UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR DE

INGENIEROS DE TELECOMUNICACIÓN

TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Desarrollo en GPU de algoritmos paralelos de procesado de imagen aplicados a la simulación del fenómeno de resonancia magnética

Autora:

Dª. Irene Fernández Arias

Tutores:

Dr. D. Carlos Alberola López
Dr. D. Federico Simmross Wattenberg

Valladolid, 19 de septiembre de 2022

TÍTULO:	**Desarrollo en GPU de algoritmos paralelos de procesado de imagen aplicados a la simulación del fenómeno de resonancia magnética**
AUTORA:	**Dª. Irene Fernández Arias**
TUTORES:	**Dr. D. Carlos Alberola López** **Dr. D. Federico Simmross Wattenberg**
DEPARTAMENTO:	**Departamento de Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

Tribunal

PRESIDENTE:	**Dr. D. Carlos Alberola López**
SECRETARIO:	**Dr. D. Federico Simmross Wattenberg**
VOCAL:	**Dr. D. Manuel Rodríguez Cayetano**
SUPLENTE 1:	**Dr. D. Marcos Martín Fernández**
SUPLENTE 2:	**Dr. D. Rodrigo de Luis García**

FECHA:

CALIFICACIÓN:

Agradecimientos

En primer lugar, quiero agradecer a mis tutores Carlos Alberola López y Federico Simmross Wattenberg por su confianza para realizar este Trabajo de Fin de Grado y por su ayuda durante estos meses, sin la cual no habría sido posible cumplir los objetivos planteados.

En segundo lugar, a David por hacer más fácil el camino recorriéndolo juntos.

En último lugar, a mis padres por su apoyo incondicional en cada uno de mis pasos.

Resumen

La simulación del fenómeno de resonancia magnética (RM) resulta de especial interés tanto desde el punto de vista de la investigación científica como desde el de la formación de personal técnico. El diseño de simuladores de RM requiere considerar de forma simultánea la velocidad y el realismo con el fin de proporcionan interactividad, sin renunciar a su uso como herramientas de investigación.

Este Trabajo de Fin de Grado presenta la implementación de un simulador de resonancia magnética sobre el *framework* OpenCLIPER. Se parte de los fundamentos de la simulación del fenómeno empleados por el simulador MRIsim.jl, el cual presenta resultados muy competitivos frente a otros simuladores existentes.

Sin embargo, el diseño del simulador MRIsim.jl no se efectúo con el principal objetivo de alcanzar el máximo rendimiento posible. Cabe mencionar adicionalmente que, aunque el simulador funciona tanto en dispositivos CPU como GPU, estas últimas se encuentran limitadas a un único fabricante. El empleo de OpenCLIPER pretende solventar los dos inconvenientes anteriores, de forma que se posibilite el uso del simulador en cualquier dispositivo de cualquier fabricante, al mismo tiempo que se supere al simulador de partida en términos de rendimiento.

Finalmente, el funcionamiento del simulador se valida de forma visual a partir de las imágenes reconstruidas por ambos simuladores ante un mismo proceso de simulación. La consecución del objetivo planteado se evalúa a través de la comparación de tiempos de ejecución entre ambos simuladores, la cual pone de manifiesto una reducción significativa de los tiempos en la nueva implementación desarrollada.

Palabras clave

MRI, simulación, Julia, OpenCLIPER, GPU

Abstract

Magnetic resonance (MR) simulation is an area of particular interest both for scientific research and for the training of technical personnel. The design of MR simulators requires taking into account speed and realism at the same time in order to offer interactivity without giving up on the use of the simulator as a research tool.

This Thesis presents the implementation of a MR simulator using the framework OpenCLIPER. The previously existing simulator MRIsim.jl is taken as the starting point in terms of the physical basis of the MR phenomenom due to its competitive results in comparison with other existing simulators.

Nevertheless, the design of MIRsim.jl was not thought with maximum performance as the main goal. In addition, although this simulator works both in CPU and GPU devices, GPUs are limited to one certain vendor. OpenCLIPER arises as an answer for both previous disadvantages in order to make the use of the simulator in any device from any vendor possible and achieve higher performance at the same time.

Finally, results are validated visually by means of the images produced by both simulators after an identic simulation. The achievement of the stated goal is assesed by the comparison of the executing times in each simulator, which shows a significant reduction in the new implementation presented.

Keywords

MRI, simulation, Julia, OpenCLIPER, GPU

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

La imagen de resonancia magnética (MRI, *Magnetic Resonance Imaging*) es una de las modalidades de imagen médica más potentes entre las existentes hoy en día. Esta técnica de diagnóstico permite obtener imágenes de cualquier tejido del cuerpo humano y extraer información sobre sus características internas, tanto físicas como químicas, a través del fenómeno NMR (*Nuclear Magnetic Resonance*) [1].

Comparada con otras técnicas de obtención de imágenes médicas, la MRI requiere de unos elevados conocimientos acerca de la física subyacente para poder adquirir e interpretar las imágenes [2]. En ese sentido, el manejo de un escáner de MRI no resulta una tarea fácil, puesto que existe un gran número de parámetros que se deben tomar en consideración por su fuerte influencia sobre las adquisiciones de imágenes, y es por ello que deben ser calibrados de forma adecuada para obtener imágenes con calidad óptima y evitar la aparición de artefactos en las mismas. Los parámetros a calibrar dependen de las secuencias de pulsos magnéticos que se programan en el escáner, de las cuales existe una amplia variedad y suponen, en la actualidad, un campo abierto de investigación que precisa de un uso intensivo de escáneres reales. Sin embargo, las demandas del ámbito clínico y los altos costes requieren de un uso constante de los escáneres, lo cual limita las posibilidades de programar sesiones de aprendizaje para los técnicos en formación o para la investigación de nuevas secuencias de pulsos. En consecuencia, el desarrollo de simuladores computacionales se presenta como una alternativa necesaria.

1.1. Motivación

La simulación ha estado presente en el campo de la MRI desde sus primeros días. A pesar de que la mayor parte de los experimentos relativos a la MRI pueden ser estudiados de forma analítica a través de las ecuaciones de Bloch, la extrema complejidad de esta técnica convierte la resolución analítica en una vía ineficiente. La simulación, en cambio, permite encontrar una solución general de dichas ecuaciones, así como aproximar de forma sencilla los valores de las señales resultantes del proceso. Por tanto, resulta útil desde un punto de vista de adquisición, para la comprensión de distintos fenómenos que pueden tener lugar en las imágenes y poder así

ayudar a mejorar su calidad. Desde un punto de vista médico, resulta también útil para determinar la adecuación de las distintas técnicas de obtención de las imágenes en función de los parámetros que se deseen medir [3].

Entre los múltiples simuladores existentes, resulta de especial relevancia por presentar unos resultados muy competitivos el simulador MRIsim.jl [4], desarrollado en la Pontificia Universidad Católica de Chile. Este simulador está escrito en el lenguaje de programación Julia [5] y dispone de un motor de simulación realista. Actualmente, es capaz de realizar operaciones paralelas, si bien solamente soporta dispositivos GPU de un único fabricante.

El grupo de investigación Laboratorio de Procesado de Imagen (LPI), en el cual se desarrolla este Trabajo de Fin de Grado, ha contribuido a este campo a través de un simulador educativo basado en web con un sistema inteligente de tutela [6] [7]. Este simulador imita el trabajo típico que un técnico lleva a cabo en la práctica e incluye distintos procedimientos para eliminar artefactos comunes. Sin embargo, por tratarse de un simulador destinado al entrenamiento de personal clínico y de técnicos de adquisición, el énfasis se puso en la interactividad y, en ese sentido, el motor de simulación se diseñó para ser rápido y simple, lo cual impide desarrollar con él nuevas secuencias de pulsos. Es por esto que su utilidad como herramienta de investigación queda limitada.

Por otro lado, se ha desarrollado recientemente, también en el grupo de investigación LPI, el *framework* OpenCLIPER [8]. Este *framework* está diseñado como un conjunto de clases de C++ centradas en proporcionar al programador diversas funcionalidades, como la gestión del dispositivo de computación, la carga y guardado de datos y la gestión de los algoritmos, y tiene como objetivo facilitar el desarrollo de aplicaciones en el lenguaje OpenCL [9]. El empleo de OpenCL hace que el código desarrollado cumpla el paradigma WORA (*write once, run anywhere*), de forma que se posibilita que el mismo código pueda ser empleado en cualquier dispositivo (CPU, GPU, FPGA, etc.).

Todo lo anterior da lugar, de forma natural, a plantear la posibilidad de escribir el núcleo del simulador MRIsim.jl [4] en OpenCL, haciendo uso de OpenCLIPER, de forma que las ventajas anteriormente comentadas sobre dicho simulador puedan verse reforzadas por la potencia de OpenCLIPER. Esto posibilitaría que los simuladores basados en principios físicos puedan ser empleados tanto con finalidades docentes como con finalidades investigadoras, y sobre cualquier dispositivo, con independencia del fabricante de que se trate.

1.2. Objetivos

El objetivo perseguido en la elaboración de este Trabajo de Fin de Grado consiste en la implementación del proceso de simulación del fenómeno MRI en OpenCL, empleando para ello el *framework* OpenCLIPER, y tomando como punto de partida en cuanto a la simulación fidedigna del fenómeno MRI el simulador MRIsim.jl [4].

Por tanto, se pretende, aparte naturalmente de que la simulación resultante produzca los mismos resultados que el simulador MRIsim.jl, que el proceso de simulación se efectúe con

mayor rapidez que en dicho simulador.

1.3. Fases y Métodos

La elaboración de este Trabajo de Fin de Grado requiere de diversas fases para la consecución del objetivo planteado. Estas fases se describen a continuación:

A) Etapa de formación:

i. Estudio de la programación paralela haciendo uso del lenguaje OpenCL [9].

ii. Estudio de los conceptos básicos relativos al fenómeno MRI.

iii. Análisis y comprensión del funcionamiento del simulador MRIsim.jl [4].

B) Etapa de diseño e implementación:

i. Diseño de la estructura de clases por las que va a estar constituido el simulador e implementación de las mismas.

ii. Implementación de una serie de secuencias básicas, esenciales para la simulación del fenómeno MRI.

iii. Implementación del proceso de simulación tomando como base el simulador MRIsim.jl y aprovechando las facilidades proporcionadas por el *framework* OpenCLIPER [8].

C) Etapa de validación y evaluación:

i. Validación visual y numérica de la imagen reconstruida por el simulador desarrollado.

ii. Evaluación del rendimiento del simulador en comparación con el simulador MRIsim.jl, en términos de tiempos de ejecución.

1.4. Medios

La realización de este Trabajo de Fin de Grado precisa de la utilización de distintas herramientas *software* y *hardware*.

Las herramientas *software* empleadas son:

- KDevelop [10], un moderno entorno de desarrollo integrado (IDE, *Integrated Development Environment*) para distintos lenguajes de programación, de entre los cuales se hará uso de C++ para el desarrollo del simulador en OpenCLIPER [8].
- Atom [11], un editor de código fuente de código abierto, que se empleará para la ejecución del simulador MRIsim.jl [4], implementado lenguaje de programación Julia [5]. Por tanto, se requerirá también del manejo de este lenguaje.

- Overleaf [12], un editor de textos colaborativo que hace uso del sistema de composición de textos LaTeX [13], que se empleará para la redacción del presente Trabajo de Fin de Grado.
- MATLAB R2022a [14], una plataforma de programación y cálculo numérico empleada para el análisis de datos, el desarrollo de algoritmos y la creación de modelos. Se empleará como herramienta de comprobación durante el proceso de desarrollo del simulador.

Las herramientas *hardware* empleadas son:

- Ordenador portátil con las siguientes características:
  - Intel® Core™ i7-8550U CPU @ 1.80 GHz × 8.
  - 7.5 GB de memoria RAM.
  - Disco duro de 256 GB.
  - GPU Nvidia GeForce MX150 con 2 GB de memoria RAM.
- Otros dispositivos CPU y GPU para la realización de pruebas de rendimiento:
  - GPU Nvidia Quadro RTX 6000.
  - Intel® Core™ i7-4790 CPU @ 3.60 GHz.
  - GPU AMD Radeon RX 5700 XT.

1.5. Estructura del documento

El resto del documento se estructura como se detalla a continuación.

En el *Capítulo 2* se plantea una revisión del estado del arte relativo a los fundamentos del fenómeno MRI, al *framework* OpenCLIPER y a los fundamentos del lenguaje de programación Julia y de la simulación de MRI.

En el *Capítulo 3* se presenta el desarrollo del simulador en OpenCLIPER, a partir de la descripción de las estructuras de datos en él empleadas, las secuencias de la simulación y el propio proceso de simulación.

En el *Capítulo 4* se valida y evalúa el simulador desarrollado a través de un análisis comparativo de las prestaciones de ambos simuladores en un proceso de simulación común.

Finalmente, en el *Capítulo 5* se comentan las principales conclusiones extraídas tras la elaboración del Trabajo de Fin de Grado y se plantean posibles líneas de trabajo futuro.

Capítulo 2

Estado del arte

2.1. Fundamentos de la MRI

La imagen por resonancia magnética (MRI, *Magnetic Resonance Imaging*) es una técnica tomográfica que permite la obtención de imágenes de cualquier tejido del cuerpo humano, de forma que proporciona información sobre sus características internas, tanto físicas como químicas [1]. La obtención de estas imágenes se basa en el fenómeno NMR (*Nuclear Magnetic Resonance*). En esta sección se lleva a cabo una descripción de la física involucrada en el proceso de obtención de este tipo de imágenes.

2.1.1. Fenómeno NMR

El fenómeno NMR constituye la base de la obtención de imágenes de resonancia magnética. Los procesos que tienen lugar a nivel nuclear son el origen de las señales generadas y detectadas para la posterior formación de la imagen [1]. Se comienza por una primera descripción del nivel nuclear para posteriormente analizar los fenómenos que tienen lugar a nivel macroscópico.

Los núcleos de los átomos con pesos atómicos y/o números atómicos impares poseen una propiedad que posibilita que tenga lugar el fenómeno NMR, que consiste en que disponen de un momento angular $\vec{J}$, conocido como *spin*, que puede visualizarse como una rotación del núcleo alrededor de un determinado eje. Este es el caso de los átomos de hidrógeno, cuyo núcleo está formado por un único protón, y que resulta de especial interés en el ámbito de la resonancia magnética debido a su abundancia en el cuerpo humano. El núcleo de un átomo de hidrógeno actúa, por tanto, como una carga positiva en continua rotación [15].

En línea con lo anterior, es sabido que toda carga en movimiento da lugar a un campo magnético a su alrededor, conocido como momento magnético, y que viene representado por $\vec{\mu}$. Existe una relación directamente proporcional entre el momento magnético y el momento angular

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

donde γ es la constante giromagnética, que es dependiente del núcleo en consideración y que toma para el caso del hidrógeno el valor $2{,}675{\cdot}10^8 rad/s/T$ [1].

Puesto que el momento magnético es una magnitud vectorial, su caracterización completa requiere especificar tanto su magnitud como su dirección. En primer lugar, su magnitud viene dada por

$$|\vec{\mu}| = \gamma\hbar\sqrt{I(I+1)} \tag{2.2}$$

donde $\hbar$ es la constante de Planck dividida por 2π e I es el número cuántico del *spin* nuclear. En concreto, para los átomos de hidrógeno el valor de I es $\frac{1}{2}$. A diferencia de la magnitud, que queda determinada por la expresión anterior, la dirección es aleatoria en ausencia de un campo magnético externo. Esta aleatoriedad supone que los campos magnéticos individuales de los *spins* se cancelen, dando lugar a un campo magnético resultante nulo [16].

Por el contrario, si los *spins* son sometidos a un campo magnético externo, estos tienden a alinearse en la dirección de dicho campo. Por convención, se considera la aplicación de un campo magnético estático denominado $\vec{B}_0$ en la dirección del eje z. La alineación de los *spins* no tiene lugar en la dirección exacta del campo externo, sino que estos experimentan un movimiento de precesión en dicha dirección, mostrado en la *Figura 2.1*. La frecuencia de precesión es proporcional al campo magnético externo y viene dada por la ecuación de Larmor [15]

$$\omega_0 = \gamma B_0 \tag{2.3}$$

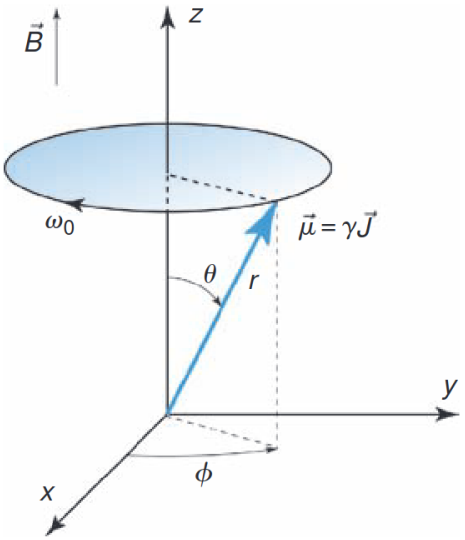


Figura 2.1: Precesión *spin* en presencia de un campo magnético estático [17]

La orientación del movimiento de precesión puede ser paralela o anti-paralela respecto al sentido del vector de campo magnético externo $\vec{B}_0$, tal como se refleja en la *Figura 2.2*.

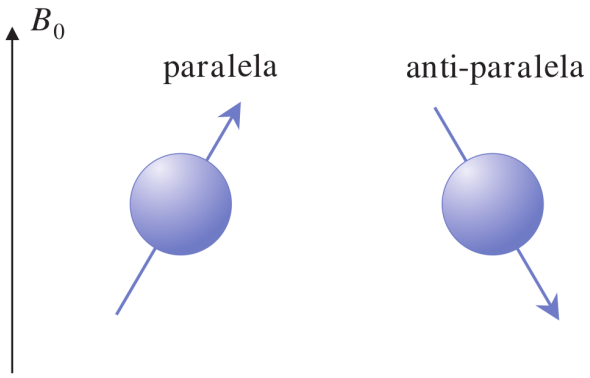


Figura 2.2: Posibles orientaciones *spin* [15].

Analizado el nivel nuclear, se considera a continuación el nivel macroscópico para describir el comportamiento de un sistema de *spins*. Se introduce un nuevo vector de magnetización $\vec{M}$ [1], que viene dado por la suma de todos los momentos magnéticos del objeto

$$\vec{M} = \sum_{n=1}^{N_s} \vec{\mu}_n \tag{2.4}$$

donde $\vec{\mu}_n$ representa el momento magnético del n-ésimo *spin* y N_s es el número total de *spins* en el objeto en consideración. El vector de magnetización neta se puede descomponer como

$$\vec{M} = M_x\vec{i} + M_y\vec{j} + M_z\vec{k} = (\sum_{n=1}^{N_s} \vec{\mu}_{x,n})\vec{i} + (\sum_{n=1}^{N_s} \vec{\mu}_{y,n})\vec{j} + (\sum_{n=1}^{N_s} \vec{\mu}_{z,n})\vec{k} \tag{2.5}$$

donde los dos primeros términos son nulos, puesto que los *spins* tienen una fase aleatoria en su precesión alrededor del eje z resultando en la cancelación de sus componentes transversales en la suma vectorial.

La distribución estadística que siguen los *spins* respecto a la orientación de su precesión determina que hay un mayor número de protones en el estado paralelo al campo magnético externo, que es el que requiere una menor cantidad de energía. Consecuentemente, el vector de magnetización neta apunta exactamente en la dirección del eje z positivo [1].

La generación de una señal detectable requiere introducir un segundo campo magnético denominado $\vec{B}_1(t)$, en este caso oscilante, y que es perpendicular al campo estático anterior $\vec{B}_0$. Los *spins*, antes de verse sometidos a este nuevo campo magnético, se encuentran rotando con fases aleatorias. Mediante la aplicación de este segundo campo magnético se pretende conseguir que todos los *spins* se encuentren en fase, es decir, que haya resonancia. Para ello, la condición necesaria que debe cumplir el campo aplicado es que su frecuencia de oscilación sea igual a la frecuencia de Larmor ω_0 antes definida en la *Ecuación 2.3*.

De forma general, se denomina pulso de radiofrecuencia (RF) a este campo magnético debido a que su duración es escasa, de unos pocos microsegundos o milisegundos, y a que oscila en el rango de la radiofrecuencia. Se asumirá, en adelante, que el pulso de RF adopta la siguiente

forma

$$B_1(t) = B_1^e(t)e^{-i(\omega_{rf}t+\phi)} \tag{2.6}$$

donde $B_1^e(t)$ representa la amplitud de la envolvente, ω_{rf} la frecuencia de la portadora y ϕ la fase, que se considera 0 por simplicidad.

El comportamiento del vector de magnetización $\vec{M}$ ante la aplicación del campo magnético $\vec{B}_1$ se describe a través de la ecuación de Bloch

$$\frac{d\vec{M}}{dt} = \gamma\vec{M} \times \vec{B} - \frac{M_x\vec{i} + M_y\vec{j}}{T_2} - \frac{(M_z - M_z^0)\vec{k}}{T_1} \tag{2.7}$$

donde $\vec{B}$ representa el campo magnético total ($\vec{B}_0 + \vec{B}_1(t)$), M_z^0 es el valor de equilibrio térmico de $\vec{M}$ en presencia única del campo $\vec{B}_0$, y T_1 y T_2 son constantes de tiempo que caracterizan el proceso de relajación de un sistema de *spins* tras haber sido perturbado, las cuales se comentarán más adelante. Habitualmente, la duración del pulso de RF es menor que T_1 y T_2, por lo que la *Ecuación 2.7* puede reducirse al primer término

$$\frac{d\vec{M}}{dt} = \gamma\vec{M} \times \vec{B} \tag{2.8}$$

Con el objetivo de facilitar el análisis del efecto del pulso de RF sobre el vector de magnetización, resulta conveniente definir un marco de referencia rotatorio. Este consiste en un sistema de coordenadas cuyo plano transversal está rotado en el sentido de las agujas del reloj a una frecuencia angular, concretamente la frecuencia de Larmor ω_0, como se muestra en la *Figura 2.3*. Para distinguirlo del marco de referencia del laboratorio se emplea la notación x', y' y z' para los ejes y la notación $\vec{i}'$, $\vec{j}'$ y $\vec{k}'$ para los vectores unitarios [1].

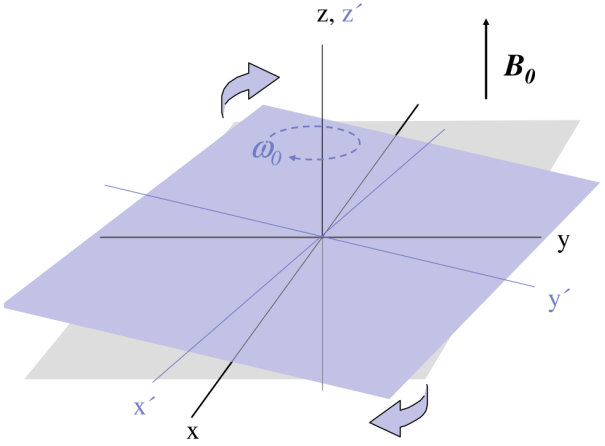


Figura 2.3: Marco de referencia rotatorio [15].

El movimiento del vector de magnetización en el nuevo marco de referencia se puede describir como

$$\frac{\partial\vec{M}_{rot}}{\partial t} = \frac{d\vec{M}}{dt} - \bar{\omega} \times \bar{M}_{rot} \tag{2.9}$$

donde $\vec{M} = \vec{M}_{rot}$, $\frac{d\vec{M}}{dt}$ representa la tasa de cambio de $\vec{M}$ en el marco de referencia inicial y $\frac{\partial \vec{M}_{rot}}{\partial t}$ la tasa de cambio de $\vec{M}$ en el nuevo marco de referencia.

La ecuación de Bloch introducida en la *Ecuación 2.8* se puede expresar en este nuevo marco rotatorio de la siguiente forma

$$\frac{\partial \vec{M}_{rot}}{\partial t} = \gamma \vec{M}_{rot} \times \vec{B}_{rot} - \vec{\omega} \times \vec{M}_{rot} = \gamma \vec{M}_{rot} \times (\vec{B}_{rot} + \frac{\vec{\omega}}{\gamma}) = \gamma \vec{M}_{rot} \times \vec{B}_{eff} \tag{2.10}$$

Considerando de nuevo la ecuación de Bloch general sin supresión de los dos últimos términos a la que se hizo referencia en la *Ecuación 2.7*, esta se puede reescribir en el marco rotatorio como

$$\frac{\partial \vec{M}_{rot}}{\partial t} = \gamma \vec{M}_{rot} \times \vec{B}_{eff} - \frac{M_{x'}\vec{i'} + M_{y'}\vec{j'}}{T_2} - \frac{(M_{z'} - M_z^0)\vec{k'}}{T_1} \tag{2.11}$$

En las dos ecuaciones anteriores se introduce el término $\vec{B}_{eff}$, que representa el campo magnético efectivo que el vector de magnetización experimenta en el marco rotatorio y que se puede expresar como

$$\vec{B}_{eff} = B_0\vec{k'} + \vec{B}_1^e(t)\vec{i'} - \frac{\vec{\omega}_{rf}}{\gamma} = (B_0 - \frac{\vec{\omega}_{rf}}{\gamma})\vec{k'} + \vec{B}_1^e(t)\vec{i'} \tag{2.12}$$

Si la excitación cumple la condición de resonancia $\omega_{rf} = \omega_0 = \gamma B_0$, se produce el fenómeno conocido como *on-resonance* que supone, sustituyendo en la *Ecuación 2.12*, que $\vec{B}_{eff} = \vec{B}_1^e(t)\vec{i'}$. Determinado el campo magnético efectivo al que se ve sometido el vector de magnetización, obtenemos, sustituyendo en la *Ecuación 2.10*

$$\frac{\partial \vec{M}_{rot}}{\partial t} = \gamma \vec{M}_{rot} \times \vec{B}_1^e(t)\vec{i'} \tag{2.13}$$

Esto indica que, en el marco rotatorio de referencia, el efecto de la excitación en condición de resonancia es la precesión del vector de magnetización neta alrededor del eje x'.

A pesar de que se asume que la mayor parte de las excitaciones cumplen la condición de resonancia, en la práctica las inhomogeneidades del campo magnético no son despreciables y el campo magnético efectivo toma la siguiente expresión

$$\vec{B}_{eff} = (B_0 - \frac{\vec{\omega}_{rf}}{\gamma})\vec{k'} + \vec{B}_1^e(t)\vec{i'} = \frac{\Delta\omega_0}{\gamma}\vec{k'} + \vec{B}_1^e(t)\vec{i'} \tag{2.14}$$

donde $\Delta\omega_0 = \omega_0 - \omega_{rf}$ mide el grado de *off-resonance*. Esto tiene como consecuencia que la precesión del vector de magnetización neta ya no sea exactamente alrededor del eje x'.

La supresión del campo externo $\vec{B}_1$ supone que el sistema de *spins* retorne a su estado inicial tras la perturbación causada por el pulso de RF. La desaparición del pulso de RF da lugar a

tres procesos: La precesión de $\vec{M}$ alrededor del campo $\vec{B}_0$, conocida como precesión libre; la recuperación de la componente longitudinal M_z, conocida como relajación longitudinal, y la desaparición de la componente transversal M_{xy}, conocida como relajación transversal.

En relación con los dos últimos procesos, se introducen las constantes de tiempo $T1$ y $T2$, que son parámetros específicos de cada tejido [16]. $T1$ se define como el tiempo necesario para que la componente longitudinal alcance el 63 % de su valor final, mientras que $T2$ se define como el tiempo necesario para que la componente transversal se reduzca al 37 % de su valor inicial. En la *Figura 2.4* se representa gráficamente el significado de ambos parámetros y es posible comprobar en ella la mayor duración de $T1$ frente a $T2$.

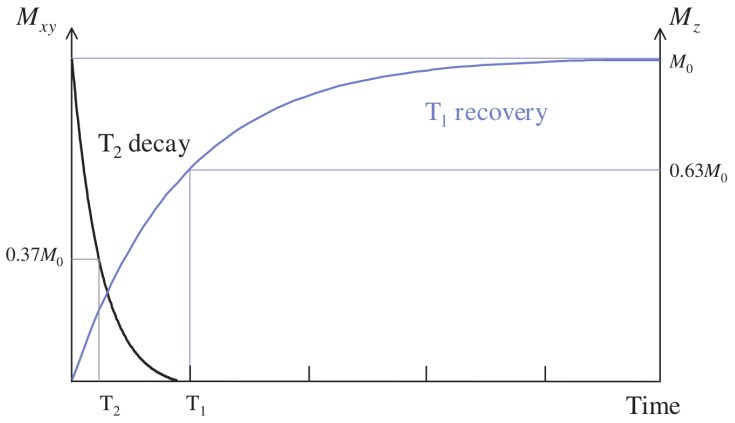


Figura 2.4: Constantes de tiempo $T1$ y $T2$ [15].

La evolución temporal de las componentes transversal y longitudinal se puede obtener a partir de las siguientes expresiones, extraídas de la *Ecuación 2.11* con $\vec{B}_{eff} = 0$

$$\begin{cases} \frac{dM_{z'}}{dt} = -\frac{M_{z'} - M_z^0}{T_1} \\ \frac{dM_{x'y'}}{dt} = -\frac{M_{x'y'}}{T_2} \end{cases} \tag{2.15}$$

A partir de la resolución de la ecuación anterior, se obtiene finalmente la evolución con el tiempo de ambas componentes

$$\begin{cases} M_{x'y'}(t) = M_{x'y'}(0_+)e^{-t/T_2} \\ M_{z'}(t) = M_z^0(1 - e^{-t/T_1}) + M_{z'}(0_+)e^{-t/T_1} \end{cases} \tag{2.16}$$

donde $M_{x'y'}(0_+)$ y $M_{z'}(0_+)$ corresponden con las magnetizaciones en el plano transversal y en el eje z inmediatamente después del pulso de RF. Como reflejan ambas expresiones y como se visualiza en la *Figura 2.4*, ambas siguen una función exponencial.

2.1.2. Detección y localización de la señal de MRI

Hasta ahora se ha descrito cómo situando un objeto en un campo magnético externo y estimulándolo mediante un segundo campo magnético oscilatorio, se induce un magnetismo

macroscópico en forma de una magnetización en rotación en el objeto. El siguiente paso consiste en detectar ese magnetismo para convertir la magnetización en rotación en señales eléctricas.

La detección de la señal de MRI se basa en la Ley de Faraday de la inducción electromagnética, que establece que un flujo magnético variante con el tiempo que atraviesa una bobina receptora induce en la misma una fuerza electromagnética o voltaje igual a la tasa a la que el flujo magnético está variando. En la MRI, el vector de magnetización neta está precesionando en el rango de la radiofrecuencia y cualquier bucle conductor resonando a esa frecuencia puede emplearse como bobina receptora [1]. La señal recibida corresponde a la superposición de las señales procedentes de todas aquellas partes del cuerpo que hayan sido excitadas por el pulso de RF, por lo que no contiene ninguna información espacial [17].

Lo anterior determina que obtenida la señal, resulte necesario diferenciar las señales generadas por distintas zonas locales del objeto en consideración. Los pasos necesarios para posibilitar la localización de la señal son la excitación selectiva, la codificación de fase y la codificación en frecuencia, los cuales se describen a continuación [1].

Una de las formas más habituales de excitación selectiva en MRI es la selección de corte o *slice*. Para poder excitar selectivamente los *spins* en un determinado *slice* se requiere la presencia de un gradiente. La justificación se encuentra en que, dado que el pulso RF solo es capaz de seleccionar la frecuencia, *spins* ubicados en diferentes posiciones espaciales son excitados de la misma forma si se encuentran precesionando a la misma frecuencia. Por tanto, para que el pulso de RF sea localmente selectivo, se requiere que la frecuencia de resonancia de los *spins* sea dependiente de la posición y, en concreto, linealmente variante a lo largo de la dirección del *slice* seleccionado.

Esto se consigue superponiendo al campo $\vec{B}_0$ durante el período de excitación un gradiente lineal de campo magnético, al que se denomina gradiente de selección de *slice*. Este gradiente consiste en un campo magnético en la dirección del eje z con una amplitud que varía linealmente en una dirección concreta a la que se denomina ($\vec{\mu}_G$), que coincidirá con la dirección del *slice* a seleccionar. El gradiente toma la siguiente expresión

$$\vec{G_{ss}} = (G_x, G_y, G_z) = G_{ss}\vec{\mu}_G \tag{2.17}$$

La introducción del gradiente hace que se modifique la frecuencia de precesión, de manera que si se considera un gradiente G_z, dicha frecuencia es

$$\omega(z) = \omega_0 + \gamma G_z z \tag{2.18}$$

Seleccionado el corte, la información espacial puede codificarse en la señal empleando la codificación de frecuencia y de fase. Comenzando por la codificación de frecuencia, esta hace que la frecuencia de precesión de los *spins* del *slice* seleccionado dependa de su posición espacial. Para ello, tras la excitación se aplica al campo estático $\vec{B}_0$ un gradiente de codificación de frecuencia adicional. Consideramos la aplicación de un gradiente $G_x x$, de forma que la frecuencia de Larmor es dependiente de la posición en el eje x

$$\omega(x) = \omega_0 + \gamma G_x x \tag{2.19}$$

La señal generada localmente por los *spins* en un intervalo infinitesimal dx en el punto x es

$$dS(x,t) = \rho(x)dxe^{-i\gamma(B_0+G_x x)t} \tag{2.20}$$

donde $\rho(x)$ es la distribución de los *spins* del objeto. Se dice que la señal está codificada en frecuencia puesto que su frecuencia de oscilación, determinada en la *Ecuación 2.19*, está relacionada linealmente con su localización espacial.

La señal total recibida del objeto en presencia de este gradiente es

$$S(t) = \int_{object} dS(x,t) = \int_{-\infty}^{\infty} \rho(x)e^{-i\gamma(B_0+G_x x)t}dx = \left[\int_{-\infty}^{\infty} \rho(x)e^{-i\gamma G_x xt}dx\right] e^{-i\omega_0 t} \tag{2.21}$$

Tras la demodulación de la señal que tiene lugar en la antena receptora, la señal demodulada toma la siguiente forma

$$S(t) = \int_{-\infty}^{\infty} \rho(x)e^{-i\gamma G_x xt}dx \tag{2.22}$$

En cuanto a la codificación de fase, esta consiste en la aplicación de un último gradiente G_y durante un corto intervalo de tiempo T_{pe}, de forma que señales procedentes de distintas posiciones del eje y acumulan diferentes fases tras un intervalo T_{pe}. Con esto se consigue que la fase esté linealmente relacionada con la localización de la señal

$$\phi(y) = -\gamma G_y y T_{pe} \tag{2.23}$$

La señal total recibida será

$$S(t) = \left[\int_{-\infty}^{\infty} \rho(x)e^{-i\gamma G_{pe} y T_{pe}}dy\right] e^{-i\omega_0 t} \tag{2.24}$$

De nuevo, como consecuencia de la demodulación, la señal resultante es

$$S(t) = \int_{-\infty}^{\infty} \rho(y)e^{-i\gamma G_{pe} y T_{pe}}dy \tag{2.25}$$

Finalmente, es posible establecer una conexión entre la codificación espacial, en frecuencia y en fase, y la transformada de Fourier. Para ello se emplea la notación del espacio K, estableciendo en primer lugar las expresiones de las variables k_x y k_y

$$k_x = \frac{\gamma}{2\pi} G_x t \tag{2.26}$$

$$k_x = \frac{\gamma}{2\pi} G_y T_{pe} \tag{2.27}$$

Sustituyendo estas variables en las *Ecuaciones 2.22 y 2.25*, respectivamente. se obtiene

$$S(k_x, k_y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \rho(x,y)e^{-i2\pi(k_x x + k_y y)}dxdy \tag{2.28}$$

Finalmente, la imagen de resonancia magnética se puede obtener a partir de la información en el espacio K mediante la aplicación de la transformada inversa de Fourier.

2.2. OpenCLIPER

Las técnicas de obtención de imágenes médicas han evolucionado en gran medida en las últimas décadas. Esta evolución ha dado lugar a una mayor rapidez, facilidad de uso y precisión, no sin esto suponer un incremento de la carga computacional. Los elevados costes computacionales han tratado de paliarse mediante el uso de GPUs (*Graphic Processing Units*), pero, dado que estos dispositivos funcionan de forma asíncrona con respecto a la CPU, su uso plantea importantes cuestiones relativas a la comunicación y sincronización que impiden un desarrollo directo del *software*.

En respuesta a lo anterior surge OpenCLIPER [8], un *framework* basado en el lenguaje de programación OpenCL cuyo objetivo fundamental es evitar que los programadores tengan que hacer frente a otras cuestiones diferentes del propio desarrollo de los algoritmos. Este *framework* permite obviar tareas improductivas como pueden ser inicializaciones, carga y guardado de datos, visualización o depuración. El hecho de utilizar OpenCL como lenguaje de programación garantiza, además, la independencia del código con respecto al dispositivo empleado, por lo que el código es multidispositivo (CPU, GPU, FPGA, etc.).

En esta sección se describen las principales características de este *framework* y su arquitectura interna para, en último lugar, enumerar los pasos a seguir en el desarrollo de un programa haciendo uso de OpenCLIPER.

2.2.1. Características

El *framework* OpenCLIPER está diseñado como un conjunto de clases de C++ centradas en proporcionar al programador como funcionalidades principales la gestión del dispositivo de computación, la carga y guardado de datos y la gestión de los algoritmos. Se procede a describir con mayor detalle estas tres funcionalidades [8].

En primer lugar, la necesidad de facilitar la gestión del dispositivo de computación surge por no ser automática la selección del dispositivo en OpenCL, lo cual supone uno de sus principales inconvenientes. Para solventarlo, OpenCLIPER permite realizar la selección en una única llamada a partir de una combinación de criterios. Por otro lado, OpenCLIPER hace uso de la versión 1.2 de OpenCL, adoptada por la mayor parte de los fabricantes, lo cual garantiza que el *framework* sea soportado por una amplia lista de dispositivos de computación.

En segundo lugar, un requerimiento esencial en cuanto a la carga y guardado de los datos es proporcionar un rápido manejo de los mismos. En ese sentido, OpenCLIPER maximiza la velocidad de transferencia de datos con el dispositivo de computación mediante la utilización automática de memoria mapeada en *host*[1] y la realización de las transferencias en una única

[1]El dispositivo *host* va a ser típicamente una CPU conectada al dispositivo de computación, que podrá ser una GPU. El uso de memoria mapeada (*pinned memory*) en *host* facilita el traslado de datos entre *host* y el dispositivo, puesto que permite realizar las transferencias mediante acceso directo a memoria (DMA) en lugar de a través de la CPU. Las transferencias realizadas mediante DMA suelen ser del orden de 2 veces más rápidas que las realizadas a través de la CPU.

llamada. Por otro lado, OpenCLIPER desconoce la organización interna de los datos, lo cual posibilita que objetos de datos complejos puedan ser transferidos en una única llamada. En cuanto a la gestión de la memoria, OpenCLIPER maneja de forma automática las posiciones de los datos y sus tamaños tanto en *host* como en el dispositivo de computación. Los objetos de datos se mantienen alineados y contiguos en memoria permitiendo un procesado en *batches*, puesto que la posición incial y el tamaño de cada componente son conocidos y pueden ser utilizados por los *kernels* de OpenCL.

En último lugar, se facilita la gestión de los algoritmos de forma que la carga y compilación de los *kernels* se lleva a cabo de forma automática en una única llamada. El trabajo con los *kernels* requiere como únicas acciones establecer sus datos de entrada y de salida, establecer sus parámetros y ejecutar. Con el objetivo de favorecer el rendimiento, se separa la fase de inicialización y ejecución. Todo esto se realiza a través de la clase `Process`, que se comentará más adelante. Por último y como se ha mencionado con anterioridad, estar basado en OpenCL hace posible que el mismo código fuente sea válido para todos los dispositivos de computación.

2.2.2. Arquitectura interna

OpenCLIPER está diseñado como un conjunto de clases de C++ que se presentan en la *Figura 2.5*. A continuación, se procede a la enumeración y descripción de los principales aspectos de las mismas [8].

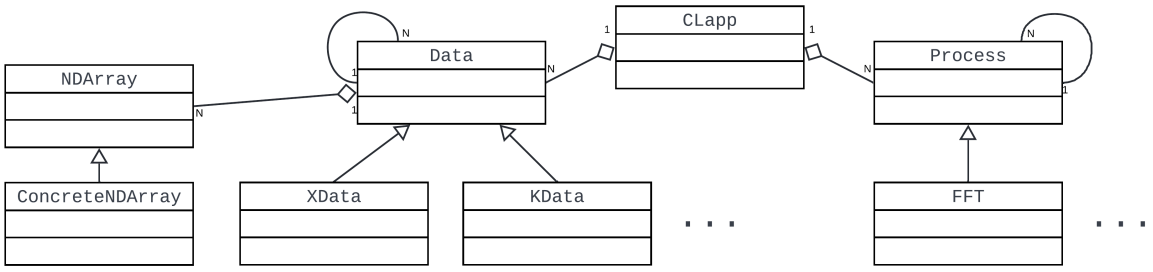


Figura 2.5: Diagrama de clases de OpenCLIPER [8].

La clase principal de OpenCLIPER es `CLapp`. Esta clase actúa como interfaz para el dispositivo OpenCL y, como tal, mantiene información sobre las actuales plataformas y dispositivos de OpenCL. Dispone, además, de la lista de objetos de datos a ser procesados en el dispositivo de computación y gestiona tanto su memoria como sus transferencias de datos.

Por otro lado, se establecen tres clases básicas involucradas en el almacenamiento de los datos. La primera de ellas es la clase `Data` que se define como una clase abstracta, de forma que será necesario derivar especializaciones de la misma para las distintas implementaciones finales. Los objetos de sus clases derivadas contendrán diferentes estructuras de datos n-dimensionales como imágenes o volúmenes. Se predefinen dos primeras especializaciones básicas a través de las clases `XData`, para datos con una interpretación física directa, y `KData`, para datos en el espacio K.

Los objetos de las clases derivadas de `Data` estarán formados por uno o más objetos de una segunda clase, `NDArray`. Estos objetos pueden tener distintos tamaños y dimensiones, de forma que datos heterogéneos pueden ser almacenados en un único objeto. La clase `NDArray` es también una clase abstracta, con atributos y métodos limitados de forma que sean aplicables a cualquier tipo de datos. La existencia de esta clase facilita el desarrollo de nuevas especializaciones de la clase `Data`, puesto que se evita que el desarrollador deba prestar atención a aspectos específicos del almacenamiento de los datos.

La clase `ConcreteNDArray` completa el almacenamiento de los datos, puesto que es la clase que los almacena verdaderamente. No está pensada para ser empleada por los usuarios de OpenCLIPER, dado que contiene datos en bruto y detalles dependientes del tipo de datos, que constituyen particularidades de bajo nivel que se pretenden aislar del usuario.

En último lugar se encuentra la clase `Process`, que actúa a modo de interfaz para los algoritmos de procesado de datos. Se define como una clase abstracta de la que será necesario derivar para los procesos propios de las distintas implementaciones a desarrollar.

2.2.3. Uso de OpenCLIPER

El desarrollo de un programa a través del *framework* OpenCLIPER requiere, de forma general, de los siguientes pasos [8]:

1. Crear un objeto `CLapp` y cargar en él los *kernels* correspondientes.
2. Crear o cargar un objeto `Data` de entrada y crear un objeto `Data` de salida.
3. Crear los objetos `Process` necesarios y establecer sus entradas y salidas a los objetos `Data` anteriores.
4. Inicializar y ejecutar los procesos.
5. Visualizar y/o guardar los datos resultantes.

2.3. Fundamentos de Julia y de la simulación de MRI

2.3.1. Julia

El lenguaje de programación Julia surge con el propósito de proporcionar un nuevo enfoque acerca de la computación numérica buscando la simplicidad, pero sin renunciar a la rapidez [5]. Se trata de un lenguaje de programación de alto nivel en cuyo diseño se ha buscado garantizar de forma simultánea la productividad y el rendimiento.

Según sus autores, ha existido la concepción de que la consecución de un buen rendimiento requería del uso de lenguajes de programación estáticos, caracterizados por la escasa flexibilidad de su código y de sus estructuras lógicas, como pueden ser C o Fortran. En cambio, los lenguajes

de programación dinámicos ofrecen una mayor flexibilidad y facilidad en su aprendizaje, entre los que se encuentran Python o MATLAB [18]. Los lenguajes de programación dinámicos han experimentado durante los últimos años un crecimiento importante, favorecido en gran medida por la simplicidad en su uso. Sin embargo, con carácter general, en el diseño de estos lenguajes no se ha planteado como objetivo principal alcanzar un alto rendimiento [5].

En este contexto, la innovación que introduce Julia reside en la combinación del rendimiento y la productividad, a través de distintas características, entre las que destacan un sistema de tipos que permite definirlos de forma opcional, el *multiple dispatch*[2] y la metaprogramación[3] [5]. En cuanto a la interacción de los usuarios con Julia, esta puede realizarse a través de una REPL (*read-eval-print loop*), a través de comandos en un fichero `.jl` o tecleando directamente en un *notebook* de Jupyter [5].

En definitiva, el lenguaje de programación Julia destaca por ofrecer a los programadores la posibilidad de escribir código claro, de alto nivel, genérico y abstracto, con una apariencia muy próxima a las fórmulas matemáticas a las que corresponde, generando al mismo tiempo un código máquina de bajo nivel, rápido y eficiente.

2.3.2. Simulador MRIsim.jl

MRIsim.jl es un *framework* para la simulación del fenómeno MRI desarrollado en la Pontificia Universidad Católica de Chile [4]. Se diseña con el propósito de dar respuesta a la falta de entornos de simulación de MRI de código abierto, rápidos y escritos en un lenguaje de alto nivel que posibiliten una fácil extensión de los mismos. En él, se realizan simulaciones de extremo a extremo a nivel nuclear para poder evaluar los distintos procesos de obtención de la imagen de resonancia magnética, desde la fase de adquisición hasta la fase de reconstrucción [19]. Cuenta con un motor de simulación realista y ofrece resultados muy competitivos con respecto a otros simuladores existentes.

El *framework* está escrito en el lenguaje de programación Julia. La elección de este lenguaje de programación viene justificada, en primer lugar, por tratarse de un lenguaje de computación científica rápido y con una sintaxis *math-friendly*, es decir, el código a escribir se aproxima a las expresiones matemáticas correspondientes; en segundo lugar, el código a elaborar es claro y conciso y fácilmente modificable, y, en último lugar, permite combinar de forma sencilla operaciones con bucles y operaciones matriciales [20]. Sin embargo, este lenguaje de programación de alto nivel abstrae la disposición de los datos en memoria, lo cual puede afectar negativamente en el rendimiento del simulador. Por otra parte, aunque actualmente es capaz de realizar operaciones paralelas, únicamente soporta dispositivos GPU de un único fabricante.

En cuanto a su uso, el simulador dispone de una interfaz gráfica de usuario (GUI, *Graphical User Interface*) desde la cual poder iniciar la simulación, mostrada en la *Figura 2.6*.

[2]El *multiple dispatch* es una funcionalidad que permite determinar el método a utilizar teniendo en cuenta todos los argumentos del mismo y no solo el primero de ellos.

[3]La metaprogamación es una facilidad que posibilita la escritura de código que cree o modifique el propio código en Julia.

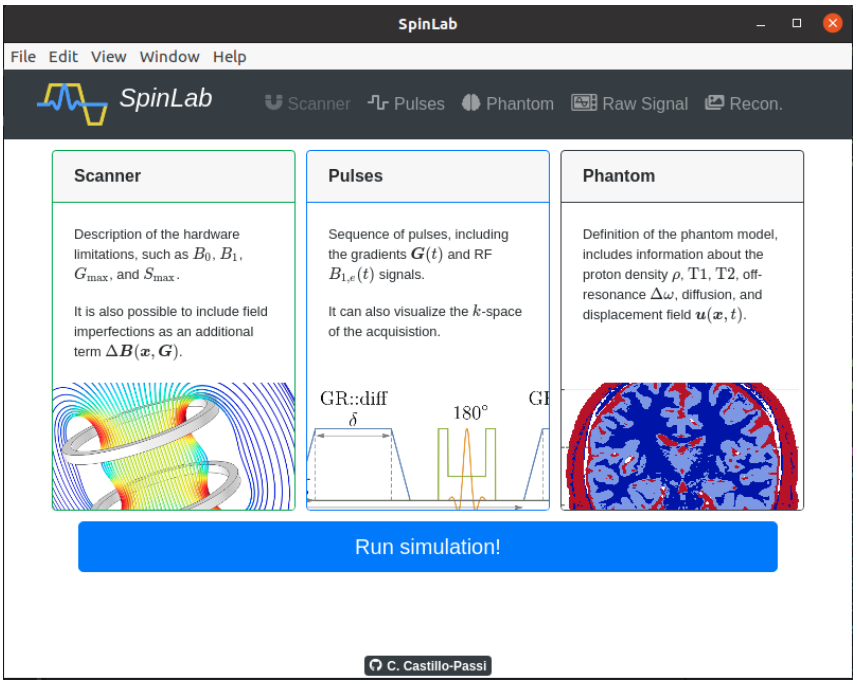
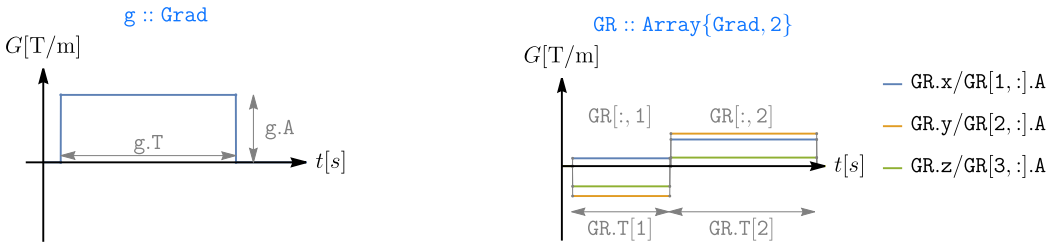


Figura 2.6: GUI del simulador MRIsim.jl [4].

El simulador se diseña a partir de una serie de estructuras de datos correspondientes a distintos conceptos del fenómeno MRI. Estas estructuras pueden verse reflejadas en el esquema simplificado del diseño del simulador, mostrado posteriormente en la *Figura 2.11*. Se realiza a continuación una breve descripción de las mismas:

- `Grad`: Estructura formada por dos reales correspondientes a la amplitud (`A`) y duración (`T`). Las formas de ondas de los gradientes se construyen a partir de la concatenación de estos bloques. La *Figura 2.7* muestra la representación de este bloque y su concatenación para formar los gradientes en cada dirección (x, y, z).

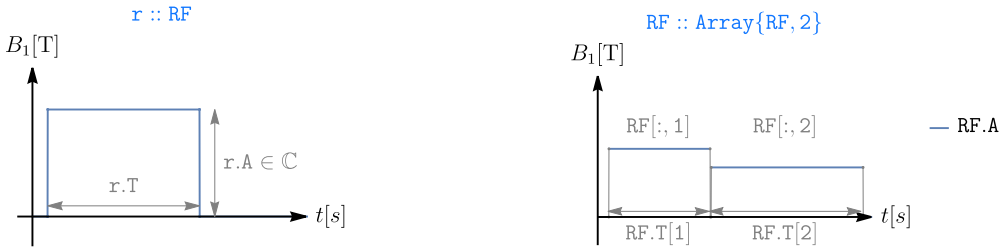


(a) Bloque `Grad`.

(b) *Arrays* de bloques `Grad` en cada dirección.

Figura 2.7: Representación estructura `Grad` [21].

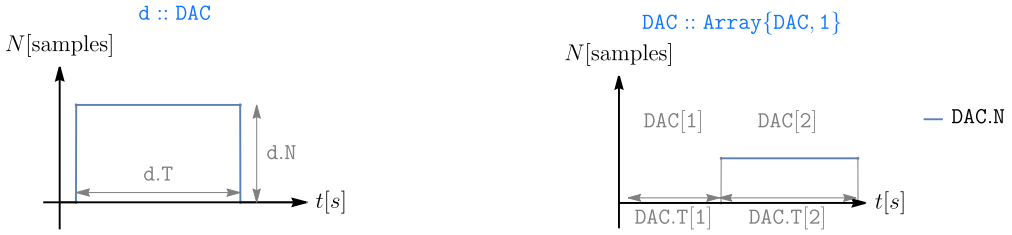
- `RF`: Estructura formada por un complejo correspondiente a la amplitud (`A`) del pulso de RF ($B_{1,x} + jB_{1,y}$) y un real correspondiente a la duración (`T`). La *Figura 2.8* muestra la representación de este bloque y su concatenación.



(a) Bloque `RF`. (b) *Array* de bloques `RF`.

Figura 2.8: Representación estructura `RF` [21].

- `DAC`: Estructura formada por dos reales correspondientes al número de muestras (`N`) y a la duración (`T`). Indica, por tanto, los instantes de tiempo de lectura de la señal de MRI. La *Figura 2.9* muestra gráficamente la representación de este bloque y su concatenación.



(a) Bloque `DAC`. (b) *Array* de bloques `DAC`.

Figura 2.9: Representación estructura `DAC` [21].

- `Sequence`: Estructura constituida por las formas de onda de los gradientes, los pulsos de RF y los tiempos de adquisición de datos. Por tanto, integra tres *arrays* de bloques `Grad`, `RF` y `DAC`, como se muestra en la *Figura 2.10*.

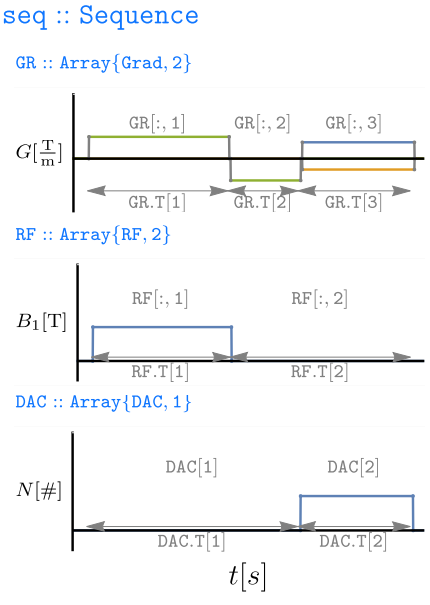


Figura 2.10: Representación estructura `Sequence` [21].

- `Phantom`: Representa el objeto virtual sobre el que se aplica la simulación del fenómeno MRI. Se trata de una estructura que tiene como campos las posiciones espaciales iniciales de los *spins* (x, y, z), la densidad de los protones ρ, las constantes de tiempo $T1$ y $T2$, entre otros, mayoritariamente definidos como vectores.
- `Mag`: Representa el vector de magnetización y consiste en una estructura formada por un complejo y un real correspondientes a la componente transversal (M_{xy}) y longitudinal (M_z), respectivamente.
- `Spinor`: Estructura formada por dos complejos (α y β), empleada para la aplicación de una rotación al vector de magnetización durante el proceso de simulación y, así, evitar el empleo de matrices para tal fin.

La *Figura 2.11* muestra un esquema básico para facilitar la comprensión del diseño del simulador. En él, los cuadros con línea continua y fondo blanco representan diferentes estructuras de datos empleadas en el simulador; los cuadros de texto curvados con fondo morado hacen referencia al fichero concreto involucrado, y las franjas sombreadas en gris representan las distintas etapas del proceso.

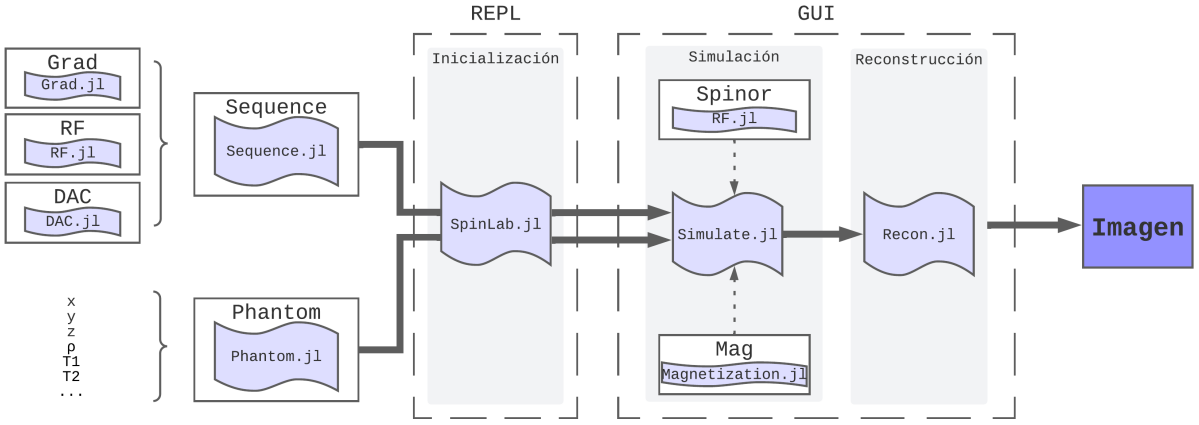


Figura 2.11: Esquema básico del simulador MRIsim.jl.

El proceso de simulación comienza con una primera fase de inicialización que tiene lugar a través del fichero `SpinLab.jl`. El simulador requiere de dos estructuras de datos principales, un elemento `Sequence`, compuesto por bloques `Grad`, `RF` y `DAC`, y un bloque `Phantom`, con sus distintas componentes. Tanto la secuencia como el fantoma se deben definir en dicho fichero.

Desde la REPL, se debe llamar a la función `SpinLab()` para la creación la interfaz gráfica (mostrada en la *Figura 2.6*), en la que el usuario debe pulsar sobre `Run simulation!` para que comience la simulación, en la que se hará uso de la secuencia y el fantoma anteriores.

Las operaciones que tienen lugar en la fase de simulación se encuentran en el fichero `Simulator.jl` y consisten, de forma simplificada, en la aplicación a cada *spin* de la ecuación de Bloch en el marco rotatorio, vista en la *Ecuación 2.11*. Finalmente, la reconstrucción de la imagen se realiza a través del fichero `Recon.jl`.

La simulación del fenómeno MRI y la reconstrucción de la imagen involucran una serie de funciones y la evaluación de distintas condiciones que se representan en el diagrama de flujo de la *Figura 2.12*, en el que se han realizado algunas simplificaciones para facilitar el análisis. Se realiza a continuación una descripción de ambos procesos empleando el diagrama como referencia:

- La fase de simulación se inicia, como se ha comentado con anterioridad, a través del botón `Run simulation!` de la GUI. Con esto, se produce la llamada a la función `simulate`, en la que, en primer lugar, se obtienen los instantes de tiempo de la simulación a partir del *array* de `DACs` que integra la secuencia de entrada. Para ello se emplean las funciones `get_uniform_times` o `get_variable_times`, en función del valor que se haya establecido del parámetro `step`. Adicionalmente, se obtiene el objeto `Mag` correspondiente al vector de magnetización inicial a partir de la densidad de los protones, que corresponde a la componente ρ del bloque `Phantom` de entrada.

- A continuación, se efectúa la llamada a la función `run_sim_time_iter`. Las operaciones posteriores se efectúan por intervalos de tiempo. Para ello, sea llama a la función `kfoldperm`, que proporciona los índices de los elementos del vector de tiempos que integran cada una de las `parts_t` particiones. Para cada intervalo de tiempo, se comprueba si se encuentra activo el pulso de RF de la secuencia de entrada a través de la función `is_RF_on`, y se llama a la función `run_spin_excitation_parallel` o a la función `run_spin_precession_parallel`, según corresponda. En cualquier caso, la llamada a la función que corresponda incluye como parámetro el vector de magnetización que se habrá visto modificado en el intervalo anterior.

- Las dos funciones anteriores difieren en las operaciones a realizar, pero tienen un funcionamiento análogo. Ambas llaman de nuevo a la función `kfoldperm`, en este caso para la fragmentación del bloque `Phantom`, que también se divide en partes. Las operaciones se aplican sobre el fantoma mediante la llamada a las funciones `run_spin_excitation` o `run_spin_precession`. Las señales correspondientes a distintas particiones del fantoma para un mismo instante de tiempo se suman, de forma que se obtiene un valor de la señal para cada instante temporal.

- Una vez finalizado el proceso para todos los intervalos de tiempo, se habrá obtenido la señal `S`. Está señal es interpolada en los instantes de tiempo de lectura de la señal que determinan los bloques `DAC` de la secuencia a partir de los valores de número de muestras y duración. Para ello, se emplea la función `LinearInterpolation`.

- Obtenida la señal interpolada (`S_interp`), comienza la fase de reconstrucción. Se realizan una serie de transformaciones sobre la señal para obtener los datos en el espacio K (`kdata`) y, finalmente, se llama a la función `ifftc`, que realiza esencialmente una Transformada Inversa de Fourier para la obtención de la imagen final.

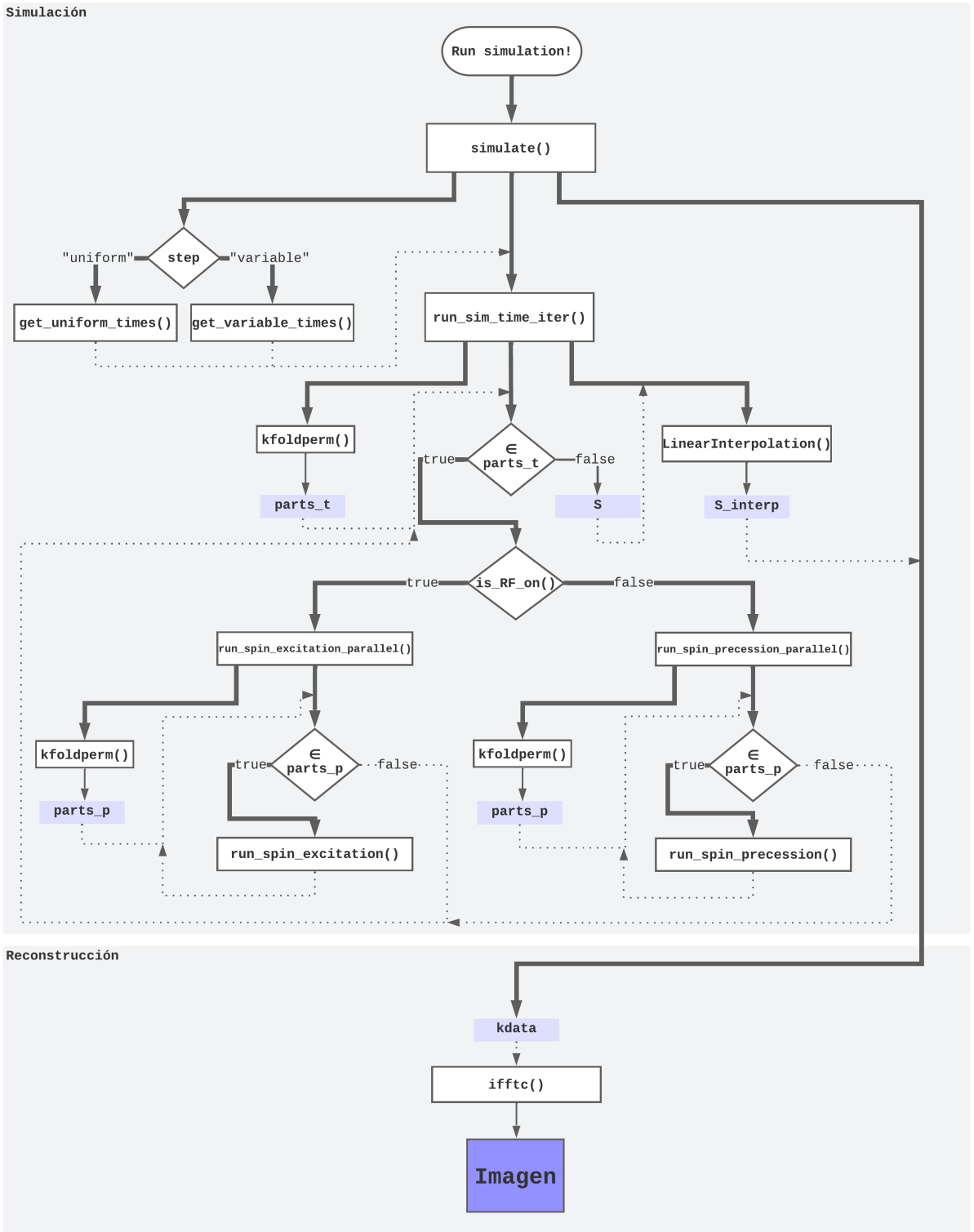


Figura 2.12: Diagrama de flujo del simulador MRIsim.jl.

Capítulo 3

Desarrollo del simulador en OpenCLIPER

El simulador MRIsim.jl [4] está implementado en Julia, un lenguaje de alto nivel que abstrae la disposición de los datos en memoria, lo cual permite un rápido desarrollo del código, pero puede afectar de forma negativa al rendimiento del simulador. En cambio, el diseño del simulador que se describe en esta sección se ha efectuado con el objetivo de proporcionar el máximo rendimiento posible, para lo cual se minimizan las transferencias *host*-dispositivo y los movimientos de datos durante las operaciones, y se maximiza la cohesión de los datos, lo cual favorece el uso eficiente de las cachés. Todo esto se ve facilitado por las ventajosas funcionalidades que ofrece el *framework* OpenCLIPER en cuanto estructuras de datos y gestión de algoritmos, descritas anteriormente en la *Sección 2.2*.

3.1. Estructuras de datos

El simulador hace uso de una serie de estructuras de datos que representan distintos conceptos del fenómeno MRI. Estas estructuras se han definido mayoritariamente como clases, las cuales se muestran en el diagrama de clases del simulador de la *Figura 3.1*. Para la creación de las nuevas clases se ha hecho uso de las clases preexistentes, definidas por el *framework* OpenCLIPER. A partir de estas se han derivado otras nuevas como es el caso de las clases `Phantom` y `Mag`, que derivan de la clase abstracta `Data` de OpenCLIPER. De forma independiente de las clases predefinidas por OpenCLIPER se han creado las clases `Sequence`, `Grad`, `RF` y `DAC`.

A continuación, se realiza una descripción de cada una de estas seis clases fundamentales para el manejo de los datos del simulador. En esta descripción se parte de las clases que corresponden a estructuras de datos más elementales para, posteriormente, definir aquellas estructuras más complejas.

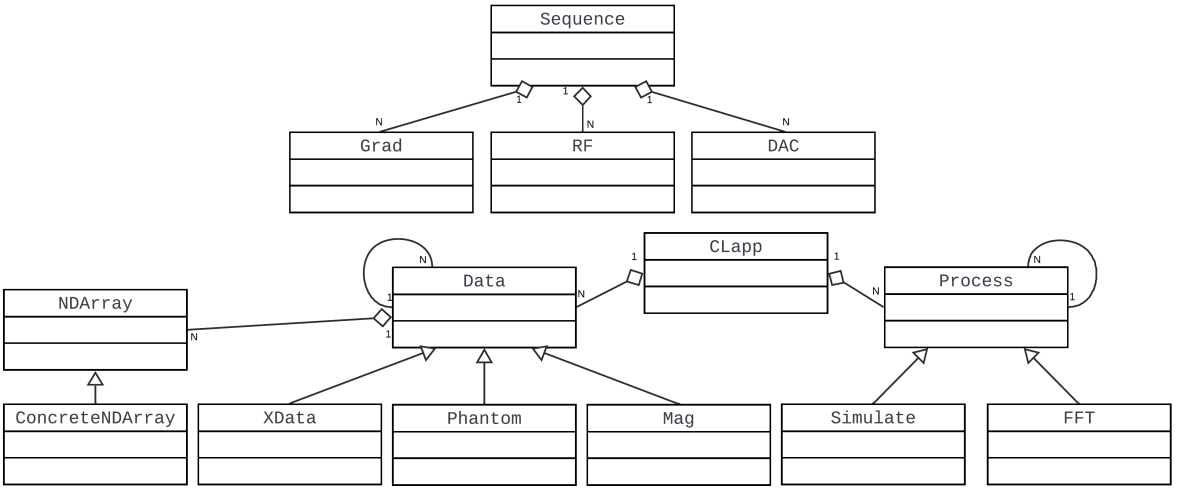


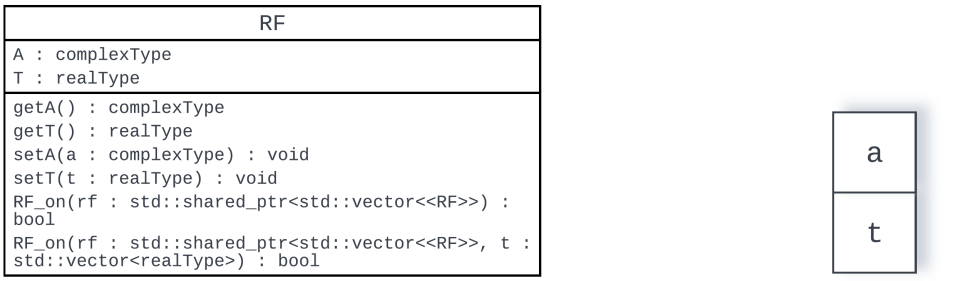
Figura 3.1: Diagrama de clases del simulador en OpenCLIPER.

En primer lugar, se encuentra la clase `Grad`, formada por dos valores reales correspondientes a su amplitud (`A`) y su duración (`T`). Esto se refleja tanto en la descripción de su clase como en la representación proporcionada de un objeto `Grad`, mostradas ambas en la *Figura 3.2*. A partir de la concatenación de objetos de esta clase se podrán construir las secuencias del simulador, a las que se hará referencia posteriormente. En cuanto a los métodos de esta clase, se definen los métodos básicos *get* y *set*, tanto para su amplitud como para su duración.

(a) Clase `Grad`. (b) Objeto `Grad`.

Figura 3.2: Representación clase `Grad`.

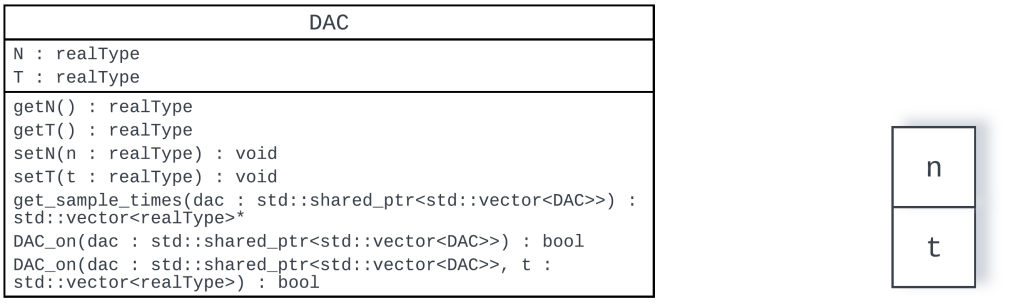
En segundo lugar, se define la clase `RF`, cuya descripción y representación se muestran en la *Figura 3.3*, que va a permitir construir los pulsos de RF. Cada objeto está formado por un valor complejo (`A`) correspondiente a la amplitud del pulso ($B_{1,x} + jB_{1,y}$), y un real (`T`) correspondiente a su duración. Al igual que los objetos de la clase anterior, los objetos `RF` suponen un bloque constitutivo de las secuencias del simulador. En cuanto a los métodos de esta clase, se definen de nuevo los métodos *get* y *set*. Adicionalmente, se define el método `RF_on` que indica si en un vector de objetos `RF` hay alguno activo, es decir, con amplitud mayor que 0. Este método se sobrecarga para poder determinar si hay algún `RF` activo en un cierto intervalo de tiempo del vector.



(a) Clase `RF`. (b) Objeto `RF`.

Figura 3.3: Representación clase `RF`.

En tercer lugar, se encuentra la clase `DAC`, representada en la *Figura 3.4*, constituida por un entero (`N`) correspondiente al número de instantes de lectura y un real (`T`) que corresponde a la duración del bloque. La concatenación de objetos de esta clase formará parte, de nuevo, de las secuencias del simulador, y permite determinar los instantes de tiempo en los que tiene lugar la lectura de la señal. En cuanto a los métodos de la clase, de forma adicional a los métodos *get* y *set*, se define primeramente el método `get_sample_times`, que proporciona los instantes de tiempo de lectura de la señal. Se define también el método `DAC_on` que, de forma análoga a la clase `RF`, se sobrecarga para poder determinar si en un vector de objetos `DAC` hay algún instante de lectura, es decir, algún objeto con valor de `N` mayor que 0, bien en la totalidad de la duración del vector, o bien en un intervalo de tiempo del mismo.



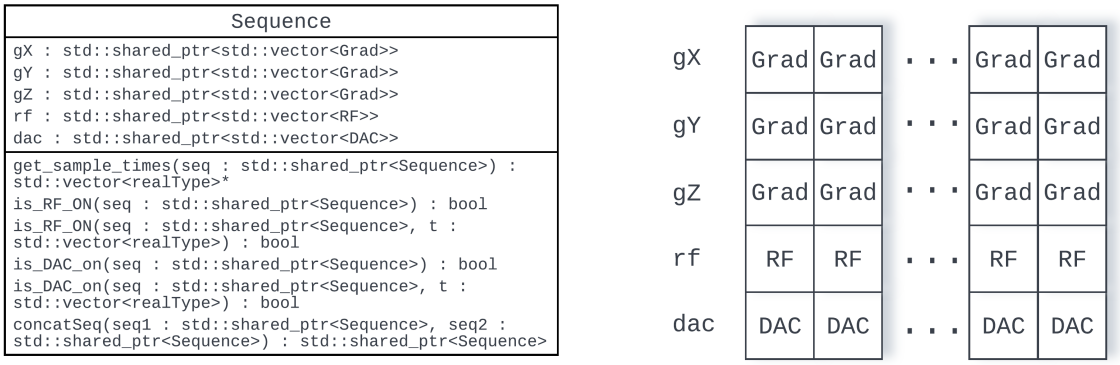
(a) Clase `DAC`. (b) Objeto `DAC`.

Figura 3.4: Representación clase `DAC`.

En cuarto lugar, se encuentra la clase `Sequence`, cuyos objetos están integrados por objetos de las tres clases anteriores. Se constituye a partir de un puntero a un vector de objetos `Grad` en cada dirección (x, y, z), un puntero a un vector de objetos `RF` y un puntero a un vector de objetos `DAC`. A efectos de esta descripción, se puede considerar que un objeto de la clase `Sequence` corresponde con la matriz mostrada en la *Figura 3.5b*. Procede destacar, además, que se considera de forma general que todos los vectores de los que consta la secuencia van a tener el mismo número de elementos con la misma duración uno a uno. La construcción de un objeto `Sequence` no requiere de los cinco vectores mostrados en la figura, sino que es posible

crear una secuencia a partir de uno o más de ellos. En ese caso, los vectores no definidos son creados con amplitud (caso de `Grad` o `RF`) o número de instantes (caso de `DAC`) igual a cero para todos sus elementos, y con duración de sus elementos idéntica al vector o vectores existentes.

Los métodos principales definidos para esta clase se muestran en la *Figura 3.5a*. En primer lugar, está el método `get_sample_times`, que hace uso del método con el mismo nombre de la clase `DAC` y que proporciona los instantes de tiempo en los que tiene lugar la lectura de la señal; en segundo lugar, el método con sobrecarga `is_RF_on`, que hace uso de los métodos `RF_on` de la clase `RF`, y que permite determinar si hay algún pulso de RF activo en la totalidad de la secuencia o en un intervalo de tiempo; en tercer lugar, el método sobrecargado `is_DAC_on`, que hace uso de los métodos `DAC_on` de la clase `DAC`, y que permite saber si hay algún instante de lectura en la totalidad de la secuencia o en un intervalo de la misma, y, por último, el método `concatSeq`, que permite concatenar dos secuencias.



(a) Clase `Sequence`.

(b) Objeto `Sequence`.

Figura 3.5: Representación clase `Sequence`.

En quinto lugar, se encuentra la clase `Phantom` que deriva de la clase abstracta `Data`, lo cual facilita su transferencia al dispositivo de computación para el posterior trabajo con los *kernels* del simulador. Como tal, consta de un único `NDArray` con 11 filas. Cada una de esas filas es un vector de números reales correspondiente a cada una de las componentes del objeto `Phantom`, tal como se muestra en la *Figura 3.6*, mientras que el número de columnas corresponde al número de *spins* del fantoma.

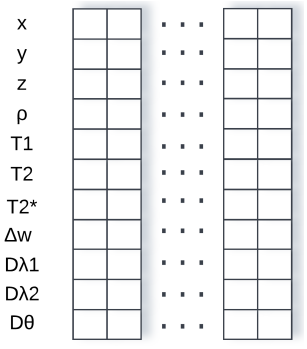


Figura 3.6: Representación clase `Phantom`.

Se proporciona un fantoma predefinido a través de la función `heart_phantom`, cuya imagen reconstruida se muestra posteriormente en la *Figura 4.2*. Este consiste en un círculo de un determinado radio r concéntrico con un anillo exterior a él de anchura $R - r$, donde $R > r$. A cada región se le asigna un valor de ρ, $T1$ y $T2$. Además, se crean campos de desplazamiento con el objetivo de simular el latido cardiaco. Es posible añadir un segundo anillo, que pretende simular la parte grasa, a través de un *flag* que se introduce como argumento de la función.

En sexto y último lugar, se encuentra la clase `Mag` que deriva también de la clase abstracta `Data`. Consta de un único `NDArray` de 2 filas correspondientes a los vectores de números complejos de la componente transversal (`Mxy`) y la componente longitudinal (`Mz`). La componente `Mz` va a tomar siempre valores reales, aunque por simplicidad se ha considerado compleja. Por ello, tal como se refleja en la *Figura 3.7*, su parte imaginaria será siempre nula.

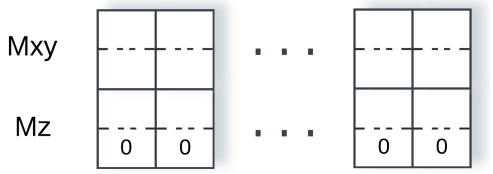


Figura 3.7: Representación clase `Mag`.

El constructor de la clase `Mag` se define a partir de un objeto `Phantom`. Por defecto, se establece la componente transversal como nula y la componente longitudinal igual al vector ρ del fantoma. Es por esto que el número de columnas del objeto `Mag` corresponde con el número de *spins* del fantoma a partir del cual se obtiene.

3.2. Secuencias

Las secuencias constituyen una parte esencial de la MRI y, como tal, también de los simuladores de este fenómeno. La descripción de la clase `Sequence`, realizada en la sección anterior, pone de manifiesto la posibilidad de construir mediante el simulador en OpenCLIPER cualquier secuencia deseada a partir de objetos `Grad`, `RF` y `DAC`. Con el objetivo de facilitar su construcción, se proporcionan una serie de secuencias predefinidas.

Las secuencias se han diseñado a partir de la creación de una clase vacía, denominada `PulseDesigner`, para la cual se definen una serie de funciones generadoras de secuencias. En concreto, esta clase contiene las funciones `RF_hard`, `DIF_base`, `DIF_null`, `radial_base` y `EPI_base`, que se describen en detalle a continuación:

- La función `RF_hard` crea una secuencia a partir de un único objeto `RF` con amplitud compleja `B1` y duración `T`. Como se indicó en la descripción de la clase `Sequence`, el resto de componentes de la secuencia se establecen a 0, pero con duración idéntica a la del objeto `RF`. Esta secuencia se presenta gráficamente en la *Figura 3.8*, en la que únicamente se muestra la componente no nula de la secuencia, `rf`.

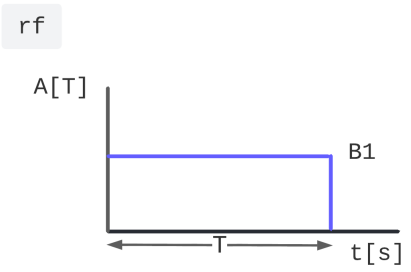


Figura 3.8: Secuencia `RF_hard`.

- La función `DIF_base` crea la secuencia básica de difusión sin empleo de la técnica *moment nulling*, a partir de dos objetos gradientes rectangulares con amplitudes opuestas y duración δ en la dirección x. Los gradientes rectangulares están separados por una duración de valor $\Delta - \delta$, tal como se muestra en la *Figura 3.9*.

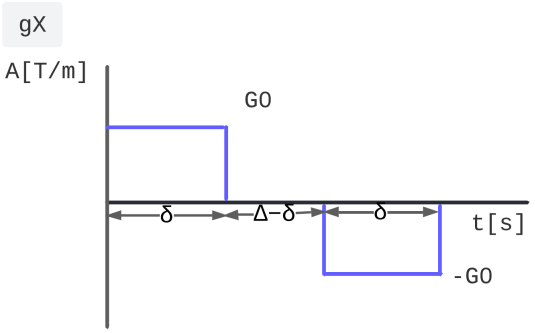
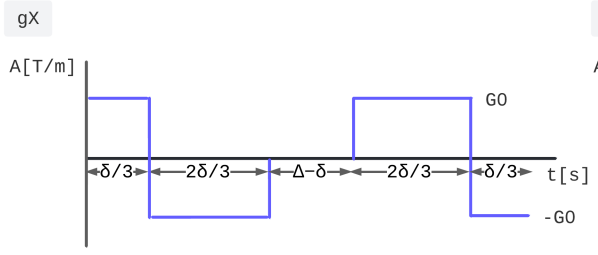
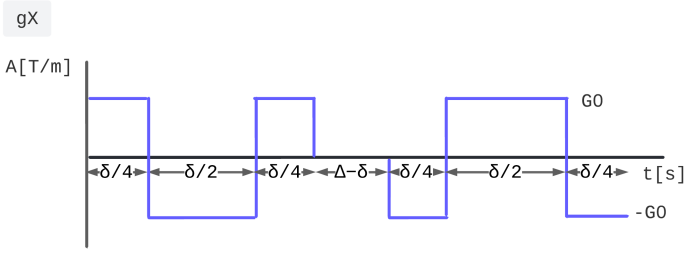


Figura 3.9: Secuencia `DIF_base`.

- La función `DIF_null` crea una secuencia de difusión con empleo de la técnica *moment nulling*, que pretende eliminar mediante la introducción de objetos gradientes adicionales en la secuencia, con duraciones y amplitudes adecuadas, efectos de patrones determinísticos de movimiento de los protones, como velocidad o aceleración constante, que pueden ocasionar la aparición de artefactos en la imagen [15]. Esta función da lugar a dos posibles secuencias en función del resultado de comparar $\Delta - 4\frac{\delta}{3}$ con el valor 10^{-6}, compuestas en ambos casos por un vector de objetos `Grad` en la dirección x. Ambas secuencias se muestran en la *Figura 3.10*.



(a) Si $\Delta - 4\frac{\delta}{3} < 10^{-6}$.



(b) Si $\Delta - 4\frac{\delta}{3} \geq 10^{-6}$.

Figura 3.10: Secuencia `DIF_null`.

- La función `radial_base` crea una secuencia para muestreo radial. Consta de un objeto `Grad` en la dirección x y un objeto `DAC` que establece durante el intervalo de tiempo de ese gradiente el número de instantes de lectura que tiene lugar. La secuencia se complementa con una secuencia previa y posterior de ajuste de fase para localizar el punto de lectura en el lugar correcto [21], que consisten también en un objeto `Grad` en la dirección x. En la *Figura 3.11* se representan las componentes no nulas de esta secuencia.

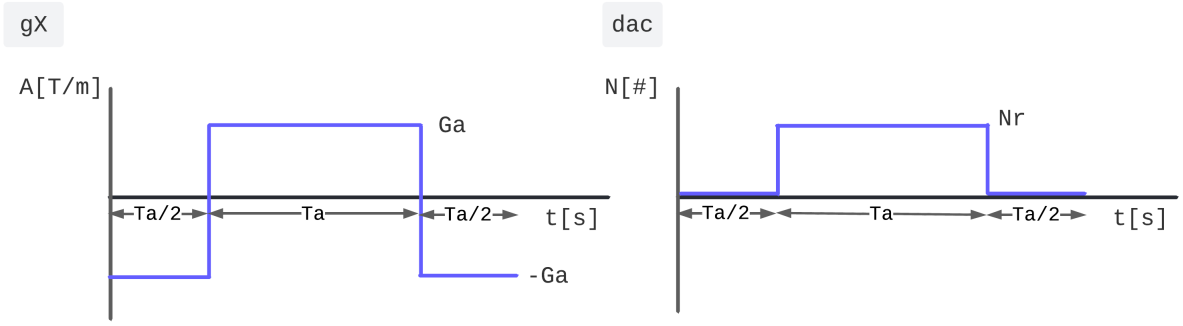


Figura 3.11: Secuencia `radial_base`.

- La función `EPI_base` genera una secuencia EPI (*echo-planar imaging*) cartesiana. La secuencia se construye de forma que el espacio K va a constar de Ny líneas y Nx puntos por línea, siendo $Nx = Ny$. El período de muestreo toma el valor Δt.

  La secuencia está formada por dos vectores de objetos `Grad` en las direcciones x e y y un vector de objetos `DAC`, con $2Ny - 1$ elementos cada vector. El vector de gradientes en la dirección x está formado por objetos `Grad` con amplitud Ga y duración Ta para las posiciones pares (alternando el signo de la amplitud entre líneas), y amplitud nula y duración Δt en posiciones impares. El vector de gradientes en la dirección y está formado por objetos `Grad` con amplitud nula y duración Ta en las posiciones pares, y amplitud Ga y duración Δt en las impares. Por último, el vector de objetos `DAC` toma valor N muestras y duración Ta en las posiciones pares, y 0 muestras y duración Δt en las posiciones impares. La secuencia es completada por dos secuencias de ajuste de fase, una previa, formada por un objeto `Grad` en la dirección x, y una posterior, formada por un objeto `Grad` en la dirección x y otro en la dirección y, mediante las cuales se desplaza el punto de lectura a la posición correcta del espacio K.

3.3. Simulación

La simulación del fenómeno MRI en OpenCLIPER requiere de la definición de una nueva clase a la que se ha denominado `Simulate`, que deriva de la clase `Process` de OpenCLIPER, tal como se muestra anteriormente en la *Figura 3.1*. Para que dicha simulación pueda tener lugar, esencialmente se debe proporcionar al proceso un objeto `Phantom`, que se establece como objeto `Data` de entrada y un objeto `Sequence`, establecido como parámetro de inicialización.

Como se ha indicado en la *Sección 2.2.1*, OpenCLIPER permite separar la inicialización y ejecución del proceso con el objetivo de favorecer el rendimiento. En la fase de inicialización del proceso `Simulate` tienen lugar las siguientes operaciones:

- Obtención de los instantes de tiempo en los que va a tener lugar la simulación a partir del vector de objetos `DAC` de la secuencia de entrada. El parámetro de inicialización `step` determina si la distribución de instantes de tiempo de simulación es uniforme o variable.
- Fragmentación del vector de instantes temporales anterior, puesto que la simulación se va a efectuar por intervalos de tiempo. Se hace uso para ello de la función `kfoldperm`, que proporciona los índices de los elementos del vector de tiempos correspondientes a cada intervalo a partir de la longitud del vector de tiempos y del número de intervalos deseados.
- Creación del objeto `Mag` correspondiente al vector de magnetización a partir de la componente ρ del objeto `Phantom` de entrada.
- Creación de los objetos `XData` necesarios para el posterior procesado a través de *kernels*, de forma que se facilita la transferencia de las estructuras de datos al dispositivo de computación.

Finalizada la inicialización del proceso `Simulate`, en la fase de ejecución se realiza un procesado en bucle para cada partición de tiempo obtenida. En términos generales, la simulación se basa en la resolución para cada *spin* de la ecuación de Bloch en el marco rotatorio a la que se hizo referencia en la *Ecuación 2.7*. Para cada intervalo de tiempo se lleva a cabo el proceso mostrado en el diagrama de la *Figura 3.12*, el cual se detalla a continuación:

- En primer lugar, se comprueba si hay un pulso de RF activo en el intervalo de tiempo en consideración. En caso afirmativo, se estaría ante *forced precession*, mientras que en caso contrario, se estaría ante *free precession*. En ambos casos se hace uso de diversos *kernels*, cuya descripción se va a relacionar con las operaciones a las que corresponden, explicadas con detalle en la *Sección 2.1*.
- Las operaciones que tienen lugar en el caso de *forced precession* son las siguientes:
  - El primer paso consiste en determinar las posiciones espaciales (xt, yt, zt) de los *spins* del fantoma en cada uno de los instantes de tiempo de la partición en consideración a partir de las funciones de desplazamiento correspondientes a dicho fantoma. Para ello, se emplea el *kernel* `spinsPosition`.
  - Los *kernels* `elementProd`, `constantDiv` y `elementSum` se emplean para obtener la componente z del campo magnético efectivo. Esta componente toma la siguiente expresión
  $$Bz = \vec{G}(t) \cdot \vec{x}(t) + \frac{\Delta\omega(t)}{\gamma} \tag{3.1}$$
  - Los *kernels* `sumVectorToMat` y `elementSqrt` se emplean para obtener el campo magnético efectivo con sus tres componentes ($\vec{B}$), a partir del vector de objetos `RF` de la secuencia de entrada, que proporciona las componetes x e y, y la componente z antes obtenida. El campo magnético efectivo es, por tanto
  $$\vec{B} = \left(B_{1,x}(t), B_{1,y}(t), \vec{G}(t) \cdot \vec{x}(t) + \frac{\Delta\omega(t)}{\gamma} \right) \tag{3.2}$$

- El vector de magnetización se ve afectado por la aplicación del campo magnético $\vec{B}$, tal como se refleja en la *Ecuación 2.7*. Esto se implementa por medio de la aplicación de una rotación al objeto `Mag`. Para ello, en primer lugar se hace uso del *kernel* `prodB`, mediante el cual se obtiene el ángulo de rotación. En segundo lugar, se obtiene una estructura de datos a la que se va a denominar *spinor* mediante los *kernels* `functionQ` y `spinorsProd`. Esta estructura *spinor* se emplea, finalmente, para efectuar la operación de rotación del vector de magnetización a través del *kernel* `magBySpinor`.

- En el caso de *free precession*, el proceso que tiene lugar es el siguiente:

  - De nuevo, en primer lugar, se utiliza el *kernel* `spinsPosition` para determinar las posiciones espaciales (xt, yt, zt) de los *spins* del fantoma en cada uno de los instantes de tiempo de la partición en consideración a partir de las funciones de desplazamiento correspondientes a dicho fantoma.
  - A continuación, se obtiene la componente z del campo magnético $\vec{B}$ empleando los *kernels* `elementProd` y `elementSumPrec`. En este caso, las componentes x e y son nula, puesto que no hay un pulso de RF activo.
  - El campo $\vec{B}$ se emplea para el cálculo de la fase de la exponencial compleja que se aplicará posteriormente a la componente transversal del vector de magnetización, y que toma la siguiente expresión

    $$\phi = -\gamma||\vec{B}||\Delta t \tag{3.3}$$

    La obtención de la fase requiere del uso del *kernel* `productColumns` para el producto con Δt, y del *kernel* `phaseAngle` para el cálculo de la fase inicial.
  - El cálculo de la fase total se efectúa de forma distinta dependiendo de si en el intervalo de tiempo en consideración existe un objeto `DAC` con número de muestras mayor que 0. En cualquier caso, se calcula la fase establecida en la *Ecuación 3.3* mediante el *kernel* `cumsum`/`sum`; se calcula la fase total como la fase inicial más la fase anterior con el *kernel* `sustractPhase_cumsum`/`sustractPhase_sum`, y se realiza el cálculo de la exponencial compleja mediante el *kernel* `complexExp_cumsum`/ `complexExp_sum`. La fase obtenida se aplica en un *kernel* posterior a la componente transversal del vector de magnetización.
  - La evolución temporal de las componentes del vector de magnetización tras la supresión del campo $\vec{B}$ se expresa en la *Ecuación 2.16*. Mediante los *kernels* `divByT2` y `multByDeltaW` se calculan los exponentes de las exponenciales a aplicar. Tras esto, la evolución de la componente transversal se implementa a través del *kernel* `mxy`, que aplica adicionalmente la exponencial compleja correspondiente a la fase antes calculada. Posteriormente, se emplea el *kernel* `mxyUpdate` para actualizar la componente transversal del objeto `Mag`. La componente longitudinal se actualiza mediante el *kernel* `mzUpdate`.
  - Por último, la señal es obtenida a partir del resultado del *kernel* `mxy`, mediante la suma de los valores de todos los *spins* para cada instante de tiempo.

Finalizado el proceso descrito para todos los intervalos, se obtiene la señal correspondiente a todos los instantes de tiempo de la simulación. Para la obtención a partir de esta señal de la imagen reconstruida, se emplean una serie de *kernels* mostrados en la *Figura 3.13*, que se describen a continuación:

- En primer lugar, se efectúa una interpolación lineal de la señal obtenida en los instantes de tiempo de lectura determinados por los objetos `DAC` de la secuencia mediante el *kernel* `linearInterpolation`.
- Tras esto, se redimensiona la señal mediante el *kernel* `kdata` a una matriz de dimensiones $Nx \times Ny$, ambos valores definidos como parámetros de inicialización.
- Se revierten las columnas de la matriz leídas en sentido inverso por la secuencia EPI mediante el *kernel* `flipKdata`. En principio, se está considerando el uso únicamente de secuencias de tipo EPI.
- El *kernel* `redim` redimensiona la matriz de datos en el espacio K a una matriz de dimensiones 128×128 para favorecer el empleo del posterior proceso `FFT`.
- Se emplea el *kernel* `ifftshift` para retornar la componente de frecuencia cero situada en el centro de la matriz a su posición original.
- A continuación, se emplea la clase `FFT`, que deriva de la clase `Process` de OpenCLIPER, para efectuar la Transformada Inversa de Fourier.
- Se desplaza, ahora, la componente de frecuencia cero al centro de la matriz mediante el *kernel* `fftshift`.
- Finalmente, el *kernel* `renorm` multiplica los elementos de la matriz por el número de *spins* del fantoma.

Finalizadas las operaciones anteriores, se obtiene la imagen final reconstruida.

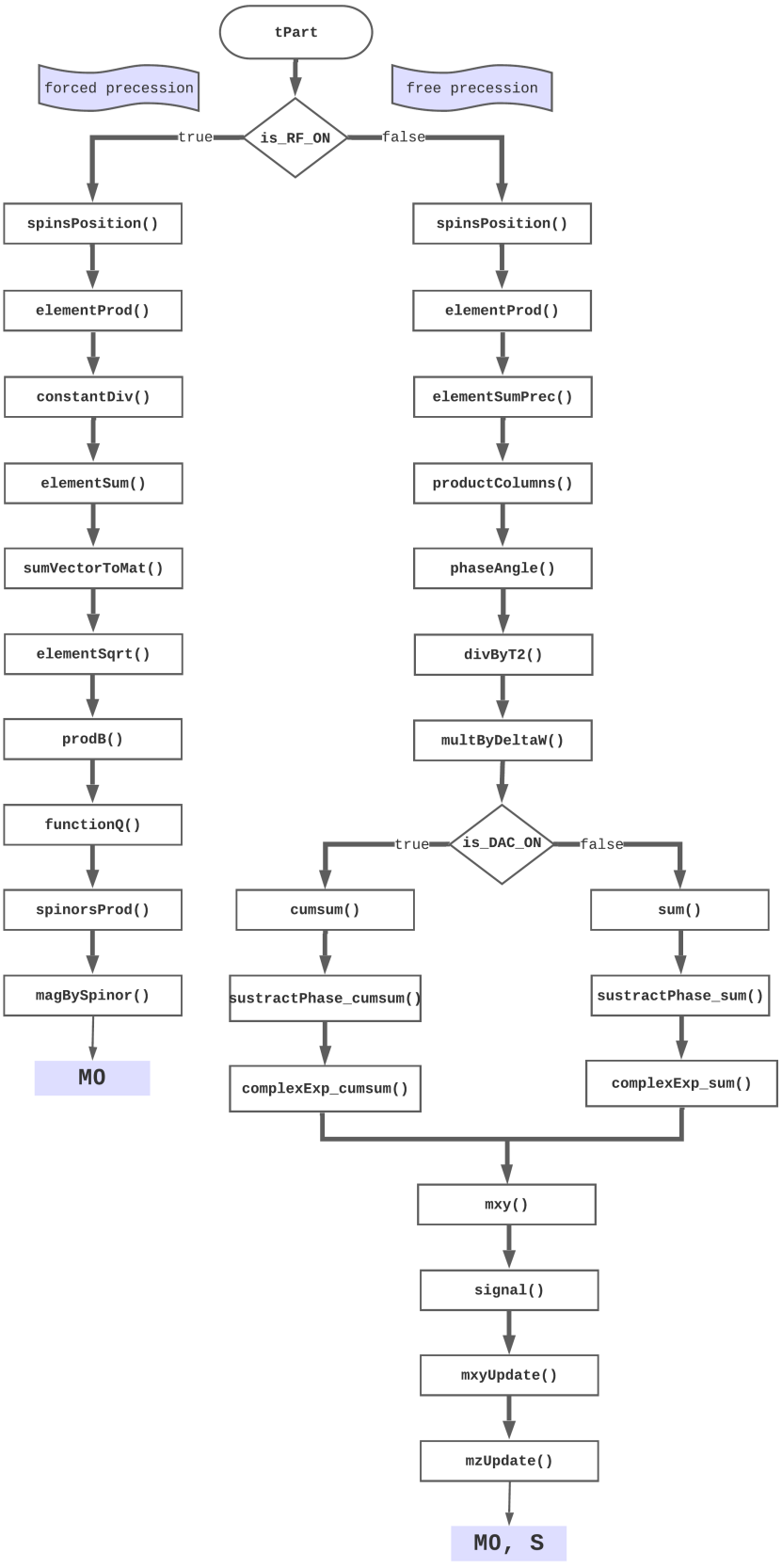


Figura 3.12: Diagrama de flujo de los *kernels* de la simulación en OpenCLIPER.

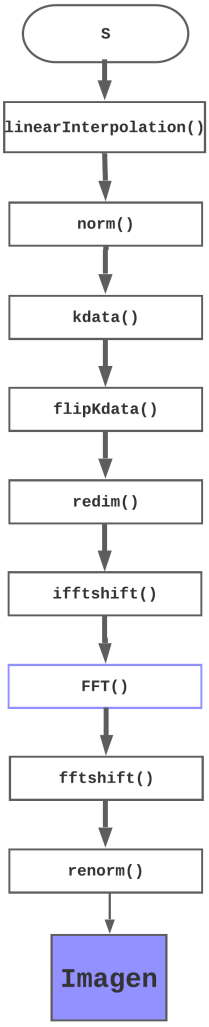


Figura 3.13: Diagrama de flujo de los *kernels* de la reconstrucción en OpenCLIPER.

Capítulo 4

Validación y evaluación de prestaciones

Este capítulo tiene como fin describir las fases de validación y evaluación del simulador desarrollado en OpenCLIPER. Ambas fases se llevan a cabo mediante la comparación del mismo con el simulador MRIsim.jl. Para ello, se ejecuta el proceso de simulación en ambos simuladores con una secuencia común y un mismo fantoma. La secuencia empleada corresponde con la generada por la función `EPI_base`, descrita en la *Sección 3.2*, mientras que el fantoma empleado es el generado por la función `heart_phantom`, descrito en la *Sección 3.1*.

4.1. Validación del simulador

El proceso de simulación que tiene lugar en ambos simuladores ofrece como resultado una imagen reconstruida correspondiente al fantoma de entrada. Esto permite que la validación del correcto funcionamiento del simulador se efectúe de forma visual, a través de la comparación de la imagen reconstruida por el simulador desarrollado en OpenCLIPER y por el simulador MRIsim.jl. Como se ha indicado antes, el fantoma empleado en la simulación es el generado por la función `heart_phantom` y la secuencia corresponde con la obtenida a través de la función `EPI_base`.

La imagen proporcionada por el simulador MRIsim.jl se muestra en la *Figura 4.1*, mientras que la imagen resultante del simulador en OpenCLIPER corresponde con la *Figura 4.2*.

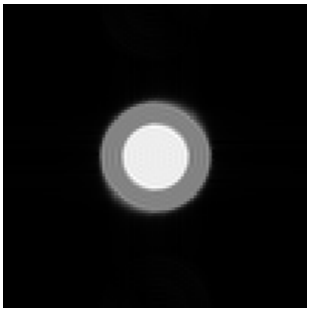


Figura 4.1: Imagen reconstruida por el simulador MRIsim.jl.

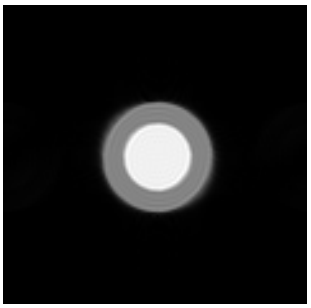


Figura 4.2: Imagen reconstruida por el simulador en OpenCLIPER.

La comparación visual de las dos imágenes anteriores permite intuir una similitud importante entre ambas. En la *Figura 4.3* se muestra la imagen resultante de calcular la diferencia entre ambas en valor absoluto.

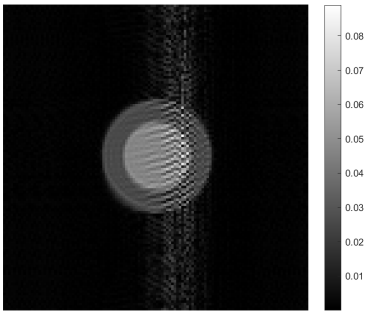


Figura 4.3: Diferencia entre las imágenes de ambos simuladores.

El rango de diferencias que muestra la escala de la imagen anterior es reducido en comparación con el rango dinámico de las dos imágenes iniciales, el cual oscila aproximadamente entre 1.8e-05 y 6.8. Entre las posibles causas de las diferencias entre las imágenes, destaca el hecho de que el simulador MRIsim.jl emplea precisión doble, mientras que en el simulador en OpenCLIPER la precisión es simple.

En definitiva, se pone de manifiesto la validez del proceso de simulación efectuado por el simulador desarrollado en OpenCLIPER, puesto que ambos simuladores proporcionan resultados prácticamente idénticos ante un mismo proceso de simulación.

4.2. Evaluación de prestaciones

La evaluación de los resultados del simulador desarrollado se efectúa a partir de la obtención de medidas de rendimiento en ambos simuladores. El fantoma empleado en el proceso de simulación (`heart_phantom`) consta de 12714 *spins*, a partir de los que se obtiene una primera señal que se interpola posteriormente en 10201 instantes de tiempo. La señal interpolada se redimensiona a una matriz de dimensiones 101×101 y, de forma adicional en el simulador en OpenCLIPER, esta matriz se redimensiona nuevamente mediante la introducción de ceros a

una matriz de 128×128. La justificación de seleccionar estas dimensiones se encuentra en que el proceso `FFT`, requerido para la obtención de la imagen final, presenta una mayor rapidez ante dimensiones que sean potencia de 2.

Las medidas de rendimiento se efectúan en términos de tiempos de ejecución con la media y la varianza como parámetros, a partir de 100 ejecuciones del proceso de simulación realizadas en tres dispositivos distintos. En los dos primeros dispositivos se ejecutan ambos simuladores, y en el tercero únicamente el desarrollado en OpenCLIPER, debido a que actualmente MRIsim.jl no está preparado para funcionar en una GPU AMD. En la *Tabla 4.1* se muestran los tiempos obtenidos para cada dispositivo, medidos en segundos.

		Media	Varianza
Nvidia Quadro RTX 6000	OpenCLIPER	0.789284	0.159327
	MRIsim.jl	4.634991	0.059775
Intel Core i7-4790	OpenCLIPER	2,545896	0,080647163
	MRIsim.jl	7,024032	0,086608
AMD Radeon RX 5700 XT	OpenCLIPER	2,701813	0,000027

Tabla 4.1: Tiempos de ejecución de los simuladores en distintos dispositivos (en segundos).

De acuerdo con la *Tabla 4.1*, se aprecia la importante ganancia en términos de tiempos de ejecución que ha supuesto la implementación del simulador en OpenCLIPER, así como la posibilidad de extender los dispositivos en que puede ejecutarse. Los valores reducidos de la varianza en todos los casos permiten determinar la fiabilidad de los resultados obtenidos.

Capítulo 5

Conclusiones y líneas futuras

Este capítulo recoge las principales conclusiones extraídas durante la elaboración de este Trabajo de Fin de Grado. Asimismo, se plantean las líneas futuras que surgen a partir del trabajo desarrollado.

5.1. Conclusiones

En este Trabajo de Fin de Grado se ha presentado la implementación de un simulador de MRI sobre el *framework* OpenCLIPER, empleando el simulador MRIsim.jl como base respecto a la simulación fidedigna del fenómeno MRI. Esto ha requerido el diseño de distintas clases asociadas a los conceptos propios de la MRI, a partir de las cuales poder desarrollar el proceso de simulación.

El objetivo perseguido ha consistido en reforzar las ventajas del simulador MRIsim.jl mediante OpenCLIPER, de forma que se obtuviera un simulador con rendimiento superior. El trabajo con OpenCLIPER para tal fin ha resultado complejo y ha requerido de un aprendizaje constante durante los meses de desarrollo del simulador. A pesar de esto, los resultados obtenidos ponen de manifiesto la consecución del objetivo planteado, puesto que se ha logrado superar al simulador de partida en tiempos de ejecución, a la vez que se mantiene su grado de realismo y se permite el funcionamiento sobre sobre dispositivos de cualquier fabricante, tanto CPU como GPU.

No obstante, la implementación actual presenta algunas limitaciones. Entre ellas, destaca el hecho de que únicamente se proporciona un fantoma con el que realizar el proceso de simulación y, en cuanto al proceso de reconstrucción, se ha contemplado únicamente la posibilidad de efectuarlo en caso de que la secuencia sea de tipo EPI.

En cualquier caso, se ha comprobado que las facilidades que presenta Julia en comparación con OpenCL en el desarrollo de *software*, se ven ensombrecidas en términos de rendimiento. A la finalización de este Trabajo de Fin de Grado se dispone una implementación inicial de un simulador de MRI sobre OpenCLIPER, el cual proporciona unos resultados satisfactorios y abre la posibilidad a futuras mejoras.

5.2. Líneas futuras

Las limitaciones de la implementación comentadas en la sección anterior abren la puerta a futuras líneas de trabajo que permitan ampliar y mejorar el funcionamiento del simulador. En ese sentido, queda abierta la posibilidad de ampliar el banco de pruebas con nuevos fantomas a partir de los cuales extender la experimentación con el simulador. Adicionalmente, el proceso de simulación debe contemplar la posibilidad de reconstruir una imagen con independencia del tipo de secuencia empleada.

En cuanto al rendimiento del simulador, los resultados alcanzados sugieren la posibilidad de futuras mejoras. Por un lado, la implementación actual no hace uso de memoria local, cuyo uso junto con otras técnicas de optimización podría dar lugar a una reducción de los tiempos de ejecución aún mayor. Por otro lado, en la simulación se emplean distintos *kernels* que realizan operaciones muy simples. Dado que la llamada a cada uno de los *kernels* requiere de un cierto tiempo, la reducción del número de *kernels* incrementando la complejidad de las operaciones realizadas por los mismos podría resultar también en una reducción de los tiempos de ejecución.

Finalmente, surge también como objetivo futuro la integración de la implementación desarrollada en OpenCLIPER junto con una interfaz gráfica que permita un uso interactivo del simulador.

Bibliografía

[1] Z.-P. Liang y P. C. Lauterbur, *Principles of magnetic resonance imaging*. SPIE Optical Engineering Press Bellingham, 2000.

[2] C. G. Xanthis, I. E. Venetis, A. V. Chalkias, y A. H. Aletras, "MRISIMUL: A GPU-Based Parallel Approach to MRI Simulations," *IEEE Transactions on Medical Imaging*, vol. 33, no. 3, 2014.

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

[4] C. Castillo-Passi y P. Irarrazaval. (2021, Sep.) cncastillo/komamri.jl. Disponible en: https://doi.org/10.5281/zenodo.6627503

[5] J. Bezanson, A. Edelman, S. Karpinski, y V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[6] D. Treceño-Fernández, J. Calabia-del Campo, M. L. Bote-Lorenzo, E. Gómez-Sánchez, R. Luis-García, y C. Alberola-López, "A web-based educational magnetic resonance simulator: Design, implementation and testing," *Journal of Medical Systems*, vol. 44, 2019.

[7] D. Treceño-Fernández, J. Calabia-del Campo, M. L. Bote-Lorenzo, E. Gómez-Sánchez, R. Luis-García, y C. Alberola-López, "Integration of an intelligent tutoring system in a magnetic resonance simulator for education: Technical feasibility and user experience," *Computer Methods and Programs in Biomedicine*, vol. 195, 2020.

[8] F. Simmross-Wattenberg, M. Rodríguez-Cayetano, E. Moya-Sáez, M. Martín-Fernández, y C. Alberola-López, "OpenCLIPER: An OpenCL-based C++ framework for overhead-reduced medical image processing and reconstruction on heterogeneous devices," *IEEE journal of biomedical and health informatics*, vol. 23, pp. 1702–1709, 2019.

[9] J. E. Stone, D. Gohara, y G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[10] KDevelop - A cross-platform IDE for C, C++, Python, QML/JavaScript and PHP. Disponible en: https://www.kdevelop.org/

[11] Atom. Disponible en: https://atom.io/

[12] Overleaf. Disponible en: https://www.overleaf.com/

[13] The LaTeX Proyect. Disponible en: https://www.latex-project.org/

[14] MATLAB & Simulink - MathWorks. Disponible en: https://www.mathworks.com

[15] D. W. McRobbie, E. A. Moore, M. J. Graves, y M. R. Prince, *MRI from Picture to Proton*. Cambridge University Press, 2017.

[16] R. A. Pooley, "Fundamental physics of MR imaging," *Radiographics*, vol. 25, no. 4, pp. 1087–1099, 2005.

[17] P. Suetens, *Fundamentals of medical imaging*. Cambridge University Press, 2017.

[18] A. Pang, C. Anslow, y J. Noble, "What programming languages do developers use? A theory of static vs dynamic language choice," en *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2018, pp. 239–247.

[19] C. Castillo-Passi, R. Coronado, S. Uribe, y P. Irarrazaval, "MRIsim. jl: A framework for end-to-end spin-level MRI simulations with GPU acceleration."

[20] C. Castillo-Passi, "MRIsim.jl: Toolbox for spin-level simulations. Project presentation," 2020.

[21] C. Alberola López, "An overview of MRIsim.jl," 2022.