imagen y posteriormente la imagen de resonancia magnética final.

2.2.1 Selección de Slice

Para conseguir que el pulso de RF excite únicamente un determinado *slice* del objeto, se requiere una excitación selectiva espacial que se consigue haciendo que la frecuencia de resonancia de los espines sea dependiente de la posición [12]. Una forma simple de conseguir esto es incrementar el campo magnético estático B_0 con un gradiente lineal (conocido como gradiente selector de *slice*) durante el periodo de excitación de la señal.

2.2.2 Codificación espacial

La localización espacial de las señales es esencial para convertir las señales de MR en imágenes de resonancia magnética. Para ello, se utilizan dos métodos de codificación espacial:

- **Codificación de frecuencia:** mediante la aplicación de gradientes del campo magnético, se consigue que la tasa de precesión de los espines del *slice* excitado dependa linealmente de la localización

espacial [1]. Considerando un gradiente de codificación de frecuencia aplicado en la dirección x (G_x), se consigue una variación de la frecuencia de resonancia ω de los espines con la posición, dada por:

$$\omega(x) = \omega_0 + \gamma G_x x \tag{2.13}$$

La señal generada por los espines en un intervalo infinitesimal dx del punto x es:

$$dS(x,t) = \rho(x)dx e^{-i\omega(x)t} \tag{2.14}$$

Donde $\rho(x)$ es la densidad de espines. Integrando en (2.14) se puede obtener la señal recibida de todo el objeto:

$$S(t) = \int_{-\infty}^{\infty} \rho(x) e^{-i\omega(x)t} dx = e^{-i\omega_0 t} \int_{-\infty}^{\infty} \rho(x) e^{-i\gamma G_x t} dx \tag{2.15}$$

La señal recibida en la antena receptora se demodula, por lo que se elimina el término de la portadora $e^{-i\omega_0 t}$. Como consecuencia, la señal demodulada resultante es:

$$S(t) = \int_{-\infty}^{\infty} \rho(x) e^{-i\gamma G_x t} dx \tag{2.16}$$

- **Codificación de fase:** se activa un gradiente durante un periodo de tiempo corto (T_{pe}) y después se desactiva, con lo que se consigue que la fase de la señal dependa de la localización espacial [1]. Considerando un gradiente de codificación de fase aplicado en la dirección y (G_y), se consigue que después de un tiempo T_{pe} el vector de magnetización tenga un desfase dependiente de la posición, dado por:

$$\phi(y) = \gamma G_y y T_{pe} \tag{2.17}$$

Siendo $\rho(y)$ la densidad de espines, la señal recibida de todo el objeto viene dada por:

$$S(t) = \int_{-\infty}^{\infty} \rho(y) e^{-i(\omega_0 t + \phi(y))} dy = e^{-i\omega_0 t} \int_{-\infty}^{\infty} \rho(y) e^{-i\phi(y)} dy \tag{2.18}$$

Igual que en el caso anterior, la señal recibida en la antena receptora se demodula, por lo que se elimina el término de la portadora $e^{-i\omega_0 t}$. Como consecuencia, la señal demodulada resultante es:

$$S(t) = \int_{-\infty}^{\infty} \rho(y) e^{-i\phi(y)} dy \tag{2.19}$$

2.2.3 Espacio K

Se puede observar que existe una relación de Fourier entre la variación espacial de la densidad de espines y la señal recibida, sin más que sustituir en (2.16) y (2.22) respectivamente:

$$k_x = \frac{\gamma}{2\pi} G_x t \tag{2.20}$$

$$k_y = \frac{\gamma}{2\pi} G_y T_{pe} \tag{2.21}$$

Esto es lo que se conoce como espacio K, que constituye el dominio sobre el que se distribuye la señal original obtenida. El espacio K de la imagen bidimensional tras la aplicación de la codificación en frecuencia y en fase de los ejes x e y respectivamente puede ser expresado como:

$$S(k_x, k_y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \rho(x, y) e^{-i2\pi(k_x x + k_y y)} dx dy \tag{2.22}$$

Una vez obtenidos los datos en el espacio K, la imagen de resonancia magnética final deseada puede ser obtenida mediante postprocesado aplicando la transformada inversa de Fourier.

2.3 Secuencia Echo Planar Imaging

La secuencia *Echo Planar Imaging* (EPI) es una secuencia rápida que se caracteriza por la obtención de un tren de ecos mediante la aplicación de gradientes en lugar de pulsos de RF. Esta secuencia se puede implementar a partir de otras secuencias clásicas como *Spin echo*, *Gradient echo* y otras basadas en el paradigma de inversión a continuación de recuperación [11]. Por otro lado, en función de lo gradientes variantes con el tiempo que se utilicen, se pueden llevar a cabo diferentes trayectorias para la lectura del espacio K, como la trayectoria en zigzag, la rectilínea o la espiral. Esto tiene su importancia, ya que desempeña un papel fundamental en el contraste de la imagen final, el cual viene determinado por el momento en el que se adquieran las bajas frecuencias, es decir, las centrales del espacio K [11].

La principal ventaja que presenta la secuencia de pulsos EPI es su rapidez en la obtención de las imágenes, ya que, con la tecnología actual, es capaz de producir imágenes bidimensionales en solo unas decenas de milisegundos [12]. No obstante, esta secuencia presenta también algunos inconvenientes importantes. Entre ellos cabe destacar la mayor atenuación de la señal y como consecuencia de las sucesivas líneas del espacio K al obtenerlas mediante una única excitación de RF; y, por otra parte, que esta secuencia es más propensa a la presencia de artefactos en las imágenes que otras como *Spin echo* o *Gradient echo* [12].

Capítulo 3

Fundamentos de la computación en GPU

En este capítulo se pretende proporcionar una visión general de los principios básicos en los que se basa la computación en GPU, así como de los lenguajes de programación existentes actualmente para llevarlo a cabo. Por último, se describe en profundidad 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 computación en GPU

Actualmente, los entornos de computación se presentan cada vez más heterogéneos. La presencia en las arquitecturas de ordenadores de microprocesadores multinúcleo, unidades de procesamiento central (CPUs), procesadores digitales de señal (DSPs), dispositivos lógicos programables (FPGAs) y unidades de procesamiento gráfico (GPUs) amplían las oportunidades de programación para los desarrolladores de software [13]. Por otro lado, tanto las CPUs que progresan aumentando el número de núcleos, como las GPUs que han evolucionado desde dispositivos de renderizado a procesadores paralelos programables, tienden a aumentar el paralelismo como principal vía para mejorar el rendimiento [14] . En este contexto, la introducción de una programación en GPU se presenta como una solución eficiente para mejorar el desarrollo de aplicaciones que supongan una elevada carga computacional, tales como, el procesado de imagen o la simulación entre otras.

En esencia, la CPU y la GPU son similares: circuitos integrados que realizan cálculos matemáticos con números binarios. No obstante, su diseño es distinto. Debido a esto, se suelen utilizar para la ejecución de aplicaciones que persiguen objetivos diferentes. Como se puede observar en la Figura 3.1, la CPU se compone de uno o pocos núcleos muy complejos, por lo que se utiliza para el procesamiento en serie de programas con una elevada cantidad de flujos de control, mientras que la GPU se compone de cientos o miles de núcleos sencillos que permiten la ejecución de programas con flujos de control simples pero con un elevado grado de paralelismo [15]. Por otro lado, en la CPU cualquier operación toma típicamente del

orden de 20 ciclos por segundo entre entrar y salir de la pipeline, mientras que 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* [16].

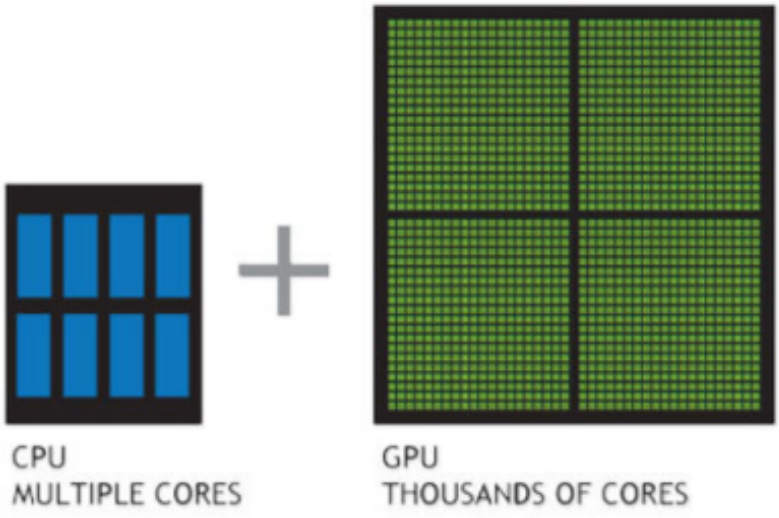


Figura 3.1: Diseño de una CPU vs. GPU [15].

3.1.1 Lenguajes para la computación en GPU

Existen numerosos lenguajes de programación que permiten realizar programación en GPU, 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 [17]. Dado que en este Trabajo Fin de Grado nos interesan las aplicaciones de propósito general nos centraremos 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.

3.2 OpenCL

OpenCL es un *framework* de computación heterogénea gestionado por el consorcio tecnológico Khronos Group [14]. 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 y un lenguaje de programación. Como ya se ha mencionado, las principal ventaja de OpenCL en comparación con otros lenguajes de programación de GPU, está en la independencia del código generado con el fabricante del *hardware* subyacente sobre el que se ejecute, permitiendo el desarrollo de *software* portable [14].

Para describir las ideas básicas de OpenCL se utilizan puntos de vista de distintos *modelos*:

- **Modelo de *Platform*:** la *platform* esta formada por un *Host* (típicamente la CPU), que es el encargado de gobernar todo el sistema por lo que está conectado a uno o más *OpenCL devices*. Cada *OpenCL device* se divide en uno o más *Compute Units (CUs)* que a su vez se dividen en uno o más *Processing Elements (PEs)*. Los cálculos en un *device* se producen dentro de los PEs [14]. La organización de un sistema OpenCL según el modelo *Platform* se puede observar en la Figura 3.2.

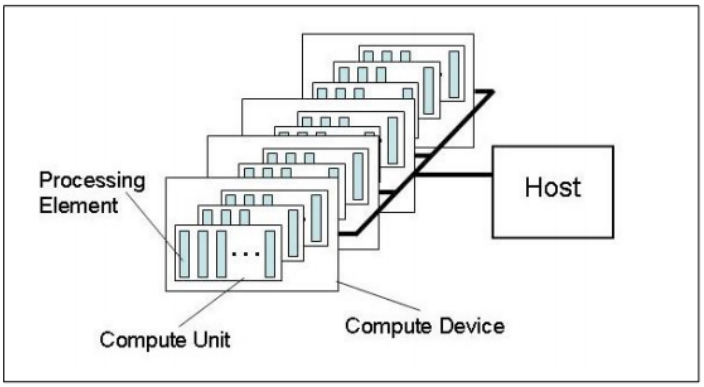


Figura 3.2: Modelo de *Platform* de OpenCL [14].

- **Modelo de ejecución:** en la ejecución de un programa en OpenCL se distinguen dos partes: los *kernels*, que son ejecutados por los *devices*, y el *programa host*, que es ejecutado por el *host* y se encarga de gestionar la ejecución de los *kernels* [14]. Para ello, el *host* define un *context* (creado y manipulado con funciones de la API de OpenCL) que incluye los siguientes recursos:

  - ***Devices*:** colección de *OpenCL devices* que ejecutan los *kernels*.
  - ***Kernels*:** funciones escritas en lenguaje OpenCL que se ejecutan en los *devices*.

- ***Command Queues*:** estructuras de datos para coordinar la ejecución de los *kernels* en cada *device*. Es necesario destacar, que deberá existir una *command queue* por cada *device* utilizado.
- ***Program Objects*:** conjunto de *kernels* de un mismo fichero fuente.
- ***Memory Objects*:** 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), imágenes (bidimensionales) o volúmenes (tridimensionales).

Cuando un kernel es enviado desde el *host* para ejecutarse en el *device* se define un espacio de índices, de forma que una instancia del kernel (*work-item*) es ejecutada para cada índice de ese espacio. Cada *work-item* ejecuta el mismo código pero la ruta de ejecución y los datos operados pueden variar. Por otro lado, los *work-items* están organizados en *work-groups*. Todos los *work-items* de un *work-group* dado se ejecutan simultáneamente en los *processing elements* de una única *compute unit* [14].

- **Modelo de memoria:** los *work-items* que ejecutan instancias de un kernel tienen acceso a cuatro regiones de memoria:
  - **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:** formada por el conjunto de registros conectados a una ALU concreta. Esta zona de memoria solo es accesible para un único *work-item*.
- **Modelo de programación:** OpenCL soporta dos modelos de programación paralela:
  - **Paralelización de datos:** la secuencia de instrucciones es aplicada de forma paralela a los múltiples elementos de un objeto de memoria. El espacio de índices asociado al modelo de

ejecución define los *work-items* y como los datos se asignan a estos. En una programación estrictamente paralela de datos se tendría una asociación de uno a uno entre cada *work-item* y cada elemento de un objeto de memoria. No obstante, OpenCL permite una implementación más relajada en la que no es necesaria una paralelización tan estricta.

- **Paralelización de tareas:** existen varias tareas concurrentes, donde cada tarea ejecuta una única instancia de un *kernel* independientemente del espacio de índices. Las tareas concurrentes pueden ejecutar diferentes *kernels*. 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 *compute unit*.

3.3 Estructura de un programa en OpenCL

La estrategia general que se sigue en un programa en OpenCL es la siguiente [17]:

1. Obtener las *platforms* existentes y escoger una.
2. Obtener los *devices* disponibles en la *platform* y elegir uno o varios.
3. Crear un *context* con los *devices* escogidos.
4. Asociar un *command-queue* a cada *device* del *context*.
5. Crear un objeto *program* a partir del fichero con extensión *.cl* que contiene el código fuente del *kernel*.
6. Compilar el *program* para los *devices* del *context*.
7. Cargar los datos de entrada en la memoria del *host*.
8. Copiar los datos a objetos de memoria (*buffers*) del *context* de OpenCL y reservar un objeto de memoria para los datos de salida.
9. Crear un objeto *kernel* a partir del objeto *program* compilado.
10. Apuntar los parámetros del *kernel* a los objetos de memoria.
11. Lanzar el *kernel*.
12. Esperar a que el *kernel* termine de ejecutarse.
13. Copiar el objeto de memoria de salida a la memoria del *host*.

14. Liberar los recursos previamente reservados.

En la implementación realizada en este Trabajo Fin de Grado se sigue ordenadamente la secuencia característica de un programa en OpenCL. En cuanto a la *platform* se selecciona la primera disponible por simplicidad, y en lo referente a los *devices* se selecciona la GPU siempre que exista una disponible. Posteriormente, se crean el *context* y la *command-queue* (solo existirá una dado que solo se utiliza un *device*). A continuación, se crean los objetos de memoria y los *kernels* (véase Secciones 4.1.3 y 4.1.4). Y una vez estos han sido creados se apuntan los parámetros del *kernel* a los objetos de memoria y se ejecutan los *kernels*. Por ultimo, se leen los objetos de memoria de salida y se liberan los recursos.

Capítulo 4

Diseño e Implementación

En todo problema relativo a la ingeniería de *software* es necesario abordar las fases de análisis, diseño e implementación. En este caso, se puede considerar que el análisis del problema es intrínseco al propio título del Trabajo Fin de Grado, por lo que en este capítulo se pretende proporcionar una visión de las fases de diseño e implementación.

4.1 Diseño del algoritmo

En esta sección se modela una solución que satisfaga los requisitos detectados que debe cumplir el algoritmo de simulación de síntesis de imágenes de resonancia magnética. Concretamente, se describe el algoritmo diseñado, así como las variables, objetos de memoria y *kernels* necesarios.

4.1.1 Descripción del algoritmo

En este Trabajo Fin de Grado se lleva a cabo la simulación de la secuencia de pulsos EPI a partir de una secuencia *Spin echo* monoeco (SE-EPI). En esta secuencia, el tren de gradientes de lectura en la dirección de codificación de frecuencia (dirección X) y el tren de gradientes blip en la dirección de codificación de fase (dirección Y) permiten adquirir todas las líneas del espacio K a partir de un único pulso de RF, lo que provoca una reducción considerable en los tiempos de captación de las imágenes en comparación con otras secuencias de pulsos.

Los pasos que se siguen en la simulación de esta secuencia son los que se detallan a continuación:

1. Se parte del 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.

2. Se simula una inhomogeneidad del campo magnético, que al ser función de cada punto del *slice* excitado dará lugar a un efecto de *off-resonance*.

3. Se aplica un número variable de pulsos preparatorios, los cuales consisten en una combinación de pulsos de 90º (pulsos de saturación) y 180º (pulsos de inversión) junto con las respectivas relajaciones de la señal.

4. Se aplica el pulso de saturación (90º) que se traduce en un giro del vector de magnetización, lo cual es implementado mediante matrices de rotación (ver apéndice A).

5. Se simula una rotación debida a la inhomogeneidad del campo y a los gradientes aplicados en las direcciones de codificación de frecuencia (dirección X) y de fase (dirección Y). Estos gradientes son los encargados de situarnos en el punto deseado del espacio K para comenzar la lectura del mismo.

6. Simulación de la relajación de los tejidos entre el pulso de saturación (90º) y el de inversión (180º).

7. Se aplica el pulso de inversión (180º) que provoca un efecto similar al pulso anterior pero variando los ángulos de rotación.

8. Se vuelve a simular otro efecto de relajación de los tejidos.

9. Llegados a este punto, comienza la lectura del espacio K, el cual se recorre siguiendo una trayectoria rectilínea como se puede observar en la Figura 4.1. Para obtener cada línea del espacio K es necesario aplicar el gradiente de lectura en la dirección de codificación de frecuencia (dirección X). La primera línea se obtiene por el eco generado mediante los pulsos de saturación e inversión. Para las restantes líneas es necesario aplicar por cada una un gradiente blip en la dirección Y que permita ir desplazándose en sentido vertical por el espacio K.

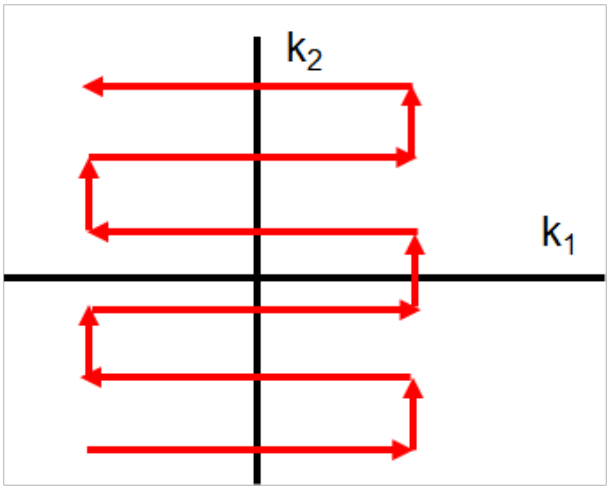


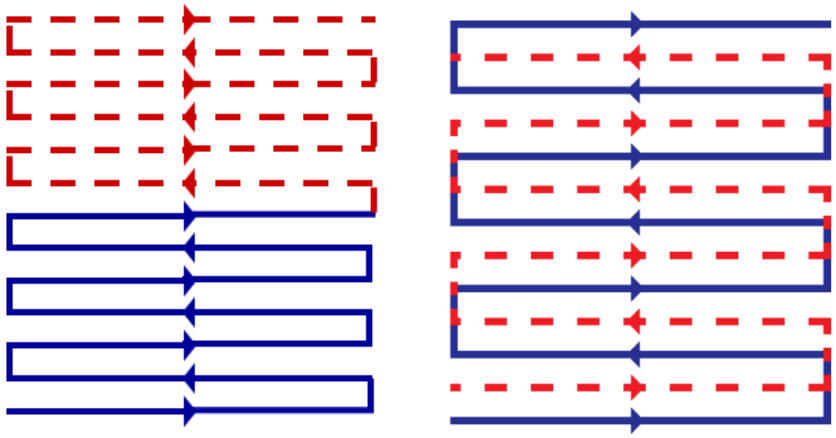
Figura 4.1: Recorrido de escritura del espacio K con la secuencia *single-shot SE-EPI* implementada.

Los pasos explicados anteriormente se corresponderían con una secuencia SE-EPI básica (*single-shot SE-EPI*), en la que todo el espacio K se adquiere a partir de una única excitación de RF. No obstante, según se ha diseñado la simulación, existe la posibilidad de simular también secuencias *multi-shot SE-EPI*, en la que el espacio K se adquiere por bloques a partir de varias excitaciones de RF. En este segundo tipo de secuencias EPI, el número de líneas que se adquieren en cada excitación se conoce como *ETL (Echo Train Length)*, *EPI factor* o *shot factor*. En el diseño realizado, este parámetro es variable y podrá ser escogido por el usuario, con la limitación de que el tamaño total de la imagen deberá ser divisible por el valor de este parámetro.

En este segundo caso tiene sentido la aplicación de los pulsos preparatorios, ya que de esa forma se consigue que todos los bloques del espacio K se obtengan con la misma cantidad de magnetización excitada. Por otro lado, para conseguir este diseño será necesario la introducción de una iteración en el algoritmo que permita repetir los pasos 4, 5, 6, 7, 8 y 9 tantas veces como números de bloques del espacio K se desee tener.

Al introducir la posibilidad de simular secuencias *multi-shot SE-EPI*, surge el problema de cómo ir escribiendo cada bloque del espacio K dentro de la trayectoria rectilínea. En relación con esto se contemplan dos posibilidades distintas de escritura de los bloques del espacio K:

- **Secuencial**: los bloques se escriben de forma secuencial como está representado en la Figura 4.2.
- **Interlineado**: las líneas de cada bloque se entrelazan como está representado en la Figura 4.2.



(a) Secuencial (b) Interlineado

Figura 4.2: Recorrido de escritura del espacio K con la secuencia *multi-shot EPI*.

4.1.2 Inicialización de parámetros

Lo primero que se lleva a cabo en el programa es la inicialización de los parámetros necesarios posteriormente, los cuales se pueden agrupar en distintos tipos:

- **Inicialización del entorno de OpenCL**: obligado en OpenCL para preparar el entorno dónde después se ejecutarán los *kernels*. Se distinguen varios:

  - ***Platform***: se obtiene con la función *clGetPlatformIDs()* de la API de OpenCL. Aunque existe la posibilidad de trabajar utilizando varias *platforms*, en este programa, por simplicidad, siempre se va a seleccionar la primera disponible como única *platform* de trabajo.
  - ***Devices***: se obtiene con la función *clGetDeviceIDs()* de la API de OpenCL. EL *device* de trabajo será la GPU, siempre que haya alguna disponible. No obstante, si no hay ninguna GPU disponible, se selecciona la CPU como *device*.
  - ***Context***: se crea un *context* asociado a cada *device*. Para ello se utiliza la función *clCreateContext()*.
  - ***CommandQueue***: se crea una *commandQueue* asociada con cada *context* mediante la función *clCreateCommandQueue()*.
  - ***Program***: se leen todos los archivos *.cl* que contienen el código fuente de los kernels mediante la función *convertToString()*. El código fuente de todos los *kernels* se concatena en una única cadena. Posteriormente, se crea el objeto *program* a partir de esta cadena con la función *clCreateProgramWithSource()* y este se compila en tiempo de ejecución, mediante la función *clBuildProgram()*.

- **Inicialización de parámetros de la simulación**

  Los modelos anatómicos del T_1 (Tiempo de relajación longitudinal), T_2 (Tiempo de relajación transversal) y PD (Densidad de protones) se leen como archivos *.raw*, que han sido escritos previamente desde MATLAB. Por otro lado, el mapa de inhomogeneidad del campo magnético también se obtiene de forma similar.

4.1.3 Creación de objetos de memoria

Los datos que se procesan en OpenCL deben residir en objetos de memoria para que puedan ser manipulados en los *devices* (en este caso la GPU siempre que exista una disponible). Como se describió en el capítulo 3, los objetos de memoria en OpenCL pueden ser *buffers* o *imágenes*. En este caso, por simplicidad, se optó por definir los objetos de memoria como *buffers*. Los objetos de memoria creados mediante la función *clCreateBuffer()* en este programa son los siguientes:

- **pdIncioBuffer**: se inicializa con el valor inicial del PD, y permanece invariable a lo largo de toda la simulación. Se define con un tamaño de $3 \times sizeof(float) \times width \times height$.
- **pdBuffer**: se inicializa con el valor inicial del PD, pero va cambiando de valor a lo largo de la simulación. Se define con un tTamaño de $3 \times sizeof(float) \times width \times height$.
- **t1Buffer**: se inicializa con el valor del T_1 y permanece invariable a lo largo de toda la simulación. Se define con un tamaño de $sizeof(float) \times width \times height$.
- **t2Buffer**: se inicializa con el valor del T_2 y permanece invariable a lo largo de toda la simulación. Se define con un tamaño de $sizeof(float) \times width \times height$.
- **deltaBrBuffer**: se inicializa con el mapa de inhomegeneidad del campo y permanece invariable a lo largo de toda la simulación. Se define con un tamaño de $sizeof(float) \times width \times height$.
- **angleGradBuffer**: se inicializa a NULL ya que en este objeto de memoria se escriben datos calculados por los *kernels*, concretamente los ángulos de rotación para cada píxel como consecuencia del gradiente después del pulso de 90^o y del gradiente blip. Se define con un tamaño de $sizeof(float) \times width \times height$.
- **angleGradXBuffer**: se inicializa a NULL ya que en este objeto de memoria se escriben datos calculados por los *kernels*, concretamente los ángulos de rotación para cada píxel como consecuencia del gradiente de lectura en la dirección de codificación de frecuencia X. Como este gradiente depende del tiempo, las dos primeras dimensiones de este objeto de memoria hacen referencia al espacio y la tercera al tiempo. Se define con un tamaño de $sizeof(float) \times width \times height \times depth$.
- **tiempoBuffer**: se inicializa con el valor del vector de tiempo y permanece invariable a los largo de toda la simulación. Se define con un tamaño de $sizeof(float) \times depth$.
- **temporalBuffer**: se inicializa a NULL ya que este objeto de memoria se utiliza para que el *kernel* escriba el valor del PD para cada instante de tiempo, después de aplicar el gradiente de lectura en X, de forma que las dos primeras dimensiones del objeto de memoria hacen referencia al espacio y la tercera al tiempo. Se define con un tamaño de $sizeof(float) \times width \times height \times depth \times 3$.
- **espacioKrealBuffer**: se inicializa a NULL ya que este objeto de memoria se utiliza para escribir la salida del *kernel* que suma por columnas la componente X (que se corresponderá con la parte real del espacio K) para cada instante de tiempo. Se define con un tamaño de $sizeof(float) \times width \times height$.
- **espacioKimagBuffer**: se inicializa a NULL ya que este objeto de memoria se utiliza para escribir la salida del *kernel* que suma por columnas la componente Y (que se corresponderá con la parte imaginaria del espacio K) para cada instante de tiempo. Se define con un tamaño de $sizeof(float) \times width \times height$.

- **espacioKrealBuffer2**: se inicializa a NULL ya que este objeto de memoria se utiliza para escribir la salida del *kernel* que suma por filas la componente X para cada instante de tiempo. El resultado de la suma se corresponde con la parte real de una fila del espacio K y esto es lo que se almacena en este objeto de memoria que representa la parte real final del espacio K. Se define con un tamaño de *sizeof(float)*×*width*×*height*.

- **espacioKimagBuffer2**: se inicializa a NULL ya que este objeto de memoria se utiliza para escribir la salida del *kernel* que suma por filas la componente Y para cada instante de tiempo. El resultado de la suma se corresponde con la parte imaginaria de una fila del espacio K y esto es lo que se almacena en este objeto de memoria que representa la parte imaginaria final del espacio K. Se define con un tamaño de *sizeof(float)*×*width*×*height*.

4.1.4 Kernels

Los *kernels* son los programas escritos mediante el lenguaje de programación de OpenCL que se ejecutan en el *device* (el cual será la GPU siempre que exista una disponible). Todos los *kernels* están escritos buscando un paralelismo a nivel de datos. Ya que, como se explico en el capítulo 1, el problema es intrínsecamente paralelo al requerir realizar las mismas operaciones para obtener cada punto de la imagen.

El principal objetivo que se persiguió diseñar los *kernels* fue reducir el tiempo de computación necesario para obtener la imagen de resonancia magnética final. No obstante, siempre se buscó un compromiso entre reducir estos tiempos y que el código fuera lo más genérico posible. De forma que fuera sencillo reutilizarlo para implementar, por ejemplo, otras secuencias de pulsos.

A continuación se describen en más detalle cada uno de los *kernels* implementados:

- **Pulso**: este *kernel* se utiliza para simular la aplicación de un pulso de α grados. En el caso de la secuencia de pulsos EPI implementada en este código, se aplican pulsos de 90^{o} (pulso de saturación) y 180^{o} (pulso inversión). La aplicación del pulso se simula como un giro del vector de magnetización, lo cual es implementado mediante matrices de rotación (ver apéndice A).

- **Relajacion**: este *kernel* se utiliza para simular la relajación de la señal en los pulsos preparatorios, después del pulso de 90^{o}, después del pulso de 180^{o}, después de cada gradiente Blip y desde la última línea de cada bloque del espacio K hasta el tiempo de repetición (TR).

- **CreateGrad90**: este *kernel* se encarga de calcular el ángulo de rotación del vector de magnetización como consecuencia de la inhomogeneidad del campo y los dos gradientes aplicados en las

direcciones de codificación de fase y frecuencia durante un tiempo $T_{pe} = T_{acq}/2$ tras el pulso de 90º.

- **CreateGradBlip**: este *kernel* se encarga de calcular el ángulo de rotación del vector de magnetización como consecuencia del gradiente blip aplicado en la dirección de codificación de fase durante un tiempo $T_{pe} = T_{acq}/2$.

- **CreateGradX**: este *kernel* se encarga de calcular el ángulo de rotación del vector de magnetización como consecuencia de la inhomogeneidad del campo y del gradiente de lectura en la dirección de codificación de frecuencia.

- **Grad**: este *kernel* se encarga de simular la rotación del vector de magnetización como consecuencia de la aplicación de los gradientes tras el pulso de 90º y de los gradientes blip.

- **Grad2**: este *kernel* se encarga de simular la rotación del vector de magnetización como consecuencia de la aplicación del gradiente de lectura en la dirección de codificación de frecuencia (dirección X).

- **CopiarMatriz**: este *kernel* se utiliza para copiar una matriz (*temporalBuffer*) de dimensiones $sizeof(float) \times width \times height \times depth \times 3$ a otra matriz (*pdBuffer*) de dimensiones $sizeof(float) \times width \times height \times 3$.

- **EspacioK1**: este *kernel* suma por columnas la componente X (parte real del espacio K) y la componente Y (parte imaginaria del espacio K) para cada instante de tiempo. El resultado de la suma de cada componente lo guarda en dos objetos de memoria diferentes (*espacioKrealBuffer* y *espacioKimagBuffer*).

- **EspacioK2**: este *kernel* suma por filas la componente X (parte real del espacio K) y la componente Y (parte imaginaria del espacio K) para cada instante de tiempo. El resultado es la obtención de una fila del espacio K, en la que cada instante de tiempo hace referencia a un punto de la fila. Cada una de las componentes se guardan en dos objetos de memoria diferentes, *espacioKrealBuffer2* y *espacioKimagBuffer2*, que representan la parte real e imaginaria final del espacio K simulado.

4.1.5 Archivos de salida

Mediante la función *clEnqueueReadBuffer* de la API de OpenCL, los objetos de memoria *espacioKrealBuffer2* y *espacioKimagBuffer2* que representan respectivamente la parte real e imaginaria del espacio K, se copian desde la memoria de la GPU a las variables *EspacioK_real* y *EspacioK_imag* de la memoria del *host*.

Posteriormente, los datos de estas variables se escriben en dos archivos de salida *.raw*. Estos archivos son leídos en MATLAB donde se juntan las partes reales e imaginarias de cada punto del espacio K y se realiza la transformada inversa de Fourier para obtener la imagen de resonancia magnética final.

4.1.6 Diagrama de flujo del algoritmo

Finalmente, el algoritmo completo diseñado es el representado en el diagrama de flujo de la Figura 4.3.

4.2 Implementación del algoritmo

A partir del diseño planteado en la sección anterior, en esta sección se proporciona una visión de la implementación realizada. Para ello se describe la clase creada con sus correspondientes atributos y métodos.

4.2.1 Clase *Simulator*

En la implementación del algoritmo de simulación de síntesis de imágenes de resonancia magnética diseñado se utiliza programación orientada a objetos con C++. Con ello se persigue, por un lado, dotar al código de cierta flexibilidad creando métodos que implementen partes diferenciadas de la secuencia de pulsos y que puedan ser reutilizadas fácilmente en la implementación, por ejemplo, de otras secuencias de pulsos. Si bien es cierto que esto se podría haber llevado a cabo con una programación orientada a procedimientos, dado que un objeto posee un estado propio, la utilización de programación orientada a objetos permite obtener esta funcionalidad sin necesidad de pasar demasiados argumentos a las funciones 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.

Concretamente, en la implementación realizada se crea una clase denominada *Simulator*, representada en la Figura 4.4, que incluye los atributos y métodos que se detallan a continuación.

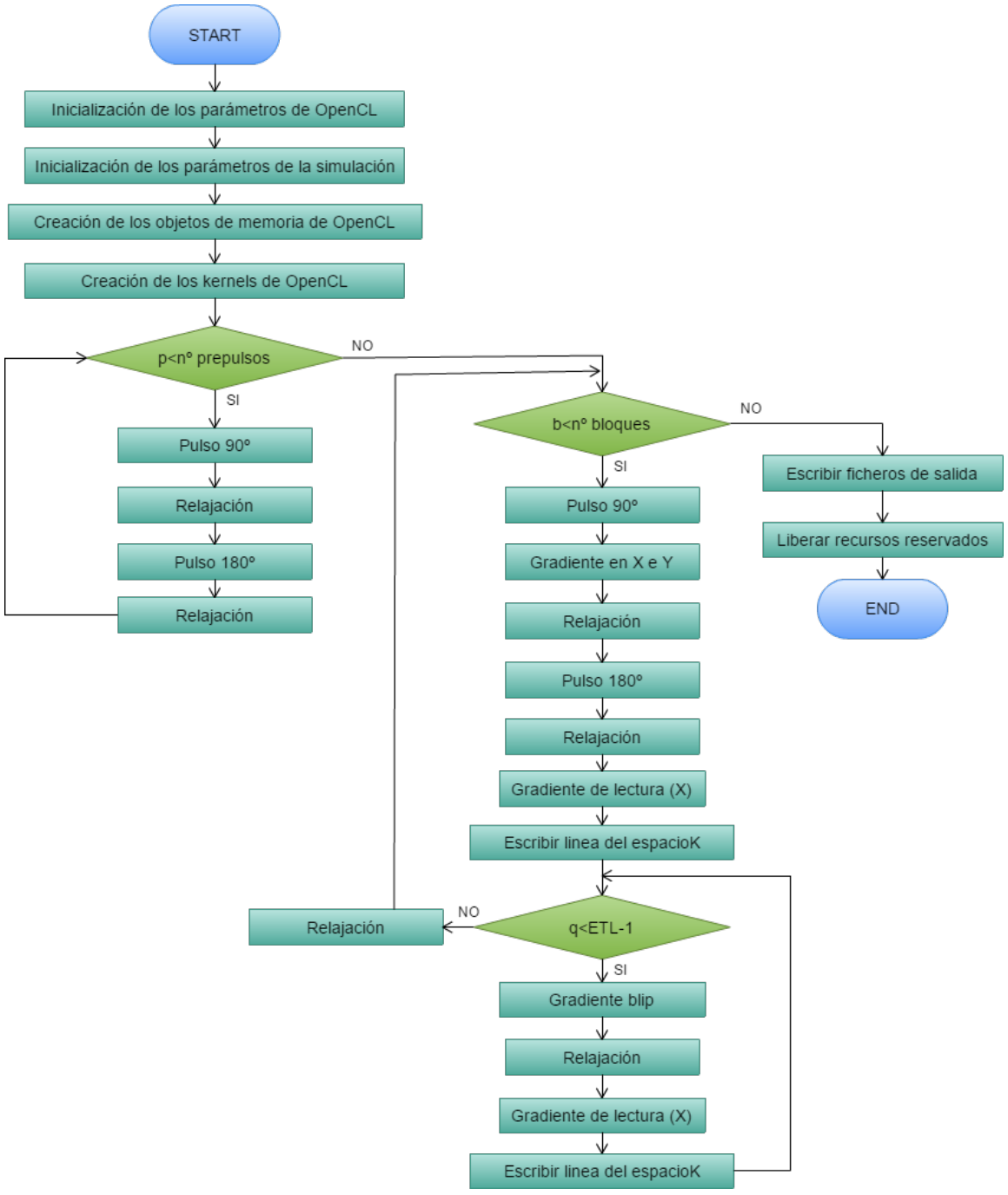


Figura 4.3: Diagrama de flujo del algoritmo diseñado.

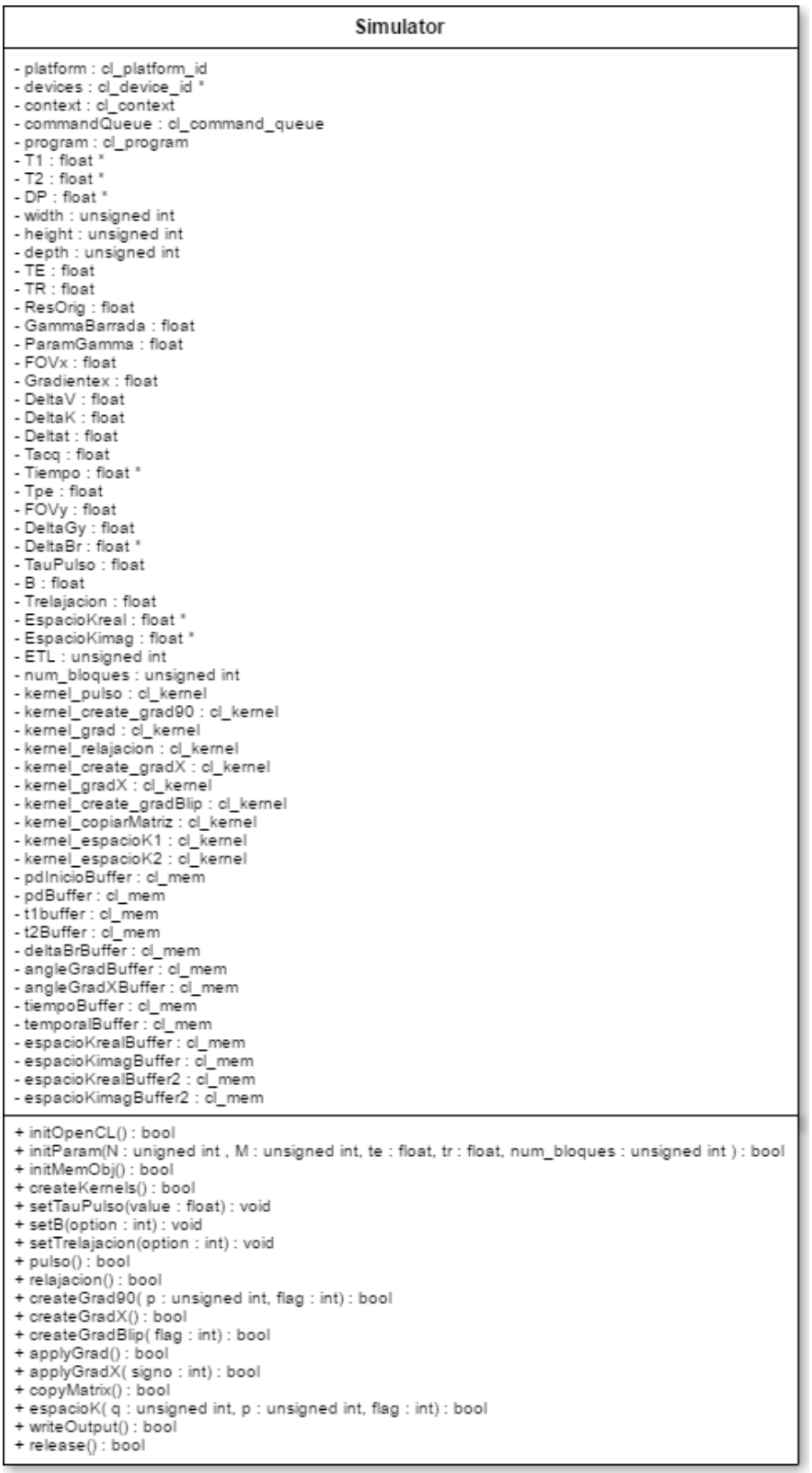


Figura 4.4: Clase *Simulator*.

Atributos

Con el objetivo de cumplir la propiedad de encapsulamiento característica de la programación orientada a objetos, evitando así manipulaciones indebidas, intencionadas o no, del estado de los objetos, la totalidad de los atributos de esta clase van a ser privados, es decir, el acceso a los mismos solo podrá darse desde el interior de la clase que los posee.

Entre los atributos de esta clase se encuentran:

- Entorno de OpenCL.
- Parámetros de la simulación.
- *Kernels* de OpenCL.
- Objetos de Memoria de OpenCL.

Métodos

Al contrario que los atributos, la totalidad de los métodos de esta clase van a ser públicos, por lo que el acceso al mismo puede darse dentro del interior de la clase, dentro de una subclase, y desde un objeto instanciado de cualquiera de estas. Esto es necesario para que los métodos sean accesibles desde otras partes del programa.

Por otro lado, todos los métodos, salvo los mutadores (aquellos que permiten establecer el valor de un atributo de la clase), devuelven un valor de tipo booleano que permite llevar a cabo la gestión de errores de OpenCL. Según se ha realizado la implementación, cada uno de los métodos se va a encargar de fijar los argumentos y ordenar la ejecución de los kernels correspondientes.

Los métodos implementados en la clase *Simulator*, son los que se enumeran a continuación:

- ***Simulator()***: constructor de la clase.
- ***bool initOpenCL()***: método para inicializar los atributos de la clase que permiten inicializar el entorno de OpenCL.
- ***bool initParam(unsigned int N, unsigned int M, float te, float tr, int num_bloques)***: método para inicializar los atributos de la clase que son parámetros de la simulación. A este método se le pasan como argumentos las dimensiones de las imágenes (`N` y `M`), el tiempos de eco (`te`), el tiempo de repetición (`tr`) y el número de bloques en los que se quiere escribir el espacio K (`nun_bloques`), ya que estos parámetros podrán ser seleccionados por el usuario mediante la interfaz.

- ***bool initMemObj()***: método para crear los objetos de memoria de OpenCL.
- ***bool createKernels()***: método para crear los *kernels* de OpenCL.
- ***void setTauPulso(float value)***: método para establecer el valor del atributo `TauPulso` de la clase, que hace referencia al tiempo durante el cual se aplica el pulso. A este método se le pasa como argumento la variable `value` que permite determinar el valor al que se debe inicializar la variable `TauPulso` dependiendo del tipo de pulso que se quiera aplicar.
- ***void setB(int option)***: método para establecer el valor del atributo `B` de la clase, que hace referencia a la amplitud del pulso aplicado. A este método se le pasa como argumento la variable `option` que dependiendo de su valor permitirá simular la aplicación de distintos tipos de pulsos.
- ***void setTrelajacion(int option)***: método para establecer el valor del atributo `Trelajacion` de la clase, que hace referencia a el tiempo durante el cual se simula la relajación de la señal. A este método se le pasa como argumento la variable `option` que dependiendo de su valor permitirá simular distintos tiempos de relajación de la señal.
- ***bool pulso()***: método para simular la aplicación del pulso de α grados. En la secuencia SE-EPI implementada se utiliza para aplicar el pulso de 90^o (pulso de saturación) y el 180^o (pulso de inversión).
- ***bool relajacion()***: método para simular la relajación de la señal.
- ***bool createGrad90(unsigned int p, int flag)***: método para crear el gradiente en las direcciones de codificación de frecuencia y fase (direcciones X e Y respectivamente) que se aplica después del pulso de 90^o. A este método se le pasan como argumentos la variable `p` que indica el número de bloque del espacio K que se va a escribir, y la variable `flag` que permite determinar si la escritura del espacio K se va a realizar de forma secuencial o con interlineado.
- ***bool createGradX()***: método para crear el gradiente de lectura (dirección X).
- ***bool createGradBlip(int flag)***: método para crear el gradiente blip. A este método se le pasa como argumento la variable `flag` que permite determinar si la escritura del espacio K se va a realizar de forma secuencial o con interlineado.
- ***bool applyGrad()***: método para aplicar el gradiente. En esta implementación se utilizará para aplicar tanto el gradiente posterior al pulso de 90^o como los gradientes blip.
- ***bool applyGradX(int signo)***: método para aplicar el gradiente de lectura (dirección X). A este método se le pasa como argumento la variable `signo` que indica si el gradiente de lectura en la dirección de codificación de frecuencia X se tiene que aplicar positivo o negativo dependiendo de si la línea del espacio K que se va a escribir es impar o par respectivamente.

- ***bool copyMatrix()***: método para copiar una matriz de *width*×*height*×*depth*×*3* a una matriz *width*×*height*×*3*.
- ***bool espacioK(unsigned int q, unsigned int p, int flag)***: método para escribir el espacio K. A este método se le pasan como argumentos las variables `q` que indica el número de línea dentro de cada bloque del espacio K que se va a escribir, la variable `p` que indica el número de bloque del espacio K que se va a escribir y la variable `flag` que permite determinar si la escritura del espacio K se va a realizar de forma secuencial o con interlineado.
- ***bool writeOutput()***: método para copiar la salida a la memoria del *host* y escribir los archivos de salida.
- ***bool release()***: método para liberar los recursos previamente asignados.

4.3 Diseño e implementación de la GUI

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, algunos parámetros de la simulación y, a su vez, observar las imágenes de salida, se crea una GUI (*Graphical User Interface)* mediante MATLAB. Esta interfaz de usuario es la representada en la Figura 4.5.

El funcionamiento interno de la interfaz implementada se puede describir en tres pasos:

1. Cuando el usuario pulsa el botón *Cargar* se leen los tres archivos especificados 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.
3. 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. Y mediante el comando *system()* de MATLAB se ejecuta el ejecutable del código en 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.

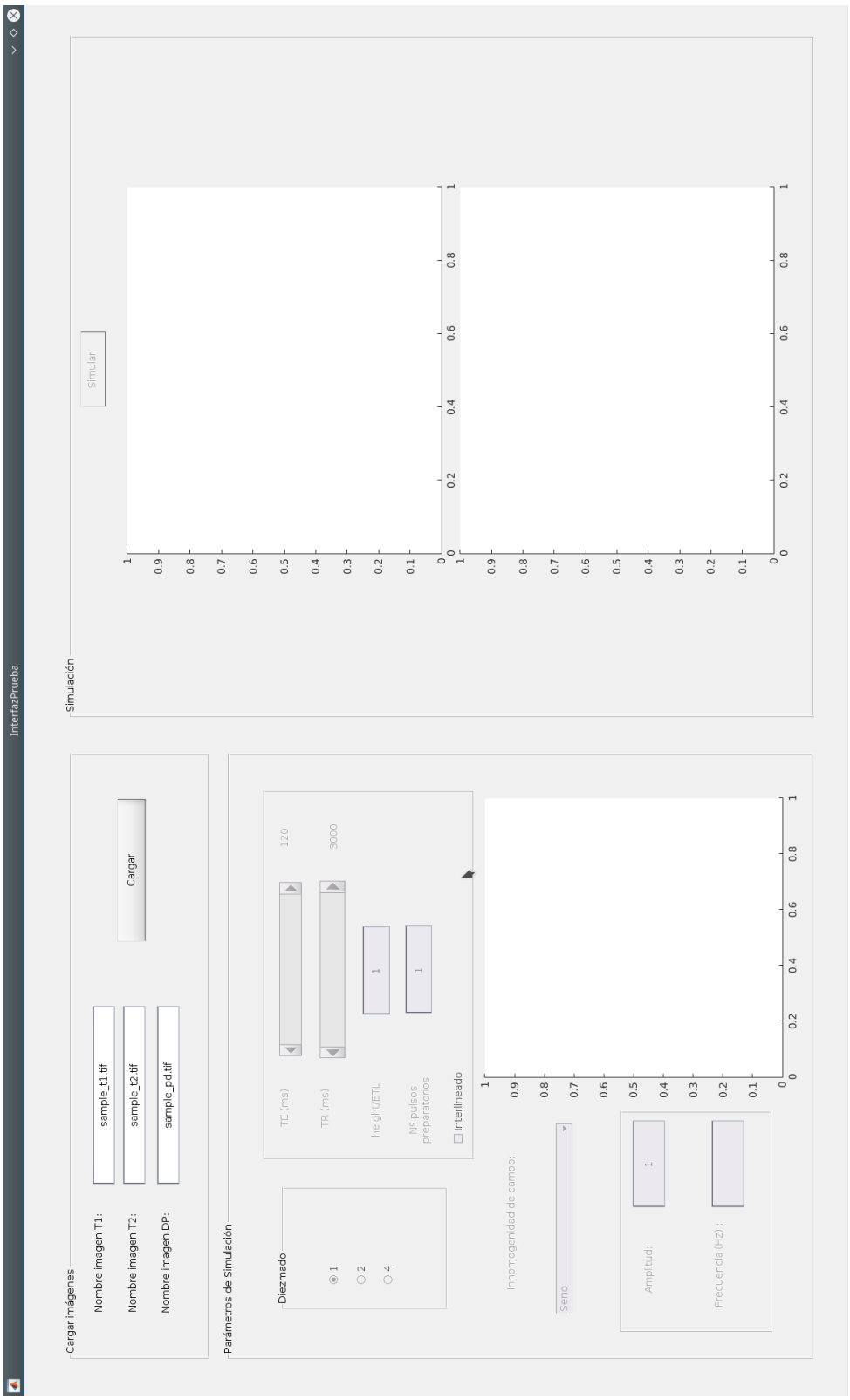


Figura 4.5: Interfaz de usuario creada mediante MATLAB.

Por último, una vez el código en OpenCL ha terminado de ejecutarse, se leen los 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.

4.3.1 Manual 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 archivos.

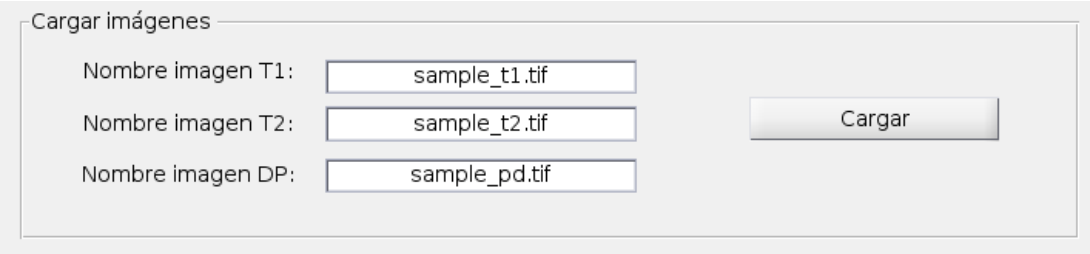


Figura 4.6: Cargar modelos anatómicos en la interfaz.

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). 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. Por otro lado, también es necesario fijar un mapa de inhomogeneidad del campo magnético, para lo que hay siete opciones disponibles:

- Sin inhomogeneidad.
- Constante.
- Gradiente en la dirección X.
- Gradiente en la dirección Y.
- Función sinusoidal.
- Función sinusoidal transpuesta.
- Función sinc.

En todos ellos, salvo en el primero, existe la posibilidad de fijar la amplitud de la inhomegeneidad. Además, en las funciones sinusoidales también se debe fijar la frecuencia de la misma.

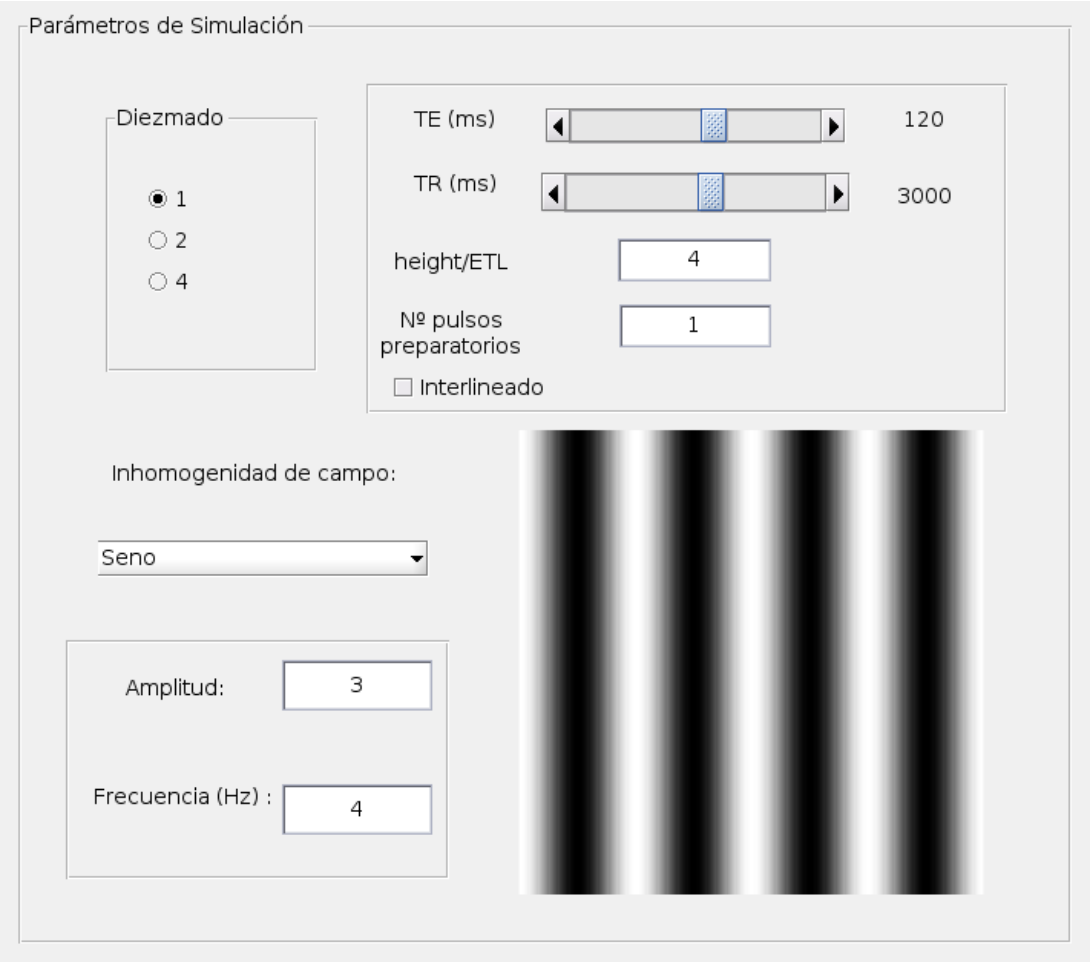


Figura 4.7: Fijar parámetros de simulación en la interfaz.

Por último, una vez todos los parámetros han sido fijados y la función de inhomogeneidad del campo representada, se habilita el último panel, representado en la Figura 4.8. En el cual, cuando el usuario pulse el botón *Simular* se podrá observar el espacio K simulado y su correspondiente imagen de resonancia magnética final.

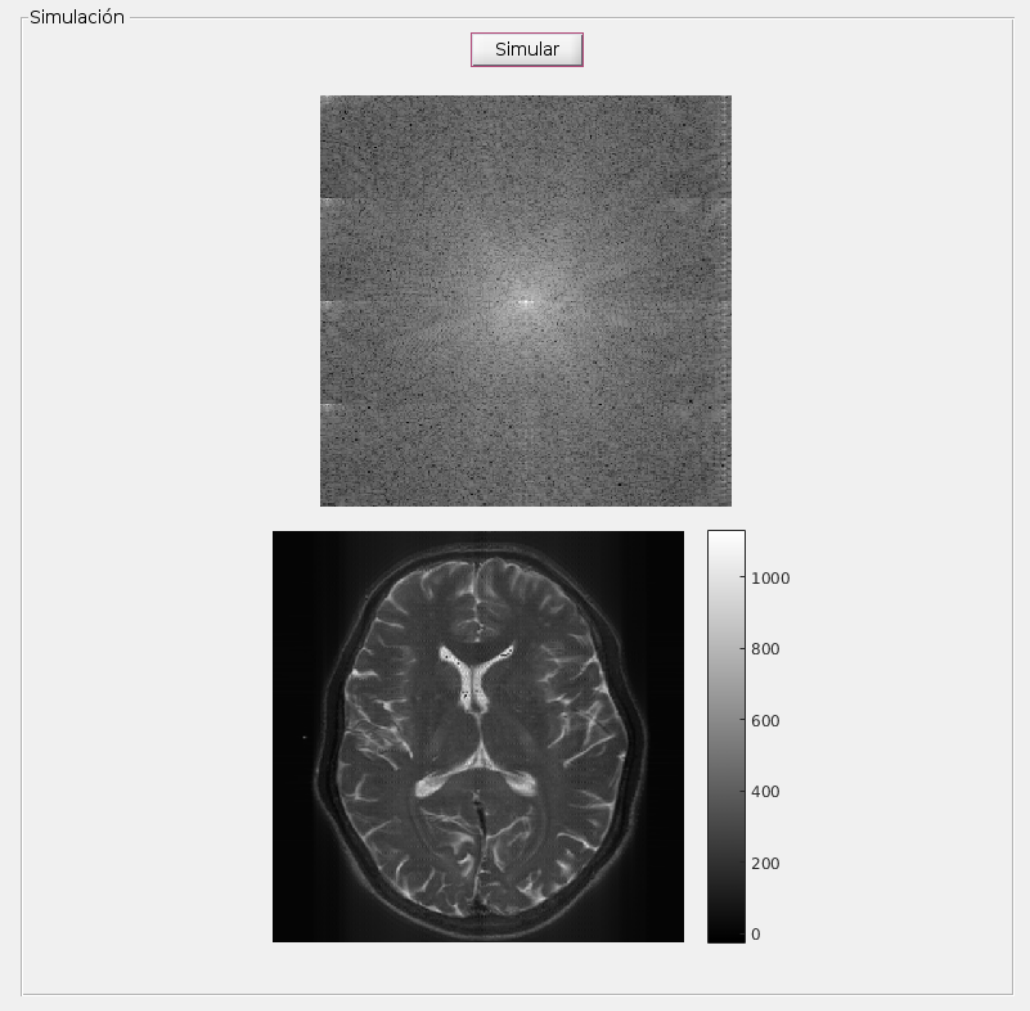


Figura 4.8: Imágenes de salida en la interfaz.

Capítulo 5

Resultados

En este capitulo se analizan los resultados obtenidos, tanto a nivel de validación visual de las imágenes de resonancia magnética como a nivel de evaluación de las medidas de rendimiento del algoritmo.

5.1 Metodología de evaluación de resultados

Para evaluar los resultados se toma el modelo anatómico de un slice del cerebro formado por las imágenes de tiempo de relajación longitudinal (T1), tiempo de relajación transversal (T2), y densidad de protones (PD) de los diferentes tejidos del mismo. Estos datos son obtenidos de la base de datos disponible en el grupo en el que se realiza el trabajo, el LPI.

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

I. **Comprobación visual**, se validarán de forma visual las imágenes de resonancia magnética obtenidas para tamaños de 256×256 píxeles. En la comprobación se variarán tanto el número de excitaciones utilizadas para escribir el espacio K correspondientes a las imágenes (una y dos excitaciones) como las funciones de inhomogeneidades del campo simuladas, que serán las que se enumeran a continuación:

   - Sin inhomogeneidad.
   - Constante.
   - Gradiente en la dirección X.
   - Gradiente en la dirección Y.
   - Función sinusoidal.
   - Función sinusoidal transpuesta.

- Función sinc.

II. **Medidas de rendimiento**, se comprobarán los tiempos de ejecución para distintos tamaños de imágenes (64×64, 128×128 y 256×256 píxeles).

5.2 Comprobación visual de resultados

A continuación se analizarán de forma visual las imágenes de resonancia magnética obtenidas.

En la Figura 5.1 se muestran las imágenes obtenidas con una y dos excitaciones sin inhomogeneidad del campo. En ese caso, no se producirá el efecto de *off-resonance* ni se tendrá en cuenta la inhomogeneidad del campo en los gradientes aplicados. Por otro lado, como era de esperar, al aumentar el número de excitaciones, el nivel de intensidad de la imagen es mayor dado que la señal se atenúa menos.

En la Figura 5.2 se presentan las imágenes de resonancia magnética obtenidas con una función de inhomogeneidad del campo constante. En ese caso, el efecto producido sobre la imagen consiste en un desplazamiento hacia abajo, es decir, en la dirección de codificación de fase (dirección Y).

En la Figura 5.3 se muestran las imágenes de resonancia magnética obtenidas cuando se aplica una inhomogeneidad del campo en forma de gradiente en la dirección de codificación de frecuencia (dirección X). Como consecuencia de esta inhomogeneidad se produce un escalado de la imagen final, concretamente la imagen se expande dado que el gradiente es positivo. Por el contrario, en la Figura 5.4 las imágenes mostradas se corresponden con la aplicación de una inhomogeneidad del campo en forma de gradiente en la dirección de codificación de fase (dirección Y). En ese caso, el efecto producido en las imágenes es consiste en una cizalladura.

En la Figura 5.5, las imágenes de resonancia magnética representadas se corresponden con una función de inhomogeneidad sinusoidal en la dirección X. Esto se podría considerar como un conjunto de varios gradientes en esta dirección (algunos positivos y otros negativos), y como consecuencia se producen varios escalados en la imagen (en algunos lugares se expande y en otros se comprime). Por otro lado, en la Figura 5.6 las imágenes se corresponden con una función de inhomogeneidad sinusoidal en la dirección Y, equivalente también a la aplicación de varios gradientes en esa dirección. Por lo que, en ese caso el efecto sobre la imagen consiste en varias cizalladuras.

Para finalizar, en la Figura 5.7 se muestran los resultados de aplicar como inhomogeneidad de campo

una función sinc. En este caso, el mapa de inhomogeneidad es máximo en el centro, por lo que la distorsión sobre la imagen se produce principalmente en esa zona.

Nótese que todos los efectos producidos por las inhomogeneidades sobre las imágenes se manifiestan, principalmente, en la dirección de codificación de fase Y, dado que en la dirección de codificación de frecuencia, los efectos de los gradientes dominan sobre los efectos de la inhomogeneidad.

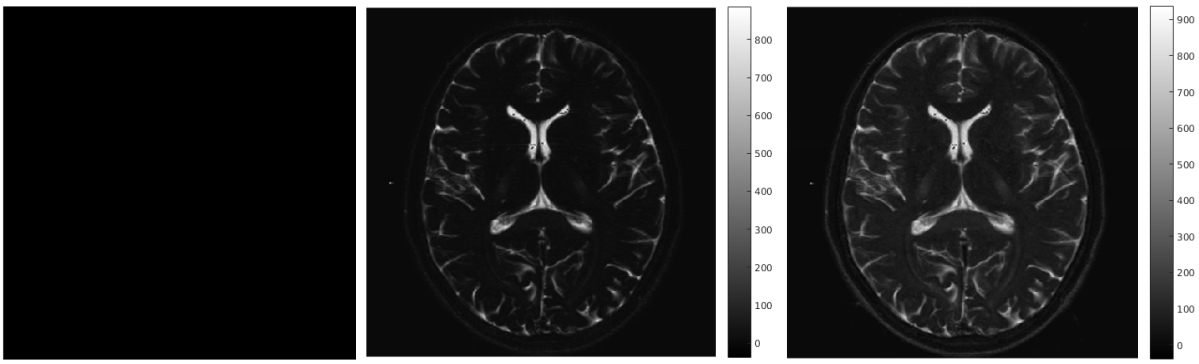


Figura 5.1: De izquierda a derecha, mapa de la inhomogeneidad del campo simulada (sin inhomogeneidad), reconstrucción simulada de la imágenes con una única excitación, y reconstrucción simulada de la imagen con dos excitaciones.

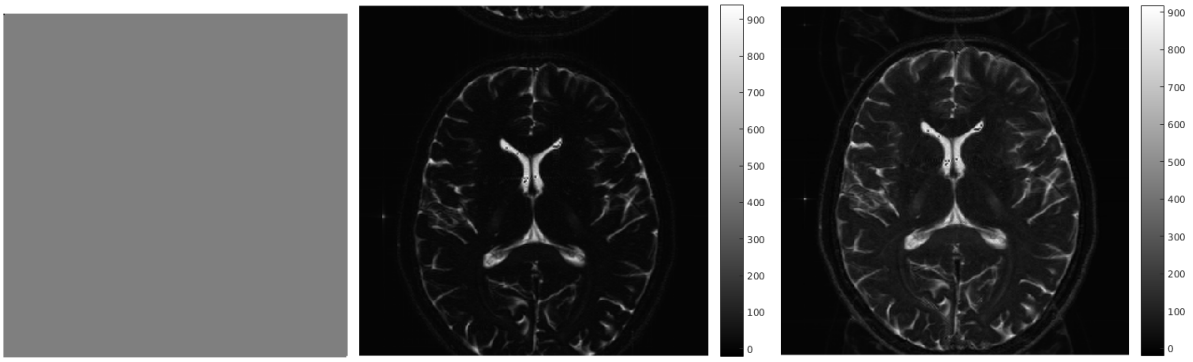


Figura 5.2: De izquierda a derecha, mapa de la inhomogeneidad del campo simulada (función constante), reconstrucción simulada de la imágenes con una única excitación, y reconstrucción simulada de la imagen con dos excitaciones.

Visualmente, los resultados son los esperados, ya que resultan coherentes con los presentados en [18]. Aunque en ese caso los efectos en las imágenes se producen por efecto de las *eddy current* en los gradientes, el considerar una inhomogeneidad de campo magnético al calcular los ángulos de rotación de la magnetización como consecuencia de la aplicación de los gradientes resulta equivalente.

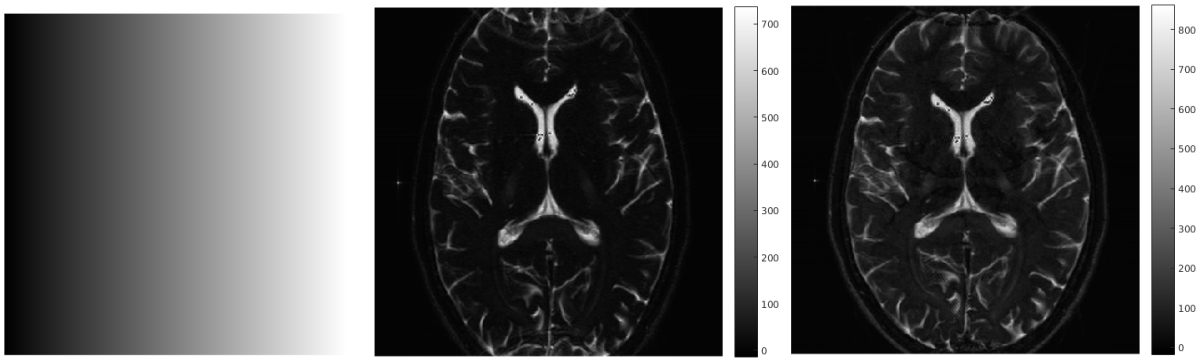


Figura 5.3: De izquierda a derecha, mapa de la inhomogeneidad del campo simulada (gradiente en la dirección de codificación de frecuencia X), reconstrucción simulada de la imágenes con una única excitación, y reconstrucción simulada de la imagen con dos excitaciones.

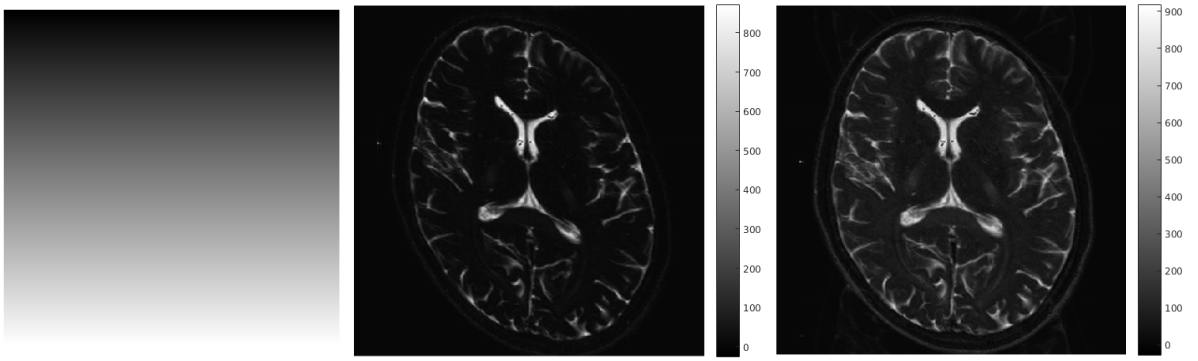


Figura 5.4: De izquierda a derecha, mapa de la inhomogeneidad del campo simulada (gradiente en la dirección de codificación de fase Y), reconstrucción simulada de la imágenes con una única excitación, y reconstrucción simulada de la imagen con dos excitaciones.

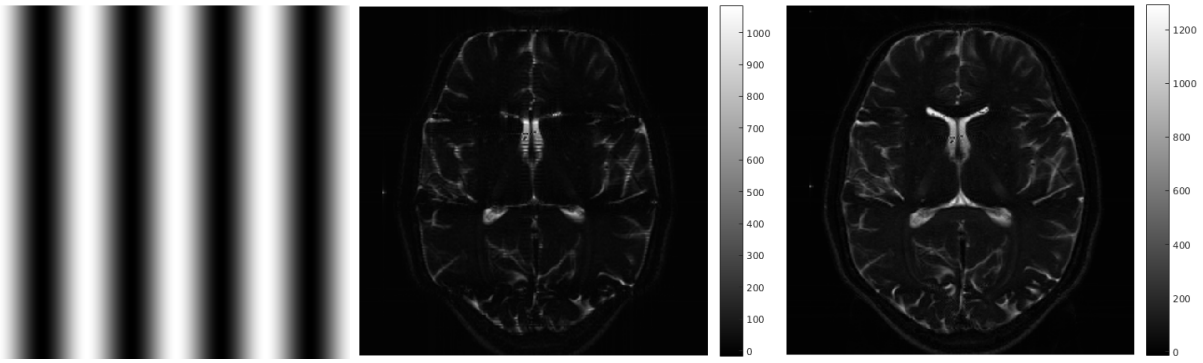


Figura 5.5: De izquierda a derecha, mapa de la inhomogeneidad del campo simulada (función seno), reconstrucción simulada de la imágenes con una única excitación, y reconstrucción simulada de la imagen con dos excitaciones.

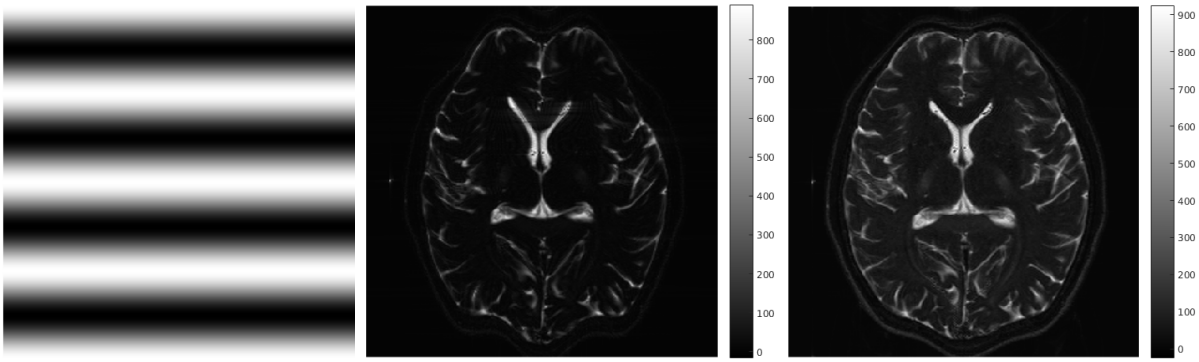


Figura 5.6: De izquierda a derecha, mapa de la inhomogeneidad del campo simulada (función seno transpuesto), reconstrucción simulada de la imágenes con una única excitación, y reconstrucción simulada de la imagen con dos excitaciones.

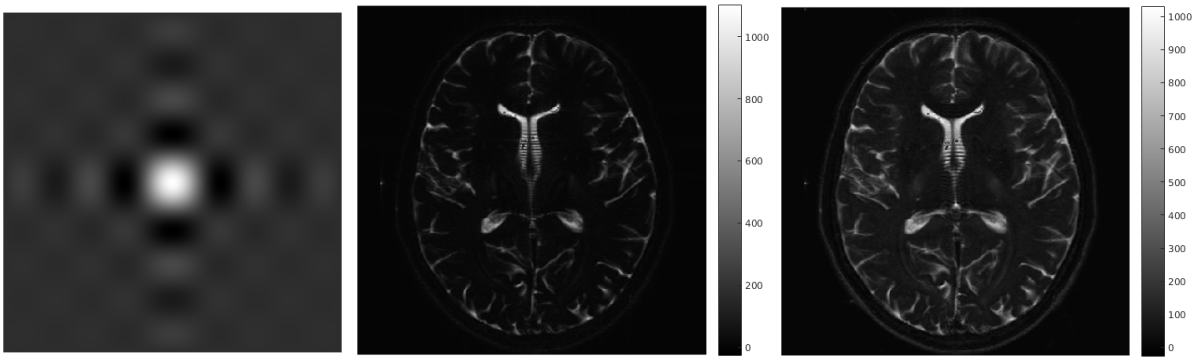


Figura 5.7: De izquierda a derecha, mapa de la inhomogeneidad del campo simulada (función sinc), reconstrucción simulada de la imágenes con una única excitación, y reconstrucción simulada de la imagen con dos excitaciones.

5.3 Medidas de rendimiento

Las medidas del rendimiento del algoritmo se realizan en términos de tiempos de ejecución, los cuales son estimados a partir de 1000 mediciones del mismo y teniendo en cuenta únicamente el núcleo de la simulación. Tras aplicar a los datos el test de Kolmogorov-Smirnov y comprobar que no se cumplía la hipótesis de normalidad con un nivel de significación del 5 %, se optó por presentar los resultados de forma no paramétrica mediante la mediana y el rango intercuartil (IQR).

En la Tabla 5.1 se muestran los tiempos de ejecución del algoritmo de síntesis de imágenes de resonancia magnética mediante una implementación secuencial con MATLAB [19], una implementación con matrices *Sparse* que permiten llevar a cabo operaciones de rotación sobre vectores apilados [19], y la implementación paralela con programación en GPU que en este trabajo se plantea. Dado que en la literatura existente no se ofrecen datos sobre el IQR, este no se ha contemplado en la realización de la Tabla 5.1.

	Tiempos de ejecución (segundos)					
Tamaño	**64 x 64**		**128 x 128**		**256 x 256**	
	Mediana	IQR	Mediana	IQR	Mediana	IQR
Secuencial	73.65	-	1040.30	-	16762.31	-
Sparse	23.80	-	413.76	-	7335.70	-
Paralelo	0.1454	0.0037	0.2007	0.0032	1.1815	0.0044

Tabla 5.1: Tiempos de ejecución, en segundos, para las distintas implementaciones y tamaños de las imágenes.

Por otro lado, la ganancia en cuanto a tiempos de ejecución que ha supuesto la implementación paralela con respecto a la implementación secuencial está representado en la Figura 5.8.

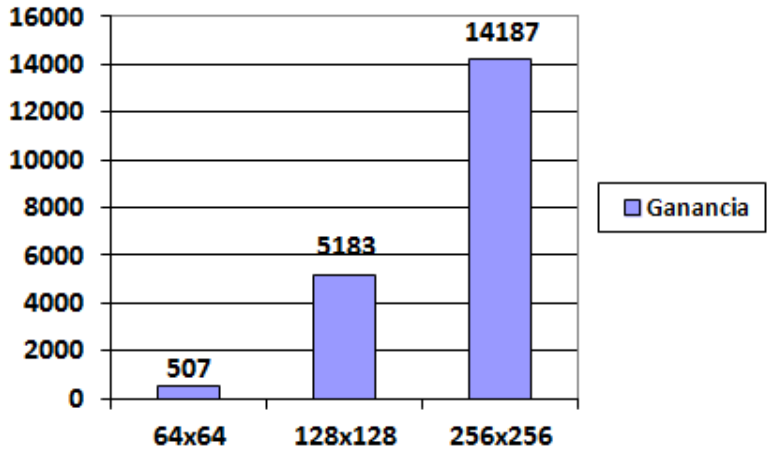


Figura 5.8: Ganancia en los tiempos de ejecución (secuencial/paralelo) para los diferentes tamaños de imágenes.

Como se puede observar, la introducción de programación paralela en GPU mediante OpenCL supone una reducción drástica en los tiempos de computación. Además, esta disminución se hace mucho más notable a medida que el tamaño de las imágenes aumenta.

Capítulo 6

Conclusiones

6.1 Conclusiones

En este Trabajo Fin de Grado, se plantea la implementación de un simulador de síntesis de imágenes de resonancia magnética mediante técnicas de computación paralela en GPU, concretamente haciendo uso de la API y el lenguaje de programación de OpenCL. Para ello, se tomó como punto de partida un código programado previamente en MATLAB en el cual estaba implementado la secuencia de pulsos SE-EPI. El objetivo principal que se perseguía consistía en una reducción de los elevados tiempos de ejecución que suponía la implementación secuencial del mismo. No obstante, en la implementación que aquí se plantea se añaden ciertas mejoras, tales como la posibilidad de simular secuencias *sigle-shot SE-EPI* o *multi-shot SE-EPI*, la implementación de dos formas distintas de escritura del espacio K (secuencial y con interlineado) o la creación de una GUI permitiendo obtener una aplicación cerrada.

No obstante, también es cierto que la implementación adolece de ciertas limitaciones, como por ejemplo que no se han considerado los desfases que pudiera introducir la inhomogeneidad del campo magnético en algunos intervalos de tiempo (desde el final del pulso de saturación al inicio de los gradientes en las direcciones de codificación de frecuencia X y de fase Y, o desde el final de estos al comienzo del pulso de inversión), o que no se ha simulado el efecto de *Chemical Shift*. Por otro lado, es importante señalar que la relajación de la señal entre las lineas del espacio K que se adquieren en una única excitación de RF se ha realizado a razón de T_2 en lugar de T_2^*.

En todo caso, se ha demostrado que la implementación de esta secuencia de pulsos para la simulación de la imagen de resonancia magnética mediante programación en GPU con OpenCL permite una reducción drástica de los tiempos de ejecución con respecto a la implementación secuencial de la misma mediante MATLAB, lo cual permite abrir nuevas posibilidades para este tipo de simulaciones.

6.2 Líneas futuras

Siguiendo el trabajo presentado en esta memoria, podría ser posible añadir ciertas mejoras a la implementación que permitieran realizar simulaciones aún más realistas.

De acuerdo con las limitaciones mencionadas en la sección anterior de este capítulo, una posible línea futura de investigación prodría ser la validación de la hipótesis de que los desfases que pudiera introducir la inhomogeneidad del campo magnético en los intervalos en los que no se ha tenido en cuenta este hecho no es relevante. Y en el caso de que dicha hipótesis no se cumpliera modificar la implementación realizada para considerarlos. También podría ser interesante la introducción en la simulación de otros fenómenos que se producen en la práctica, como es el caso del *Chemical Shift*.

Así mismo, dado que en la implementación realizada se considera que los pulsos y gradientes son ideales con una forma totalmente rectangular, la introducción de gradientes con formas más realistas en las que se incluyera fenómenos de *slew rate*, daría lugar a la obtención de imágenes que se acercaran más a las que se producen en la práctica.

Otro posible trabajo futuro, podría estar relacionado con la simulación de otras secuencias de pulsos de RF o otras posibles trayectorias de adquisición del espacio K.

Bibliografía

[1] Z. P Liang and P. C. Lauterbur. *Principles of Magnetic Resonance Imaging*. IEEE Press, 2000.

[2] C. G. Xanthis, I. E. Venetis, A. V. Chalkias, and A. H. Aletras. MRISIMUL: A GPU-Based Parallel Approach to MRI Simulations. *IEEE Transactions on Medical Imaging*, 33(3):607–617, 2014.

[3] J. Bittoun, J. S. Taquin, and M. Sauzade. A computer algorithm for the simulation of any nuclear magnetic resonance (NMR) imaging method. *Magnetic Resonance Imaging*, 2:113–120, 1984.

[4] D. Rundle, S. S. Kishore, S. Seshadri, and F. Wehrli. Magnetic resonance imaging simulator: A teaching tool for radiology. *Journal of Digital Imaging*, 3(4):226–229, 1990.

[5] The MathWorks - MATLAB and Simulink for Technical Computing. Website. Último acceso jun. 2017. `http://www.mathworks.com/`.

[6] The Eclipse Foundation. Website. Último acceso jun. 2017. `https://eclipse.org/`.

[7] LaTeX3 Proyect. Website. Último acceso jun. 2017. `https://www.latex-project.org/latex3/`.

[8] R. A. Poley. Fundamentals physics of MR Imaging. *RadioGraphics*, 25(4):1087–1099, 2005.

[9] P. Suetens. *Fundamentals of Medical Imaging*. Cambridge, 2006.

[10] L. Cordero. *Estimación del tensor de esfuerzo del miocardio. Integración de propiedades físicas del problema e imagen de resonancia magnética multimodal en un modelo estocástico*. PhD thesis, Universidad de Valladolid, 2011.

[11] L. Oleaga and J. Lafuente. *Aprendiendo los fundamentos de la resonancia magnética*. Panamericana, 2009.

[12] M. A. Bernstein, K. F. King, and Zhou X. J. *Handbook of MRI Pulse Sequence*. Elsevier Academic Press, 2004.

[13] B. L. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL*. Elsevier, 2013.

[14] Khronos Group. *The OpenCL Specification. Versión: 1.2. Revisión: 19*, 2012. `https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf`.

[15] What is GPU-Accelerated computing?. Website. Último acceso jun. 2017. `http://www.nvidia.com/object/what-is-gpu-computing.html`.

[16] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Preceedings of the IEEE*, 96(5):879–899, 2008.

[17] F. Simmross. *Introducción a la programación heterogénea CPU-GPU en OpenCL.*

[18] P. Jezzard, A. S. Barnett, and C. Pierpaoli. Characterization of and Correction for Eddy Current Artifacts in Echo Planar Diffusion Imaging. *Magnetic Resonance in Medicine*, 39:801–812, 1998.

[19] D. Treceño, J. Calabia, R. de Luis, and C. Alberola. Una implementación eficiente no paralela de secuencias de resonancia magnética mediante matrices *Sparse*. *CASEIB. Valencia, España*, 2016.

Apéndice A

Operaciones de rotación

Las matrices de rotación en torno a cada uno de los ejes en el sistema móvil de referencia son los siguientes:

$$\mathbf{R}_{x'}(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha) & sin(\alpha) \\ 0 & -sin(\alpha) & cos(\alpha) \end{bmatrix} \tag{A.1}$$

$$\mathbf{R}_{y'}(\alpha) = \begin{bmatrix} cos(\alpha) & 0 & -sin(\alpha) \\ 0 & 1 & 0 \\ sin(\alpha) & 0 & cos(\alpha) \end{bmatrix} \tag{A.2}$$

$$\mathbf{R}_{x'}(\alpha) = \begin{bmatrix} cos(\alpha) & sin(\alpha) & 0 \\ -sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{A.3}$$