ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

CONSTRUCCIÓN DE UN EDITOR Y SIMULADOR DE SECUENCIAS DE IMAGEN DE RESONANCIA MAGNÉTICA CON Qt

AUTOR:
Pablo Villacorta Aylagas

TUTORES:
Carlos Alberola López y Manuel Rodríguez Cayetano

Título:	**Construcción de un Editor y Simulador de Secuencias de Imagen de Resonancia Magnética con Qt**
Autor:	**Pablo Villacorta Aylagas**
Tutores:	**Carlos Alberola López y Manuel Rodríguez Cayetano**
Departamento:	**TSCeIT**

Tribunal

Presidente:	**Carlos Alberola López**
Secretario:	**Federico Simmross Wattenberg**
Vocal:	**Manuel Rodríguez Cayetano**
Suplente:	**Santiago Aja Fernández**
Suplente:	**Pablo Casaseca de la Higuera**

Fecha:	
Calificación:	

Abstract

This Batchelor thesis is focused on the improvement of a software tool to simulate the magnetic resonance image (MRI) acquisition and reconstruction process, by adding a graphical user interface (GUI) that incoporates a sequence editor as well as advanced graphical capabilities to guide the user.

The application idea arises from the analysis of an existing MRI simulation tool, `KomaMRI.jl`, developed in the Julia programming language. The tool has enormous potential in terms of simulation methods, but lacks a GUI that eases sequence creation and observation of the slice selection gradient effects. With this motivation, the main objective of this project is set: to design a GUI to make the composition of MRI sequences easier for the user.

The application has been developed from the combination of C++/Qt, QML and Julia, and allows creating arbitrary MRI sequences by arranging and grouping blocks. These blocks implement the basic sequence elements, which are essentially gradients and radiofrecuency electromagnetic pulses. A tool has also been integrated in order to visualize the slice selected by the gradients in three dimensions.

As results, both the images generated by the simulation and the three-dimensional models provided by the 3D visualization tool have been verified. In addition, both the development and usage of the tool give rise to new future work perspectives, suach as the creation of a web application and its integration with a simulator version developed in OpenCL by members of the same research group in which this project has been carried out.

Keywords

MRI, sequence, Julia, EPI, Graphical user interface (GUI).

Resumen

En este Trabajo de Fin de Grado se plantea la mejora de una herramienta software de simulación del proceso de adquisición de imágenes por resonancia magnética (MRI) mediante la adición de una interfaz gráfica de usuario (GUI) que disponga de un editor de secuencias, así como de capacidades gráficas avanzadas para que sirvan de guía al usuario.

La idea de esta aplicación surge del estudio de una herramienta de simulación de MRI ya existente, `KomaMRI.jl`, desarrollada en el lenguaje de programación Julia. La herramienta cuenta con un enorme potencial en cuanto a metodología de simulación se refiere, pero carece de una GUI que facilite la creación de secuencias y la observación del efecto de los gradientes de selección de *slice*. Con esta motivación se fija el objetivo principal de este proyecto: diseñar una GUI para que la composición de secuencias de MRI resulte más sencilla de cara al usuario.

Se ha desarrollado una aplicación a partir de la combinación de los lenguajes C++/Qt, QML y Julia, que permite crear secuencias de MRI arbitrarias mediante la concatenación y agrupación de bloques. Estos bloques implementan los elementos básicos de una secuencia, que son, esencialmente, gradientes y pulsos electromagnéticos de radiofrecuencia. Asimismo, se ha integrado una herramienta que permite visualizar en tres dimensiones el corte seleccionado por los gradientes.

Como resultados, se han verificado tanto las imágenes generadas por la simulación como los modelos tridimensionales proporcionados por la herramienta de visualización 3D. Por otra parte, tanto el desarrollo como el uso de la herramienta nos han proporcionado nuevas perspectivas de trabajo futuro, entre las que se plantea la creación de una aplicación web y su integración con una versión del simulador desarrollada en OpenCL por miembros del mismo grupo de investigación en el que se ha realizado este proyecto.

Palabras Clave

MRI, secuencia, Julia, EPI, Interfaz gráfica de usuario (GUI).

Agradecimientos

En primer lugar, agradecer a mis tutores Carlos Alberola López y Manuel Rodríguez Cayetano por darme la oportunidad de realizar este Trabajo de Fin de Grado y brindarme apoyo constante durante su realización. Así mismo, merece especial mención mi tutor de prácticas, Federico Simmross Wattenberg, quien me ha ayudado, de igual manera, durante todo el desarrollo del proyecto.

Quiero dar las gracias a mis compañeros de laboratorio, por su amabilidad y predisposición a ayudarme desde el primer momento.

También han sido fundamentales mis amigos, que me han servido como soporte y vía de escape cuando más lo he necesitado.

Por último, agradecerle todo a mi familia, especialmente a mis padres, a mi hermana y a mis abuelos, ya que sin su apoyo incondicional no habría sido capaz de finalizar este Trabajo.

Índice general

1. Introducción **1**
- 1.1. Motivación . . . 1
- 1.2. Objetivos . . . 1
- 1.3. Fases y Métodos . . . 2
- 1.4. Medios . . . 2
- 1.5. Estructura del documento . . . 3

2. Estado del Arte **4**
- 2.1. Introducción . . . 4
- 2.2. Fundamentos de la MRI . . . 4
  - 2.2.1. Fundamentos físicos . . . 4
  - 2.2.2. Formación y adquisición de la imagen de resonancia magnética . . . 11
  - 2.2.3. Secuencias de MRI . . . 16
- 2.3. Fundamentos del lenguaje Julia . . . 19
  - 2.3.1. Características de Julia . . . 19
  - 2.3.2. Métodos de interacción con Julia . . . 20
  - 2.3.3. Módulos . . . 20
  - 2.3.4. Carga de código en Julia . . . 20
  - 2.3.5. Entornos y Proyectos . . . 21
  - 2.3.6. Integración de Julia en C/C++ . . . 21
- 2.4. Simulación de MRI . . . 22
  - 2.4.1. Herramientas de simulación de MRI en la actualidad . . . 22
  - 2.4.2. Análisis del simulador `KomaMRI.jl` . . . 22
- 2.5. Desarrollo de aplicaciones gráficas . . . 27
  - 2.5.1. El marco Qt . . . 27
  - 2.5.2. QML . . . 29
  - 2.5.3. Compilación de aplicaciones basadas en el entorno Qt . . . 30
  - 2.5.4. El estándar WebAssembly . . . 31
- 2.6. Visualización de volúmenes y cortes en 3D . . . 32

3. Análisis, Diseño e Implementación **34**
- 3.1. Análisis de los requisitos de la aplicación . . . 34
  - 3.1.1. Requisitos funcionales . . . 34
  - 3.1.2. Requisitos no funcionales . . . 35
- 3.2. Diseño de la aplicación . . . 35
  - 3.2.1. Elección de las tecnologías empleadas . . . 35
  - 3.2.2. Descripción general de funcionamiento . . . 36
  - 3.2.3. Estructura de la secuencia . . . 37
  - 3.2.4. Definición de los parámetros de la simulación . . . 39
  - 3.2.5. Diseño de la interfaz gráfica . . . 40
- 3.3. Implementación . . . 41
  - 3.3.1. Planteamiento del modelo de datos en QML . . . 41
  - 3.3.2. Pasarelas entre lenguajes . . . 43
  - 3.3.3. *Slots* de la clase `Backend` . . . 45

3.3.4. Funciones de `QtSlots.jl` . . . 47
3.4. Generación del `Phantom` a partir de un fichero NIfTI . . . 50

4. Resultados **51**
4.1. Apariencia de la interfaz y funcionamiento . . . 51
4.2. Resultados de la simulación . . . 52
4.2.1. Simulación de una secuencia GE-EPI *single-shot* . . . 52
4.2.2. Simulación de una secuencia FSE . . . 53
4.3. Visualización 3D . . . 54

5. Conclusiones y líneas futuras **56**
5.1. Conclusiones . . . 56
5.2. Aportaciones . . . 56
5.3. Líneas futuras de trabajo . . . 57

Apéndices **58**
A. Rotación de secuencias . . . 59
A.1. Direcciones de codificación de fase y de frecuencia en secuencias rotadas . . . 59
A.2. Cálculo de la matriz de rotación . . . 60
B. Funciones utilizadas en QML . . . 61
B.1. Función `createScanner` . . . 61
B.2. Función `createSeq` . . . 61
B.3. Función `saveSeq` . . . 62
C. Implementación de *slots* en C++ . . . 63
C.1. Función `plot3D` . . . 63
C.2. Función `simulate` . . . 63
C.3. Función `saveSeq` . . . 64
C.4. Función `displaySeq` . . . 66
D. Fichero `QtSlots.jl` . . . 67
D.1. Función `plot3D` . . . 67
D.2. Función `sim` . . . 68
D.3. Función `displaySeq` . . . 70
E. Función `nii_brain_phantom3D` . . . 71
F. Versión Web: Diseño . . . 73

Bibliografía **76**

Índice de figuras

2.1. Protón de hidrógeno en rotación. El protón presenta un movimiento de rotación en torno a un eje, y actúa de forma similar a la de un imán, con polos norte y sur. 5
2.2. Influencia del campo magnético $\boldsymbol{B}_0$ en el alineamiento de los espines. 6
2.3. Ley de Ampére aplicada a dos geometrías de conductor [9] 6
2.4. Movimiento de precesión de un espín sometido a un campo magnético estático. 7
2.5. Efecto producido en el vector magnetización neta por la aplicación de un pulso RF [12]. 9
2.6. Comportamiento del vector magnetización tras desactivar el pulso de 90^{o}. $\boldsymbol{M}$ comienza a inclinarse hacia $\boldsymbol{B}_0$ mientras precesa a la frecuencia de Larmor [12] 9
2.7. Relajación T1, longitudinal o espín-red 10
2.8. Relajación T2, transversal o espín-espín 10
2.9. Detección de la señal MR mediante una bobina [12] 11
2.10. Representación de la configuración del conjunto de bobinas de gradiente magnético de MRI [13] 12
2.11. Selección de slice a lo largo de z 13
2.12. Codificación espacial mediante gradientes de codificación de fase y de frecuencia [12] . . 14
2.13. (a) *Raw data* en el espacio K. (b) Imagen correspondiente [14] 15
2.14. Diagrama temporal de una secuencia *spin echo* típica [12] 17
2.15. Efecto producido por el pulso re-orientador de 180^{o} (a) en la magnetización, (b) en el tiempo de relajación transversal 17
2.16. Diagrama temporal de una secuencia *gradient echo* típica [12] 18
2.17. Diagrama temporal de una secuencia GE-EPI y trayectoria producida en el espacio K . 19
2.18. Esquema de funcionamiento general del simulador `KomaMRI.jl`[3] 22
2.19. Representación gráfica de (a) una instancia de `Grad` y (b) un *array* bidimensional formado por instancias de esta misma estructura 23
2.20. Representación gráfica de (a) una instancia de `RF` y (b) un *array* bidimensional formado por instancias de esta misma estructura 24
2.21. Representación gráfica de (a) una instancia de `ADC` y (b) un *array* unidimensional formado por instancias de esta misma estructura 24
2.22. Diagrama en el que se muestra la conexión de varios objetos en Qt (derivados de la clase `QObject`) mediante señales y *slots* [26] 27
2.23. Fichero main.cpp 28
2.24. (a) Fichero QML y (b) su correspondiente resultado 29
2.25. Implementación de un botón con QML 30
2.26. Relación entre los elementos que permiten la visualización de datos en QML [26] 31
2.27. Renderizado 3D mediante (a) *Surface Rendering* y (b) *Volume Rendering* (obtenido a partir de tomografía computarizada (CT)) [31] 33

3.1. Diagrama de funcionamiento del proyecto a realizar 37
3.2. Boceto de la interfaz gráfica de la aplicación, en el que están resaltados cada uno de los paneles 40
3.3. Ejemplo de estructura de bloques que conforman una secuencia 42
3.4. Código QML correspondiente a dos instancias de `ListElement` pertenecientes a una secuencia. (a): bloque de excitación. (b): bloque de tipo grupo con 4 bloques "hijos" (ver cuadro 3.1). 43
3.5. Esquema de conexiones entre los lenguajes que componen la aplicación 44

3.6. Esquema gráfico del gradiente selectivo 49

4.1. Apariencia final de la interfaz gráfica tras la simulación de una secuencia EPI 51
4.2. Salida de la aplicación durante una simulación, mostrada en la consola de Qt Creator . 52
4.3. (a) Secuencia GE-EPI creada mediante el bloque `EPI` y (b) parámetros de configuración del bloque 52
4.4. (a) Imagen MRI cerebral de 101x101 píxeles obtenida a partir de una secuencia GE-EPI. (b) Espacio K correspondiente 53
4.5. Jerarquía de bloques para una secuencia *Fast Spin Echo* (FSE) 53
4.6. Adquisición de 6 líneas del espacio K mediante una secuencia FSE con ETL= 3 54
4.7. A la izquierda, la representación 3D del *slice* seleccionado cuando se activa un gradiente en z. A la derecha, la imagen reconstruida correspondiente 54
4.8. A la izquierda, la representación 3D del *slice* seleccionado cuando se activan gradientes en y y z. A la derecha, la imagen reconstruida correspondiente 55

F.1. Diagrama de funcionamiento de la aplicación web. Las flechas naranjas representan los comandos del protocolo establecido entre el cliente y el servidor (HTTP), mientras que las flechas verdes corresponden a las llamadas a las funciones de Julia 74

Índice de cuadros

3.1. Correspondencia entre el tipo de bloque y la propiedad `cod` en la lista de QML 41
3.2. Líneas de código utilizadas para cargar el fichero QML desde C++ 44
3.3. Líneas de código utilizadas para hacer que el objeto `backend` esté disponible en QML . 44
3.4. Código C++ utilizado para inicializar Julia y el módulo `KomaMRI` 45

Siglas

ADC	*A*nalog-to-Digital Converter.
API	*Application Programming Interface*.
BW	*Bandwidth*.
CPU	*Central Processing Unit*.
EM	*Electromagnetic Energy*.
EPI	*Echo Planar Imaging*.
ETL	*Echo Train Length*.
FEG	*Frecuency Encoding Gradient*.
FID	*Free Induction Decay*.
FOV	*Field Of View*.
FRQ	*Functional Requirement*.
FSE	*Fast Spin Echo*.
GE	*Gradient Echo*.
GPU	*Graphics Processing Unit*.
GUI	*Graphical User Interface*.
IR	*Inversion Recovery*.
LPI	Laboratiorio de Procesado de Imagen.
MIT	*Massachusetts Institute of Technology*.
MR	*Magnetic Resonance*.
MRI	*Magnetic Resonance Image*.
NEX	*Number of Excitations*.
NFR	*Non-Functional Requirement*.
NIfTI	*Neuroimaging Informatics Technology Initiative*.
NMR	*Nuclear Magnetic Resonance*.
PEG	*Phase Encoding Gradient*.
QML	*Qt Meta Language*.
RAM	*Random Access Memory*.
REPL	*Real-Eval-Print Loop*.
RF	Radiofrecuencia.

SE	*Spin Echo*.
SERAM	Sociedad Española de Radiología Médica.
SNR	*Signal to Noise Ratio*.
SSG	*Slice Select Gradient*.
TE	Tiempo de eco.
TFG	Trabajo Fin de Grado.
TI	Tiempo de inversión.
TR	Tiempo de repetición.
VNC	*Virtual Network Computing*.

Capítulo 1

Introducción

1.1. Motivación

La imagen por resonancia magnética (MRI, por sus siglas en inglés) es una técnica de imagen basada en el fenómeno de la resonancia magnética nuclear (NMR) y proporciona un medio para obtener imágenes detalladas y de alto contraste de los tejidos y órganos del cuerpo, que son utilizadas por la comunidad médica con fines de diagnóstico. Esta técnica se diferencia de otras tecnologías de imagen como la radiografía, en que en este caso se utiliza radiación no ionizante, basada en campos magnéticos estáticos y espacialmente variantes y en la aplicación de energía electromagnética (EM) [1].

Es una herramienta muy versátil, lo que la convierte en la opción más adecuada en muchas situaciones clínicas. Sin embargo, no es tarea fácil operar con un sistema de MRI y son muchos los factores que contribuyen a estas dificultades [2]. Estas trabas, sumadas al alto coste de un sistema real de MRI, hacen que la tarea de aprendizaje sobre esta tecnología se vuelva poco accesible.

Con esta primera motivación surgió la idea de los simuladores de imagen de resonancia magnética, capaces de facilitar una mejor comprensión de los elementos involucrados en el proceso, además de permitir un número ilimitado de usos. Desde el surgimiento de esta idea, no han sido pocos los simuladores que han ido apareciendo, por lo que cabe cuestionar la necesidad de otro simulador más. Aquí es donde entra en juego un simulador ya existente (`KomaMRI.jl`) que constituirá un extraordinario punto de arranque para este proyecto.

`KomaMRI.jl` [3] (antes MRIsim.jl) es un paquete de Julia [4] pensado para simular, de forma sencilla, escenarios generales de MRI. Cuenta con una interfaz gráfica que permite visualizar los resultados (imagen, espacio K, señal en "crudo", diagrama de secuencia, etc.) de simulaciones realizadas a partir de secuencias[1] preprogramadas.

No obstante, la herramienta carece de un editor de secuencias propiamente dicho, por lo que las secuencias de entrada al simulador se generan mediante la ejecución de ficheros en Julia (extensión .jl) o mediante la carga de ficheros de secuencia (.seq) ya existentes. Esta, por tanto, es la principal motivación para la realización de este proyecto de fin de Grado.

1.2. Objetivos

El objetivo principal de este Trabajo de Fin de Grado es la creación de un editor de secuencias de resonancia magnética que resulte sencillo de usar, que sea versátil y cuyo uso sea compatible con fines tanto docentes como investigadores.

Como objetivos secundarios, perseguimos:

1. El diseño de una interfaz gráfica (GUI) para que la composición de secuencias de MRI resulte más sencilla de cara al usuario,

2. La creación de "pasarelas", *wrappers*, entre los distintos lenguajes a emplear (C++/Qt, QML, Julia), de forma que sea posible la comunicación entre ellos.

[1] En la sección 2.2.3 se explicará en detalle el concepto de secuencia de MRI.

3. Dotar al simulador con la funcionalidad que permita visualizar, en tres dimensiones, el corte excitado en la simulación.

1.3. Fases y Métodos

Las fases que se han seguido para la consecución de los objetivos marcados han sido las siguientes:

1. **Fase de formación**: esta fase no suele incluirse en el ciclo de vida de un proyecto software, pero en este caso es completamente necesaria, ya que se requiere adquirir determinados conocimientos previos antes de entrar en el proyecto como tal. Estos son:
   - Fundamentos básicos de resonancia magnética, tanto los principios físicos que permiten generar las señales, como las técnicas de obtención de imágenes a partir de estas.
   - Repaso de fundamentos de programación, ya trabajados anteriormente en asignaturas previas del grado. No obstante, ninguno de los lenguajes utilizados en este proyecto se ha estudiado explícitamente en la titulación, por lo que será de especial importancia la familiarización con ellos. Estos lenguajes son:
     - Julia [4]
     - C++/Qt
     - QML
   - Adicionalmente, para agilizar el proceso de desarrollo, será también de gran ayuda la familiarización con el sistema de control de versiones Git, así como el repaso de conceptos básicos de administración de redes y sistemas en Linux.

   Las siguientes etapas corresponden a las fases generales de desarrollo de un proyecto software, particularizadas para el caso que nos ocupa:
2. **Análisis y especificación de los requisitos**: se definirán de forma detallada los requisitos funcionales y no funcionales.
3. **Diseño**: se plantean el diseño conceptual de la aplicación, el bocetaje de la interfaz gráfica, así como la elección de las tecnologías a utilizar.
4. **Implementación**: ejecución de las tareas de codificación que sigan el diseño que se habrá realizado al término de la etapa anterior.
5. **Pruebas generales y medidas de rendimiento**: se realizarán pruebas con distintas secuencias y se comparará la salida de la aplicación con la del simulador propiamente dicho.
6. **Mantenimiento**: la fase de mantenimiento, habitual en el desarrollo software, no forma parte de este proyecto, salvo cuestiones evidentes que surjan durante el período de desempeño del trabajo.

1.4. Medios

Para la realización de este Trabajo de Fin de Grado se ha hecho uso de los siguientes medios:

Hardware

- Ordenador de sobremesa perteneciente al Laboratorio de Procesado de Imagen (LPI), con las siguientes características:
  - CPU: AMD Ryzen 7 5800X 8-Core 3.8 GHz
  - GPU: NVIDIA Quadro RTX 4000 8GB
  - RAM: 32 GB
- Acceso remoto a los servidores disponibles en el Laboratorio de Procesado de Imagen. Este acceso es posible tanto por ssh como a través de un cliente VNC. Además, el sistema de ficheros del ordenador utilizado reside en estos servidores.

Software

- Sistema operativo Ubuntu 20.04.1 LTS
- Qt Creator [5]: entorno de desarrollo (IDE) multiplataforma orientado a la programación de aplicaciones con Qt [5].
- Repositorio Git en el que se almacenarán tanto el proyecto principal como todos los proyectos fruto del aprendizaje de cada tecnología.

Datos

En cuanto a los datos de partida para la simulación, el Laboratorio de Procesado de Imagen cuenta con modelos anatómicos cerebrales y del miocardio de alta calidad. Estos modelos serán usados como entrada al simulador.

1.5. Estructura del documento

El presente documento se divide en cinco capítulos, comenzando por la Introducción. El segundo de ellos va asociado a la primera de las fases del proyecto y se centra en el estudio de los conceptos previos necesarios para su desarrollo. Concretamente, se exponen los fundamentos de la técnica de imagen por resonancia magnética y se presentan algunas de las herramientas utilizadas en la actualidad para simularla. Entre ellas se encuentra el simulador utilizado en este proyecto, cuyo funcionamiento es también analizado. Adicionalmente, se introducen dos lenguajes de programación: Julia, utilizado en el simulador, y Qt, con el que se desarrollará la interfaz gráfica. Por último, se presenta el concepto de *rendering* 3D y se mencionan dos herramientas útiles para llevarlo a cabo.

Respecto al tercer capítulo, comienza con un análisis de los requisitos, tanto funcionales como no funcionales, que debe cumplir la aplicación desarrollada. Después, dentro del diseño del sistema, se justifican las decisiones tomadas en cuanto a la elección de las tecnologías y algunas decisiones adicionales relacionadas con la estructura del programa, la definición de parámetros y la apariencia visual de la aplicación. En cuanto a la implementación, se describe el uso concreto de las tecnologías escogidas, el modelo de datos utilizado y se hace un análisis de las funciones y métodos empleados para conectar lenguajes e implementar todas las funcionalidades.

En el capítulo 4 se muestran los resultados obtenidos. En concreto, se visualizará la interfaz, su apariencia final y las imágenes resultado de la simulación. Se realizarán pruebas con diferentes secuencias y se compararán los resultados generados por el simulador cuando es llamado a través de nuestra interfaz con los que genera el simulador mediante la programación de los *scripts* de llamada correspondientes. Adicionalmente, se muestra el modelo tridimensional generado a partir de la secuencia creada.

El capítulo 5 recoge las principales conclusiones extraídas del proyecto y las líneas futuras de trabajo que podrían ser seguidas durante las próximas etapas académicas.

Por último, en los apéndices del documento se recogen algunos fragmentos de código utilizados en el proyecto y algunas operaciones, como la de rotación de secuencias (apéndice A), útiles para su comprensión. Adicionalmente, en uno de estos apéndices se expone la introducción al diseño de una versión web de la aplicación.

Capítulo 2

Estado del Arte

2.1. Introducción

Para poder comprender lo realizado durante la ejecución del proyecto, así como el porqué de las decisiones tomadas durante su desarrollo, es necesario entender cada una de las tecnologías empleadas. Así mismo, para justificar las motivaciones expuestas en la introducción de este documento, hace falta conocer los últimos avances producidos en este ámbito, las tecnologías más usadas en la actualidad y la razón de su utilización.

En este capítulo describiremos estas tecnologías. En primer lugar, se abordarán los aspectos fundamentales en el estudio de la resonancia magnética. A continuación, se hará una pequeña introducción al lenguaje de programación empleado para el desarrollo del simulador propiamente dicho (Julia), así como a la herramienta de simulación que hace uso de este lenguaje (`KomaMRI.jl`). El capítulo continuará con una explicación de avances y tecnologías utilizadas en el desarrollo de interfaces de usuario, concretando en los lenguajes utilizados en la realización de este proyecto. Por último, se hará una mención a alguna de las tecnologías utilizadas para la visualización de volúmenes en tres dimensiones.

2.2. Fundamentos de la MRI

Una de las piezas clave en el desarrollo de la aplicación, más concretamente en la fase de formación, es la de la familiarización con los principios físicos de la resonancia magnética. Al estar trabajando con un simulador, es necesario conocer los procesos, tanto de formación de señales de NMR, como de adquisición de imágenes a partir de estas señales, para así poder aplicar estos conceptos a la herramienta software que usaremos. Cabe destacar que, aunque partamos de un simulador ya desarrollado (por lo que no es nuestra tarea desarrollar los algoritmos que simulen los procesos físicos), será de gran ayuda tener los conceptos claros, para así diseñar la interfaz de la forma más lógica posible y poder pasar al simulador unos parámetros adecuados.

2.2.1. Fundamentos físicos

Espines y momentos magnéticos nucleares

Los núcleos atómicos que presentan número atómico impar poseen un momento angular $\boldsymbol{J}$ conocido como espín. En el modelo vectorial clásico, esta magnitud se define como un vector que representa la rotación de los núcleos atómicos alrededor de un determinado eje [6]. Estos núcleos cargados en rotación generarán un campo magnético a su alrededor, conocido como momento magnético $\boldsymbol{\mu}$. La relación entre este momento magnético y el espín viene dada por la ecuación

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

Siendo γ la denominada constante giromagnética, una constante física característica de cada tipo de núcleo. En nuestro caso, nos interesan especialmente los núcleos de hidrógeno (^{1}H), compuestos por un único protón, y presentes en abundancia en forma de H_2O en los tejidos biológicos. Los núcleos

de hidrógeno presentan una constante giromagnética $\gamma = 42{,}58$ MHz/T. La figura 2.1 muestra una interpretación de lo que podría ser un núcleo de hidrógeno en rotación.

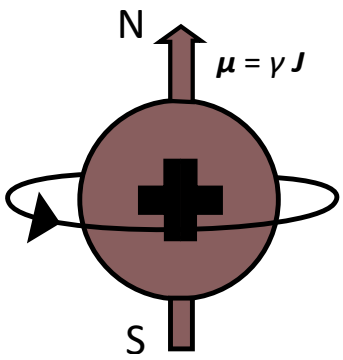


Figura 2.1: Protón de hidrógeno en rotación. El protón presenta un movimiento de rotación en torno a un eje, y actúa de forma similar a la de un imán, con polos norte y sur.

Como cualquier otro vector, el momento magnético está formado por un módulo y una dirección:

$$\boldsymbol{\mu} = \mu \hat{u} \tag{2.2}$$

Sabemos que el módulo es dependiente del denominado número cuántico del espín, I, característico de cada tipo de núcleo, y que debe ser no nulo para que se pueda producir resonancia. En el caso de los núcleos de hidrógeno, $I = \frac{1}{2}$.

Por otra parte, la dirección $\hat{u}$ del momento magnético es, a priori, aleatoria, ya que en ausencia de cualquier campo magnético externo, esta dirección solo dependerá del movimiento térmico aleatorio de los núcleos [7]. Por lo tanto, cuando nos encontremos ante un tejido compuesto principalmente por núcleos de hidrógeno, cada uno de sus momentos magnéticos apuntará en una dirección distinta. Al ser estas direcciones aleatorias (y podemos suponer equiprobables), se producirán cancelaciones mutuas entre los espines, y el campo magnético total del sistema (conocido como magnetización neta $\boldsymbol{M}$) será nulo.

Sin embargo, bajo la aplicación de un campo magnético estático $\boldsymbol{B}_0$), los núcleos de hidrógeno tienden a alinearse con la dirección de este campo, pudiendo adoptar, cada uno de los espines, dos posibles orientaciones:

- Paralela: misma dirección y mismo sentido que $\boldsymbol{B}_0$ (por convención, será la dirección $\hat{z}$).
- Antiparalela: misma dirección pero distinto sentido que $\boldsymbol{B}_0$ $(-\hat{z})$.

Aunque no todos los núcleos se alineen y algunos de ellos sigan cancelándose mutuamente, existirá un pequeño exceso de protones que se alinearán con el campo magnético, produciendo una magnetización neta no nula y paralela a ese campo [8]. Esto se debe a que, de las dos posibles orientaciones, la orientación paralela supone una menor energía que la antiparalela, por lo que habrá un mayor número de núcleos rotando con esta orientación. La figura 2.2 muestra esta influencia del campo $\boldsymbol{B}_0$ en el sistema de espines. Cabe destacar que, para que el vector magnetización neta apunte en la dirección del campo, no es necesario que cada uno de los núcleos esté orientado en esa dirección; basta con que la contribución de cada protón al campo magnético total tenga más peso en la dirección paralela al campo. De esta forma, las componentes en el resto de direcciones se seguirán cancelando y solo quedará una componente neta apuntando en dirección paralela a $\boldsymbol{B}_0$.

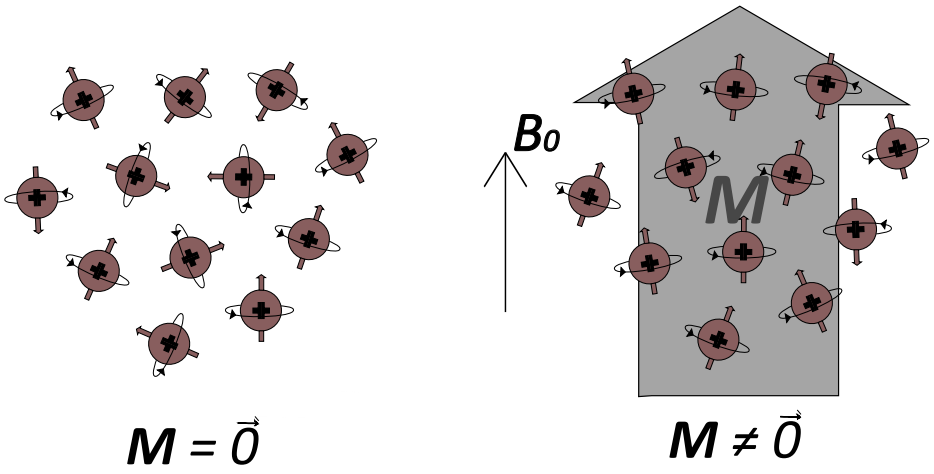
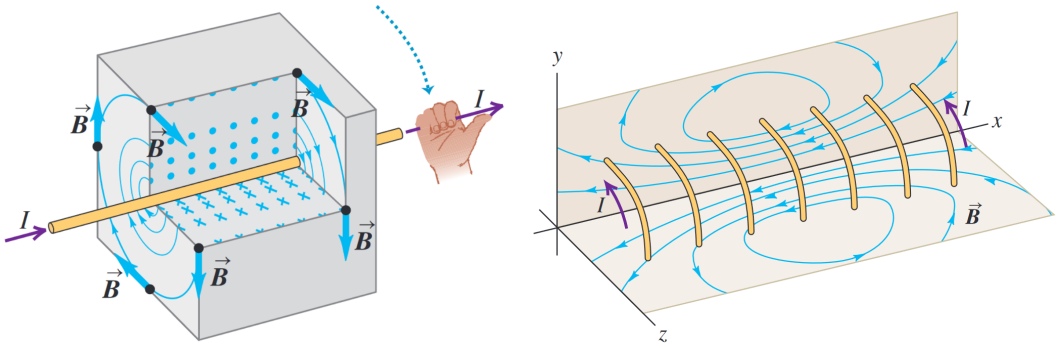


Figura 2.2: Influencia del campo magnético $\boldsymbol{B}_0$ en el alineamiento de los espines.

Para entender cómo se produce el campo magnético constante $\boldsymbol{B}_0$ utilizado, es necesario conocer la ley de Ampère, que establece que la circulación del campo magnético a lo largo de una línea cerrada es proporcional a la suma algebraica de las intensidades de las corrientes que atraviesan la superficie delimitada por la línea cerrada:

$$\oint \boldsymbol{B} dl = \mu_0 \sum I \tag{2.3}$$

Esto, llevado al caso de un hilo conductor infinito, produce unas líneas de campo magnético con forma circular, concéntricas con respecto al conductor (ver figura 2.3a). Como en nuestro caso, lo que buscamos son líneas de campo rectas (al menos en la región de interés), deberemos generar el campo magnético, no con un conductor rectilíneo, sino con un solenoide. El campo magnético en su interior cumplirá nuestros requisitos. La figura 2.3b muestra la idea básica del funcionamiento del solenoide, en cuyo interior se colocaría al paciente.



(a) Campo magnético generado por un conductor rectilíneo

(b) Campo magnético generado por un solenoide

Figura 2.3: Ley de Ampére aplicada a dos geometrías de conductor [9]

Precesión de los espines

Además del movimiento de rotación previamente explicado, los espines presentan un segundo tipo de movimiento: el de precesión. Para comprenderlo de forma sencilla, podemos imaginarnos un espín como si fuese una peonza. Una peonza gira sobre su eje pero, por otra parte, la fuerza de la gravedad intenta tirarla para que caiga hacia abajo. Los efectos combinados de la gravedad y el movimiento de

giro hacen que la peonza precesione [8]. De forma análoga ocurre con los espines, salvo que para este caso no es la fuerza de gravedad la que actúa, sino la fuerza del campo magnético externo. La figura 2.4 ilustra este fenómeno.

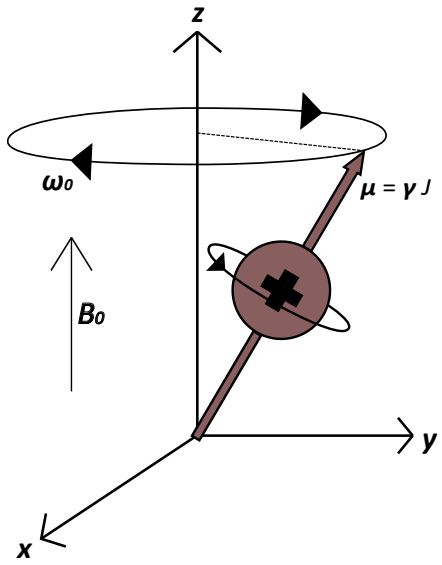


Figura 2.4: Movimiento de precesión de un espín sometido a un campo magnético estático.

La frecuencia a la que los espines precesionan (es decir, el número de ciclos que completan en un segundo) viene dada por la frecuencia de Larmor ω, proporcional al campo magnético $\boldsymbol{B}_0$ [10]:

$$\omega_0 = -\gamma B_0 \tag{2.4}$$

Esta proporcionalidad es clave para poder comprender el funcionamiento de los procesos de generación y adquisición de imágenes de resonancia magnética.

Magnetización neta

Tal y como ya se ha mencionado anteriormente, el vector magnetización neta $\boldsymbol{M}$ está formado por la suma de todos los momentos magnéticos microscópicos individuales:

$$\boldsymbol{M} = \sum_{n=1}^{N_s} \boldsymbol{\mu_n} = (M_x, M_y, M_z) \tag{2.5}$$

Siendo N_s el número total de espines del cuerpo, y (M_x, M_y, M_z), las componentes del vector magnetización neta en las direcciones $\hat{x}$, $\hat{y}$ y $\hat{z}$, respectivamente.

Previamente ya se ha visto que, bajo la acción del campo magnético estático $\boldsymbol{B}_0$, la única componente del vector magnetización neta no nula será aquella paralela al campo. Por lo tanto, siguiendo la convención de que $\boldsymbol{B}_0$ apunta en la dirección longitudinal $\hat{z}$:

$$M_x^0 = M_y^0 = 0, \qquad M_z^0 = \frac{1}{2}(N_\uparrow - N_\downarrow)\gamma\bar{h} \tag{2.6}$$

Donde los superíndices ^{0} indican que estamos en presencia únicamente de $\boldsymbol{B}_0$. $N_\uparrow$ es el número de espines que apuntan en la dirección paralela, $N_\downarrow$ el número de espines que apuntan en la dirección antiparalela y $\bar{h}$ es la constante de Planck reducida o constante de Dirac [7]:

$$\bar{h} = \frac{h}{2\pi}, \qquad Constante\ de\ Plank\ h = 6{,}33 \cdot 10^{-34} J \cdot s \tag{2.7}$$

Excitación RF y *flip angle*

Con todo lo que hemos visto hasta ahora, aun no estamos en condiciones de producir ni adquirir ninguna señal de resonancia magnética. Esto se debe a que, aunque la componente longitudinal del vector de magnetización neta sea no nula, esta es muy pequeña en relación a $\boldsymbol{B}_0$, por lo que resulta imposible hacer mediciones directamente en esa dirección. Para detectar correctamente la magnetización necesitamos, por tanto, que esta también esté presente en el plano transversal, es decir, que las componentes M_x y M_y sean no nulas. Para ello, se requiere la aplicación de un segundo campo magnético $\boldsymbol{B}_1(t)$, en la dirección transversal, cuyo módulo es [7]:

$$B_1(t) = B_1^e(t) \cdot e^{-j(w_{rf}t+\phi)} \tag{2.8}$$

Siendo $B_1^e(t)$ la amplitud de la envolvente compleja del campo, ω_{rf} la frecuencia de la portadora y ϕ la fase inicial, que se supondrá nula.

Este campo $B_1(t)$ debe oscilar a una frecuencia igual a la frecuencia de Larmor (es decir, $\omega_{rf} = \omega_0$). De esta forma, los espines entrarán en resonancia y se producirá una transferencia eficiente de energía [8]. La frecuencia de precesión de los núcleos de hidrógeno cuando, por ejemplo, $B_0 = 1{,}5\,T$, es de aproximadamente 64 MHz, dentro del rango RF. Es por este motivo por el que al campo B_1 se lo conoce normalmente como pulso RF.

La aplicación de este pulso RF hará que el vector de magnetización neta del sistema $\boldsymbol{M}$ se mueva con respecto a su orientación inicial, paralela a $\boldsymbol{B}_0$, y comience a tener componentes transversales. La evolución de este proceso en el tiempo viene descrita por la ecuación de Bloch, que da cuenta de la variación de $\boldsymbol{M}$ [10]:

$$\frac{d\boldsymbol{M}}{dt} = \boldsymbol{M} \times \gamma\mathbf{B} - \frac{M_x\hat{i} + M_y\hat{j}}{T_2} - \frac{(M_z - M_z^0)\hat{k}}{T_1} \tag{2.9}$$

Siendo M_z^0 el valor de equilibrio de $\boldsymbol{M}$ únicamente bajo la presencia del campo $\boldsymbol{B}_0$. T_1 y T_2 son constantes temporales que caracterizan el proceso de relajación del sistema de espines [7], y de las que hablaremos más adelante.

Un concepto adicional que es importante destacar es el del ángulo de giro, *flip angle* α, que hace referencia a la cantidad de giro que experimenta el vector magnetización con respecto a la dirección longitudinal tras la aplicación de un pulso RF de duración τ_p. Se obtiene a partir de la siguiente expresión [11]:

$$\alpha = \gamma \int_0^{\tau_p} |B_1^e| dt \tag{2.10}$$

La figura 2.5 (en la cual podemos interpretar M_0 como el módulo del vector $\boldsymbol{M}$) muestra la influencia del pulso RF en el vector magnetización, y con ello, la aparición del *flip angle*. Cabe destacar que, para ciertos casos comunes del *flip angle* (45º, 90º, 180º...), existen valores concretos de $|B_1^e(t)|$ y de τ_p que hacen que se den justamente estos ángulos. Así, si por ejemplo queremos aplicar un pulso RF que produzca un ángulo de giro de 90º, podríamos decir, simplemente, que aplicamos un pulso de 90º (*90º pulse*) La figura 2.5c da cuenta de ello.

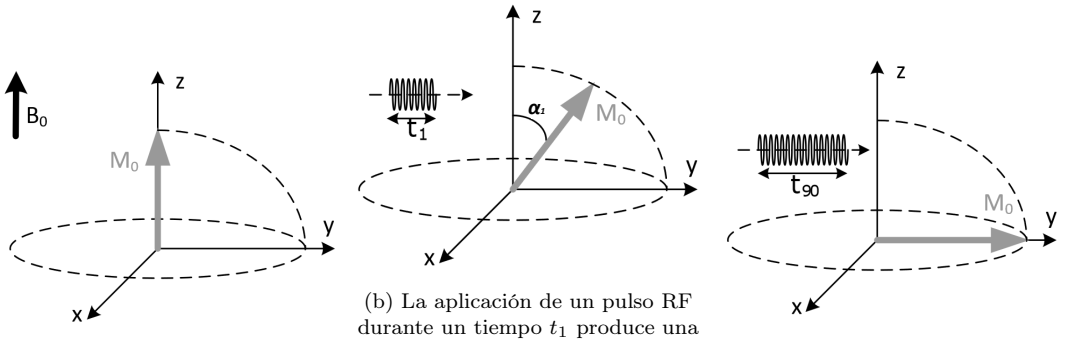
Si asumimos que la duración del pulso RF es pequeña comparada con T_1 y T_2, la ecuación 2.9 se puede simplificar como:

$$\frac{d\boldsymbol{M}}{dt} = \boldsymbol{M} \times \gamma\mathbf{B} \tag{2.11}$$

Asimismo, si el pulso es lo suficientemente corto como para poder considerar su aplitud constante, 2.10 queda reducida a [11]:

$$\alpha = \gamma |B_1^e(t)| \tau_p \tag{2.12}$$

Obsérvese que al duplicar la amplitud también se duplica el ángulo de giro.



(a) No se ha aplicado ningún pulso RF. En este caso $M_z = M_0$ y $M_x = M_y = 0$

(b) La aplicación de un pulso RF durante un tiempo t_1 produce una inclinación de la magnetización en un ángulo α_1. En este caso $M_z = M_0 cos\alpha_1$ y $M_{xy} = M_0 sin\alpha_1$

(c) Se aplica un pulso RF de 90º, dando como resultado un $M_z = 0$ y un $M_{xy} = M_0$

Figura 2.5: Efecto producido en el vector magnetización neta por la aplicación de un pulso RF [12].

Relajación

Si el pulso RF se apaga, los momentos magnéticos de los núcleos y la correspondiente magnetización neta $\boldsymbol{M}$ continúan precesando alrededor de $\boldsymbol{B}_0$, a la frecuencia de Larmor ω_0. Simultáneamente, $\boldsymbol{M}$ comienza de nuevo a inclinarse hacia la alineación paralela a $\boldsymbol{B}_0$ llegando, finalmente, a la condición de equilibrio térmico que tenía antes de la excitación RF [12]. Este efecto, mostrado en la figura 2.6, produce una trayectoria en espiral.

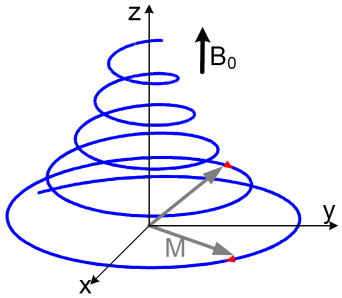
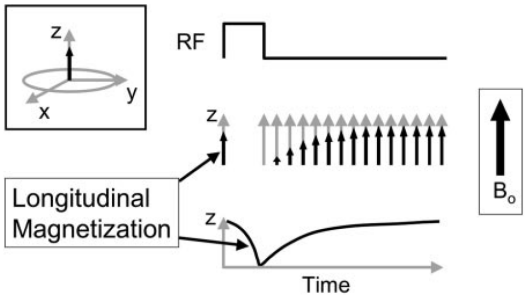


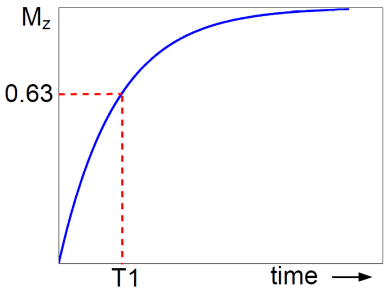
Figura 2.6: Comportamiento del vector magnetización tras desactivar el pulso de 90º. $\boldsymbol{M}$ comienza a inclinarse hacia $\boldsymbol{B}_0$ mientras precesa a la frecuencia de Larmor [12]

Podemos dividir el proceso de relajación en tres subprocesos, que ocurren de forma simultánea:

- **Precesión libre**: es el movimiento de precesión de $\boldsymbol{M}$ alrededor de $\boldsymbol{B}_0$ [7].
- **Relajación $\boldsymbol{T_1}$, longitudinal o espín-red**: debida a un intercambio de energía entre los espines y el ambiente circundante. Provoca la recuperación de la magnetización longitudinal [7] y queda caracterizada por T_1, un parámetro dependiente de cada tejido, pero que también depende de la intensidad del campo magnético principal. Un ejemplo de este efecto se muestra en la figura 2.7a. Siendo algo más concretos, podemos definir T_1 como el tiempo que tarda la magnetización longitudinal en alcanzar el 63 % de su valor final (ver figura 2.7b), suponiendo que se ha aplicado un pulso RF de 90º. La magnetización de los tejidos con diferentes valores de T1 volverá a crecer en la dirección longitudinal a diferentes velocidades [8].
- **Relajación $\boldsymbol{T_2}$, transversal o espín-espín**: debida a la pérdida de coherencia de fase en la precesión de los espines. Da lugar a la pérdida de las componentes transversales de $\boldsymbol{M}$ [7] y viene caracterizada por T_2. Recordemos que, durante el pulso RF, los protones comienzan a precesar juntos (se ponen en "fase"). Inmediatamente después del pulso, los protones siguen en fase pero comienzan a desfasarse debido a diversos efectos, entre los que está la interacción espín-espín. Podemos definir T_2 como el tiempo que tarda la magnetización transversal en caer hasta el 37 % de su valor original [8]. La figura 2.8 da cuenta este efecto.

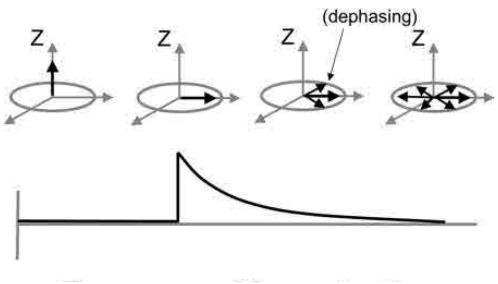


(a) Relajación longitudinal (T1) tras la aplicación de un pulso de 90º [8]

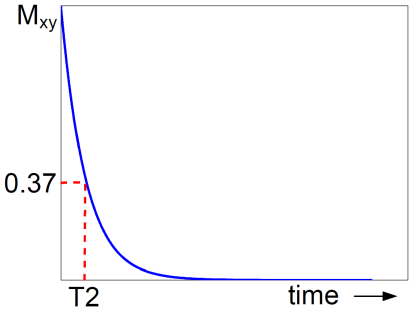


(b) Definición de T1 [12]

Figura 2.7: Relajación T1, longitudinal o espín-red



(a) Relajación transversal (T2) tras la aplicación de un pulso de 90º [8]



(b) Definición de T2 [12]

Figura 2.8: Relajación T2, transversal o espín-espín

Las figuras 2.7b y 2.8b ya nos han adelantado que el vector de magnetización neta se puede dividir en dos componentes: transversal y longitudinal:

$$\boldsymbol{M} = \boldsymbol{M}_{xy} + \boldsymbol{M}_z = M_{xy}\hat{u} + M_z\hat{z} \tag{2.13}$$

Siendo $\hat{u}$ un vector unitario con dos componentes (u_x, u_y). La variación con el tiempo de M_z y M_{xy} tras desactivar el pulso RF queda descrita por las siguientes ecuaciones [6]:

$$M_z(t) = M_z^0(1 - e^{-\frac{t}{T_1}}) + M_z^{0+}e^{-\frac{t}{T_1}} \tag{2.14}$$

$$M_{xy}(t) = M_{xy}^{0+}e^{-\frac{t}{T_2}} \tag{2.15}$$

Siendo M_z^0 la magnetización longitudinal justo antes del pulso RF y M_z^{0+} y M_{xy}^{0+} las magnetizaciones longitudinal y transversal justo después de la aplicación del pulso. Para el caso de un pulso RF de 90º, M_z^{0+} sería nula, por lo que la ecuación 2.14 quedaría reducida a [12]:

$$M_z(t) = M_z^0(1 - e^{-\frac{t}{T_1}}) \tag{2.16}$$

Por otra parte, se ha mencionado previamente que el desfase de los protones no se debe solo a las interacciones espín-espín (si solo tuviésemos en cuenta este efecto, podríamos seguir afirmando que el tiempo de relajación transversal se corresponde con T_2), sino que existen otros fenómenos que pueden reducir más este tiempo, tales como:

- Las inhomogeneidades ΔB_0 en el campo magnético
- La susceptibilidad magnética
- El denominado *chemical shift*

Por ahora, no vamos a entrar en detalle en ninguno de estos efectos, simplemente vamos a tenerlos en cuenta en el proceso de relajación transversal. Contando con los cuatro efectos y no solo con las interacciones espín-espín, los protones se desfasan más rápidamente, por lo que el tiempo de relajación transversal disminuye y ya no se denomina T_2, sino T_2^*. En la práctica, y por simplicidad, para calcular T_2^* basta con tener en cuenta las inhomogeneidades del campo magnético:

$$T_2^* = \frac{1}{\frac{1}{T2} + \gamma \Delta B_0} \tag{2.17}$$

Detección de la señal MR

Tal y como se ha mencionado anteriormente, resulta imposible detectar la componente longitudinal de la magnetización debido a su pequeño valor con respecto al campo $\boldsymbol{B}_0$. La única señal que es posible detectar y medir es la procedente de la componente transversal M_{xy}. Es por ello por lo que se aplican pulsos de radiofrecuencia, tal y como se ha visto.

Para detectar esta magnetización transversal, se coloca una espira con un campo magnético perpendicular a $\boldsymbol{B}_0$ lo suficientemente cerca del cuerpo, tal y como se muestra en la figura 2.9. Entonces, la magnetización neta generará un voltaje inducido en la espira, tal y como enuncia la ley de Faraday. Este voltaje será una sinusoide de frecuencia ω_0 y su amplitud disminuirá según lo haga M_{xy}, es decir, a razón de T_2^*. Esta es la denominada señal MR, conocida comunmente como *Free Induction Decay* (FID) [12].

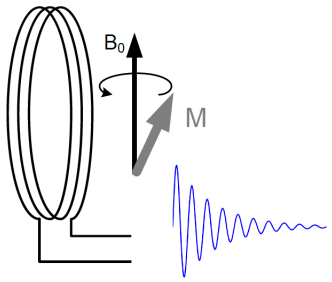


Figura 2.9: Detección de la señal MR mediante una bobina [12]

La señal recibida por nuestro escáner, por lo tanto, no nos dará información sobre T_2, sino que lo que podremos medir, a priori, será T_2^*. No obstante, se puede calcular el valor real de T_2 aplicando una secuencia de pulsos RF conocida como *spin echo*, la cual se explicará en detalle más adelante.

2.2.2. Formación y adquisición de la imagen de resonancia magnética

En la sección anterior de fundamentos físicos hemos analizado el proceso de generación y detección de una señal MR. Llegados a este punto debemos recalcar que aun estamos hablando de señal, y no de imagen; es importante tener clara la diferencia entre estos dos conceptos:

- **Señal de resonancia magnética (*MR signal*)**: es la señal analógica de voltaje inducido por la espira, y generada por la componente transversal de la magnetización neta. Tal y como hemos visto, es una señal sinusoidal de frecuencia ω_0 y cuya amplitud decae a razón de T_2^*. Esta señal no nos aporta demasiada información, menos aun si queremos analizarla con fines médicos.
- **Imagen de resonancia magnética (*MR Image*, MRI)**: es una imagen digital, es decir, la representación gráfica de una matriz bidimensional. Esta matriz se obtiene mediante el procesado, en distintas etapas (demodulación, muestreo, transformadas inversas de Fourier, etc.) de la señal MR, y es la que nos aporta verdadera información de cara al diagnóstico médico.

El primer reto a la hora de adquirir una imagen a partir de la señal MR es el de poder localizar espacialmente la parte del cuerpo de la cual queremos obtener la señal ya que, de lo contrario, la señal adquirida no se correspondería con una parte del cuerpo concreta (cabeza, corazón, hígado...), sino

que estaría producida por la excitación de todos y cada uno de los espines del cuerpo. Vamos a ver que esta localización espacial se puede realizar por medio de gradientes.

Gradientes

Los gradientes de campo magnético añaden o quitan intensidad al campo magnético $\boldsymbol{B}_0$ en cada una de las tres dimensiones cartesianas (x, y, z), de forma que la frecuencia de Larmor de cada espín y su correspondiente momento magnético se vuelve dependiente de su posición [1]. Además, cada uno de los tres gradientes se aplica con una espira distinta, pudiendo así activar o desactivar cada gradiente de forma independiente con respecto del resto. La figura 2.10 muestra una idea de cómo se pueden integrar las bobinas que generan los gradientes en un equipo de resonancia real.

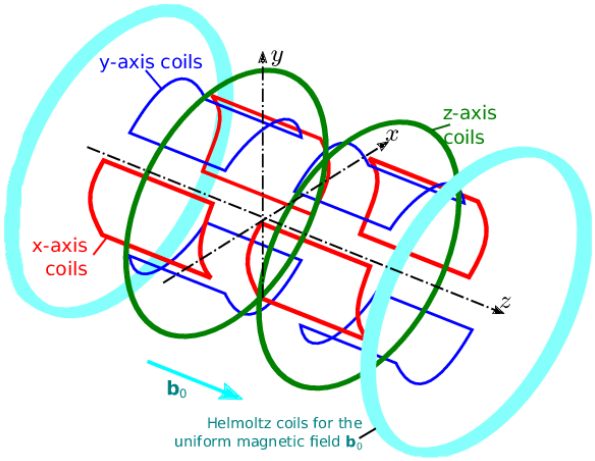


Figura 2.10: Representación de la configuración del conjunto de bobinas de gradiente magnético de MRI [13]

Estos gradientes, además de hacer que la frecuencia de precesión de cada espín sea dependiente de su posición, pueden configurarse para que esa dependencia esté también presente en su fase. De esta forma podríamos localizar cada espín, por ejemplo, en la dimensión x fijándonos en su frecuencia y en la dimensión y fijándonos en su fase. Estas técnicas se denominan codificación de frecuencia y de fase, y se explicarán más adelante.

Si dividimos el proceso de obtención de una imagen de MR en dos subprocesos, podemos distinguir la fase de excitación (selección de *slice*) y la fase de adquisición (recorrido del espacio K). En ambas es necesario el uso de gradientes, aunque se emplean con propósitos distintos.

Fase de excitación: selección de *slice*

En esta fase se activa el pulso RF, por lo que ocurre todo lo que se ha explicado hasta el momento: excitación de los espines, aparición de la componente transversal de la magnetización, relajación, etc.

No obstante, si en vez de excitar todos los espines del cuerpo, se quiere excitar únicamente una "tajada" (*slice* en inglés) concreta, será necesario, además, activar un gradiente lineal (*Slice Select Gradient*, SSG [12]) a la vez que el pulso RF está activado. De esta forma, la frecuencia de resonancia de los espines será dependiente de su posición, y solo serán excitados aquellos espines cuya frecuencia de precesión coincida con la del pulso RF [7]. Podemos seleccionar un slice a lo largo de cualquier dirección arbitraria simmplemente controlando el valor que toma cada uno de los tres gradientes (G_x, G_y, G_z) durante la excitación. Además, si dotamos al pulso RF de cierto ancho de banda BW, podremos seleccionar un slice con un grosor proporcional a ese ancho.

Veamos el ejemplo más común: si queremos seleccionar un *slice* perpendicular a la dirección longitudinal (de pies a cabeza), solo será necesario activar G_z. Entonces, el campo magnético neto en la dirección $\hat{z}$ será:

$$\boldsymbol{B}_{net} = (B_0 + G_z z)\hat{z} \tag{2.18}$$

Ahora, el pulso RF que activemos (ver ecuación 2.8) deberá tener una frecuencia central:

$$w_{rf}(z) = \gamma B_{net} = \gamma(B_0 + G_z z) \tag{2.19}$$

La figura 2.11 muestra esta selección. Podemos ver cómo G_z corresponde a la pendiente de la función. Esto quiere decir que cuanto mayor sea G_z, más sensible será la frecuencia a variaciones a lo largo de z y, por tanto, más estrecho será el *slice*. Para $z = 0$, la frecuencia central del pulso se corresponde con la frecuencia de Larmor ω_0. Este punto es conocido como isocentro.

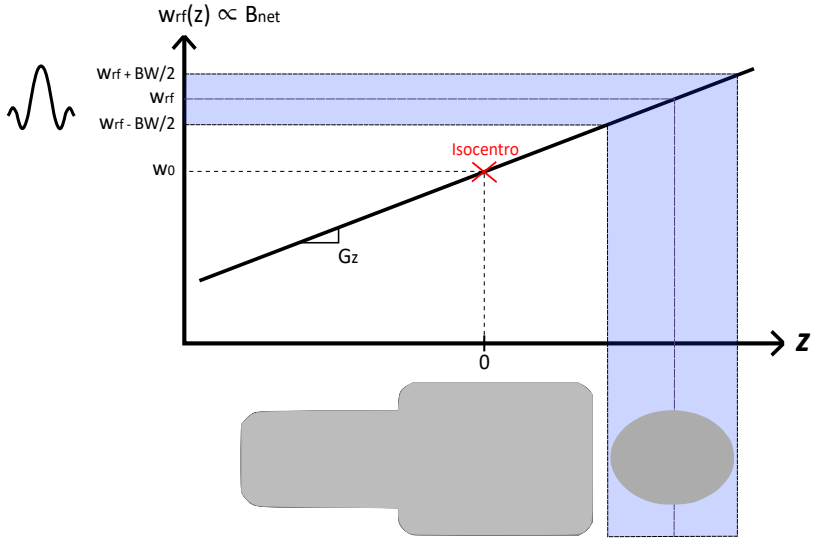


Figura 2.11: Selección de slice a lo largo de z

Llevándolo ahora al caso general, supongamos que se aplica un pulso RF de amplitud compleja $B_1 = B_{1x} + jB_{1y}$ durante un intervalo Δt lo suficientemente corto como para poder considerar la amplitud constante. Supongamos, además, que se aplican unos gradientes $\mathbf{G} = (G_x, G_y, G_z)$ y que existen inhomogeneidades ΔB_0 en el campo magnético. Entonces, el campo magnético local, dependiente de la posición, será el siguiente:

$$\mathbf{B} = \begin{bmatrix} B_{1x}(\mathbf{r}) \\ B_{1y}(\mathbf{r}) \\ \mathbf{G}^T\mathbf{r} \ + \ \Delta B_0 \end{bmatrix} \tag{2.20}$$

En consecuencia, el efecto producido por el pulso en la posición del punto $\mathbf{r}$ es una rotación[1] del vector magnetización con un ángulo ϕ alrededor del eje $\mathbf{n}$:

$$\phi = -\gamma\,\Delta t\,\sqrt{|B_1|^2 + (\mathbf{G}^T\mathbf{r} + \Delta B_0(\mathbf{r}))^2} \tag{2.21}$$

$$\mathbf{n} = \frac{\gamma\Delta t}{|\phi|}\begin{bmatrix} B_{1x}(\mathbf{r}) \\ B_{1y}(\mathbf{r}) \\ \mathbf{G}^T\mathbf{r} + \Delta B_0(\mathbf{r}) \end{bmatrix} = \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} \tag{2.22}$$

Esto se traduce en que, si aplicamos unos gradientes $\mathbf{G}$ durante la excitación, la máxima componente transversal aparecerá en todos aquellos puntos $\mathbf{r} \in \mathbb{R}^3$ para los que $\mathbf{G}^T\mathbf{r} = 0$. Esta es la ecuación en 3D de un plano cuyo vector normal es $\mathbf{G}$ y que contiene al punto $(0, 0, 0)$ cuando $\Delta B_0(\mathbf{r}) = 0$.

Aunque $\Delta B_0(\mathbf{r})$ caracterice las inhomogeneidades, hemos visto que también puede ser elegido explícitamente para ser distinto de 0 y así poder seleccionar un slice separado del isocentro. Si el pulso

[1]La rotación en la ecuación 2.21 sigue la regla de la mano derecha, mientras que la rotación en la ecuación 2.10 sigue la regla de la mano izquierda; esto explica el signo negativo en la ecuación 2.21, que hace que las rotaciones sean equivalentes [11]

RF tiene una frecuencia $\omega_{rf} = \omega_0 + \Delta\omega$, siendo $\omega_0 = \gamma B_0$, entonces $\Delta B_0(\mathbf{r}) = -\frac{\Delta\omega}{\gamma}$ y el slice excitado satisfará:

$$\mathbf{G}^T\mathbf{r} - \frac{\Delta\omega}{\gamma} = 0 \tag{2.23}$$

Esta es la ecuación del plano perpendicular al vector $\mathbf{G}$ que está separado del origen (en la dirección de $\mathbf{G}$) una distancia igual a $\frac{\Delta\omega}{\gamma|\mathbf{G}|}$.

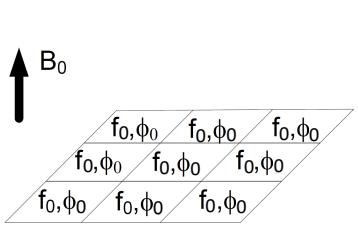
Además, como ya se ha mencionado, la anchura del *slice* será función, en esencia, del ancho de banda del pulso, es decir, del ancho de banda (temporal) de la envolvente compleja $B_{1x}(\mathbf{r},t) + jB_{1y}(\mathbf{r},t) = B_1^e(\mathbf{r},t)e^{j\varphi(\mathbf{r},t)}$. La forma del pulso puede ser aproximada mediante pulsos cortos de amplitud constante, de forma que las ecuaciones 2.21 y 2.22 puedan seguir cumpliéndose [11].

Fase de adquisición: recorrido del espacio K

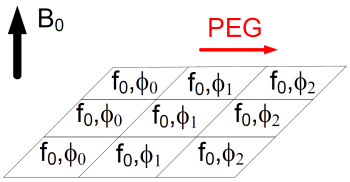
Una vez seleccionado y excitado el *slice* (por simplicidad, supondremos que tiene anchura 0), y justo después de desactivar los gradientes y el pulso RF, todos los espines del *slice* precesarán con aproximadamente la misma fase y frecuencia. Es ahora cuando entra en juego la denominada codificación espacial, que nos permitirá localizar la señal producida por cada uno de estos espines en las dos coordenadas que definen el plano excitado. La codificación espacial se divide en dos procesos sucesivos:

- **Codificación de fase**: durante esta fase se aplica un gradiente (*Phase Encoding Gradient*, PEG) a lo largo de uno de los ejes que definen el plano excitado (en el caso del ejemplo sencillo anterior, en el que se seleccionó un slice a lo largo de z, el PEG podría activarse, por ejemplo, en y). Durante su aplicación, los espines del slice precesarán a frecuencias diferentes según su posición en el eje del PEG, por lo que también se desfasarán. Tras desactivar el gradiente, los espines volverán a precesar todos con la misma frecuencia aproximada, pero ahora, con fases diferentes [12]. Esta diferencia de fase se podrá cuantificar, permitiéndonos localizar, en una de las dos direcciones, la procedencia espacial de la señal MR recibida. La figura 2.12b de cuenta del efecto de esta codificación.

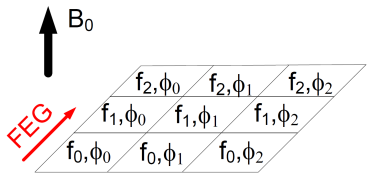
- **Codificación de frecuencia**: hemos visto cómo después de aplicar el PEG, los espines precesan con aproximadamente la misma frecuencia pero con fases diferentes. Ahora, se aplica un gradiente (*Frecuency Encoding Gradient*, FEG) a lo largo del tercer eje, que es mutuamente perpendicular a los ejes del SSG y del PEG (en el caso del ejemplo, este tercer eje se correspondería con x). En este caso los espines, que ya están precesando con fases diferentes, comienzan a precesar también con diferentes frecuencias según su posición en el eje del FEG. Ya estamos, por tanto, en condiciones para registrar la señal, ya que esta contiene la información de fase y frecuencia de los espines de cada píxel de la imagen deseada. La señal es registrada durante la aplicación del FEG, y es por ello por lo que a este gradiente de codificación de frecuencia también se le denomina gradiente de lectura (*readout*) [12]. La figura2.12c muestra el efecto producido por esta codificación.



(a) Slice seleccionado antes de activar los gradientes de codificación espacial. Todos los espines del slice tienen la misma frecuencia y fase.



(b) La aplicación del PEG provoca un cambio en la fase de los momentos magnéticos de cada espín, proporcional a su posición en el eje del gradiente.



(c) Durante la aplicación del FEG, los momentos magnéticos mostrarán distintas frecuencias junto a las distintas fases debidas al PEG. Por lo tanto, la señal MR puede registrarse durante la activación del PEG

Figura 2.12: Codificación espacial mediante gradientes de codificación de fase y de frecuencia [12]

Si consideramos que se aplica un PEG en la dirección y durante un tiempo T_{pe}, el desfase dependiente de la posición será:

$$\phi(y) = \gamma G_y y T_{pe} \tag{2.24}$$

Y la señal recibida de todo el objeto vendrá dada por:

$$S(t) = \int_{-\infty}^{\infty} \rho(y) e^{-j(\omega_0 t + \phi(y))} dy \tag{2.25}$$

Donde $\rho(y)$ corresponde a la densidad de espines del objeto a lo largo de y. Tras demodular (se elimina la portadora):

$$S(t) = \int_{-\infty}^{\infty} \rho(y) e^{-j\phi(y)} dy = \int_{-\infty}^{\infty} \rho(y) e^{-j(\gamma G_y T_{pe})y} dy \tag{2.26}$$

Aplicando, ahora, un FEG en la dirección x de forma independiente, sabemos que la frecuencia dependiente de la posición será:

$$w(x) = \omega_0 + \gamma G_x x \tag{2.27}$$

Por lo que la señal recibida en este caso sería:

$$S(t) = \int_{-\infty}^{\infty} \rho(x) e^{-j(\omega_0 t + \gamma G_x x)} dx \tag{2.28}$$

Tras demodular:

$$S(t) = \int_{-\infty}^{\infty} \rho(x) e^{-j\gamma G_x t x} dx \tag{2.29}$$

Si definimos los términos k_x y k_y tal que:

$$k_x = \frac{\gamma}{2\pi} G_x t\,, \qquad k_y = \frac{\gamma}{2\pi} G_x T_{pe} \tag{2.30}$$

Y sustituimos en 2.29 y 2.26 respectivamente, podremos observar que existe una relación de Fourier entre la variación espacial de la densidad de espines y la señal recibida:

$$S(k_x, k_y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \rho(x, y) e^{-j2\pi(k_x x + k_y y)} dx dy \tag{2.31}$$

Esto es lo que se conoce como espacio K, y constituye el dominio sobre el que se distribuye la señal original (*raw data*) obtenida [7]. Además, la señal en el espacio K se corresponde con la transformada de Fourier de la imagen deseada, por lo que bastará con hacer la transformada inversa de la matriz del espacio K para poder obtener la imagen. En la figura 2.13 se muestra el ejemplo de una imagen y de la matriz del espacio K a partir de la cual se obtuvo [14].

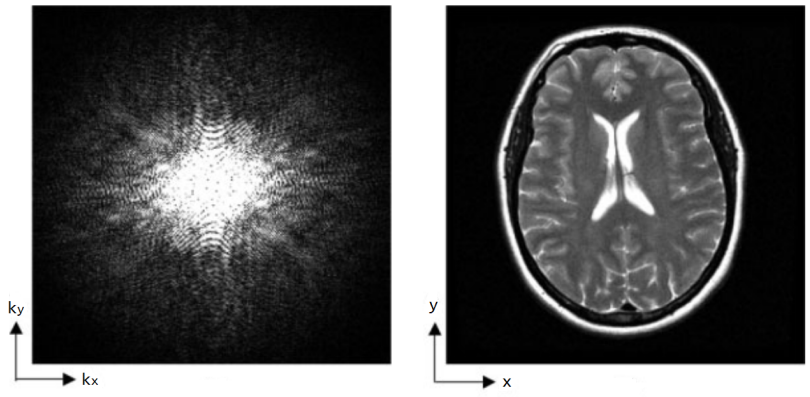


Figura 2.13: (a) *Raw data* en el espacio K. (b) Imagen correspondiente [14]

La ubicación en el espacio K (k_x, k_y) a la que pertenecen los datos de cualquier punto muestreado se rige por el efecto acumulado de los gradientes desde que se produce la excitación RF hasta el momento en el que se recoge esa muestra [14]. Esto quiere decir que, aunque las ecuaciones previas resulten útiles para comprender el proceso analítico de formación del espacio K, lo que realmente nos va a ser de utilidad es el hecho de saber que la aplicación de gradientes durante la fase de adquisición produce un desplazamiento en el espacio K, convencionalmente de la siguiente forma:

- Los gradientes activados en la dirección vertical (en el ejemplo, y) corresponderán a los gradientes de codificación de fase. Durante su aplicación nos moveremos verticalmente por el espacio K.
- Los gradientes activados en la dirección horizontal (en el ejemplo, x) se corresponderán con los gradientes de codificación de frecuencia. Nos permiten desplazarnos horizontalmente por el espacio K y durante su aplicación se realizará la lectura de los datos.

La combinación, por tanto, de estos dos tipos de gradientes, nos va a permitir seguir cualquier trayectoria arbitraria a la hora de adquirir los datos. Los gradientes, junto con la(s) excitación(es) RF previa(s), definen lo que se conoce como secuencia de resonancia magnética.

2.2.3. Secuencias de MRI

Sabemos que para crear una imagen es necesaria la aplicación de pulsos de excitación de RF durante el proceso de relajación. Inmediatamente después, se mide la señal obtenida, generalmente en forma de eco. Para la obtención de estas señales de eco puede ser necesaria la aplicación de uno o más pulsos de refase de RF, o bien de gradientes. El conjunto de cada pulso de excitación de RF y los pulsos o gradientes de refase posteriores necesarios para producir una señal medible se denomina ciclo de pulsos, y se corresponde con lo que en este documento hemos denominado fase de excitación.

Junto a los pulsos de RF es necesaria la aplicación de gradientes de campo magnético para la localización y codificación espacial de la señal, tal y como hemos visto en la fase de adquisición. En MRI, es necesario repetir estos ciclos 64, 128, 256, 512 ó 1024 veces para rellenar el espacio K o matriz de datos crudos y la posterior reconstrucción de la imagen. Se denomina secuencia a la repetición de ciclos de pulsos y gradientes asociados [15].

Aunque existen decenas de siglas y acrónimos de secuencias, la mayoría son modificaciones, variantes y combinaciones de las secuencias básicas que se van a describir a continuación [15]. Según la forma en la que se genere la señal MR (fase de excitación), las secuencias se pueden dividir en dos grandes "familias": *spin echo* y *Gradient echo*. Además, debemos saber que también existen versiones "rápidas" de estas secuencias (*fast spin echo*, *fast gradient echo*), así como otras estrategias de adquisición, como la que se sigue en las secuencias EPI (*Echo Planar Imaging*).

Secuencia *Spin Echo* (SE)

Esta secuencia comienza con un gradiente SSG que se aplica simultáneamente con un pulso RF de 90^{o} para excitar un determinado corte. Tras descativar ambos, se aplica un PEG para conseguir una diferencia de fase entre los momentos magnéticos del corte seleccionado según su posición en el eje del gradiente. A continuación, pasado un tiempo igual a $TE/2$ (siendo TE un parámetro característico del ciclo de pulsos conocido como tiempo de eco [7]), se enciende el SSG simultáneamente con un pulso RF de 180^{o}. De esta forma, se compensa el desfase de los momentos magéticos debido a las inhomogeneidades ΔB_0 del campo magnético estático [12] (ver figura 2.15a) y, tras un tiempo (desde la aplicación del pulso de 90^{o}) igual a TE, se genera una señal conocida como eco. Esta es la señal que necesitamos leer, por lo que, mientras esté presente, se activará un gradiente FEG a la vez que se registra la señal. De esta forma habremos leído una línea del espacio K. Para leer más líneas, se repite (pasado un tiempo TR conocido como tiempo de repetición) la secuencia, cambiando la magnitud del PEG un paso más arriba (o más abajo) en cada repetición. La figura 2.14 muestra un diagrama temporal de una secuencia *spin echo* típica.

Se ha mencionado que gracias al pulso de 180^{o} se compensa el desfase producido por ΔB_0, y vimos anteriormente que esas inhomogeneidades ΔB_0 eran las principales responsables de que la amplitud de la componente transversal de la magnetización decayera a razón de T_2^* y no de T_2. Por tanto, lo que se está consiguiendo en realidad en una secuencia *spin echo* es "simular" un tiempo de relajación

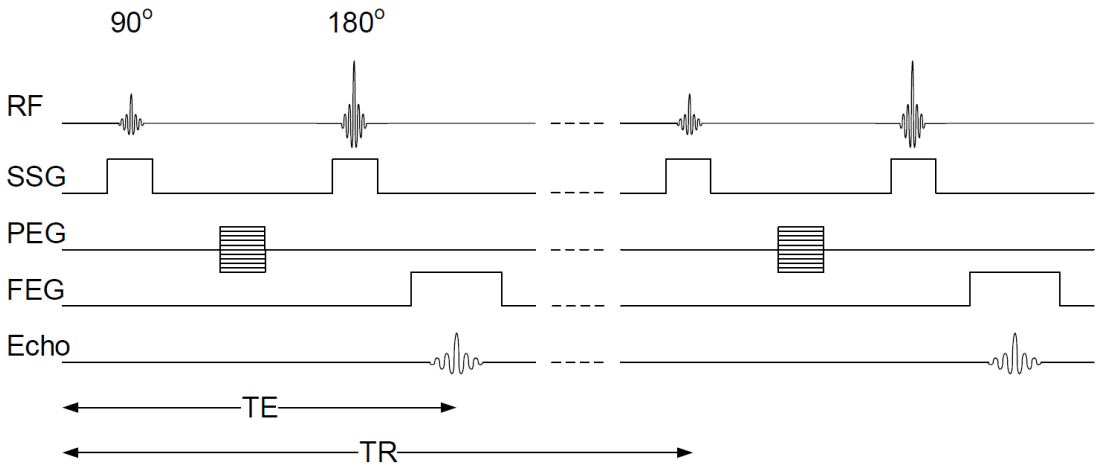


Figura 2.14: Diagrama temporal de una secuencia *spin echo* típica [12]

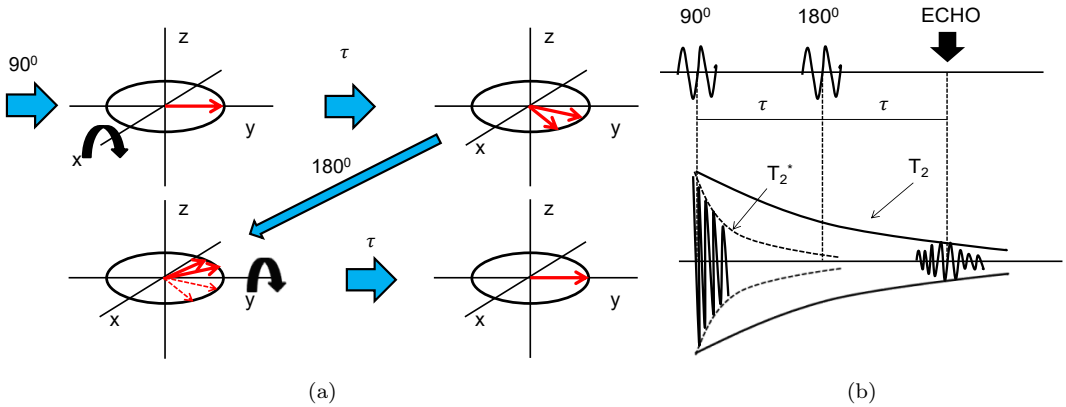


Figura 2.15: Efecto producido por el pulso re-orientador de 180º (a) en la magnetización, (b) en el tiempo de relajación transversal

transversal mayor, ya que, aunque la señal FID siga decayendo a razón de T_2^*, la amplitud del eco decaerá a razón de T_2 (ver figura 2.15b).

Existen numerosas variantes de la secuencia *Spin Echo*, entre las que se puede destacar la *Fast Spin Echo*, basada en el mismo principio de creación de ecos a partir de pulsos RF de reenfoque. Esta secuencia se caracteriza por obtener más de un eco por cada pulso RF de 90º, proporcionando un tiempo total de adquisición mucho menor. En una *spin echo* convencional, cada uno de los pulsos de 90º iba seguido por un único pulso de 180º. En una FSE, en cambio, el pulso de 90º puede ir sucedido por un número arbitrario de pulsos de reenfoque, conocido como *Echo Train Length* (ETL).

Secuencia *Gradient Echo* (GE)

La secuencia *Gradient Echo* (GE) comienza con un pulso RF que se aplica a la vez que el gradiente SSG. Después, se aplica un gradiente PEG para conseguir la codificación de fase. De forma simultánea a este PEG, se aplica un FEG negativo para compensar el desfase que se espera generar durante el gradiente de lectura, y acto seguido se activa el gradiente de lectura (FEG) propiamente dicho. Durante este gradiente, los momentos magnéticos ya desfasados (debido al FEG negativo) comienzan a reenfocarse y, de forma inherente, la señal de MR comienza a aumentar hasta alcanzar el máximo. A continuación, con el paso del tiempo, los momentos magnéticos comienzan a desfasarse de nuevo, lo que provoca una disminución de la amplitud de la señal de MR. Ahora es evidente que la principal diferencia entre las dos secuencias es que la se emplea un pulso de RF de alta potencia de 180° para para generar el eco de MR, mientras que la GE utiliza un FEG negativo para reenfocar los momentos

magnéticos y generar el eco [12]. La figura 2.16 muestra el diagrama temporal de una secuencia *gradient echo*.

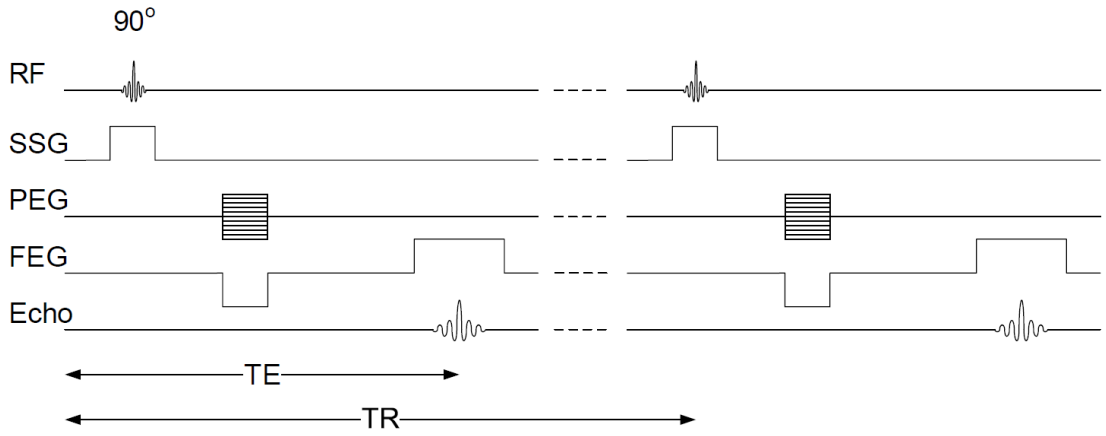


Figura 2.16: Diagrama temporal de una secuencia *gradient echo* típica [12]

Secuencia *Echo Planar Imaging* (EPI)

La secuencia EPI es una forma de adquisición ultrarrápida, cuya rapidez deriva de la adquisición de múltiples líneas del espacio K tras el pulso de excitación. En su forma más pura se adquieren todas las líneas del *slice* tras un único pulso (*single-shot*). En estas condiciones, y con una matriz de baja resolución, la adquisición de la imagen puede durar 50-100 ms. También se puede adquirir únicamente una parte del espacio K tras cada pulso de excitación (segmentación del espacio K). En este caso estaríamos hablando de la técnica de múltiples disparos (*multi-shot*). Se puede decir, genéricamente, que en todas las secuencias de MR la rapidez de la adquisición es inversamente proporcional a la calidad de la imagen; esto es, si cabe, más manifiesto en la secuencia EPI. Por ello, en las aplicaciones actuales en estudios de cuerpo el relleno del espacio K en la secuencia EPI se realiza de forma segmentada [15].

En EPI, los ecos se obtienen a partir de la FID, aplicando muy rápidamente gradientes de lectura alternativos de signo inverso (GE-EPI), alternándolos también con gradientes PEG que van desplazando el puntero línea a línea por el espacio K. Como la FID decae a razón de T_2^* (es decir, muy rápidamente), surgirán problemas, tales como una SNR baja, una mayor sensibilidad a artefactos o la necesidad de aplicar gradientes muy intensos y rápidos [15].

No obstante, los ecos también pueden obtenerse a partir de un pulso RF de 180º, tal y como ocurría en *spin echo* (SE-EPI). Esto, como ya se ha visto, nos permitiría tener más tiempo para la adquisición.

La figura 2.17 muestra esquemáticamente el diagrama temporal de una secuencia EPI, junto con su correspondiente trayectoria en el espacio K. Se sigue una trayectoria rectilínea, aunque esta secuencia también admite otras trayectorias, tales como en zigzag o en espiral.

Conceptos adicionales: parámetros de la secuencia y parámetros del escáner

Habiendo estudiado ya algunas de las principales secuencias de resonancia magnética, a continuación se mencionan dos conceptos importantes relacionados con ellas:

- TI: tiempo de inversión. Hace referencia al tiempo que transcurre desde que se aplica el pulso inversor de 180º hasta que se aplica el pulso de 90º en las denominadas secuencias *Inversion Recovery* (IR).
- NEX (*Number of Excitations*): es un parámetro de medida, utilizado para representar el número de veces que se adquiere la misma línea del espacio K. Se refiere, por lo tanto, al número de lecturas completas realizadas. El propósito de hacer más de una lectura (y por tanto, de adquirir más de una imagen del mismo corte), es el de mejorar la SNR (*Signal-to-Noise Ratio*), ya que

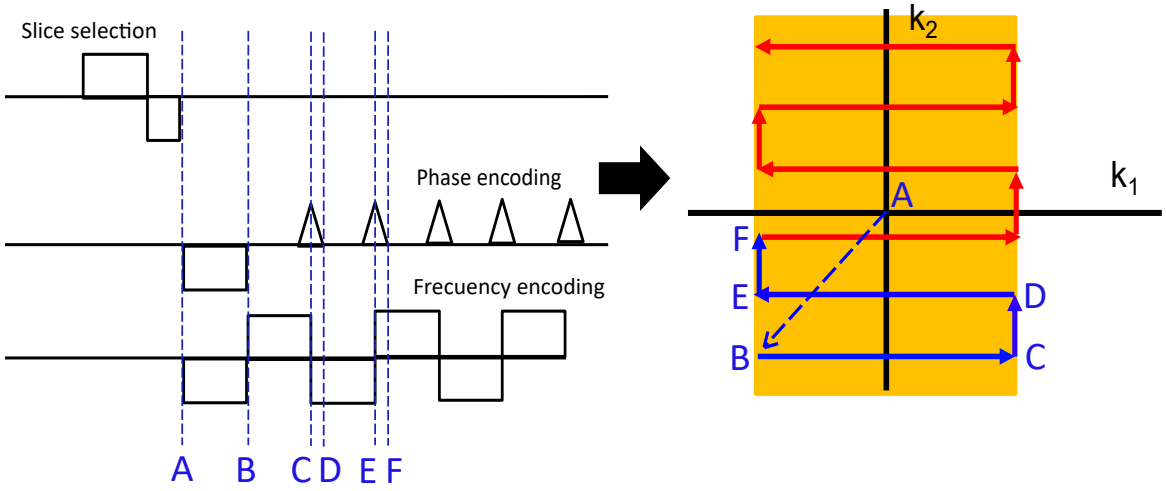


Figura 2.17: Diagrama temporal de una secuencia GE-EPI y trayectoria producida en el espacio K

se puede realizar una media entre todas las imágenes adquiridas y "cancelar", por tanto, el ruido proveniente del movimiento molecular y de la resistencia eléctrica de las bobinas.

Estos tres parámetros, junto con TE y TR, determinan lo que se conoce como parámetros de la secuencia, y vienen dados por el software del sistema de MRI. Por otro lado, existen los parámetros del escáner, determinados por el hardware. Entre ellos destacan la intensidad del campo magnético estático B_0, la intensidad máxima de los gradientes o el *slew-rate*.

Estos dos tipos de parámetros pueden ser agrupados bajo el nombre de parámetros de la simulación.

2.3. Fundamentos del lenguaje Julia

Entre la infinidad de lenguajes de programación que existen, los que permiten un uso dinámico e interactivo resultan especialmente útiles para desarrollar programas de forma ágil, probando, descartando y ajustando soluciones de forma continua. Asimismo, según avanza la tecnología y la capacidad de registrar y manejar información, es necesario que los programas sean más eficientes, para poder así llevar a cabo cálculos complejos con bases de datos voluminosas.

Este es, precisamente, el propósito de Julia [4], un lenguaje dinámico de alto nivel desarrollado por el MIT, utilizado para cálculos en múltiples ámbitos como la física, biología, ingeniería, matemáticas y finanzas, entre otros. Es un lenguaje fácil de extender y componer, con un "ecosistema" de paquetes versátil e igualmente extensible. Además, uno de sus puntos fuertes es la resolución de problemas numéricos complejos con grandes cantidades de datos, que otros lenguajes de su misma categoría no pueden abordar de forma eficiente a no ser que se combinen con código compilado en un segundo lenguaje (típicamente C o C++).

2.3.1. Características de Julia

El lenguaje Julia posee determinadas características diferenciadoras con respecto a otros lenguajes. Estas características pretenden crear un balance entre las necesidades del usuario y las necesidades del ordenador [4]. Destacan:

- **Definición opcional de los tipos de datos**: en los lenguajes de programación estáticos, como C o Fortran, todos los tipos de datos deben definirse, para poder así ser comprobados estáticamente durante la fase de compilación. El resultado es un excelente rendimiento. Por otro lado, los lenguajes dinámicos prescinden de las definiciones de tipos, lo que conduce a una mayor productividad de cara al programador pero a un menor rendimiento, ya que el compilador no cuenta a priori con la información de tipos, esencial para producir un código rápido [4].

El sistema de tipos de Julia se caracteriza por la posibilidad de especificar tipos cuando sea necesario, dando la opción al programador de omitir la definición del tipo de dato o, por el contrario, de definirlo si quiere optar por un mayor rendimiento.

- ***Multiple Dispatch***: esta característica se refiere a la selección de la implementación de una función basándose en el tipo de dato de cada argumento de entrada a esa función [4]. De esta forma es posible crear múltiples funciones con el mismo nombre, que procesen los datos de distinta manera dependiendo de su tipo. En los ficheros base de Julia ya existen numerosas funciones que hacen uso de esta propiedad; un ejemplo es el operador `*` que, aplicado sobre dos números, realiza su multiplicación, pero aplicado sobre dos cadenas de texto, realiza una concatenación. Si quisiésemos definir una nueva función –por ejemplo, la función `add`– que realizase la suma de dos números enteros o la concatenación de dos cadenas de texto, podríamos escribir las siguientes dos líneas de código:

```
add(a::Int64, b::Int64) = a + b
add(a::String, b::String) = a * b
```

- **Meta-programación**: se refiere a la posibilidad de transformar el código de un programa desde ese mismo programa, lo cual permite la generación de código sofisticado sin necesidad de pasos adicionales en la compilación [16].

2.3.2. Métodos de interacción con Julia

Los usuarios pueden interactuar con Julia por medio del entorno denominado REPL (*Real-Eval-Print Loop*), similar al empleado por lenguajes como Python, R o MATLAB [4]. El funcionamiento del REPL es análogo al de una línea de comandos, en la que se ejecutan las directivas introducidas por el usuario y se devuelve su resultado por pantalla.

Asimismo, existen otras dos alternativas para interaccionar con Julia: por medio de ficheros (`.jl`) que se ejecutan como *scripts*, o mediante la escritura directa en Jupyter Notebook [4].

2.3.3. Módulos

Los módulos en Julia ayudan a organizar el código dentro de unidades coherentes. Sintácticamente, están delimitados entre las directivas (`module NombreDelModulo`) y (`end`), y tienen las siguientes características [16]:

- Los módulos definen espacios de nombres (*namespaces*) separados, cada uno de los cuales introduce un nuevo ámbito global. Esto resulta útil porque permite la utilización sin conflicto del mismo nombre para diferentes funciones o variables globales, siempre y cuando estas estén en módulos separados.
- Los módulos facilitan la gestión detallada del espacio de nombres: cada módulo define un conjunto de nombres que exporta –mediante la directiva `export`– y puede importar nombres de otros módulos con `using` e `import`.
- Los módulos pueden ser precompilados para una carga más rápida, y contienen código para la inicialización durante el tiempo de ejecución.

2.3.4. Carga de código en Julia

Julia cuenta con dos mecanismos para cargar código [16]:

- **Inclusión de código** mediante el comando `include("source.jl")`. Esta expresión hace que el contenido de `source.jl` sea evaluado dentro del ámbito global del módulo en el que se produce la llamada a `include`. Si se llama a `include("source.jl")`, el fichero `source.jl` se evaluará varias veces. El argumento de entrada a esta función es interpretado como una ruta relativa al fichero en el que se produce la llamada; si se hace la llamada desde el REPL, esta ruta será relativa al directorio de trabajo actual, `pwd()`.

- **Carga de paquetes** mediante los comandos `import X` y `using X`. Este mecanismo permite cargar un paquete[2] y hace que el módulo resultante esté disponible con el nombre `X` dentro del módulo importador. Si el mismo paquete `X` se importa varias veces en la misma sesión de Julia, solo se cargará la primera vez.

2.3.5. Entornos y Proyectos

Al trabajar con múltiples proyectos en Julia, se recomienda usar un entorno (*environment*) diferente para cada uno de ellos, de forma que se facilite la reproducibilidad y se minimicen las dependencias de los paquetes. Un entorno en Julia viene definido por dos ficheros que residen en el directorio del proyecto: `Project.toml` y `Manifest.toml`. El primero de ellos describe el proyecto en alto nivel; por ejemplo, define las dependencias y las restricciones de compatibilidad del proyecto. Los campos más importantes presentes en este fichero son el nombre (`nombre`) del proyecto, su autor (`name`), la versión (`version`) de Julia utilizada, o el campo `uuid`, necesario para definir cada uno de los paquetes que han de instalarse. Por otra parte, el fichero `Manifest.toml` es un registro absoluto del estado de los paquetes en el entorno. Incluye información exacta sobre las dependencias –directas e indirectas– del proyecto y es generado automáticamente, por lo que nunca debería ser modificado de forma manual [16].

Dado un par (`Project.toml` + `Manifest.toml`), es posible instanciar exactamente el mismo entorno de un proyecto, lo cual resulta extremadamente útil para su reproducibilidad [16].

2.3.6. Integración de Julia en C/C++

Existe una API que permite integrar expresiones, variables y funciones de Julia en programas desarrollados en C/C++ (ver sección "Embedding Julia", en [16]). Destacamos las funciones:

- `jl_init`: inicializa el motor de Julia dentro de la aplicación. Es la primera función a la que hay que llamar para poder integrar Julia. Es fundamental, además, que Julia esté instalado en la máquina y que el programa conozca su directorio de instalación.

- `jl_eval_string`: hace que Julia interprete literalmente el contenido de la cadena de texto que se pase como argumento de entrada. Esta función, por lo tanto, es equivalente a escribir un comando en el REPL.

- `jl_call`: permite llamar desde a funciones implementadas en Julia. Estas funciones deberán hacerse "visibles" previamente en C++, mediante órdenes como `jl_get_global`. Dependiendo del número de argumentos de entrada necesarios, esta función podrá tener las siguientes formas: `jl_call_0`, `jl_call_1`, `jl_call_2`, etc. Para pasar más de tres argumentos, se utilizará simplemente `jl_call` y se pasarán los argumentos en forma de *array*, indicando, además, el tamaño de este *array* [16].

Adicionalmente, esta API también define nuevos tipos de datos. Entre ellos:

- `jl_value_t`: permite guardar, en forma de puntero, cualquier elemento de Julia, ya sea una variable, una función o incluso un módulo.

- `jl_function_t`: permite guardar una función de Julia en forma de puntero. Guardando las funciones en este formato será posible usar posteriormente la función `jl_call`.

Existen algunos tipos de datos y funciones más que pueden ser útiles, por ejemplo, para definir un *arrays* en un formato adecuado para Julia, y cuya implementación se estudiará en la sección 3.3.3.

[2] Véase paquete como una colección independiente y reutilizable de código Julia, envuelta en un módulo [16]. La manipulación de paquetes puede realizarse mediante el gestor de paquetes de Julia, accesible pulsando la tecla `]` desde el REPL o mediante la directiva `using Pkg`

2.4. Simulación de MRI

2.4.1. Herramientas de simulación de MRI en la actualidad

En la actualidad, existe una gran variedad de herramientas disponibles para la simulación de MRI. Algunas de ellas, como las descritas en [17] y [18], fueron creadas con el objetivo de enseñar conceptos teóricos de resonancia magnética, como la ecuación de Bloch (sección 2.2.1) o conceptos de mecánica cuántica. No obstante, estos programas no contaban con fines médicos directos [2]. Otros simuladores, por ejemplo, Virtual MRI [19], pueden emular algunos de los principales elementos del proceso de adquisición en un escáner de MR real. El simulador Torheim [20] tiene un comportamiento muy similar al que acabamos de mencionar, y tiene, igualmente, fines didácticos [2].

En cuanto a los simuladores orientados a la investigación, debemos hablar de JEMRIS [21], un simulador de código abierto enfocado, como el aquí presentado, al desarrollo de secuencias. Considera muchas propiedades de interés en MR, para ser lo más realista posible; sin embargo, presenta el inconveniente de que sólo utiliza ejecución multihilo en CPU. Otras alternativas son BlochSolver [22], MRISIMUL [23] –ambos de código cerrado– o MRILab [24], uno de los simuladores de MRI más recientes.

2.4.2. Análisis del simulador `KomaMRI.jl`

`KomaMRI.jl`[3] es un simulador que pretende ser rápido, fácil de usar, extensible, de código abierto y multiplataforma. Este simulador ha sido desarrollado en Julia, por lo que incluye precompilación y soporte para GPU; lo primero debería hacer que `KomaMRI.jl`sea tan eficiente como otros simuladores escritos en lenguajes C/C++ y lo segundo hace posible una ejecución rápida.

Esta sección pretende describir los principios de funcionamiento del simulador `KomaMRI.jl`, de manera que se pueda comprender una de las formas que existen de recrear computacionalmente el fenómeno de la resonancia magnética.

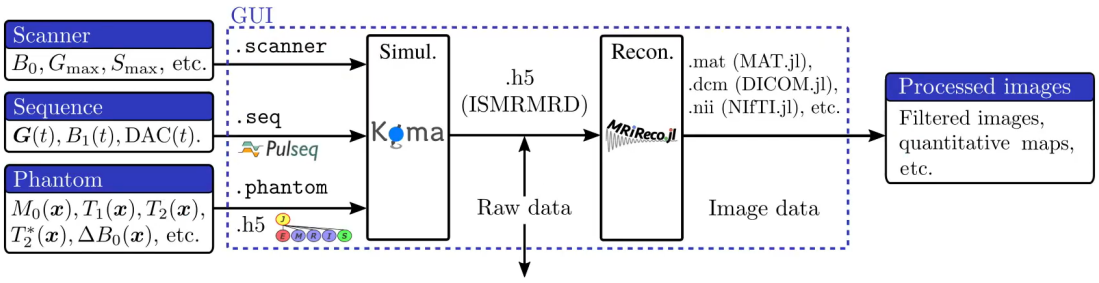


Figura 2.18: Esquema de funcionamiento general del simulador `KomaMRI.jl`[3]

Módulo `KomaMRI`

`KomaMRI.jl` es un módulo de Julia. La idea general de funcionamiento del simulador se muestra en la figura 2.18. En ella se puede ver un diagrama completo de toda la aplicación, incluyendo los parámetros de entrada, el simulador propiamente dicho, la interfaz gráfica y un módulo adicional llamado `MRIReco.jl` [25], con el que se convierten las señales en crudo a imágenes mediante un proceso llamado reconstrucción. Distinguimos tres parámetros de entrada:

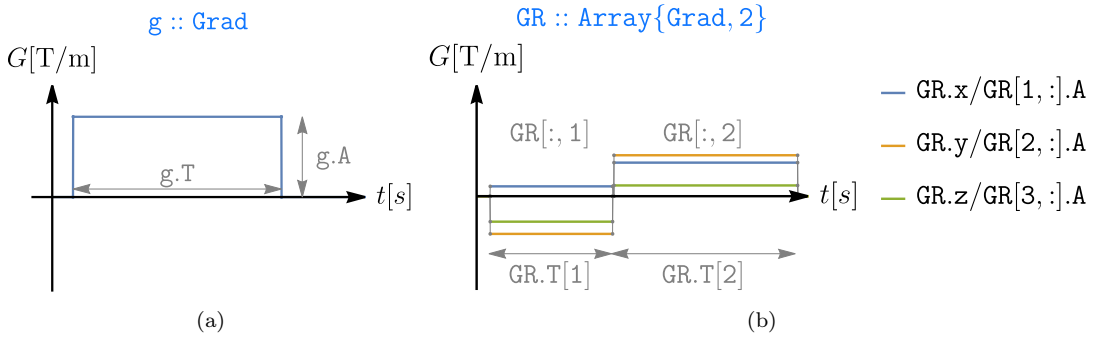
- `Scanner`: se corresponde con la estructura de datos que contiene los parámetros globales de la simulación, tales como la intensidad del campo magnético estático B_0, el tiempo de rasterización Δt o el *slew-rate* máximo, entre otros. La información de esta estructura se guarda en ficheros con extensión `.scanner`.

- `Phantom`: esta estructura simula lo que en un escáner real sería el cuerpo (o parte del cuerpo) del paciente. Contiene los datos (tiempos T_1, T_2, T_2^* de cada espín y densidad de protones ρ,

entre otros) sobre los que se aplica la secuencia. En `KomaMRI.jl`, los datos correspondientes a un `Phantom` se almacenan en ficheros con extensión `.phantom` o `.h5`.

- `Sequence`: contiene la información de la secuencia MRI propiamente dicha (gradientes, pulsos RF y ADCs, principalmente). Esta información puede ser almacenada en ficheros con extensión `.seq`. Una secuencia está compuesta por tres estructuras principales, cada una de ellas definida en un fichero (`.jl`) diferente:

  1. **Fichero** `Grad.jl`: en este fichero se implementa la estructura `Grad`. En el simulador, los gradientes consistirán en una concatenación de instancias de esta estructura. Esto quiere decir que `Grad` es simplemente un bloque de construcción del gradiente y, por tanto, de una secuencia. La estructura `Grad` consta, esencialmente, de dos valores escalares reales[3]: `A`, correspondiente a la amplitud, y `T`, que hace referencia a la duración temporal del gradiente. La figura 2.19a muestra la representación gráfica de esta estructura.



(a) (b)

Figura 2.19: Representación gráfica de (a) una instancia de `Grad` y (b) un *array* bidimensional formado por instancias de esta misma estructura

En cuanto a la figura 2.19b, es importante entender que en una secuencia puede haber activados simultáneamente gradientes en las tres direcciones del espacio, y que la intensidad (o amplitud) de estos gradientes puede variar con el tiempo. Para implementar esta idea en el simulador, se recurre a matrices formadas por elementos de la "clase" (en realidad, es una estructura) `Grad`, de forma que cada una de las filas se corresponda con una de las direcciones del espacio (x, y o z) y cada columna corresponda a un instante o intervalo de tiempo determinado. Por tanto, la matriz correspondiente a esta figura 2.19b podría ser la siguiente:

$$\texttt{GR::Array\{Grad,2\}} = \begin{bmatrix} \texttt{Grad}(1,1) & \texttt{Grad}(3,2) \\ \texttt{Grad}(-4,1) & \texttt{Grad}(4,2) \\ \texttt{Grad}(-3,1) & \texttt{Grad}(1,2) \end{bmatrix} = \begin{bmatrix} \texttt{GR.x} \\ \texttt{GR.y} \\ \texttt{GR.z} \end{bmatrix} \tag{2.32}$$

En este fichero también se definen varios constructores para la estructura `Grad`, así como funciones auxiliares que nos permiten realizar operaciones sobre los gradientes, tales como multiplicaciones por un escalar, rotaciones, concatenaciones, etc.

  2. **Fichero** `RF.jl`: en él se define la estructura `RF`, que consta de varios atributos: el primero, `A`, se corresponde con la amplitud de la envolvente compleja del pulso RF, por lo que tendrá la forma $(B_{1x} + jB_{1y})$; `T`, al igual que en la estructura `Grad`, corresponde a la duración, en segundos, del pulso. El parámetro `Δf`, añadido en una de las últimas versiones, da cuenta de la diferencia de frecuencia, con respecto a la frecuencia de Larmor, del pulso RF. Esto, tal y como mostraba la ecuación 2.23, permitía seleccionar un slice separado del punto $(0, 0, 0)$. Por último, el parámetro `delay` permite añadir un retardo al pulso RF. La figura 2.20 muestra una representación gráfica de la estructura `RF`, junto con la representación de un *array* bidimensional formado por objetos de esta clase. En este caso, mientras que la

[3] en las últimas versiones se han incluido más parámetros para poder dar forma trapezoidal a los gradientes, pero se prescindirá de ellos por simplicidad

dimensión horizontal sigue reprensentando el tiempo, la vertical ya no corresponde al espacio (a cada una de las dimensiones), sino que corresponde al número de bobinas, es decir, cada fila corresponde a una bobina. Por lo tanto, puede haber simultáneamente varios pulsos RF activados, cada uno generado por una bobina.

Además de `RF`, en este fichero también se define la estructura `Spinor`, útil para llevar a cabo la rotación del vector de magnetización.

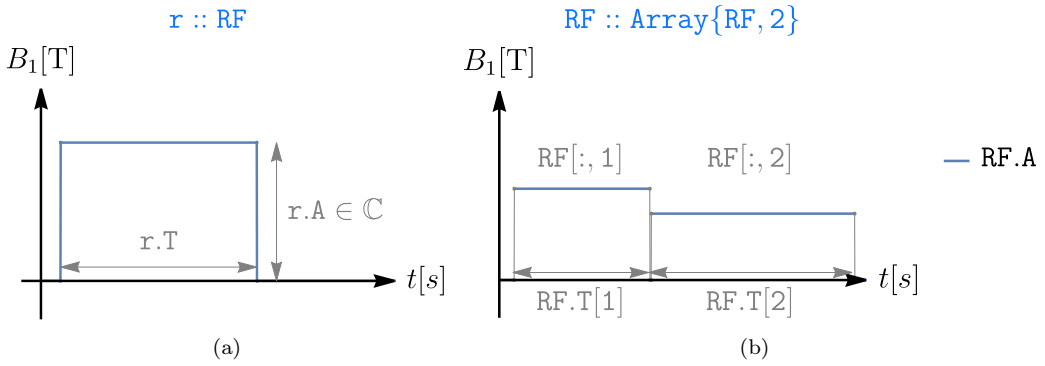


Figura 2.20: Representación gráfica de (a) una instancia de `RF` y (b) un *array* bidimensional formado por instancias de esta misma estructura

3. **Fichero** `ADC.jl`: se crea la estructura `ADC`, que indica los instantes de tiempo en los que tiene lugar la lectura. Los dos parámetros principales de esta estructura son `T`, que es la duración del intervalo, y `N`, que da cuenta del número de muestras (equiespaciadas en el tiempo) que deben adquirirse durante este intervalo de lectura.

   La función `get_sample_times` proporciona los instantes de tiempo absolutos (con respecto al comienzo de la secuencia) en los que se lleva a cabo la lectura. La función recibe como entrada una secuencia y obtiene la duración temporal de cada objeto `ADC` presente en ella. De esta forma, se devuelven los instantes de tiempo en los que se toma cada muestra.

   Además, es importante que cada secuencia tenga al menos un objeto `ADC` de amplitud (`N`) nula, para poder así localizar correctamente los tiempos de muestreo con respecto al inicio de la secuencia.

   En la figura 2.21 se representan gráficamente una instancia de la estructura `ADC` y un *array* 1D formado por instancias de este mismo tipo. La dimensión del *array* es el tiempo.

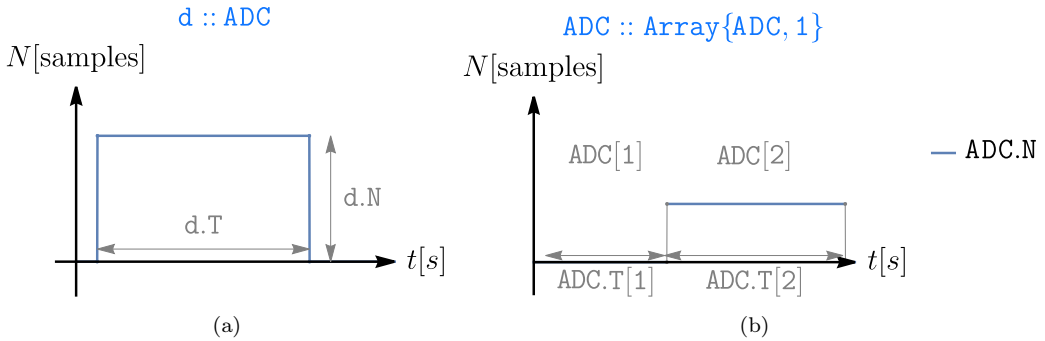


Figura 2.21: Representación gráfica de (a) una instancia de `ADC` y (b) un *array* unidimensional formado por instancias de esta misma estructura

Vistos los tres elementos principales que componen una secuencia, podemos fijarnos, ahora, en la definición de la estructura secuencia dentro del fichero `Sequence.jl`. Esta estructura está formada por:

- Un *array* 2D de elementos `Grad`, tal y como se ha visto y explicado a partir de la figura 2.19b.
- Un *array* 2D de elementos `RF`, como el mostrado en la figura 2.20b.
- Un *array* 1D de elementos `ADC`, como el mostrado en la figura 2.21b.
- Un vector `DUR`, que contiene las duraciones de cada bloque[4]. Esto permite crear retardos después de los pulsos RF para satisfacer así los denominados *ring-down times*[5].
- Una estructura diccionario `DEF`, que contiene información relevante para la reconstrucción.

En el fichero `Sequence.jl` también se definen varios constructores para la estructura, dependiendo de los parámetros de entrada que se introduzcan. También se implementan operaciones aritméticas básicas, como la concatenación[6], el producto por un escalar (con el que quedarían multiplicados solo los gradientes de la secuencia) o el producto por una matriz (con el que se pueden aplicar rotaciones a las secuencias), entre otras. Por último, nos encontramos con un conjunto de funciones auxiliares que pueden proporcionarnos información de una secuencia (obtener sus gradientes, sus pulsos RF, saber si la lectura está activada, obtener la trayectoria en el espacio K generada, etc.).

Debemos comprender que la estructura `Sequence` es, desde el punto de vista conceptual, un elemento de la secuencia de entrada al simulador. Esto quiere decir que la secuencia que se aplica al simulador consiste en una concatenación de elementos `Sequence`, pero también es, en sí misma, un elemento `Sequence`. Estaremos en este enlace

Módulo PulseDesigner

Una vez definidas las principales estructuras de datos, vamos a hacer un breve análisis del módulo `PulseDesigner` del simulador, ya que contiene funciones que nos van a resultar útiles de cara al editor de secuencias que diseñaremos. Este módulo nos permite crear determinadas secuencias (o parte de ellas) de una manera más rápida y directa que si lo hiciésemos 'manualmente". Destacamos las siguientes tres funciones:

- `RF_hard`: crea un objeto secuencia con gradientes de amplitud nula (por defecto) y un elemento `RF`. Si se desea, se puede definir un gradiente distinto de cero, que influirá en la selección de `slice`. De igual manera, se puede elegir un `Δf` distinto de cero, para que el pulso RF excite un `slice` separado del origen.
- `EPI`: esta función da lugar a una secuencia EPI cartesiana. Uno de los parámetros importantes es `Δt`, el periodo de muestreo, obtenido a partir objeto de la clase `Scanner` (recordemos que esta estructura de datos contenía parámetros globales de la simulación) pasado como argumento de entrada a la función. Para que la señal sea muestrada correctamente debemos garantizar que se cumple el teorema de Nyquist. Para saber qué implicación tiene esto, supongamos un gradiente constante con amplitud `Ga`; la frecuencia máxima de la envolvente compleja se dará cuando estemos en los bordes del espacio K. Esto puede expresarse tal que así (asumiendo que x se recorre de $-\frac{FOV}{2}$ a $\frac{FOV}{2}$):

$$\exp\left(j2\pi\left[\gamma G_a \frac{FOV}{2}\right]t\right)$$

El teorema de muestreo establece:

$$\frac{1}{\Delta t} \geq 2f_{max} = 2\gamma G_a \frac{\mathrm{FOV}}{2} \tag{2.33}$$

[4]En el simulador, se entiende por bloque al conjunto de elementos de una secuencia que ocurren durante el mismo intervalo de tiempo. Se puede acceder a cada bloque de la secuencia mediante su índice, como si de un *array* unidimensional se tratase: `seq[i]`, siendo `seq::Sequence`.

[5]Los pulsos RF de excitación utilizados en resonancia magnética producen una respuesta transitoria en la bobina del receptor que provoca lo que se conoce como respuesta *ring-down*. Esto suele requerir de un breve retardo tras la excitación, para evitar así la sobrecarga del receptor

[6]La concatenación de secuencias se implementa con el símbolo `+`. Así, al realizar, por ejemplo, la operación `seq1+seq2`, estaríamos concatenando estas dos secuencias una detrás de la otra.

$$G_a \leq \frac{1}{\gamma \cdot \Delta t \cdot \text{FOV}} \tag{2.34}$$

Lo cual fija un valor máximo para el gradiente. Dentro de la función `EPI` se comprueba si `Ga` (obtenido con un signo de igualdad en la desigualdad anterior) es mayor que el valor máximo permitido para el gradiente `Gmax`, extraído, al igual que `Δt`, del parámetro de entrada `sys::Scanner`.

Con respecto a cómo se construye la secuencia:

- El espacio K consta de `Ny` líneas (en la dirección de codificación de fase) y `Nx` puntos por línea (en la dirección de codificación de frecuencia). Dentro de la función, se "fuerza" a que estos dos valores coincidan, pero en realidad no tendrían por qué coincidir.
- `Δt` es el periodo de muestreo, es decir, la diferencia temporal entre dos muestras consecutivas en el espacio K. El tiempo que lleva escribir una línea en la dirección de codicación de frecuencia es

$$\texttt{Ta} = (\texttt{Nx} - 1)\Delta\texttt{t} \tag{2.35}$$

- En la dirección de codificación de fase, el tiempo que toma cubrir el espacio K es también `Ta`. Por tanto, como se necesitan `Ny-1` saltos para cubrir `Ny` líneas, el tiempo que lleva moverse de una línea a la siguiente será

$$\Delta\tau = \texttt{Ta}/(\texttt{Ny} - 1) \tag{2.36}$$

La secuencia en sí consiste en un vector de gradientes en las direcciones x (lectura) e y (codificación de fase); estos se concatenan verticalmente usando la función `vcat`. Como tenemos que poblar `Ny` líneas, necesitamos, como se ha indicado anteriormente, `Ny-1` saltos para movernos entre líneas. Por tanto, el contador `i` tiene `2Ny-1` valores, yendo desde 0 hasta `2Ny-2`; sus valores pares corresponderán a líneas de lectura, mientras que los impares serán saltos entre líneas. Dentro de las líneas pares, los múltiplos de 4 (incluido el cero) se desplazarán de izquierda a derecha, mientras que los demás lo harán de derecha a izquierda. Entonces:

- En cuanto a `Gx`, si `i` es par, el gradiente será igual a `Ga` durante `Ta` segundos y su signo irá alternándose cuando se avance por líneas de lectura consecutivas. Cuando `i` es impar, no se realiza ninguna lectura, por lo que solo se espera (con un gradiente nulo) durante $\Delta\tau$ segundos.
- En cuanto a `Gy`, cuando `i` es par (es decir, estamos en modo lectura), simplemente esperamos con un gradiente de amplitud cero durante `Ta` segundos, de forma que la lectura se haga en la dirección horizontal. Cuando `i` es impar, se activa el gradiente con intensidad `Ga` durante $\Delta\tau$, obteniéndose así un gradiente denominado *blip*.
- El *array* `ADC` se pone a cero para los valores impares de `i` (es decir, durante los *blips*), mientras que se activa para los valores pares, con duración `Ta` y `N=Nx`.
- La sequencia queda totalmente definida tras añadir un `PHASE` antes de la adquisición, para desplazar el puntero desde el centro hasta la esquina inferior izquierda del espacio K (ver transición desde el punto A al punto B en la figura 2.17), y un `DEPHASE` justo al final, que desplaza el puntero de vuelta al $(0, 0)$ desde la esquina superior derecha (en este caso, sería imprescrindible que el número de líneas leídas, `Ny`, fuese impar, para así terminar la lectura en la esquina superior derecha y no en la superior izquierda).

Por último, se calcula el ancho de banda por píxel en las dos direcciones de codificación (solo con fines informativos).

- `radial_base`: la función `radial_base` define una secuencia para el muestreo radial del espacio K. En ella, solo se activa un gradiente, el de la dirección de lectura, no hay gradiente que proporcione la pendiente de cada radio. Por tanto, la secuencia tiene que complementarse con una rotación alrededor del eje z para proporcionar esta pendiente.

La función establece los parámetros (similares a los de la función `EPI` en la dirección de lectura) que afectan al comportamiento de los radios. También se define el número de radios necesarios para que se cumpla el criterio de Nyquist en la dirección radial. La secuencia se complementa con dos etapas de refase para ubicar el punto de lectura en el lugar correcto. La función devuelve la secuencia, la separación angular entre radios consecutivos y el tiempo necesario para la lectura dentro del radio.

Como se ha indicado anteriormente, la secuencia funcionará siempre que haya una rotación explícita entre las llamadas a esta función.

2.5. Desarrollo de aplicaciones gráficas

2.5.1. El marco Qt

Qt [5] es un *framework* orientado al desarrollo de interfaces gráficas de usuario multiplataforma (aplicaciones de escritorio, aplicaciones web o aplicaciones móviles, entre otras). No es un lenguaje de programación en sí mismo, sino que se corresponde con un conjunto de clases, métodos, funciones y bibliotecas escritas en C++. Se utiliza un preprocesador, el MOC (*Meta-Object Compiler*) para ampliar el lenguaje C++, añadiendo así funcionalidades como las denominadas *signals & slots* [26]. Antes del paso de compilación, el MOC analiza los ficheros fuente escritos en C++ con extensiones de Qt y genera a partir de ellos ficheros fuente compatibles con el estándar C++.

Señales y *Slots*

En la programación de interfaces de usuario, cuando se modifica el estado de un objeto[7], a menudo se busca que otro objeto sea notificado. En general, queremos que los objetos de cualquier tipo puedan comunicarse entre sí. Por ejemplo, si un usuario hace clic en un botón de cierre, probablemente queramos que se llame a la función close() de la ventana [26].

Otras tecnologías emplean los denominados *callbacks*[8], pero en Qt se utilizan las denominadas *signals & slots*. Una señal es emitida cuando ocurre un determinado evento. Los objetos en Qt, concretamente, los pertenecientes a la clase `QObject`, cuentan con señales predefinidas, pero también se les puede añadir las señales que se deseen. Un *slot* es una función que es llamada en respuesta a una señal concreta. Al igual que con las señales, los objetos en Qt también poseen *slots* predefinidos, existiendo la opción de añadir los que se deseen [26].

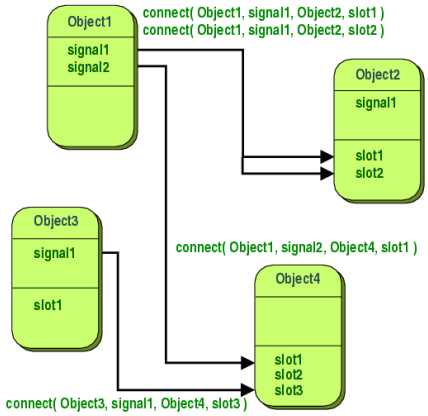


Figura 2.22: Diagrama en el que se muestra la conexión de varios objetos en Qt (derivados de la clase `QObject`) mediante señales y *slots* [26]

[7] Véase objeto como cualquier elemento de la interfaz: un botón, una caja de texto, una imagen...

[8] Un *callback* es un puntero a una función. Si se quiere que una determinada función en curso notifique algún evento, se pasa el puntero a otra función (el *callback*) a esa función en curso. La función en curso, entonces, llamará a la función apuntada por el *callback* cuando sea conveniente

Las conexiones entre objetos se realizan mediante la directiva `connect`, tal y como muestra la figura 2.22. De esta forma es posible conectar la misma señal a dos o más slots distintos, o hacer que un objeto pueda recibir varias señales provenientes de distintos objetos. De la misma manera, es posible que un objeto pueda enviar varias señales a objetos diferentes.

Módulos en Qt

En función de cómo se quiera afrontar el diseño de la aplicación gráfica y de los requisitos de la aplicación, se puede recurrir a distintos módulos de Qt. En caso de querer diseñar la interfaz directamente en C++, se puede usar el módulo `Qt Widgets`, que proporciona un conjunto de elementos (clases) que pueden ser utilizados para diseñar la interfaz. Los Qt Widgets son los elementos de interfaz de usuario típicos de los entornos de escritorio, y se integran bien en la plataforma subyacente, proporcionando un aspecto nativo en Windows, Linux y macOS. Los widgets tienen la ventaja de ser elementos robustos y ricos en características, adecuados para la mayoría de interfaces de usuario tradicionales [26]. Por este motivo, son una buena opción si se pretende crear grandes aplicaciones de escritorio, y son menos adecuados para crear aplicaciones interactivas orientadas, por ejemplo, a dispositivos móviles.

El módulo `Qt Widgets` incluye un conjunto de widgets (controles) básicos, listos para ser usados de forma directa. Ejemplos de estos widgets y sus respectivas clases pueden ser los botones (`QPushButton`, `QToolButton`), las *combobox* (`QComboBox`) o las barras de desplazamiento (`QScrollBar`), entre otras muchas clases disponibles. La comunicación entre widgets se realiza mediante los señales y *slots*, previamente explicados.

La otra opción a la hora de diseñar una interfaz de usuario es el uso del módulo `Qt Quick`, destinado a la creación de interfaces multiplataforma más fluidas y dinámicas. Es el módulo utilizado para escribir aplicaciones en QML, un lenguaje declarativo muy sencillo de utilizar. El funcionamiento básico es el siguiente: la lógica del programa residirá en C++. De hecho, en C++ es donde se implementará la función `main()`, desde la que se llamará a las funciones de Qt, y desde donde se cargará el fichero QML. Será en este fichero QML en el que se definirán todos los aspectos relacionados con la interfaz propiamente dicha (apariencia, disposición de ventanas y menús, colores, animaciones...). Un ejemplo básico de esta implementación puede verse en las figuras 2.23 y 2.24a. El resultado es el mostrado en la figura 2.24b. Tal y como se puede observar en el código, la carga del fichero `main.qml` se realiza mediante un objeto de la clase `QQuickView`, que proporciona una ventana en la que mostrar la interfaz de usuario hecha con `Qt Quick`. La llamada al método `exec()` de la clase `QGuiApplication` hace que se genere un bucle infinito, operativo siempre y cuando la aplicación esté en funcionamiento, y que escuchará todas las posibles interacciones que el usuario pueda tener con la interfaz.

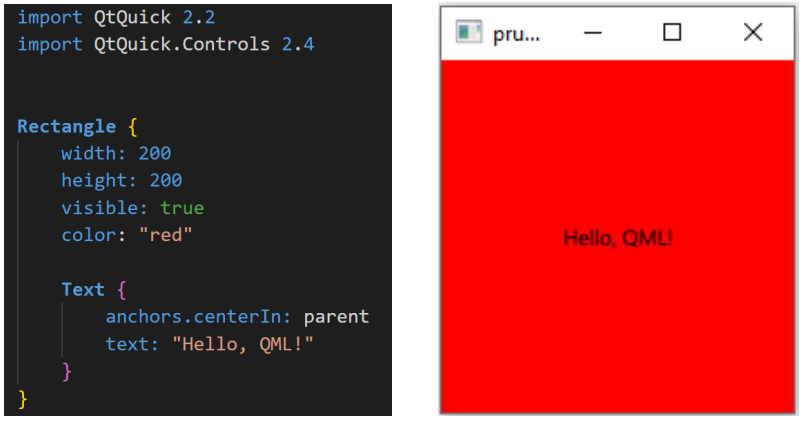
```cpp
#include <QGuiApplication>
#include <QQuickView>


int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);

    QQuickView view;
    view.setSource(QUrl("qrc:/main.qml"));
    view.show();

    return app.exec();
}
```

Figura 2.23: Fichero main.cpp

2.5.2. QML

QML (*Qt Meta Language*) es un lenguaje multi-paradigma utilizado para la creación de aplicaciones con alto grado de dinamismo. Con QML, cada uno de los bloques que componen la aplicación (ej. componentes de la interfaz de usuario) deben ser declarados, así como sus propiedades. Su sintaxis puede recordar en un primer momento a una mezcla entre HTML y CSS, ya que su estructura es bastante similar a un lenguaje de marcado, pero a su vez nos permite definir directamente cada una de las características de estilo.

Adicionalmente, QML incluye un sistema de control de eventos mediante Javascript. A partir de este lenguaje se puede controlar todo el comportamiento y la lógica de la interfaz. Es importante recalcar la diferencia entre la lógica de programa, que como ya se ha mencionado, reside en C++, y la lógica de la interfaz, relacionada con la manipulación de eventos de interacción del usuario con la interfaz, y que residirá en Javascript. Por tanto, QML se puede interpretar como una combinación de los tres lenguajes de programación web más típicos: HTML, CSS y Javascript.



(a) (b)

Figura 2.24: (a) Fichero QML y (b) su correspondiente resultado

Conceptos básicos de la sintaxis QML

Sintácticamente, un bloque en QML define un árbol de objetos. Los objetos se definen mediante declaraciones de objeto, que describen tanto el tipo de objeto como sus atributos. Cada objeto también puede declarar objetos hijo mediante declaraciones anidadas. La declaración de un objeto consiste en el nombre de su tipo, seguido de una pareja de llaves. Todos los atributos u objetos hijo se definirán en el interior de esas llaves. De esta forma, como se ha mencionado anteriormente, nos encontramos ante una sintaxis legible y fácil de interpretar, que puede ser comprendida sin prácticamente conocimientos previos de programación.

También podemos controlar la lógica de la interfaz, añadiendo funcionalidades como métodos, manejadores de eventos o los denominados *property bindings* de QML. Es aquí donde Javascript entra en juego.

A partir de un ejemplo podemos ver cada una de estas tres funcionalidades: la figura 2.25 muestra la implementación de un objeto botón. Este objeto, al igual que muchos otros en QML, tiene la propiedad de emitir señales cuando se produce determinada acción sobre él. Esas señales son escuchadas y, cuando se producen, pueden ejecutar un determinado fragmento de código que esté situado dentro de un manipulador de eventos. En nuestro caso, la señal se producirá al hacer clic sobre el botón, por lo que el manipulador de eventos será `onClicked`. Vemos que, cuando se produce este evento, se hace una llamada a la función `disp()`, cuya definición se encuentra fuera del objeto botón.

En cuanto a las *property bindings*, la línea 3 de la figura nos muestra un ejemplo de su uso. Se puede observar cómo la propiedad del ancho del botón está relacionada con el ancho del texto contenido en su interior. De esta forma, el botón tendrá la misma anchura que el texto. Sin embargo, lo especial de esta relación entre las propiedades de los objetos está en que, si durante el tiempo de ejecución

cambiase el texto contenido en el botón, la anchura del botón cambiaría con él de forma automática. De alguna forma, podríamos considerar que esas dos propiedades están ahora "ancladas".

```
1  Button{
2      height: 20
3      width: buttonText.width
4      Text{
5          id: buttonText
6          text: "Click Me!"
7      }
8      onClicked: {
9          disp();
10     }
11 }
12
13 function disp(){
14     console.log("Clicked");
15 }
```

Figura 2.25: Implementación de un botón con QML

Modelos de datos en QML

Para almacenar datos en QML se recurre a los modelos de datos. El más importante es `ListModel`, una estructura que permite guardar los datos en forma de lista y que funciona como un contenedor de definiciones de la clase `ListElement`. En cada instancia de `ListElement` se almacena la información referente a un elemento de la lista. Esta información se define en forma de los denominados *roles* [26], de una manera similar a la utilizada para definir las propiedades de cualquier objeto QML (ver figura 2.24a).

La estructura `ListModel` cuenta con una serie de métodos que permiten obtener y manipular su contenido; destacan `get()`, `append()`, `insert()`, `move()` y `set()`. Es posible, además, referirse a cada elemento de la lista mediante su índice, que vendrá dado por la posición absoluta en la que se encuentra el elemento dentro de la lista. Así, si se quiere, por ejemplo, obtener el primer elemento de un `ListModel` con identificador `list`, se puede recurrir a la siguiente directiva:

```
element = list.get(0)
```

De forma que el primer elemento de la lista quede guardado en el objeto `element`, y sus propiedades puedan ser accedidas con un punto detrás de su nombre (por ejemplo, `element.name`).

Para poder visualizar el contenido de un `ListModel`, se recurre a las vistas (`View`) de QML, que son contenedores de colecciones de elementos. Estas vistas son ricas en funciones y se pueden personalizar para cumplir con los requisitos de estilo o comportamiento deseados [26]. Además, permiten mostrar los datos en forma de lista (`ListView`) o en forma de cuadrícula (`GridView`).

Entre los atributos configurables de un `View`, destaca el denominado `Delegate`, que dicta cómo deben aparecer los datos en la vista. El `Delegate` toma cada dato del modelo y lo encapsula. Los datos de cada elemento del modelo son accesibles desde el `Delegate`. Asimismo, desde el `Delegate` también se pueden escribir datos en modelos editables [26]. La figura 2.26 da cuenta de la relación entre los datos, el modelo, la vista y el `Delegate`.

2.5.3. Compilación de aplicaciones basadas en el entorno Qt

Los principales sistemas de construcción que soporta Qt son los siguientes:

- **qmake**: es el sistema creado por la propia compañía de Qt. Proporciona un sistema orientado a proyectos para gestionar el proceso de construcción de aplicaciones, bibliotecas y otros componentes. Este enfoque da al programador control sobre los ficheros de origen utilizados, y permite que cada uno de los pasos del proceso se descriva de forma concisa, normalmente dentro de un único fichero [26].

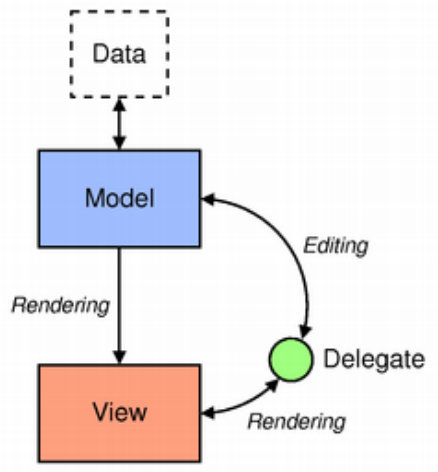


Figura 2.26: Relación entre los elementos que permiten la visualización de datos en QML [26]

Con qmake, los proyectos son descritos mediante un fichero de proyecto (`.pro`), que es utilizado para generar los ficheros estándar, tales como el *Makefile*. Los ficheros de proyecto suelen contener una lista de ficheros fuente y de cabecera, información de configuración general y cualquier detalle específico de la aplicación, como una lista de bibliotecas con las que enlazar, o una lista de adicional de rutas de inclusión (*include paths*).

Este sistema presenta ventajas como su alta compatibilidad con Qt –dado que se creó específicamente para este lenguaje– y su sintaxis sencilla, que facilita su comprensión. Sin embargo, también presenta puntos débiles:

- No existe documentación suficiente. Al ser un sistema relativamente reciente, aun no se dispone de documentación lo suficientemente amplia.
- Exclusividad para Qt: qmake no resulta apropiado cuando se quieren realizar proyectos grandes que combinan varias tecnologías.

- **CMake** [27]: es una herramienta multiplataforma de generación o automatización de código, diseñada para ser utilizada junto con el entorno nativo de compilación (como GCC o MinGW). Con CMake, los ficheros de configuración simples (`CMakeLists.txt`) son utilizados para generar los ficheros estándar que usa el compilador, como pueden ser los ficheros *Makefile* en Unix.

  CMake cuenta con las siguientes ventajas:

  - Mayor cantidad de información y documentación disponible, al ser un sistema ampliamente utilizado y que lleva instaurado ya más de dos décadas
  - Mayor compatibilidad con otros lenguajes y tecnologías

  No obstante, CMake también presenta desventajas, como la complejidad de su sintaxis, que conlleva a una mayor dificultad para su aprendizaje.

2.5.4. El estándar WebAssembly

WebAssembly [28] es un estándar abierto del consorcio de la World Wide Web (W3C), y define un formato de código binario portátil para programas ejecutables, un lenguaje de ensamblado textual correspondiente e interfaces para facilitar las interacciones entre dichos programas y su entorno de hospedaje. El código WebAssembly se ejecuta en una máquina virtual de bajo nivel, que imita la funcionalidad de los muchos microprocesadores sobre los que se puede ejecutar.

El principal motivo por el que este estándar genera interés es el deseo de ejecutar un código de computación más intensivo en los navegadores. Los usuarios, en general, están pasando cada vez más tiempo en un navegador, y menos haciendo uso de aplicaciones locales. Esta tendencia ha creado una urgencia en torno a la eliminación de barreras para ejecutar una amplia gama de programas dentro del navegador. Con esta motivación se concibió originalmente WebAssembly.

Compilación del módulo C/C++ en WebAssembly

Cuando se escribe código en C++ y se desea su ejecución en un navegador web, se necesitan las siguientes herramientas:

- Navegador con soporte para WebAssembly. Actualmente, Firefox, Chrome, Safari y Edge cuentan con este soporte.
- Compilador de C/C++ para WebAssembly. El más utilizado es Emscripten [29].
- Compilador para C/C++ (GCC en Linux).
- Servidor web local sencillo en el que alojar los ficheros resultado de la compilación.

Disponiendo de estas herramientas, se puede compilar en WebAssembly con órdenes del tipo `emcc`, con las que se podría generar un fichero `.wasm` (la extensión de WebAssembly) y un fichero (`index.js`), que cargará el fichero `.wasm` y hará de intermediario entre WebAssembly y el navegador. Adicionalmente, habría que crear un fichero `index.html` que cargase el fichero Javascript y al cual accederíamos desde el navegador.

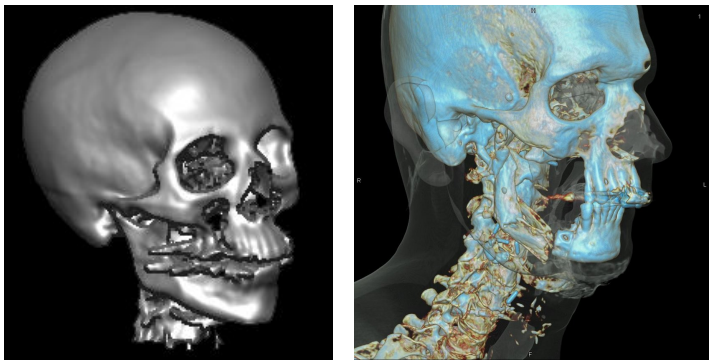
Lo anteriormente mencionado no resulta ser una tarea compleja, pero existen herramientas que pueden facilitarla aún más. Concretamente, Qt Creator el entorno de desarrollo creado por el propio equipo de Qt, contiene un kit para la compilación en WebAssembly que crea automáticamente los ficheros `.wasm`, `.js` y `.html`. La única labor ahora será, por tanto, la de arrancar un servidor web –por ejemplo, con la orden `python -m SimpleHTTPServer`– en el directorio en el que se hayan generado estos ficheros. Después, simplemente se deberá acceder a ese servidor desde el navegador, a través de la url `http://localhost:8000`.

2.6. Visualización de volúmenes y cortes en 3D

El principal problema de la visualización de datos volumétricos es el de mostrar los datos tridimensionales como una imagen en 2D sin perder demasiada información [30]. El proceso que permite esta visualización de objetos 3D en una superficie plana –es decir, en una pantalla– se denomina *rendering*.

Antes de entrar a describir algunas de las herramientas más utilizadas, se van a presentar muy brevemente los dos métodos principales de *rendering* de volúmenes en tres dimensiones:

- ***Surface Rendering***: este método consiste básicamente en construir superficies poligonales a partir de los conjuntos de datos y realizar un *rendering* de estas superficies [30]. En el caso de un volumen obtenido a partir de una matriz, podríamos, por ejemplo, extraer todos aquellos puntos que tuviesen el mismo valor y unirlos posteriormente mediante superficies poligonales. De esta forma obtendríamos un modelo 3D definido por una única superficie (denominada *isosurface*) y que podría ser, por ejemplo, impreso en 3D. La figura 2.27a muestra un ejemplo de *surface rendering*.
- ***Volume Rendering***: por otra parte, esta técnica consiste en asignar un color y un valor de opacidad a cada elemento de datos y proyectar los elementos directamente sobre el plano de la imagen, sin utilizar polígonos [30]. Este método, comparado con el `surface rendering`, es bastante más costoso a nivel computacional, ya que implica tener que realizar un *rendering* de muchos más puntos. No obstante, permite visualizar varias "capas" del volumen de forma simultánea, debido a la opacidad variable que acabamos de mencionar. La figura 2.27b muestra un *rendering* 3D realizado mediante esta técnica.



(a) (b)

Figura 2.27: Renderizado 3D mediante (a) *Surface Rendering* y (b) *Volume Rendering* (obtenido a partir de tomografía computarizada (CT)) [31]

De las múltiples herramientas que existen para realizar *rendering*, vamos a destacar, por su amplia utilización en todo tipo de programas y sus facilidades de uso:

- **VTK (*The Visualization Toolkit*)** [32]: se trata de una API de código abierto para C++ (aunque también existen versiones para lenguajes como Java o Python) consistente, principalmente, en una biblioteca de clases que permite la visualización de gráficos 3D, el *rendering* de volúmenes, la visualización científica y el trazado de gráficos 2D. Admite una gran cantidad de algoritmos de visualización y técnicas de modelado avanzadas, y aprovecha el procesamiento paralelo basado en hilos y la memoria distribuida para obtener escalabilidad y mayor velocidad de cálculo.
- **PlotlyJS.jl**: este paquete de Julia sirve como interfaz para la biblioteca de visualización plotly.js, es decir, aprovecha esta biblioteca de Javascript para construir gráficos. El paquete está orientado a la visualización, tal y como indica su nombre, de gráficos y diagramas.

Capítulo 3

Análisis, Diseño e Implementación

3.1. Análisis de los requisitos de la aplicación

Los requisitos recogen las necesidades y condiciones que han ser cubiertas por la aplicación para poder cumplir con una determinada especificación. Su definición es necesaria para el correcto enfoque de las fases posteriores del proyecto.

3.1.1. Requisitos funcionales

Los requisitos funcionales (FRQ) de un sistema describen lo que el programa debe hacer. Estos requisitos dependen del tipo de programa que se desarrolle y de lo que necesiten de él sus posibles usuarios.

Tal y como se introdujo en la sección 1.3, los requisitos funcionales son los siguientes:

- **FRQ-001**
  Un conjunto de bloques elementales para la construcción de secuencias MRI –esencialmente, pulsos RF y gradientes– debe estar disponible y listo para su uso desde un primer momento.

- **FRQ-002**
  La aplicación debe permitir crear la secuencia MRI que se desee mediante la concatenación de estos bloques básicos. Los bloques se deben poder añadir, modificar, mover y eliminar de manera totalmente libre. El movimiento de los bloques se podrá realizar, tanto con las flechas del teclado, como con acciones del tipo *drag* & *drop*[1].

- **FRQ-003**
  Debe existir la opción de generar grupos de bloques. Estos grupos, desde la perspectiva del usuario, se verán como bloques en sí mismos, por lo que todas las operaciones previamente mencionadas –crear, modificar, mover y eliminar bloques– deben estar también disponibles para los grupos. Adicionalmente, debe existir una funcionalidad para duplicar grupos previamente creados.

- **FRQ-004**
  Se debe incluir una funcionalidad que permita, tanto guardar las secuencias creadas, como cargar secuencias ya existentes.

- **FRQ-005**
  El usuario debe tener la opción de definir los parámetros globales de la simulación; concretamente, los parámetros del escáner. Estos son:

  - B_0: intensidad del campo magnético principal.
  - B_1: máxima intensidad del pulso RF.

[1] *drag* & *drop* se refiere a la acción de arrastrar y soltar objetos en la interfaz, por medio del ratón, o en su defecto, de una pantalla táctil.

- Δt: periodo de muestreo.
- $Gmax$: máxima intensidad de los gradientes.
- $Smax$: *slew-rate* máximo.

- **FRQ-006**
  La interfaz gráfica debe proporcionar, una vez realizada la simulación, una vista de la imagen reconstruida y del espacio K correspondiente, así como un diagrama temporal de la secuencia creada.

- **FRQ-007**
  Se debe mostrar una representación en 3D del corte seleccionado en la excitación RF.

- **FRQ-008**
  El simulador debe admitir entradas procedentes de ficheros NIfTI (`.nii`).

3.1.2. Requisitos no funcionales

Los requisitos no funcionales (NFR) no hacen referencia directamente a funciones específicas que deba proporcionar el programa, sino a propiedades que definen las restricciones del sistema como la fiabilidad, el tiempo de respuesta, la capacidad de almacenamiento, etc. Los NFR se originan en la necesidad del usuario, debido a restricciones el presupuesto, a las políticas de la organización, a la necesidad de interactuar con otros sistemas software o hardware o a factores externos como regulaciones de seguridad o legislaciones acerca de la privacidad.

Los requisitos no funcionales de la aplicación son los siguientes:

- **NFR-001**
  El sistema debe ser completamente funcional cuando se ejecute como aplicación de escritorio.

- **NFR-002**
  La comunicación entre todas las tecnologías empleadas en el desarrollo del sistema debe ser transparente al usuario. Concretamente:

  - Comunicación de la interfaz QML con C++/Qt.
  - Comunicación entre C++ y Julia.

- **NFR-003**
  La complejidad de las simulaciones debe ser adaptable a las necesidades del usuario.

- **NFR-004**
  Las tecnologías empleadas en el desarrollo del sistema deben permitir que este sea posteriormente transformado en una aplicación web, reutilizando lo máximo posible el código ya desarrollado para la aplicación de escritorio.

3.2. Diseño de la aplicación

3.2.1. Elección de las tecnologías empleadas

A partir del conjunto de tecnologías descritas en el capítulo 2 y de los requisitos fijados en el capítulo 3.1, se pretende exponer las tecnologías escogidas para el desarrollo del proyecto, así como la razón de su elección:

- Con respecto a la interfaz gráfica, se ha escogido Qt como marco central del trabajo. La razón principal de esta decisión es su posibilidad de desarrollar interfaces gráficas multiplataforma, con el aspecto de las aplicaciones nativas propias de cada sistema operativo, y su compatibilidad con WebAssembly, que garantiza su ejecución en los principales navegadores web usados en la actualidad. Lenguajes ampliamente usados como C++ –sobre el cual se desarrolla Qt– o Rust son probablemente los más asociados con WebAssembly hoy en día. Este hecho nos va a ser de

utilidad, además de para la aplicación de escritorio, para la posible migración a web que podría realizarse en un futuro, y que sería prácticamente directa. Adicionalmente, existe una API para C/C++ que permite integrar expresiones, funciones y variables de Julia en C++.

En cuanto al módulo de Qt, recordemos las dos opciones estudiadas en la sección 2.5.1: `Qt Widgets` y `Qt Quick`. La decisión final es el uso del segundo de estos módulos, debido a lo siguiente:

- `Qt Quick` está orientado al desarrollo de interfaces fluidas y dinámicas, facilitando así la implementación, entre otras, de la funcionalidad *drag & drop* mencionada anteriormente.
- La sintaxis de QML es más sencilla e intuitiva que el desarrollo directo en el lenguaje Qt. Recordemos que Qt era, esencialmente, C++ con extensiones.
- `Qt Quick`, además, está más preparado para el desarrollo de interfaces adaptadas a dispositivos móviles. Este hecho nos puede ser útil si, en un futuro, se contempla la opción de habilitar la aplicación para este tipo de terminales.

- Para la visualización en 3D, vamos a decantarnos por el paquete de Julia `PlotlyJS.jl`, debido a dos razones principales:
  - Sencillo de usar: la utilización de las funciones de este paquete resulta intuitiva y su sintaxis se parece, en gran medida, a la de funciones de MATLAB (`plot()`, `plot3()`, `volshow()`, etc.) ya vistas durante los estudios del Grado.
  - Al estar trabajando con un simulador hecho en Julia, no es necesario instalar ningún software ni lenguaje adicional, y la implementación de las funciones de visualización se puede realizar de forma prácticamente directa: bastaría con definir una nueva función dentro del paquete del simulador `KomaMRI.jl`. Más adelante, en la sección 3.3, veremos cómo se lleva a cabo esta tarea.

  No obstante, en el resto de aspectos –rendimiento, funcionalidades, integración con otros lenguajes, etc.–, VTK supera a PlotlyJS con diferencia, pero esto también se ve reflejado en su complejidad.
- Como sistema de construcción, se ha elegido CMake, debido a las ventajas expuestas en la sección 2.5.3
- Para facilitar las tareas de tratamiento de matrices e imágenes en C++, también se hará uso de la biblioteca OpenCV [33].
- Por último, es necesario indicar las herramientas de desarrollo software escogidas:
  - Para el desarrollo de la parte de Qt y la realización de pruebas de funcionamiento de la aplicación completa, se ha decidido utilizar Qt Creator ya que, además de estar dirigido especialmente al desarrollo con Qt, permite la compilación de aplicaciones de escritorio y de aplicaciones web con WebAssembly.
  - Cuando sea necesario centrarse en el simulador propiamente dicho, desarrollado en Julia, mejores opciones serán Visual Studio Code y Atom. Estas dos herramientas son las más adecuadas para trabajar con Julia en Windows y Linux, respectivamente. Incluyen extensiones específicas, así como un REPL integrado desde el que es posible probar las funciones y módulos desarrollados.

3.2.2. Descripción general de funcionamiento

En la sección 2.4.2 se analizó el simulador `KomaMRI.jl` junto con su esquema global de funcionamiento (ver figura 2.18). Una vez visto este funcionamiento, ahora se hará un análisis del uso que, en nuestro caso particular, se va a hacer de la aplicación. La figura 3.1 muestra un esquema sencillo.

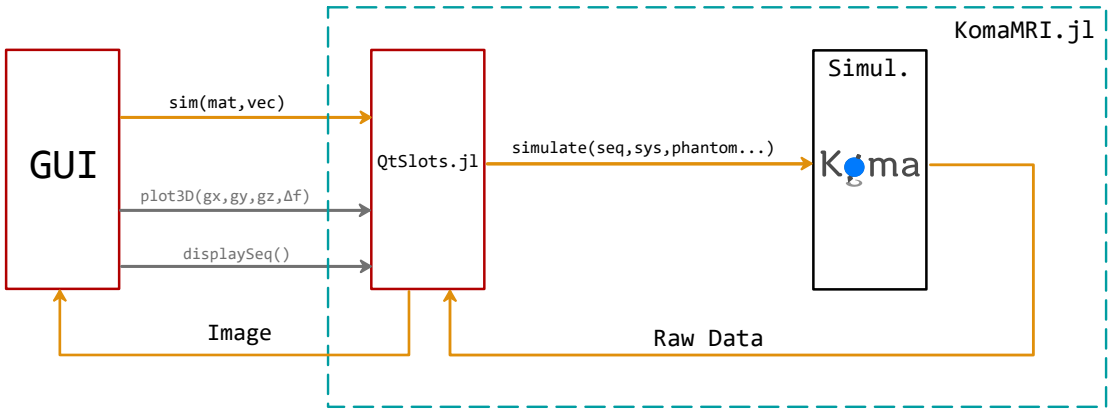


Figura 3.1: Diagrama de funcionamiento del proyecto a realizar

En este esquema se han introducido dos elementos nuevos (en línea roja), de los que se hará una breve introducción, aunque será en la sección 3.3 en la que se analizarán más en profundidad:

- **Interfaz gráfica de usuario (*Graphical User Interface*, GUI)**: es el elemento principal del proyecto. Se realizará en Qt/QML y contará con las funcionalidades descritas en el capítulo 3.1.
- **Fichero QtSlots.jl**: incluido por comodidad dentro del directorio fuente (`KomaMRI.jl/src`) del simulador, este fichero contiene tres funciones, las cuales son llamadas desde la interfaz:
  - La función `sim` hace de "intermediaria" entre la GUI y el simulador; se encarga de adaptar los datos relativos a la secuencia y a los parámetros globales (que le llegan en forma de matriz y vector, respectivamente) a un formato apto para el simulador. En definitiva, crea un objeto `seq::Sequence` a partir de la matriz `mat`, en la que están los datos de la secuencia, y un objeto `sys::Scanner` a partir del vector `vec`, en el que hay información de los parámetros de la simulación. Con estos dos objetos ya creados, hace una llamada al simulador (función `simulate`), procesa la señal generada por el simulador y devuelve el resultado.

    Tal y como se puede observar, la función `simulate` de `KomaMRI.jl` devuelve la señal MR en crudo, y es en la función `sim` en la que se realiza la reconstrucción.
  - La función `plot3D` es totalmente ajena al resto de la aplicación, y no hace llamadas al simulador, sino que hace uso del paquete `PlotlyJS.jl` para representar en tres dimensiones el slice excitado por un pulso RF.
  - La función `displaySeq` crea una matriz auxiliar a partir de la secuencia generada por la función `sim`. Esta matriz se devolverá a la interfaz y se utilizará para dibujar el diagrama temporal de la secuencia.

3.2.3. Estructura de la secuencia

En la interfaz se emplearán los términos "bloque" y "secuencia" de forma distinta al significado que tienen en el simulador `KomaMRI.jl` (ver sección 2.4.2). Concretamente:

- Con bloque nos referimos al elemento visual, con forma cuadrada, que puede ser creado, movido y eliminado de forma libre y cuyos parámetros, dependientes del tipo de bloque, son configurables. Un bloque de la interfaz puede corresponder a uno o a varios bloques del simulador, en función, también, de su tipo.
- En cuanto a la secuencia, nos referimos a ella como al conjunto de todos los bloques presentes en la interfaz. Es, por tanto, una definición más parecida a la que hacíamos en la sección 2.2.3.

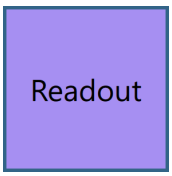
Tipos de bloques

A continuación se presentan los tipos de bloque elementales que se van a implementar en el editor:

- Bloque `Ex`: se corresponde con un bloque de excitación. Durante su aplicación, se activa el pulso RF y los gradientes de selección de *slice*. Los parámetros configurables de este bloque son su duración, la intensidad de los gradientes, la amplitud de la envolvente compleja del pulso RF ($B_{1x} + jB_{1y}$) y la desviación en frecuencia Δf. Estos parámetros, cuando se llame al simulador, serán los argumentos de entrada de la función `RF_hard`, que ya se describió anteriormente. Además, cada vez que despleguemos un bloque de este tipo, podremos visualizar en 3D el *slice* que vamos a seleccionar. De esta forma podremos comprobar, sin necesidad de realizar la simulación, que el pulso va a excitar el *slice* deseado. Hay que resaltar que, tras un bloque de excitación, el resto de bloques que lo sucedan deberán ser rotados para ser así coherentes con los gradientes de selección de slice. Este aspecto se tratará en detalle en la sección 3.3.4.

- Bloque `Delay`: aplica un retardo en la secuencia. Por ello, su único parámetro configurable es la duración. Este parámetro será el argumento de entrada del constructor de la clase `Delay`, que equivale a un objeto `Sequence` con gradientes nulos y con el pulso RF y el ADC desactivados.

- Bloque `Dephase`: este bloque hace que se apliquen gradientes durante un tiempo determinado. Tanto el pulso RF como el ADC (es decir, la lectura) están desactivados, por lo que la utilidad de este bloque es la de desplazarnos por el espacio K en una dirección arbitraria (tal y como hacen, por ejemplo, los gradientes blip en una secuencia EPI). Como ya se ha visto anteriormente, esa dirección vendrá dada por la intensidad de los gradientes en x, y y z. Con los parámetros de configuración de este bloque (duración y gradientes) se creará un objeto `GR::Grad` que servirá como argumento de entrada al constructor de `Sequence`: `DEPHASE = Sequence(GR)`.



- Bloque `Readout`: muy parecido al bloque `Dephase`, salvo porque en este caso sí que se activa la lectura. Por tanto, habrá que añadir un parámetro a mayores: el número de muestras que queremos que se tomen durante la duración del bloque. Recordemos que este era el parámetro `N` de la estructura `ADC`.

- Bloque `EPI`: hasta ahora, la correspondencia de los bloques de la interfaz con los del simulador era de 1 a 1 (es decir, un bloque de la interfaz equivalía a un bloque del simulador). No obstante, esto no es así para este caso. El bloque `EPI` simula la adquisición completa del espacio K, aplicando la secuencia descrita en la función `EPI`, anteriormente estudiada (ver sección 2.4.2). De hecho, es a esta función a la que llama cuando añadimos el bloque `EPI` a la secuencia. Por lo tanto, los parámetros configurables serán el FOV y el número de muestras por línea, `N` (al ser una adquisición cuadrada, elegir `N` equivale a elegir las dimensiones de la imagen).

Grupos de bloques

La tarea de componer una secuencia a partir de los bloques previamente descritos puede resultar tediosa en caso, por ejemplo, de necesitar varias iteraciones (esto ocurre prácticamente en todas las secuencias). Al ser la secuencia un conjunto de bloques ordenados de manera lineal, no existe a priori la opción de crear bucles en ella; no es posible, por ejemplo, crear un subconjunto de bloques que se

vaya repitiendo cada TR (funcionalidad que nos vendría bien en el caso de secuencias como la *spin echo*).

Con esta motivación surge el concepto de grupo de bloques. La idea es poder crear subconjuntos dentro de la secuencia, de forma que los bloques de su interior se repitan un número determinado de veces de forma iterativa, tal y como ocurriría en un bucle `for`. De esta forma, solo sería necesario crear una iteración, incluir sus bloques dentro de un grupo e indicar el número de veces que ese grupo se debe repetir. Además, sería también útil la funcionalidad de poder crear grupos anidados, es decir, que un grupo de bloques pueda contener a su vez otros grupos.

Podemos pensar, por ejemplo, en una secuencia EPI *multi-shot*, en la que existiría un grupo exterior, correspondiente a la excitación y un grupo interior asociado a la adquisición de una línea del espacio K. Para adquirir la imagen completa, harían falta varias excitaciones y por cada una de ellas habría que adquirir más de una línea del espacio K (si quisiésemos adquirir, por ejemplo, una imagen de 20 líneas, podríamos crear un grupo "Excitación" que se repitiese 4 veces y en su interior un grupo "Lectura" que leyese una línea y se repitiese 5 veces por cada excitación).

Todo esto queremos poder diseñarlo en la propia interfaz, por lo que lo más cómodo es que la estructura de datos asociada a la secuencia resida en QML: el modelo de datos `ListModel`, tal y como se explicó en la sección 2.5.2, nos va a permitir crear y modificar los datos de la secuencia de forma dinámica. u

3.2.4. Definición de los parámetros de la simulación

Para configurar los parámetros de la simulación, concretamente, los parámetros de la secuencia, existen dos posibles caminos a seguir:

1. En la mayoría de simuladores de MRI existentes[2], es posible fijar previamente los parámetros de la secuencia, tales como:

   - TE: tiempo de eco.
   - TR: tiempo de repetición.
   - TI: tiempo de inversión.
   - ETL (*Echo Train Length*)
   - NEX (*Number of Excitations*)

   Una vez fijados estos parámetros y habiendo elegido el tipo de secuencia a aplicar, será el simulador el que, a partir de los datos introducidos por el usuario, establezca las duraciones e intensidades adecuadas para cada elemento de la secuencia. Esto se hace de forma automática y transparente al usuario.

2. La otra alternativa es la de seguir el sentido opuesto: configurar manualmente los parámetros de cada elemento –bloque– de la secuencia: intensidad de los gradientes, duración de los retardos, intensidad de los pulsos RF... Haciendo esto, ya estamos definiendo implícitamente los parámetros de la secuencia. Adicionalmente, parámetros como el FOV (*Field Of View*) o las dimensiones de la imagen obtenida quedan igualmente definidos.

En el caso de nuestro proyecto, vamos a optar por la segunda opción, debido principalmente a dos motivos:

- Dado el fin didáctico de la aplicación, resulta mucho más interesante, de cara al aprendizaje, el hecho de tener que configurar manualmente cada uno de los bloques. De esta forma el usuario podrá notar la influencia de cada pequeño cambio, y comprender, a fin de cuentas, todo el proceso de una forma más interactiva.
- El simulador está preparado para recibir parámetros como la duración de cada elemento, y no tanto para recibir variables como pueden ser el tiempo de eco o el de repetición.

Por todo ello, no vamos a tener la opción de definir directamente los parámetros de la secuencia; no obstante, los parámetros del escáner, mencionados también en la sección 2.2.3, sí van a ser configurables.

[2]Ver, por ejemplo, el simulador creado por Daniel Treceño [2]

3.2.5. Diseño de la interfaz gráfica

Aquí se plantea el diseño, tanto visual como funcional de la interfaz propiamente dicha. A partir de todas los requisitos vistos a lo largo de este documento, se ha creado un boceto de interfaz (ver figura 3.2) en el que, por medio de paneles, se incluyen todas las funcionalidades previamente descritas. Se ha apostado, principalmente, por que la interfaz sea simple e intuitiva.

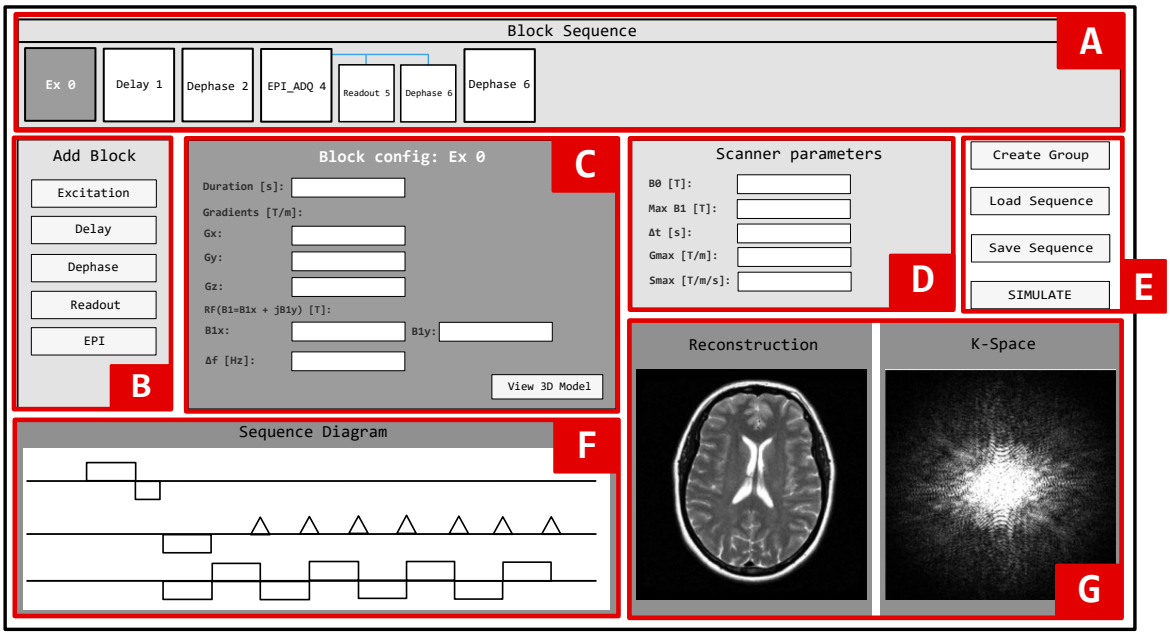


Figura 3.2: Boceto de la interfaz gráfica de la aplicación, en el que están resaltados cada uno de los paneles

El panel (A) se corresponde con el panel de secuencia. Destaca por ocupar todo el ancho de la ventana de la aplicación; con ello se pretende atraer la atención del usuario hacia la secuencia, que es el componente principal de todo el proceso. Además de esta forma se facilita la su visualización completa. No obstante, si aun así la secuencia fuese demasiado larga como para poder visualizarse de manera completa, el panel cuenta con una barra de desplazamiento horizontal.

El panel (B) proporciona la colección de bloques disponibles, que pueden añadirse a la secuencia simplemente haciendo clic en uno de los botones del panel.

Los paneles (C) y (D) contienen, respectivamente, los parámetros de configuración del bloque y del escáner. En cuanto a los primeros, al hacer clic en un bloque determinado se mostrará su menú de configuración, desde el que se pueden ajustar fácilmente los parámetros relacionados; con respecto a los parámetros del escáner, estos pueden ser definidos directamente.

El panel (E) está destinado a la gestión de secuencias; se pueden realizar operaciones sencillas como cargar y guardar secuencias. Además, el usuario puede crear grupos de bloques (ver sección 3.2.3). El botón para lanzar la simulación también está situado, por comodidad, en este panel.

Por último, los paneles inferiores, es decir, los paneles (F) y (G), están destinados a la visualización; concretamente, el panel (F) mostrará, tras la simulación, el diagrama temporal de la secuencia de pulsos creada, mientras que el panel (G) mostrará el resultado de la simulación, tanto la imagen reconstruida como el espacio K correspondiente.

3.3. Implementación

3.3.1. Planteamiento del modelo de datos en QML

Tal y como se expuso en la sección 2.5.2, el modelo `ListModel` de QML permite almacenar datos en un formato de lista dentro de la interfaz. Este modelo nos va a ser útil, tanto para almacenar la información de la secuencia, como para almacenar los valores de los parámetros del escáner.

Modelo de datos de la secuencia

Se utilizará un `ListModel` con identificador `blockList`, de forma que cada uno de sus elementos `ListElement` vaya asociado a un bloque de la secuencia. Cada elemento tendrá la misma colección de propiedades, sea cual sea el tipo de bloque al que vaya asociado; esto quiere decir que, aunque haya propiedades innecesarias para algunos tipos de bloque –por ejemplo, vimos que en un bloque `EPI` el parámetro de la duración no era necesario–, estas se definirán igualmente, inicializándose a 0.

De entre estas propiedades destacamos, inicialmente, `cod`, un número entero que hará referencia al tipo de bloque. Nos fijaremos en él para definir aspectos como el color del bloque, el texto o los parámetros visibles en el menú de configuración. El cuadro 3.1 muestra la correspondencia entre el valor de esta propiedad y el tipo de bloque.

Bloque	`cod`
Grupo	0
`Ex`	1
`Delay`	2
`Dephase`	3
`Readout`	4
`EPI`	5

Cuadro 3.1: Correspondencia entre el tipo de bloque y la propiedad `cod` en la lista de QML

En cuanto a los grupos de bloques, debemos saber que un grupo es también, en sí mismo, un bloque; será un bloque "especial", cuyos dos únicos parámetros relevantes serán el número de repeticiones (`reps`) y un *array* (`children`) que contendrá los índices de cada uno de los bloques pertenecientes al grupo. Asimismo, los bloques de tipo grupo tendrán `cod = 0`.
Planteando este atributo `children`, es posible guardar los datos de una estructura jerárquica como la que hemos planteado en un modelo lineal, como puede ser el proporcionado por `ListModel`. De esta forma no tenemos que recurrir a otras estructuras de datos más complejas como puede ser la estructura en árbol.

Para aclarar todo lo explicado referente a los grupos de bloques y a la estructura de la secuencia, podemos fijarnos en la figura 3.3, que ilustra cómo sería una estructura de bloques con dos grupos definidos, uno anidado dentro del otro. Se pueden ver los atributos `children` y `reps` de cada bloque. La secuencia resultante de la estructura mostrada en la figura sería la siguiente: `b1-b2-b3-b4-b3-b4-b3-b4-b5-b2-b3-b4-b3-b4-b3-b4-b5-b6`.

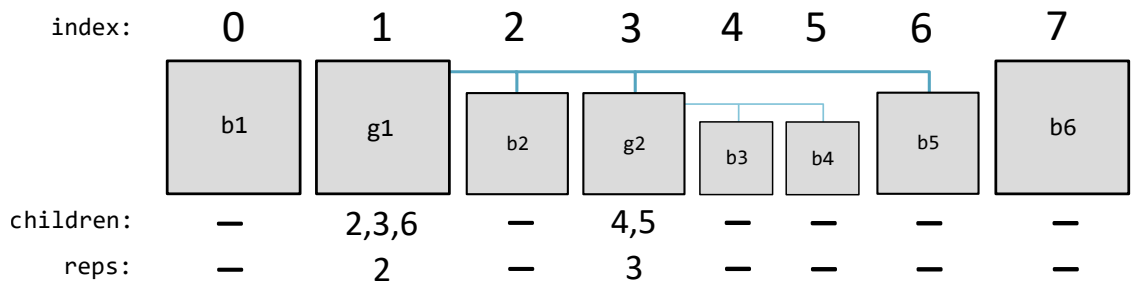


Figura 3.3: Ejemplo de estructura de bloques que conforman una secuencia

Asimismo, las figuras 3.4a y 3.4b muestran dos ejemplos de instancias de `ListElement` pertenecientes a una secuencia. Se puede comprobar cómo, efectivamente, todos los bloques tienen la misma colección de propiedades aunque, dependiendo de su tipo, estas serán o no relevantes.

Aprovechando estas figuras, se va a realizar una breve descripción de todos los parámetros, aún no mencionados, de cada bloque:

- `dur`: duración de cada bloque en segundos.
- `gx, gy, gz`: intensidad, en T/m, de los gradientes en las direcciones x, y y z, respectivamente.
- `b1x, b1y`: partes real e imaginaria de la envolvente compleja del pulso RF. Recordemos que esta envolvente compleja era de la forma $B_{1x} + jB_{1y}$. Se miden en teslas, T, y se utilizan solo en bloques de tipo `Ex`.
- `delta_f`: diferencia de frecuencia del pulso RF con respecto a la frecuencia de Larmor. Se mide en Hz, y al igual que los parámetros `b1x` y `b1y`, solo se utiliza en bloques tipo `Ex`.
- `fov`: *Field Of View*, es decir, campo de visión. Se utiliza en los bloques `EPI` para definir las dimensiones –en el espacio real– del corte que queremos visualizar.
- `n`: también usado únicamente en bloques `EPI`, define el número de líneas que se desea leer. El bloque `EPI`, tal y como se vio en la sección 3.2.3, lleva a cabo una adquisición cuadrada, por lo que el número de líneas definido con este parámetro `n` será asignado también al número de píxeles leídos por línea. Así, se obtendrá una imagen de `n` x `n` píxeles.
- `grouped`: este booleano auxiliar se utiliza cuando se activa el modo de agrupar bloques. Con este modo activo, cuando el usuario pulsa en un bloque, su propiedad `grouped` pasa de `false` a `true` (o viceversa). Sirve para dar cuenta de los bloques que han sido seleccionados para conformar un grupo.
- `ngroups`: indica el número de grupos a los que pertenece un bloque ya que, recordemos, que pueden existir grupos anidados. Es otra propiedad auxiliar que permite definir el tamaño del bloque en función del número de grupos a los que pertenezca: cuantos más grupos, menor será el tamaño del bloque (ver, por ejemplo, el bloque `b3` en la figura 3.3, perteneciente a 2 grupos).
- `collapsed`: otro booleano auxiliar, que en este caso indica si un bloque es visible o no. No será visible, por ejemplo, cuando el grupo al que pertenezca no esté desplegado.

Lógicamente, todas las propiedades auxiliares que se han descrito no se pasarán al simulador, solamente tienen utilidad para la interfaz gráfica.

(a) (b)

Figura 3.4: Código QML correspondiente a dos instancias de `ListElement` pertenecientes a una secuencia. (a): bloque de excitación. (b): bloque de tipo grupo con 4 bloques "hijos" (ver cuadro 3.1).

Modelo de datos para duplicar grupos

Para implementar la funcionalidad de poder duplicar grupos, se incluirá en el menú de configuración de este tipo de bloques un botón con el texto "*Make group duplicable*". Cuando se haga clic sobre él, aparecerá un nuevo botón en el panel B, con el nombre del grupo que hemos hecho "duplicable". Cada vez que lo pulsemos, se añadirá a la secuencia un grupo exactamente igual al que hemos duplicado.

Esto es posible gracias a un `ListModel` auxiliar con identificador `groupList`, en el que queda almacenada la información de cada uno de los grupos –y sus correspondientes "hijos"– que se han añadido al panel B. La forma de almacenar esta información es prácticamente idéntica a la que se usaba con la estructura `blockList`, salvo que ahora se omiten los parámetros auxiliares como `collapsed` o `grouped`. La información de este modelo `groupList` no tendrá ninguna relevancia para el simulador propiamente dicho.

Modelo de datos de los parámetros del escáner

Para almacenar los parámetros del escáner será todo mucho más sencillo, ya que no existirán estructuras jerárquicas, simplemente una colección de propiedades. Estas propiedades estarán contenidas en los cuadros de texto del menú de parámetros del escáner (panel D de la figura 3.2). Cuando se pulse el botón de simular, bastará con pasar el contenido de estas cajas de texto al simulador, sin necesidad de ningún modelo `ListModel`.

3.3.2. Pasarelas entre lenguajes

En la sección 3.2.2 se introdujo el funcionamiento general de la aplicación y se indicaron las conexiones entre cada uno de sus elementos. No obstante, no se analizó el modo preciso en el que tales conexiones se llevaban a cabo. En esta sección se van a estudiar los métodos con los que se realiza la comunicación entre las partes del proyecto desarrolladas en lenguajes distintos, así como el formato que se da a los datos –tanto de entrada como de salida de las funciones– para que sean compatibles con cada una de las tecnologías empleadas.

Mientras la figura 3.1 mostraba las llamadas a las funciones de cada componente de la aplicación, la figura 3.5 da cuenta de los mecanismos utilizados para conectar cada uno de esos componentes. Desde C++ se inicia la aplicación, ya que es en este lenguaje donde se implementa la función `main`. Se puede observar cómo, además, C++ hace de lenguaje intermediario entre la interfaz propiamente dicha y el simulador.

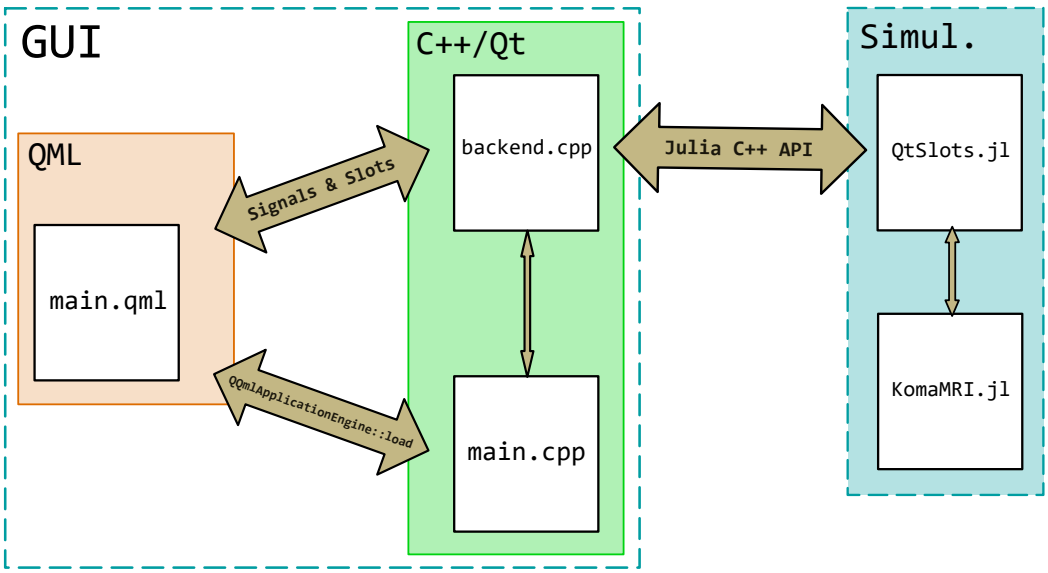


Figura 3.5: Esquema de conexiones entre los lenguajes que componen la aplicación

GUI: Comunicación de QML con C++

Existen dos situaciones en las que la parte implementada en QML se debe comunicar con la parte de C++. Concretamente:

- Vincular fichero `main.qml` con la aplicación Qt: la figura 2.23 ilustraba una forma de hacer "visible" el fichero QML desde C++, que se basaba en la definición de un objeto `QQuickView` y la posterior llamada a su método `show`.

  Sin embargo, existe un método equivalente, introducido en las versiones más recientes de Qt, que parece más apropiado. Se basa en el uso de las clases `QApplication` y `QQmlApplicationEngine`. La primera de ellas gestiona el flujo de control de la GUI y su configuración principal. En cuanto a la segunda, proporciona una forma de cargar un solo fichero QML y expone algunas funcionalidades centrales de la aplicación a QML, que una aplicación híbrida C++/QML normalmente controlaría desde C++ [26]. La carga del fichero QML se realiza mediante el método `load`. El cuadro 3.2 muestra las líneas de código involucradas en el proceso.

```
QApplication app(argc, argv);
QQmlApplicationEngine engine;
const QUrl url(QStringLiteral("qrc:/main.qml"));
engine.load(url);
return app.exec();
```

Cuadro 3.2: Líneas de código utilizadas para cargar el fichero QML desde C++

```
Backend backend;
engine.rootContext()->setContextProperty("backend",&backend);
```

Cuadro 3.3: Líneas de código utilizadas para hacer que el objeto `backend` esté disponible en QML

- Enviar señales y datos desde QML a C++. Se utilizarán señales y *slots* (ver sección 2.5.1). En concreto, se creará una clase `Backend`, derivada de `QObject`, en la que se declararán e implementarán los *slots*, en los ficheros `backend.h` y `backend.cpp`, respectivamente.

Para poder llamar a estos *slots* desde QML, se debe crear una instancia de la clase `Backend` dentro de la función main y vincularla al `QQmlApplicationEngine` mediante el método `setContextProperty`. El cuadro 3.3 muestra las dos líneas de código correspondientes, que se deberán situar antes de la llamada al método `load` previamente mencionado. Una vez realizada esta conexión, bastará con llamar a los *slots* desde QML, utilizando[3]:

```
backend.nombreDelSlot(argumentos)
```

Comunicación entre C++ y el simulador

Se hará uso de la API descrita en la sección 2.3.6. El primer paso es inicializar Julia y el módulo `KomaMRI` desde la función main, tal y como muestra el cuadro 3.4. Es necesario incluir estas líneas antes de la directiva `app.exec()`.

```
jl_int();
jl_eval_string("cd(\"../KomaMRI.jl\")");
jl_eval_string("using Pkg; Pkg.activate(\".\")");
jl_eval_string("Pkg.instantiate();");
jl_eval_string("using KomaMRI");
```

Cuadro 3.4: Código C++ utilizado para inicializar Julia y el módulo `KomaMRI`

Mediante la segunda línea se sitúa el directorio de trabajo de Julia en la carpeta donde está alojado el simulador. La tercera línea activa el entorno de `KomaMRI`, basándose en el contenido de los ficheros `Project.toml` y `Manifest.toml`. Mediante la cuarta línea se instalan los paquetes definidos en el entorno y sus dependencias. Por último, la quinta línea inicializa el módulo propiamente dicho y brinda acceso a todos sus elementos (funciones, variables globales, submódulos, etc.).

La comunicación entre C++ y Julia, una vez inicializado el simulador, se realiza mediante los *slots* de la clase `Backend`. Estos *slots*, por lo tanto, tendrán una doble tarea:

- Recibir las llamadas realizadas desde QML. Se ha explicado en el apartado anterior que estas llamadas se pueden realizar sin necesidad de conectar los *slots* a señales de ningún otro objeto. Basta con realizar una llamada como si se tratase de funciones normales[4].
- Realizar las llamadas a las funciones de Julia implementadas en el fichero `QtSlots.jl`. Concretamente:
  - *Slot* `plot3D`: realiza la llamada a la función `plot3D` del fichero `QtSlots.jl`.
  - *Slot* `simulate`: realiza la llamada a la función `sim` del fichero `QtSlots.jl`.
  - *Slot* `displaySeq`: realiza la llamada a la función `displaySeq` del fichero `QtSlots.jl`.
  - *Slot* `saveSeq`: no realiza la llamada a ninguna función de Julia.

En el siguiente apartado se explica más detalladamente de la implementación de estos *slots*, analizando cómo se reciben las llamadas desde QML y cómo se llama a las funciones de Julia.

3.3.3. *Slots* de la clase `Backend`

En esta sección se describe la implementación de cada uno de los *slots* definidos en la clase `Backend`, analizando previamente la realización de las llamadas a estos *slots* desde QML y el formato en el que se les pasan los datos.

Adicionalmente, para aclarar más su funcionamiento, la implementación completa de todas estas funciones ha sido recogida en el apéndice C.

[3]Desde QML, es posible llamar a los *slots* de C++ sin necesidad de emitir señales. Las llamadas se realizan como si de una función convencional se tratase.

[4]No obstante, es necesario que en el fichero de cabecera de la clase `Backend`, estas funciones estén declaradas como *slots*. De lo contrario, las llamadas desde QML no funcionarían.

- `void plot3D(double gx,double gy, double gz, double delta_f)`:

  Esta función recibe los valores de los gradientes en las tres dimensiones y la variación de frecuencia del pulso RF con respecto a la frecuencia de Larmor (ver sección 2.2.1). En QML, estos cuatro parámetros son recogidos directamente de los campos de texto presentes en el menú de configuración del bloque desde el que se llama a la función. La correspondencia de datos de tipo `double` entre QML y C++ es directa (ver sección "Data Type Conversion Between QML and C++", en [26]), por lo que en C++ bastará con definirlos también como *doubles*.

  Una vez recibidos estos datos, el *slot* llama a la función `plot3D` de `QtSlots.jl` para que, desde Julia, se represente en 3D el *slice* seleccionado. Por lo tanto, este *slot* no realiza ninguna tarea adicional además de la de servir de intermediario entre QML y Julia.

  La llamada a la función `plot3D` de Julia se realiza recogiendo en un puntero de tipo `jl_value_t` la salida de la orden:

  ```
  jl_eval_string("@cfunction(KomaMRI.plot3D,Any,(Cdouble,Cdouble,Cdouble,Cdouble))");
  ```

  La expresión `@cfunction` de Julia genera un puntero de función que puede ser invocado desde C/C++. En este caso se crea un puntero de la función `plot3D`, perteneciente al módulo `KomaMRI`. `@cfunction` requiere también la definición del tipo y número de argumentos, tanto de salida como de entrada. Respecto a los primeros, el tipo de dato `Any` indica que el argumento de salida de la función podrá ser de cualquier tipo, incluso nulo, como es el caso. En cuanto a los argumentos de entrada, serán 4 números de tipo double.

  La correspondencia de tipos entre C++ y Julia no es directa para los datos de tipo `double`; lo que en C++ es `double`, en Julia se corresponde con `Float64`. Sin embargo, hace falta un tipo de dato adicional en Julia para poder interpretar correctamente los datos procedentes de C/C++: el tipo `Cdouble`.

  Por último, se obtiene, a partir de la orden `jl_unbox_voidpointer`, la función `c_plot`, que podrá ser llamada directamente desde C++.

- `void simulate(QVector<double> sys, QVector<QVector<double>> seq)`:

  Este *slot* recibe dos argumentos desde QML, uno que contiene los parámetros del escáner y otro con la información de la secuencia. Ya se mencionó anteriormente que en la interfaz, los datos del escáner se obtenían directamente de los valores introducidos en las cajas de texto del menú de configuración global. Estos valores se colocan en un *array* unidimensional que será enviado al *slot*. En cuanto a los datos de la secuencia, se obtienen a partir de la estructura `blockList` y se disponen en un *array* bidimensional, de forma que cada una de sus columnas corresponda a un bloque. En el apéndice B se incluyen las dos funciones de Javascript que crean estos *arrays*: `createScanner` y `createSeq`.

  En este caso, no existe una correspondencia de tipos directa entre QML y C++, ya que los *arrays* no presentan el mismo formato en ambos lenguajes. Es por ello por lo que hay que hacer uso del tipo de dato `QVector`, un contenedor que permite agrupar variables en forma de *array*. Anidando estas estructuras `QVector`, es posible crear matrices de tantas dimensiones como se desee.

  Una vez recibidos los datos en el formato adecuado, es necesario volver a adaptarlos, en este caso, para que sean compatibles con Julia. Volviendo a consultar la sección "Embedding Julia" en [16], se comprueba que en la API existen funciones para definir *arrays* de Julia dentro de C++. Destacan las funciones `jl_alloc_array_1d`, `jl_alloc_array_2d` y `jl_apply_array_type`.

  Tras adaptar la matriz de la secuencia y el vector del escáner al formato admitido por Julia, se realiza la llamada a la función `sim`. En este caso, a diferencia de como se realizó en el *slot* `plot3D`[5], se ha hecho uso de la función `jl_call_2` para llamar a `jl_sim`, función previamente obtenida con `jl_get_global`. Tras esta llamada, se obtendrán, en forma de matriz, la imagen reconstruida y el espacio k correspondiente.

[5] La llamada a la función de Julia se ha efectuado de forma distinta a la utilizada en el *slot* `plot3D` debido, simplemente, a que se han querido probar los dos posibles métodos: uno mediante `jl_unbox_voidpointer` y otro mediante `jl_call`. No obstante, no existe un método mejor que otro.

Por último, se hace uso de la biblioteca OpenCV para convertir esas matrices en imágenes y almacenarlas en el directorio `temp`.

- `void saveSeq(QVector<QVector<QVector<double>>> arr, QString path)`:

  La función de este *slot* es la de guardar la secuencia creada por el usuario en un fichero, con el objetivo de que pueda volver a ser utilizada en futuras ocasiones. Para llevar a cabo esta tarea, en QML se debe crear un *array*, en este caso, tridimensional, a partir de la estructura `blockList`. En la tercera dimensión del *array* se almacenarán, para cada bloque de tipo grupo, los identificadores de cada uno de sus bloques "hijos". La función Javascript `saveSeq`, también incluida en el apéndice B, es la que se encarga de la creación de este *array*.

  El segundo argumento recibido por el *slot* es la ruta absoluta del fichero en el que se debe guardar la secuencia. Esta ruta se obtiene mediante la implementación de un objeto `FileDialog` en la interfaz. Se puede apreciar, además, cómo se está utilizando el tipo de dato `QString` para adaptar la cadena de texto de un lenguaje a otro.

  La secuencia se almacenará en un fichero con formato QML, concretamente, en un `ListModel`. De esta forma, la secuencia podrá ser fácilmente cargada desde la interfaz.

  El *slot* `saveSeq` no hace uso de llamadas a funciones de Julia, todo se hace directamente en C++.

- `QVector<QVector<double>> displaySeq()`:

  A diferencia de los tres anteriores, este *slot* tiene un argumento de salida: se trata de una matriz generada a partir de la secuencia creada en la función `sim` de Julia. Esta matriz será usada en la interfaz para dibujar el diagrama temporal de la secuencia.

  En QML, por lo tanto, se hará la llamada al *slot*, se recibirá la matriz y se leerán sus elementos uno a uno para dibujar la secuencia en un objeto `ChartView` [26].

3.3.4. Funciones de `QtSlots.jl`

Esta sección contiene un análisis del código utilizado en la implementación de las tres funciones del fichero `QtSlots.jl`. Es importante recalcar que este fichero está incluido dentro del directorio fuente (`.../KomaMRI.jl/src`) del simulador y que, para que sus funciones sean accesibles desde programas externos, es necesario incluir las siguientes líneas en el fichero principal (`.../KomaMRI.jl/src/KomaMRI.jl`):

```
include("QtSlots.jl")
export plot3D, sim, displaySeq
```

El nombre del fichero, `QtSlots.jl`, se debe a que las tres funciones contenidas en él son llamadas, cada una, por uno de los *slots* implementados en Qt, cuyo funcionamiento se ha analizado en la sección anterior. Las implementaciones completas de estas funciones se recogen en el apéndice D.

Función `plot3D`

Esta función recibe 4 parámetros procedentes del *slot* `plot3D`, que corresponden a los gradientes de selección de *slice* en las 3 direcciones del espacio y a la desviación en frecuencia del pulso RF. A partir de estos parámetros, representa en tres dimensiones el *slice* seleccionado junto con el volumen completo.

Se hace uso de los siguientes paquetes de Julia:

- `PlotlyJS`, explicado en la sección 2.6.
- `Blink`, utilizado para crear una ventana independiente en la que mostrar la visualización.
- `StatsBase` es un paquete de Julia que proporciona soporte básico para operaciones estadísticas.
- `NIfTI`, utilizado para la lectura/escritura de ficheros NIfTI (`.nii`).

Inicialmente, se lee el contenido del fichero NIfTI deseado y se almacenan sus datos en la variable `data`, que será una matriz tridimensional. A continuación, haciendo uso de la función `percentile`, se eliminan (en realidad, se saturan) los valores denominados *outliers*. El siguiente paso es realizar un submuestreo y normalización de los datos, para reducir la carga computacional y hacer que los valores del volumen estén dentro de un rango conocido, entre 0 y 1. El submuestreo se aplica con un factor considerable, de 5, o incluso 6, ya que, de no realizarlo, el número de puntos a representar sería demasiado elevado para el motor de Julia. Al submuestrear tan drásticamente, se producirá un descenso muy notable en la calidad de la visualización; sin embargo, este descenso no impedirá poder seguir distinguiendo los elementos principales dentro del volumen.

Para representar el volumen hay que crear una malla de puntos en las tres dimensiones con la función `mgrid`, equivalente a `meshgrid` en MATLAB. Después, se define una estructura `volume` con los datos del volumen y se fijan algunos de sus parámetros, como `isomin` e `isomax`.

En cuanto al *slice*, se define una estructura `surface`, y se configuran sus parámetros `x`, `y` y `z` para que satisfagan la ecuación del plano:

$$\mathtt{x} * \mathtt{gx} + \mathtt{y} * \mathtt{gy} + \mathtt{x} * \mathtt{gz} = \frac{\Delta \mathtt{f}}{\gamma}$$

Una vez definidos tanto el volumen como el *slice*, estos se incluyen en una estructura del tipo `GenericTrace` y se representan mediante la función `plot`. Para verlo en una ventana aparte, será necesario utilizar el objeto `Window` del paquete `Blink` e integrar el `plot` en la ventana creada.

Función `sim`

Tal y como se introdujo en la sección 3.2.2, `sim` se encarga de llamar a las funciones del simulador `KomaMRI`, creando previamente los objetos `Phantom`, `Scanner` y `Sequence` necesarios para su funcionamiento. Con respecto a cada uno de estos objetos:

- El objeto `phantom::Phantom` se crea a partir de la función `nii_brain_phantom3D` (ver sección 3.4).
- Las propiedades del objeto `sys::Scanner` se extraen del vector `vec` que llega como argumento de entrada. Concretamente, estas propiedades son:
  - `B0`: intensidad del campo magnético principal.
  - `B1`: intensidad máxima del pulso/s RF.
  - `ADC_Δt`: periodo de muestreo del ADC.
  - `Gmax`: intensidad máxima de los gradientes.
  - `Smax`: *slew-rate* máximo.
- En cuanto al objeto `seq::Sequence`, este se construye a partir de la matriz `mat`, en la que cada columna corresponde a un bloque de la secuencia creada en la interfaz. Para realizar el mapeo entre los bloques de la interfaz y los bloques del simulador[6], la función analiza el código del bloque, contenido en la primera posición de cada columna de la matriz. A medida que se vaya avanzando por columnas, se irán añadiendo –concatenando– subsecuencias a la secuencia principal `seq`.

  Cuando la columna se corresponda con un bloque de excitación (código 1), se creará una matriz de rotación `R`, tal y como se indica en el apéndice A.2. Esta matriz se utilizará para rotar los bloques de desfase y de lectura que sucedan a la excitación, de forma que las direcciones de sus gradientes sean coherentes con los gradientes de selección de *slice*.

  Para comprender mejor este último concepto, se recurre a la figura 3.6, que muestra un esquema del efecto de rotación que los gradientes de selección de *slice* producen en las direcciones de codificación de fase y de frecuencia.

  Sea $\mathbf{G}$ el vector formado por los gradientes de selección de *slice*, es posible definir un nuevo sistema de referencia a partir de los ejes x_p e y_p, que estarán contenidos en el plano excitado, y z_p, que

[6] Recordemos que el concepto de bloque no era el mismo en la interfaz que en el simulador

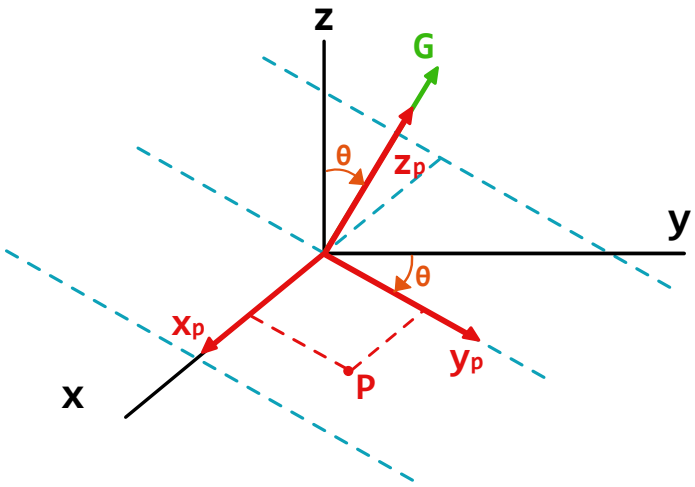


Figura 3.6: Esquema gráfico del gradiente selectivo

tendrá dirección normal a este. En estas condiciones, si existiese un punto P, cuyas coordenadas en este nuevo sistema fuesen $(1, 1, 0)$, podríamos calcular sus coordenadas con respecto al sistema canónico (x, y, z) mediante la matriz $\mathbf{R}$.

Generalizando, si partimos de un vector de posición $\mathbf{r_p}$, definido en (x_p, y_p, z_p), podemos hallar su equivalente $\mathbf{r}$ en coordenadas absolutas mediante:

$$\mathbf{r} = \mathbf{R} \cdot \mathbf{r_p}$$

La matriz $\mathbf{R}$ va a ser necesaria debido a que en el editor, todos los gradientes de desfase y adquisición de la secuencia se definen con respecto al sistema de referencia (x_p, y_p, z_p); por otra parte, en las máquinas reales todos los gradientes se aplican en las direcciones sin rotar (x, y, z). Un ejemplo es la función `EPI` (ver sección 2.4.2), en la que la adquisición se realiza siempre sobre el plano xy, sin importar el valor de los gradientes de excitación previos. Por tanto, será necesario realizar la operación `seq = R*EPI` antes de mandar la secuencia al simulador.

Esta cuestión se aborda de manera más detallada en el apéndice A.

Una vez definidos estos objetos `Phantom`, `Scanner` y `Sequence`, se hace una llamada a la función `simulate`, que devuelve la señal MR compleja en forma de vector. Para reconstruir la imagen a partir de esta señal, se utiliza la función `reshape`, que distribuye los elementos del vector en forma de una matriz NxN. En caso de haber obtenido la señal mediante una secuencia EPI, las líneas pares de la matriz se deberán invertir, ya que su adquisición se habrá producido de derecha a izquierda. Una vez hecho esto, tendremos el espacio K correspondiente a la imagen generada. Para generar esta imagen, bastará con realizar la transformada inversa de Fourier del espacio K, mediante `ifft`.

Finalmente, la función devuelve una matriz formada por la concatenación horizontal de las matrices imagen y espacio K. No se devuelve como dos matrices separadas porque este formato no sería compatible con C++, donde solo es posible un argumento de salida por función.

Función `displaySeq`

Esta función devuelve una matriz auxiliar creada a partir de la secuencia `seq`. Está formada por seis filas. La primera, correspondiente a la duración, constituye el eje de abscisas del diagrama temporal de la secuencia. En cuanto a las otras cinco, cada una representa una función que deberá ser dibujada sobre esa base de tiempos. Se dibujará una función para cada uno de los tres gradientes, otra para representar la amplitud del pulso RF, y una última función que indicará cuándo se activa y se desactiva la lectura.

3.4. Generación del `Phantom` a partir de un fichero NIfTI

Para cumplir con el último de los requisitos funcionales de la aplicación, se ha incluido en el fichero `Phantom.jl` del simulador la función `nii_brain_phantom3D`, cuya implementación completa se adjunta en el apéndice E.

La función, haciendo uso del paquete `NIfTI` de Julia, realiza la lectura de tres ficheros con extensión (`nii.gz`) que contienen los mapas de tiempos T1 y T2 y el mapa de densidad de protones (PD) del cuerpo de un paciente. Estos ficheros, obtenidos mediante técnicas de relaxometría sobre pacientes reales, pertenecen a la base de datos del Laboratorio de Procesado de Imagen de la Universidad de Valladolid.

Una vez realizada la lectura, los datos se procesan mediante submuestreo, eliminación de *outliers* y escalado. A continuación, se crea un objeto `Phantom` y se asigna a cada una de sus propiedades (T1, T2, ρ) los valores procedentes de los datos de los ficheros.

Capítulo 4

Resultados

En este capítulo se muestran y analizan los resultados obtenidos a partir de la ejecución de la aplicación, comenzando por la apariencia final de la interfaz. Se continúa con el análisis de las imágenes producidas por la simulación de las secuencias creadas. Por último, se muestran los resultados de la implementación de la herramienta de visualización 3D.

4.1. Apariencia de la interfaz y funcionamiento

La figura 4.1 muestra la apariencia final del editor, en la que apenas se aprecian diferencias con respecto al boceto de la figura 3.2. Esto se debe a que se ha respetado la distribución de paneles planteada al inicio del proyecto.

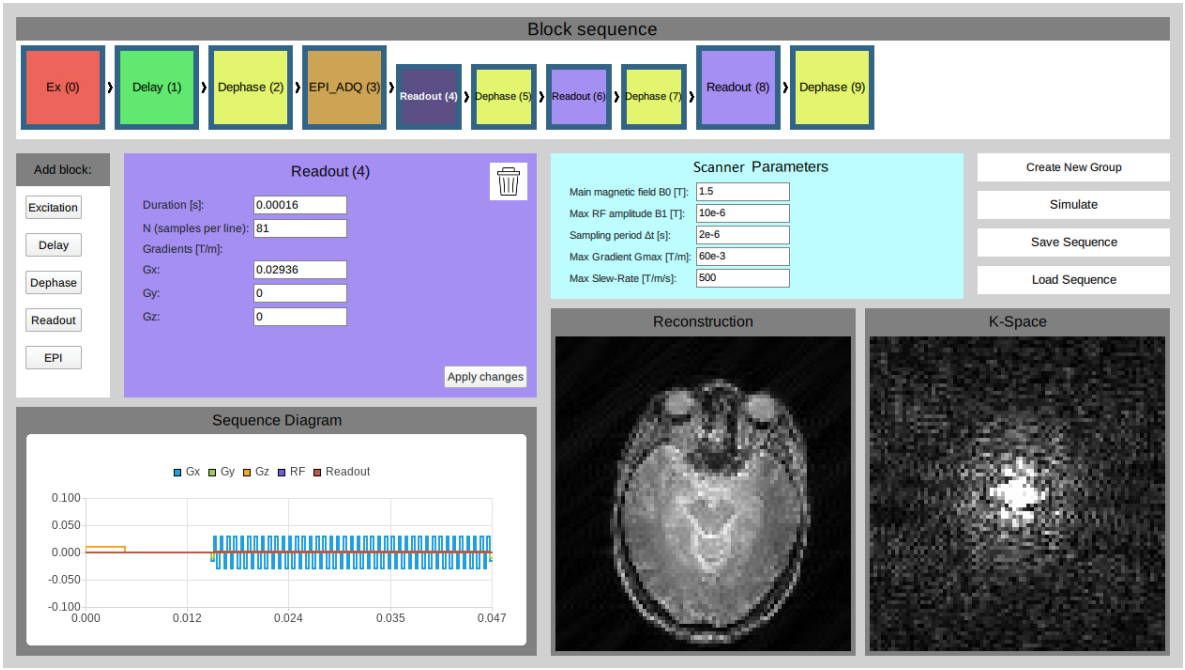


Figura 4.1: Apariencia final de la interfaz gráfica tras la simulación de una secuencia EPI

Con respecto al funcionamiento, se han implementado todas las funcionalidades descritas en el capítulo 3.1. Es posible, por tanto, crear secuencias arbitrarias, añadir bloques, moverlos, configurarlos, agruparlos y eliminarlos. Además, una vez se ha creado un grupo, este puede ser desplegado, situando el cursor encima del bloque correspondiente. De esta forma se muestran los bloques contenidos en el grupo, que son igualmente configurables.

Las opciones de guardar y cargar secuencias funcionan también correctamente, salvo un pequeño *bug* de QML en Linux que hace que los cuadros de diálogo correspondientes a ambas operaciones sean aparentemente iguales[1].

En cuanto a las imágenes, se pueden visualizar correctamente en los paneles destinados a ello. Además, el visor del diagrama de secuencia es interactivo, permitiendo al usuario ampliar la vista, reducirla o desplazarse libremente por el gráfico.

Por último, una vez pulsado el botón "Simulate", el simulador de Julia comienza a funcionar y la interfaz pierde su interactividad hasta que la simulación finaliza. Durante este periodo de tiempo, la interacción con el usuario se producirá por medio de la consola de Julia. En el caso de ejecutar el proyecto desde Qt Creator, esta salida por consola estará integrada en una de las ventanas del programa, tal y como muestra la figura 4.2. Se puede apreciar, además, que en la información proporcionada por la consola se incluye una barra de carga que indica el progreso de la simulación y el tiempo estimado restante.

```
10x165
Phantom size: 80 x 80 x 40 elements
Phantom dimensions: (0.395 x 0.395 x 0.195) m
Sequence[ τ = 47.17 ms | blocks: 166 | ADC: 81 | GR: 166 | RF: 1 | DEF: 3 ]
[ Info: Running simulation... [GPU = true, CPU = 1 thread(s)].
Dividing simulation in Nblocks=42
Starting simulation with Nspins=174182 and Nt=6899
Progress:  45%|████████████████                        |  ETA: 0:02:29[K
```

Figura 4.2: Salida de la aplicación durante una simulación, mostrada en la consola de Qt Creator

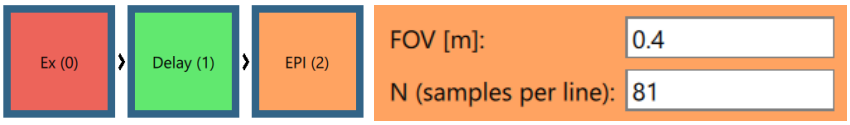
4.2. Resultados de la simulación

4.2.1. Simulación de una secuencia GE-EPI *single-shot*

En cuanto a las imágenes producidas por el simulador, en la propia figura 4.2 se muestra la salida generada por una secuencia GE-EPI *single-shot*. El resultado, de 81x81 píxeles, corresponde a una imagen de resonancia magnética en el plano transversal del cerebro.

En esta secuencia GE-EPI, el pulso RF se ha implementado con un bloque `Ex`; a continuación se añade un bloque `Delay` para controlar el TE. Después se añade un bloque `Dephase`, que lleva el cursor de lectura a la esquina inferior izquierda del espacio K. El siguiente bloque, llamado `EPI_ADQ` es un grupo que se repite 40 veces. Este grupo consiste en dos pares de bloques de lectura y desfase; el primer par realiza una lectura hacia la derecha y un desplazamiento hacia arriba; el segundo par realiza una lectura hacia la izquierda y otro desplazamiento hacia arriba. Tras las 40 repeticiones, se habrán leído 80 líneas. La lectura de la última línea se realiza mediante el último bloque `Readout` (con índice 8), que ya no forma parte del grupo. El último bloque `Dephase` no es obligatorio, ya que simplemente devuelve el punto de lectura al origen del espacio k.

La misma secuencia GE-EPI puede también ser generada directamente mediante la utilización de un bloque `EPI` (ver figura 4.3). Este hecho da cuenta de la alta versatilidad de la aplicación, ya que es posible adaptar su complejidad a las necesidades del usuario.

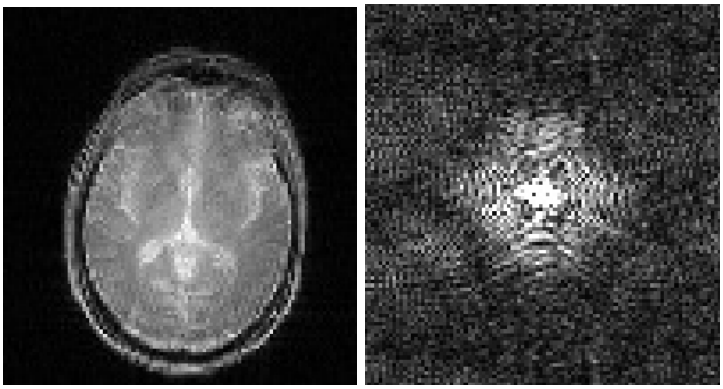


(a) (b)

Figura 4.3: (a) Secuencia GE-EPI creada mediante el bloque `EPI` y (b) parámetros de configuración del bloque

[1] El componente `FileDialog` de QML se utiliza para especificar la ruta y el nombre de los ficheros que se van a cargar o guardar. Cuenta con una propiedad, `fileMode`, que permite especificar la acción que se realizará en ese cuadro de diálogo: guardar o cargar. En Linux, aun haciendo uso de esta propiedad, los cuadros son aparentemente iguales. No obstante, se han hecho pruebas en Windows y este *bug* no se produce.

Para ilustrar algún ejemplo adicional, la figura 4.4 muestra las imágenes de 101x101 píxeles producidas por otra secuencia EPI, con la que se ha adquirido otro plano transversal del cerebro.



(a) (b)

Figura 4.4: (a) Imagen MRI cerebral de 101x101 píxeles obtenida a partir de una secuencia GE-EPI. (b) Espacio K correspondiente

4.2.2. Simulación de una secuencia FSE

Adicionalmente, es posible plantear la construcción de otros tipos de secuencias. Para ilustrar cómo es posible definir la jerarquía de bloques mediante la creación de grupos anidados, se decide construir una secuencia FSE (ver sección 2.2.3) con 10 ecos por disparo. La figura 4.5 muestra un grupo exterior, llamado `FSE`, formado por un bloque de excitación, que genera el pulso RF de 90^{o}, un retardo `Delay` y un desfase que coloca el puntero de lectura en la esquina superior derecha[2] del espacio K. Fijamos el número de repeticiones de este grupo a 8, lo que quiere decir que la secuencia constará de 8 disparos. El siguiente bloque es el grupo interior, llamado `ETL`, que controla el proceso de lectura propiamente dicho. Consta de un bloque `Ex`, que genera el pulso RF de reenfoque de 180^{o} y que invierte la posición del puntero de lectura, tomando el centro del espacio K como punto de simetría. A continuación, un bloque `Readout` realiza la lectura hacia la derecha, y un bloque `Dephase` mueve el puntero a la siguiente línea. El número de repeticiones de este grupo interior –correspondiente al *Echo Train Length*– se fija en 10, de modo que, al finalizar la simulación, se habrán leído un total de 80 líneas.

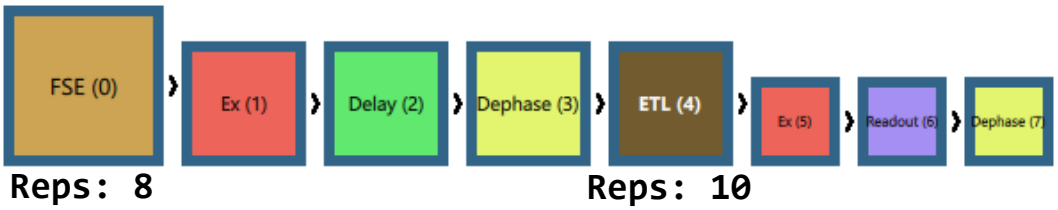


Figura 4.5: Jerarquía de bloques para una secuencia *Fast Spin Echo* (FSE)

Con respecto a la imagen resultado de la simulación de esta secuencia FSE, no ha sido posible su obtención. Para entender el motivo, podemos imaginar una secuencia FSE de dos disparos y ETL= 3, que produciría una adquisición de 6 líneas del espacio K, tal y como se muestra en la figura 4.6.

[2]Esto ocurrirá solo en la primera de las iteraciones. En las siguientes, la posición inicial del puntero de lectura será definida en función de las líneas ya leídas en iteraciones anteriores. La figura 4.6 muestra este cambio en la posición incial del puntero.

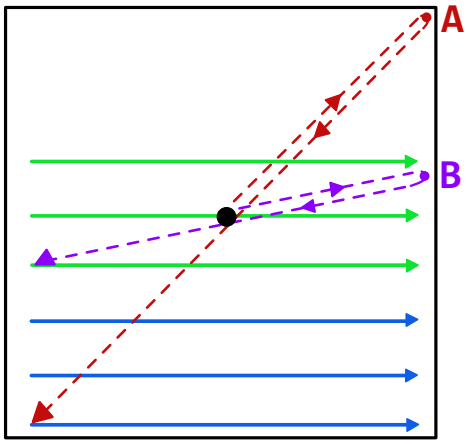


Figura 4.6: Adquisición de 6 líneas del espacio K mediante una secuencia FSE con ETL= 3

En esta secuencia se producen dos pulsos de excitación, para cada uno de los cuales se obtienen tres líneas del espacio K. A continuación del primero de ellos, y después del `Delay`, el bloque `Dephase` (que en la figura 4.5 tiene índice 3) debe desplazar el cursor hasta el punto A. Sin embargo, en la segunda iteración, ese mismo bloque debe llevar el cursor, ahora, hasta el punto B. Por tanto, el mismo bloque tendría que desplazar el puntero de lectura a dos puntos del espacio K distintos, en función de la iteración en la que se encuentre. Esta funcionalidad, por el momento, no está implementada en la aplicación.

Una posible solución pasaría por incluir una opción en los bloques de tipo `Dephase` que permitiese definir un *step*, o paso, en cada uno de sus gradientes. De esta forma, en cada iteración, los gradientes podrían ir variando a razón de este paso.

4.3. Visualización 3D

Cuando se pulsa sobre el botón "View 3D Model", presente en el menú de configuración de un bloque `Ex`, se despliega una nueva ventana. Esta ventana muestra, simultáneamente, el volumen completo y el plano que lo atraviesa, el cual representa el *slice* excitado. Las figuras 4.7 y 4.8 muestran dos adquisiciones llevadas a cabo con gradientes de excitación distintos y los modelos tridimensionales correspondientes.

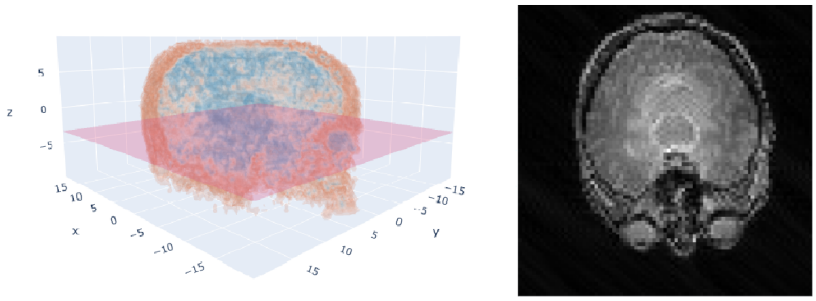


Figura 4.7: A la izquierda, la representación 3D del *slice* seleccionado cuando se activa un gradiente en z. A la derecha, la imagen reconstruida correspondiente

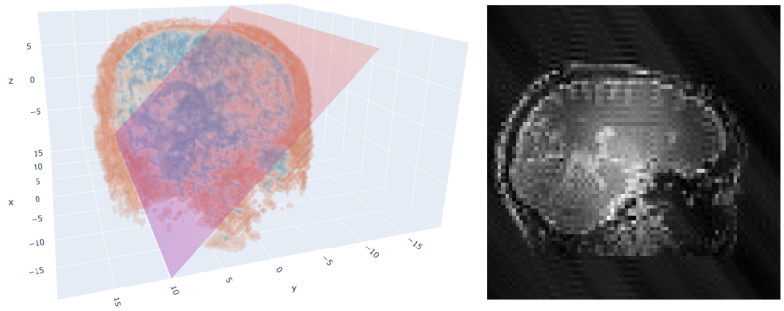


Figura 4.8: A la izquierda, la representación 3D del *slice* seleccionado cuando se activan gradientes en y y z. A la derecha, la imagen reconstruida correspondiente

Es posible evaluar esta herramienta desde dos puntos de vista:

- Calidad de la visualización: la calidad que ofrece la herramienta permite distinguir los elementos anatómicos presentes en la parte del cuerpo visualizada. No obstante, no ofrece un alto grado de detalle.
- Rendimiento: para que la carga del visor se complete, es necesario realizar un submuestreo bastante extremo (en un factor de 4 ó 5) a los datos del volumen. Adicionalmente, cuantos más puntos se desee representar, mayor será el tiempo de carga del visualizador, y más difícil será la interacción con él.

Capítulo 5

Conclusiones y líneas futuras

5.1. Conclusiones

El objetivo principal de este Trabajo de Fin de Grado (TFG) ha sido la creación de un editor de secuencias de resonancia magnética que resulte sencillo de usar, que sea versátil y cuyo uso sea compatible con fines tanto docentes como investigadores. Como hemos indicado en capítulos anteriores, hemos partido del simulador `KomaMRI.jl`, del cual hemos podido ver su enorme potencial en cuanto a metodología de simulación, pero también hemos observado que no disponía de una GUI que facilitase la creación de secuencias ni la observación del efecto de los gradientes de selección de corte de una manera sencilla e intuitiva. Ello ha sido nuestra principal motivación para llevar a cabo la tarea que hemos descrito en capítulos anteriores. Un aspecto fundamental en el desarrollo de este trabajo ha sido la necesidad de combinar fundamentos de adquisición y simulación de imagen por resonancia magnética, por una parte, con técnicas avanzadas de programación multiplataforma, por otra. Hemos perseguido, a su vez, que estas técnicas permitiesen la interoperabilidad de diferentes lenguajes de programación; por ello, la selección de las tecnologías precisas ha sido una cuestión abordada minuciosamente, lo cual nos ha llevado a elegir, por los motivos ya expuestos en el documento, C++, Qt y QML. Se ha realizado un énfasis especial en aspectos de comunicación entre estos lenguajes, tanto entre sí como con el lenguaje Julia en el cual está programado `KomaMRI.jl`.

La aplicación desarrollada permite al usuario con conocimientos de imagen por resonancia magnética, crear secuencias arbitrarias mediante la concatenación de bloques sin necesidad de crear código que interactúe con el simulador `KomaMRI.jl`. Con el objetivo de facilitar la repetición de agrupaciones de bloques se ha desarrollado también el concepto de grupo, concepto con una jerarquía asociada, el cual permite construir secuencias sin necesidad de repeticiones explícitas de operaciones similares. Ello aumenta la legibilidad de las secuencias y, en general, la elegancia en su diseño.

Por otra parte, los aspectos relacionados con la visualización han tenido un papel relevante durante la realización de este trabajo. En este sentido, el usuario no sólo dispone de los diagramas de secuencia y de los resultados de la simulación tanto en el espacio imagen como en el espacio K, sino que el resultado de los gradientes de selección se puede observar sobre un modelo tridimensional, lo cual facilita el establecimiento de los valores deseados para un determinado fin.

5.2. Aportaciones

Las aportaciones realizadas mediante la realización de este TFG ha sido:

- La concepción, diseño e implementación de una interfaz gráfica de usuario para un simulador de resonancia que, a juicio de sus autores [3], extiende el estado del arte en el campo de la simulación de resonancia magnética. Por tanto, el TFG contribuye, asimismo, a mejorar la usabilidad de dicho simulador pues, para su empleo, se requiere de conocimientos de programación en el lenguaje Julia.
- La interfaz no solamente proporciona acceso a la funcionalidad del simulador, sino que también constituye un editor de secuencias de resonancia magnética, el cual, mediante un esquema

jerarárquico, permite anidar operaciones de forma sencilla e intuitiva.

- Finalmente, se ha dotado de facilidades gráficas al simulador para poder conocer de forma bastante precisa el efecto de los gradientes de selección en el modelo sobre el cual se llevará a cabo la simulación. Por tanto, la selección del *slice*, a partir de la interfaz desarrollada, deja de ser un ejercicio a ciegas, como era hasta el momento, para pasar a ser un ejercicio guiado por la herramienta gráfica desarrollada.

Estas aportaciones se han traducido en el envío de una comunicación a un congreso nacional [34].

5.3. Líneas futuras de trabajo

Consideramos diversas posibilidades para futuros desarrollos de este trabajo. Algunas de ellas son las siguientes:

- Creación de una aplicación cliente-servidor para su funcionamiento mediante tecnologías Web. El apéndice F introduce brevemente el diseño de esta versión, y los métodos que habría que seguir para implementarla; una buena parte de los desarrollos llevados a cabo durante este TFG serían reutilizables para esta aplicación, de forma que el núcleo de los nuevos desarrollos estaría constituido por la lógica de los servidores que se muestran en la figura F.1.

- Se pretende, además, añadir nuevas funcionalidades en la interfaz. Se ha comprobado mediante pruebas de uso que la creación de secuencias *spin echo* y derivadas puede ser mejorable, debido a los motivos expuestos en la sección 4.2.2. La principal funcionalidad que se pretende añadir es, por tanto, la de poder controlar el *step*, o paso, de los gradientes incluidos dentro de grupos.

- Si bien el desarrollo gráfico realizado ha servido para cumplir uno de nuestros objetivos secundarios, el motor gráfico de Julia es un tanto limitado, tanto en una aplicación de escritorio (como ya hemos experimentado, con la necesidad de diezmar los modelos a representar) como, y fundamentalmente, en una aplicación cliente servidor. Por ello, debemos emplear nuevas tecnologías para conseguir que las facilidades gráficas estén al nivel deseado en una aplicación remota.

- El simulador `KomaMRI.jl` dispone de capacidad de paralelización; sin embargo, en el momento actual, únicamente dispone de soporte para GPUs de un único fabricante. Por otra parte, el grupo investigador ha promulgado la paralelización de este simulador en el lenguaje multidispositivo `OpenCL`. Resultado de esta labor ha sido un simulador similar a `KomaMRI.jl` pero con mejores prestaciones que el original y que puede ejecutarse en múltiples dispositivos paralelos [35]. Nuestro objetivo sería proporcionar a nuestro interfaz de una pasarela con la versión OpenCL del simulador.

- Finalmente, en el grupo investigador se ha desarrollado un simulador de consola de resonancia [2], cuya eficacia docente ha sido medida en un entorno educativo real [36] y ha sido empleado en varios cursos avalados por la Sociedad Española de Radiología Médica (SERAM). Dicho simulador tiene una máquina de simulación relativamente sencilla, pues el énfasis se ha puesto en la interactividad a pesar de la posible pérdida de realismo. Un objetivo a perseguir, en consecuencia, sería la conexión ambos simuladores así como la integración del editor de secuencias desarrollado en este TFG.

Apéndices

Apéndice A. Rotación de secuencias

A.1. Direcciones de codificación de fase y de frecuencia en secuencias rotadas

Todas las operaciones en el simulador se realizan en las direcciones (x, y, z), aunque esto requiere tener en cuenta alguna consideración. Supongamos que se activa un gradiente $\mathbf{G}$ durante la fase de excitación, por lo que esta excitación se convierte en selectiva. Podemos asumir que $\mathbf{G}$ es el resultado de rotar el vector $\mathbf{r} = (0, 0, 1)$ por una matriz de rotación $\mathbf{R}$.

Consideremos que $\mathbf{r} \in \mathbb{R}^3$ es el vector de posición de un punto con respecto a los ejes canónicos y $\mathbf{r}_p$ son las coordenadas de este punto dentro de un marco de referencia (x_p, y_p, z_p), en el que sus dos primeras componentes se encuentran en el plano definido por $\mathbf{G}$ y la tercera componente es ortogonal al plano, es decir, paralela a $\mathbf{G}$ (ver figura 3.6).

De acuerdo con la definición de la matriz $\mathbf{R}$, es sencillo comprobar que $\mathbf{r} = \mathbf{R}\mathbf{r}_p$. Si denominamos $\mathbf{G}_p$ a los gradientes aplicados durante la fase de adquisición (por ejemplo, $G_{p_{x_p}}$ al gradiente aplicado en la dirección de codificación de frecuencia y $G_{p_{y_p}}$ al gradiente de codificación de fase), será necesario rotar el vector $\mathbf{G}_p$ con la matriz $\mathbf{R}$ para que el simulador opere de la manera correcta.

Concretamente, si denominamos $\mathbf{G}_f$ al vector de gradientes a aplicar en el marco de referencia original (x, y, z), la operación que se pretende realizar es la siguiente:

$$I = \int M_{xy}(\mathbf{r})\delta\left(\mathbf{n}^T\mathbf{r}\right) e^{-j\gamma\mathbf{G}_f^T\mathbf{r}} d\mathbf{r} \tag{A.1}$$

Donde, por simplicidad, los gradientes se han considerado constantes en el tiempo. Obsérvese que la función δ garantiza la excitación de un único *slice*. Aplicando $\mathbf{r} = \mathbf{R}\mathbf{r}_p$:

$$I = \int M_{xy}(\mathbf{R}\mathbf{r}_p)\delta\left(\mathbf{n}^T\mathbf{R}\mathbf{r}_p\right) e^{-j\gamma\mathbf{G}_f^T\mathbf{R}\mathbf{r}_p} |\mathbf{R}| d\mathbf{r}_p$$

Recordemos que $|\mathbf{R}| = 1$. Asimismo, $\delta()$ solo es no nulo para puntos del plano $z_p = 0$. Así:

$$\int f(x, y, z)\delta(z)dz = f(x, y, 0).$$

Y, por lo tanto:

$$I = \int\int M_{xy}\left(\mathbf{R}\begin{bmatrix} x_p \\ y_p \\ 0 \end{bmatrix}\right) e^{-j\gamma\mathbf{G}_f^T\mathbf{R}\begin{bmatrix} x_p \\ y_p \\ 0 \end{bmatrix}} dx_p dy_p$$

Realizando el cambio $\mathbf{G}_p^T = \mathbf{G}_f^T\mathbf{R}$, podemos escribir:

$$I = \int\int M_{xy}(T(x_p, y_p)) e^{-j\gamma\left(G_{p_{x_p}}x_p + G_{p_{y_p}}y_p\right)} dx_p dy_p \tag{A.2}$$

Por lo tanto, la operación efectiva se realiza dentro del plano seleccionado, utilizando las direcciones de codificación de frecuencia y fase adecuadas. La rotación que debe sufrir la secuencia es:

$$\begin{aligned} \mathbf{G}_p^T &= \mathbf{G}_f^T\mathbf{R} \\ \mathbf{R}^T\mathbf{G}_f = \mathbf{G}_p &\implies \mathbf{G}_f = \mathbf{R}\mathbf{G}_p \end{aligned}$$

Por consiguiente, $\mathbf{G}_p$, que son los gradientes que se utilizarían en una secuencia no rotada, deben ser rotados por $\mathbf{A}$ para que la operación realizada en 3D en el marco de referencia original (es decir, la operación de la ecuación. (A.1)) sea equivalente a la que se pretende hacer directamente en el plano de la imagen (ecuación (A.2)). En consecuencia, en el simulador debemos realizar la operación `R*SEQ`.

A.2. Cálculo de la matriz de rotación

La matriz $\mathbf{R}$, utilizada para llevar a cabo una rotación con un ángulo θ alrededor del eje $\mathbf{u} = (u_x, u_y, u_z)$, tiene la forma [37]:

$$\mathbf{R} = \begin{bmatrix} cos\theta + u_x^2(1 - cos\theta) & u_x u_y(1 - cos\theta) - u_z sin\theta & u_x u_z(1 - cos\theta) + u_y sin\theta \\ u_y u_x(1 - cos\theta) + u_z sin\theta & cos\theta + u_y^2(1 - cos\theta) & u_y u_z(1 - cos\theta) - u_x sin\theta \\ u_z u_x(1 - cos\theta) - u_y sin\theta & u_z u_y(1 - cos\theta) + u_x sin\theta & cos\theta + u_z^2(1 - cos\theta) \end{bmatrix}$$

Siendo $\mathbf{u}$ un vector unitario.

Por lo tanto, para obtener la matriz $\mathbf{R}$ bastará con hallar los valores concretos de θ y (u_x, u_y, u_z). Estos valores se pueden calcular a partir del producto vectorial del vector de referencia $\mathbf{r} = (0, 0, 1)$ y el vector $\mathbf{G} = (G_x, G_y, G_z)$, formado por los gradientes activos durante la aplicación del pulso RF. Concretamente:

$$\mathbf{u} = \frac{\mathbf{r} \times \mathbf{G}}{|\mathbf{r} \times \mathbf{G}|}$$

Para calcular θ, hay que tener en cuenta que el módulo del producto vectorial se puede definir como:

$$|\mathbf{r} \times \mathbf{G}| = ||\mathbf{r}|| \cdot ||\mathbf{r}|| \cdot sin\theta$$

Entonces:

$$\theta = arcsin\left(\frac{|\mathbf{r} \times \mathbf{G}|}{||\mathbf{r}|| \cdot ||\mathbf{G}||}\right)$$

Estas operaciones se llevan a cabo cada vez que hay un bloque `Ex` en la secuencia. La implementación se realiza en la función `sim` del fichero `QtSlots.jl` (ver apéndice D).

Apéndice B. Funciones utilizadas en QML

La implementación de estas funciones se realiza en Javascript

B.1. Función `createScanner`

```
function createScanner(){
    var array = [];

    array[0] = Number(globalMenu.b0);
    array[1] = Number(globalMenu.b1);
    array[2] = Number(globalMenu.delta_t);
    array[3] = Number(globalMenu.gmax);
    array[4] = Number(globalMenu.smax);

    return array;
}
```

B.2. Función `createSeq`

```
function createSeq(){
    var array = [];
    for(var i=0; i<10; i++){
        array[i] = [];
    }

    // This function is recursive. It is called every
    // time the index n corresponds to a group block:
    function addToSeq(i){
        var children, reps;
        var j,k;
        if(i>=blockList.count){
            return;
        }

        if(isGroup(i)){
            reps = blockList.get(i).reps;
            children = blockList.get(i).children.count;
            for(j=0;j<reps;j++){
                for(k=0;k<children;k++){
                    addToSeq(blockList.get(i).children.get(k).number);
                }
            }
        } else {
            array[0].push(blockList.get(i).cod);
            array[1].push(blockList.get(i).dur);
            array[2].push(blockList.get(i).gx);
            array[3].push(blockList.get(i).gy);
            array[4].push(blockList.get(i).gz);
            array[5].push(blockList.get(i).b1x);
            array[6].push(blockList.get(i).b1y);
            array[7].push(blockList.get(i).delta_f);
            array[8].push(blockList.get(i).fov);
            array[9].push(blockList.get(i).n);
        }
    }

    for (var n=0; n<blockList.count; n++){
        if(!isChild(n)){
            addToSeq(n);
        }
    }
    return array;
}
```

B.3. Función saveSeq

```
function saveSeq(){
    var array = [];
    for(var i=0; i<15; i++){
        array[i] = [];
        for(var j=0; j<blockList.count; j++){
            array[i][j] = []; //We need to store the children so we need a 3D array
        }
    }
    for(j=0; j<blockList.count; j++){
        array[0][j].push(blockList.get(j).cod);
        array[1][j].push(blockList.get(j).dur);
        array[2][j].push(blockList.get(j).gx);
        array[3][j].push(blockList.get(j).gy);
        array[4][j].push(blockList.get(j).gz);
        array[5][j].push(blockList.get(j).b1x);
        array[6][j].push(blockList.get(j).b1y);
        array[7][j].push(blockList.get(j).delta_f);
        array[8][j].push(blockList.get(j).fov);
        array[9][j].push(blockList.get(j).n);
        array[10][j].push(blockList.get(j).ngroups);

        for(var k=0; k<blockList.get(j).name.length; k++){
            array[11][j].push(blockList.get(j).name.charCodeAt(k));
        }

        for(k=0; k<blockList.get(j).children.count; k++){
            array[12][j].push(blockList.get(j).children.get(k).number);
        }

        if(isChild(j)){
            array[13][j].push(1);
        } else {
            array[13][j].push(0);
        }

        array[14][j].push(blockList.get(j).reps);
    }

    return array;
}
```

Apéndice C. Implementación de *slots* en C++

C.1. Función `plot3D`

```
void Backend::plot3D(double gx,double gy, double gz, double delta_f){
   jl_value_t *jl_plot = jl_eval_string("@cfunction(KomaMRI.plot3D,Any,(Cdouble,Cdouble
       ↪ ,Cdouble,Cdouble))");
   void (*c_plot)(double,double,double,double) = (void (*)(double,double,double,double)
       ↪ ) jl_unbox_voidpointer(jl_plot);

   c_plot(gx,gy,gz,delta_f);
}
```

C.2. Función `simulate`

```
void Backend::simulate(QVector<double> sys, QVector<QVector<double>> seq){
  // -------------------------------- SEQUENCE -----------------------------------
  int rows = 0;
  int cols = 0;
  for(const auto& num : seq[0]){
      cols++;
  }

  for(const auto& j : seq){
      rows++;
  }

  std::cout << rows << "x" << cols << endl;


  // ------------------------- SEQUENCE MATRIX CREATION --------------------------
  // this matrix will be the input argument to our Julia function

  // Create 2D array of float64 type
  jl_value_t* array_type_seq = jl_apply_array_type((jl_value_t*)jl_float64_type, 2);
  jl_array_t* x              = jl_alloc_array_2d(array_type_seq, rows, cols);

  // Get array pointer
  double *p = (double*)jl_array_data(x);

  // Get number of dimensions
  int ndims = jl_array_ndims(x);
  // Get the size of the i-th dim
  size_t size0 = jl_array_dim(x,0);
  size_t size1 = jl_array_dim(x,1);

  // Fill array with data
  for(size_t i=0; i<size1; i++)
      for(size_t j=0; j<size0; j++)
          p[j + size0*i] = seq[j][i];

  jl_value_t* seq_arg = (jl_value_t*)x;


  // ----------------------- SYSTEM (GLOBAL PARAMETERS) --------------------------
  jl_value_t* array_type_sys = jl_apply_array_type((jl_value_t*)jl_float64_type, 1);
  jl_array_t* y              = jl_alloc_array_1d(array_type_sys, 5);

  double *q =(double*)jl_array_data(y);

  for(size_t i=0; i<jl_array_len(y); i++)
      q[i] = sys[i];

  jl_value_t* sys_arg = (jl_value_t*)y;
```

```
  // --------------------------- CALLING JULIA FUNCTION -----------------------------
  jl_value_t *mod = jl_eval_string("KomaMRI");
  jl_function_t *jl_sim=(jl_function_t*)jl_get_global((jl_module_t*)mod,jl_symbol("sim"));

  jl_array_t* output = (jl_array_t*)jl_call2(jl_sim, seq_arg, sys_arg);

  size_t size_recon0 = jl_array_dim(output,0);   //Rows
  size_t size_recon1 = jl_array_dim(output,1);   //Columns

  std::cout << size_recon0 << " x " << size_recon1 << endl;

  double *matData = (double*)jl_array_data(output);


  // -------------------------- OUTPUT REPRESENTATION -----------------------------
  // OpenCV Matrix construction
  cv::Mat mat(size_recon0,size_recon1,CV_64F);

  for(int i=0; i<size_recon1; i++)
      for(int j=0; j<size_recon0; j++){
          mat.at<double>(j,i) = matData[j + size_recon0*i];
      }

  cv::Mat recon(size_recon0,size_recon1/2,CV_64F), recon_norm;
  cv::Mat kspace(size_recon0,size_recon1/2,CV_64F), kspace_norm;

  for(int i=0; i<size_recon1/2; i++)
      for(int j=0; j<size_recon0; j++){
          recon.at<double>(j,i) = mat.at<double>(j,i);
          kspace.at<double>(j,i) = mat.at<double>(j,i+size_recon1/2);
      }

  // Normalization and conversion to 0-255 scale
  cv::normalize(recon,recon_norm,0,1,cv::NORM_MINMAX);
  cv::normalize(kspace,kspace_norm,0,1,cv::NORM_MINMAX);

  recon_norm.convertTo(recon_norm,CV_8UC3,255.0);
  kspace_norm.convertTo(kspace_norm,CV_8UC3,255.0);

  // Save recon image into temp file
  cv::imwrite("../temp/recon.jpg",recon_norm);
  cv::imwrite("../temp/kspace.jpg",kspace_norm);

 // This signal will be received by QML so that it will refresh the displayed images
 // (The ones in temp directory):
 emit imageRequested();
}
```

C.3. Función saveSeq

```
void Backend::saveSeq(QVector<QVector<QVector<double>>> seq, QString path){
    int rows = 0;
    int cols = 0;
    int depth = 0;

    int i,j;

    int children, chars;

    for(const auto& num : seq[0]){
        cols++;
    }

    for(const auto& j : seq){
        rows++;
    }
```

```
    for(const auto& k : seq[12][0]){
        depth++;
    }

    // Create and open a text file
    ofstream MyFile(path.toStdString().substr(7));

    // Write to the file
    MyFile << "import QtQuick 2.0" << endl;
    MyFile << "ListModel{" << endl;

    for(i=0;i<cols;i++){
        MyFile << " ListElement{" << endl;
        MyFile << "      cod:" << seq[0][i][0] << endl;
        MyFile << "      dur:" << seq[1][i][0] << endl;
        MyFile << "      gx:" << seq[2][i][0] << endl;
        MyFile << "      gy:" << seq[3][i][0] << endl;
        MyFile << "      gz:" << seq[4][i][0] << endl;
        MyFile << "      b1x:" << seq[5][i][0] << endl;
        MyFile << "      b1y:" << seq[6][i][0] << endl;
        MyFile << "      delta_f:" << seq[7][i][0] << endl;
        MyFile << "      fov:" << seq[8][i][0] << endl;
        MyFile << "      n:" << seq[9][i][0] << endl;
        MyFile << "      grouped:false" << endl;
        MyFile << "      ngroups:" << seq[10][i][0] << endl;

        chars = 0;
        for(const auto& m : seq[11][i]){
            chars++;
        }
        MyFile << "      name:\"";
        for(j=0;j<chars;j++){
            MyFile << char(seq[11][i][j]);
        }
        MyFile << "\"" << endl;

        MyFile << "      children:[" << endl;

        children = 0;
        for(const auto& n : seq[12][i]){
            children++;
        }
        for(j=0;j<children;j++){
            MyFile << "          ListElement{" << endl;
            MyFile << "              number:" << seq[12][i][j] << endl;
            MyFile << "          }";
            if(j!=(children-1)){
                MyFile << ",";
            }
            MyFile << endl;
        }

        MyFile << "      ]" << endl;

        if(seq[13][i][0]==0){
            MyFile << "      collapsed:false" << endl;
        } else if (seq[13][i][0]==1){
            MyFile << "      collapsed:true" << endl;
        }

        MyFile << "      reps:" << seq[14][i][0] << endl;

        MyFile << " }" << endl;
    }
    MyFile << "}" << endl;

    // Close the file
    MyFile.close();
}
```

C.4. Función displaySeq

```
QVector<QVector<double>> Backend::displaySeq(){
    jl_value_t *mod = jl_eval_string("KomaMRI");
    jl_function_t *jl_displaySeq = (jl_function_t*)jl_get_global((jl_module_t*)mod,
        ↪ jl_symbol("displaySeq"));

    jl_array_t* seq =(jl_array_t*)jl_call0(jl_displaySeq);

    size_t size_seq0 = jl_array_dim(seq,0);   //Rows
    size_t size_seq1 = jl_array_dim(seq,1);   //Columns

    double *seqData = (double*)jl_array_data(seq);

    QVector<QVector<double>> seq_mat = CreateMatrix(size_seq0,size_seq1);

    for(int i=0; i<size_seq1; i++){
        for(int j=0; j<size_seq0; j++){
            seq_mat[j][i] = seqData[j + size_seq0*i];
        }
    }

    return seq_mat;
}
```

Apéndice D. Fichero `QtSlots.jl`

D.1. Función `plot3D`

```
plot3D(gx,gy,gz,Δf) = begin

    downsample_factor = 6;

    traces = GenericTrace[];

    path = @__DIR__; path = path*"/datatypes/phantom/brain004/T1map.nii.gz";

    # Phantom
    ni = niread(path);
    data = ni.raw;

    # Clip outliers
    valorPercentil = percentile(data[:],99);
    data[data.>=valorPercentil].=valorPercentil;

    # Downsampling
    dsample_data = (data[1:downsample_factor:end,1:downsample_factor:end,1:
        ↪ downsample_factor:end])

    # Normalization
    mini, maxi = extrema(dsample_data)
    normed = Float32.((dsample_data .- mini) ./ (maxi - mini));

    M, N, L = size(dsample_data)

    # FOV in mm
    FOVx = (M-1)*downsample_factor*1e-1
    FOVy = (N-1)*downsample_factor*1e-1
    FOVz = (L-1)*downsample_factor*1e-1

    x = -size(dsample_data)[1]/2 : (size(dsample_data)[1]/2) -1;
    y = -size(dsample_data)[2]/2 : (size(dsample_data)[2]/2) -1;
    z = -size(dsample_data)[3]/2 : (size(dsample_data)[3]/2) -1;

    ratio = FOVx/M

    X,Y,Z = mgrid(x,y,z);

    phantom = volume(x=X[:].*ratio,
                     y=Y[:].*ratio,
                     z=Z[:].*ratio,
                     value = normed[:],
                     opacity=0.1,
                     surface_count= 8,
                     isomin= 0.1,
                     isomax= 0.9,
                     colorscale = colors.RdBu_3
    )

    push!(traces,phantom);


    # Slices
    X,Y = mgrid(x,y);
    Z = 0*X;

    op = 0.3;

    d = Δf/γ

    for i in 1:n_slices
```

```
        if gz == 0
            if gy == 0
                trace = surface(x=Z,y=Y,z=X,
                                opacity=op,
                                showscale = false);
            else
                trace = surface(x=X,y=-X*gx/gy,z=Y,
                                opacity=op,
                                showscale = false);
            end
        else
            trace = surface(x=X,y=Y,z=(-X*gx.-Y*gy)/gz,
                            opacity=op,
                            showscale = false);
        end
        push!(traces,trace);
    end

    # Layout and plotting
    layout = Layout(
        scene_xaxis_range = [-FOVx/2,FOVx/2],
        scene_yaxis_range = [-FOVy/2,FOVy/2],
        scene_zaxis_range = [-FOVz/2,FOVz/2],
        scene_aspectratio = attr(x=.02*size(dsample_data)[1],
                                 y=.02*size(dsample_data)[2],
                                 z=.02*size(dsample_data)[3])
    )

    w = Window();
    p = plot(traces,layout);
    body!(w,p);
end
```

D.2. Función `sim`

```
sim(mat,vec) = begin

    let
        #----------------------- Phantom --------------------------
        ss = 5
        phantom = KomaMRI.nii_brain_phantom3D(ss)


        #----------------------- Scanner --------------------------
        sys = Scanner()
        sys.B0 =        vec[1]      # Main magnetic field [T]
        sys.B1 =        vec[2]      # Max RF amplitude [T]
        sys.ADC_Δt =    vec[3]      # ADC sampling time
        sys.Gmax =      vec[4]      # Max Gradient [T/m]
        sys.Smax =      vec[5]      # Max Slew-Rate


        # --------------------- Secuence --------------------------
        global seq = Sequence()

        for i=1:size(mat)[2]

            if mat[1,i] == 1 # Excitation
                            B1 = mat[6,i] + mat[7,i]im;
                            duration = mat[2,i]
                            Δf = mat[8,i];
                            EX = PulseDesigner.RF_hard(B1,
                                    duration,
                                    sys
                                    G=[mat[3,i] mat[4,i] mat[5,i]],
                                    Δf)
```

```
                    seq += EX

                    G = EX.GR.A; G = [G[1];G[2];G[3]];
                    REF = [0;0;1];

        # Cross product:
        global cross_prod = LinearAlgebra.cross(REF,G);
        # Rotation axis (n = axb) Normalized cross product:
        n = normalize(cross_prod);
        # Rotation angle:
        θ = asin(norm(cross_prod)/((norm(REF))*(norm(G))));
        # Rotation matrix:
        global R = [cos(θ)+n[1]^2*(1-cos(θ)),
                    n[1]*n[2]*(1-cos(θ))-n[3]*sin(θ),
                    n[1]*n[3]*(1-cos(θ))+n[2]*sin(θ);
                    n[2]*n[1]*(1-cos(θ))+n[3]*sin(θ),
                    cos(θ)+n[2]^2*(1-cos(θ)),
                    n[2]*n[3]*(1-cos(θ))-n[1]*sin(θ);
                    n[3]*n[1]*(1-cos(θ))-n[2]*sin(θ),
                    n[3]*n[2]*(1-cos(θ))+n[1]*sin(θ),
                    cos(θ)+n[3]^2*(1-cos(θ))];


    elseif mat[1,i] == 2 # Delay
        DELAY = Delay(mat[2,i])
        seq += DELAY


    elseif ((mat[1,i] == 3) || (mat[1,i] == 4)) # Dephase or Readout

        AUX = Sequence()

        ζ = abs(sum([mat[3,i],mat[4,i],mat[5,i]])) / sys.Smax
        ϵ1 = mat[2,i]/(mat[2,i]+ζ)

        AUX.GR[1] = Grad(mat[3,i],mat[2,i],ζ)
        AUX.GR[2] = ϵ1*Grad(mat[4,i],mat[2,i],ζ)
        AUX.GR[3] = Grad(mat[5,i],mat[2,i],ζ)

        AUX.DUR = convert(Vector{Float64}, AUX.DUR)
        AUX.DUR[1] = mat[2,i] + 2*ζ

        # No samples during Dephase interval:
        AUX.ADC[1].N = (mat[1,i] == 3) ? 0 : trunc(Int,mat[10,i])

        # Duration must be explicitly stated:
        AUX.ADC[1].T = mat[2,i]
        AUX.ADC[1].delay = ζ

        AUX = (norm(cross_prod)>0) ? R*AUX : AUX

        if(mat[1,i]==4)
            global N_x = trunc(Int,mat[10,i])
        end

        seq += AUX

    elseif mat[1,i] == 5 # EPI
         FOV = mat[9,i]
         N = trunc(Int,mat[10,i])

         EPI = PulseDesigner.EPI(FOV, N, sys)
         EPI = (norm(cross_prod)>0) ? R*EPI : EPI
         seq += EPI

        global N_x = N
    end
end
```

```
        seq.DEF = Dict("Nx"=>N_x,"Ny"=>N_x,"Nz"=>1)

        #Simulation
        Ncores = Hwloc.num_physical_cores()
        simParams = Dict{String,Any}("Δt"=>1e-3,
                                     "Δt_rf"=>1e-4,
                                     "gpu"=>has_cuda(),
                                     "Nthreads"=>Ncores)

        global w = nothing
        signal = simulate(phantom, seq, sys; simParams, w)

        # Reconstruction
        Nx, Ny = [N_x;N_x]
        kdata = reshape(signal,(Nx,Ny))
        kdata[:,2:2:Ny] = kdata[Nx:-1:1,2:2:Ny]  # Only for EPI adquisitions
        kdata = convert(Array{Complex{Float64},2},kdata)

        kdata_abs = abs.(kdata);
        image = abs.(fftshift(ifft(kdata)))

        # Clip k-space outliers
        percentileValue = percentile(kdata_abs[:],99);
        kdata_abs[kdata_abs.>=percentileValue].=percentileValue;

        [image kdata_abs]
    end
end
```

D.3. Función displaySeq

```
displaySeq() = begin
    aux = zeros(6,length(seq))

    for i=1:length(seq)
        aux[1,i] = seq[i].DUR[1]               #duration
        aux[2,i] = seq[i].GR[1].A              #gx
        aux[3,i] = seq[i].GR[2].A              #gy
        aux[4,i] = seq[i].GR[3].A              #gz
        aux[5,i] = seq[i].RF[1].A              #RF amplitude
        aux[6,i] = seq[i].ADC[1].N>0 ? 1 : 0   #Readout on/off
    end

    aux
end
```

Apéndice E. Función `nii_brain_phantom3D`

```
function nii_brain_phantom3D(ss=1::Int)
    path = @__DIR__; path = path*"/phantom/brain004/";
    T1_path = path*"T1map.nii.gz";
    T2_path = path*"T2map.nii.gz";
    PD_path = path*"PDmap.nii.gz";

    T1_ni = niread(T1_path);
    T2_ni = niread(T2_path);
    PD_ni = niread(PD_path);

    T1_data = T1_ni.raw
    T2_data = T2_ni.raw
    ρ = PD_ni.raw

    M, N, L = size(T1_data)

    # Subsampling
    T1_data = T1_data[1:ss:end,1:ss:end,1:ss:end]
    T2_data = T2_data[1:ss:end,1:ss:end,1:ss:end]
    ρ = ρ[1:ss:end,1:ss:end,1:ss:end]
    T2s_data = T2_data.+10

    # Clip outliers
    T1_percentile = percentile(T1_data[:],99)
    T1_data[T1_data.>=T1_percentile].= T1_percentile;

    T2_percentile = percentile(T2_data[:],99)
    T2_data[T2_data.>=T2_percentile].= T2_percentile;

    ρ_percentile = percentile(ρ[:],99)
    ρ[ρ.>=ρ_percentile].= ρ_percentile;

    T2s_percentile = percentile(T2s_data[:],99)
    T2s_data[T2s_data.>=T2s_percentile].= T2s_percentile;

    # Each voxel is 1mm x 1mm x 1mm.
    # Voxels have the same size in the three dimensions (cubic voxels)
    Δx = 1e-3*ss

    M, N, L = size(T1_data)

    println("Phantom size: $M x $N x $L elements")

    FOVx = (M-1)*Δx
    FOVy = (N-1)*Δx
    FOVz = (L-1)*Δx

    println("Phantom dimensions: ($FOVx x $FOVy x $FOVz) m")

    x = -FOVx/2:Δx:FOVx/2
    y = -FOVy/2:Δx:FOVy/2
    z = -FOVz/2:Δx:FOVz/2

    X,Y,Z = mgrid(x,y,z);

    Δw = zeros(M,N,L)

    # Convert miliseconds into seconds
    T1_data .*= 1e-3
    T2_data .*= 1e-3
    T2s_data .*= 1e-3

    phantom = Phantom(name="nii_brain3D",
                          x=X[ρ.!=0],
                          y=Y[ρ.!=0],
```

```
                            z=Z[ρ.!=0],
                            ρ=ρ[ρ.!=0],
                            T1=T1_data[ρ.!=0],
                            T2=T2_data[ρ.!=0],
                            T2s=T2s_data[ρ.!=0],
                            Δw=Δw[ρ.!=0]
                            )
    phantom
end
```

Apéndice F. Versión Web: Diseño

Este apéndice incluye una introducción al diseño de la versión web de la aplicación. A pesar de que el código de la versión de escritorio es reutilizable desde una perspectiva basada en el navegador, se pretende, en este caso, que la lógica de la aplicación se ejecute en un servidor, por lo que es necesario el uso de una arquitectura cliente-servidor. Esto lleva a la necesidad de añadir elementos adicionales. La figura F.1 proporciona un diagrama detallado de estos elementos y su funcionamiento. Concretamente, deben añadirse dos elementos:

- **Un servidor Web**, capaz de proporcionar al cliente los ficheros resultantes de la compilación a WebAssembly, que son, esencialmente:
  - Un fihero (`.wasm`): contiene el código binario guardado en el formato WebAssembly.
  - Un fichero (`.html`): en él se carga el fichero Javascript.
  - Un fichero (`.js`): hace de intermediario entre WebAssembly y el navegador. Se encarga de cargar el fichero (`.wasm`).

  Estos tres ficheros serán cargados en el navegador web del cliente, de forma que este sea capaz de visualizar la interfaz e interactuar con ella.
- **Un servicio web de tipo RESTful**: este servicio ejecutará los comandos solicitados remotamente y devolverá sus resultados. La necesidad de este servidor intermediario se debe a que desde el código WebAssembly de la aplicación desarrollada no se puede llamar directamente al código Julia del simulador. El servidor de tipo RESTful se encargará, por lo tanto, de ejecutar este código Julia.

  Un servicio web RESTful funciona de manera similar a una página web, por lo que simplemente se deben enviar peticiones y recibir sus correspondientes respuestas. Este servidor soporta las siguientes funciones:
  - `POST/commands`: `POST` ejecuta un comando y guarda su salida en un fichero temporal cuyo identificador se devuelve por defecto en la cabecera `Location` de la respuesta.
  - `GET/commands/{resultID}`: `GET` devuelve el contenido del fichero con identificador local `resultID` (el obtenido en la cabecera Location de la respuesta `POST`).
  - `DELETE/comandos/resultadoID`: elimina el fichero temporal creado como resultado del comando `POST`.

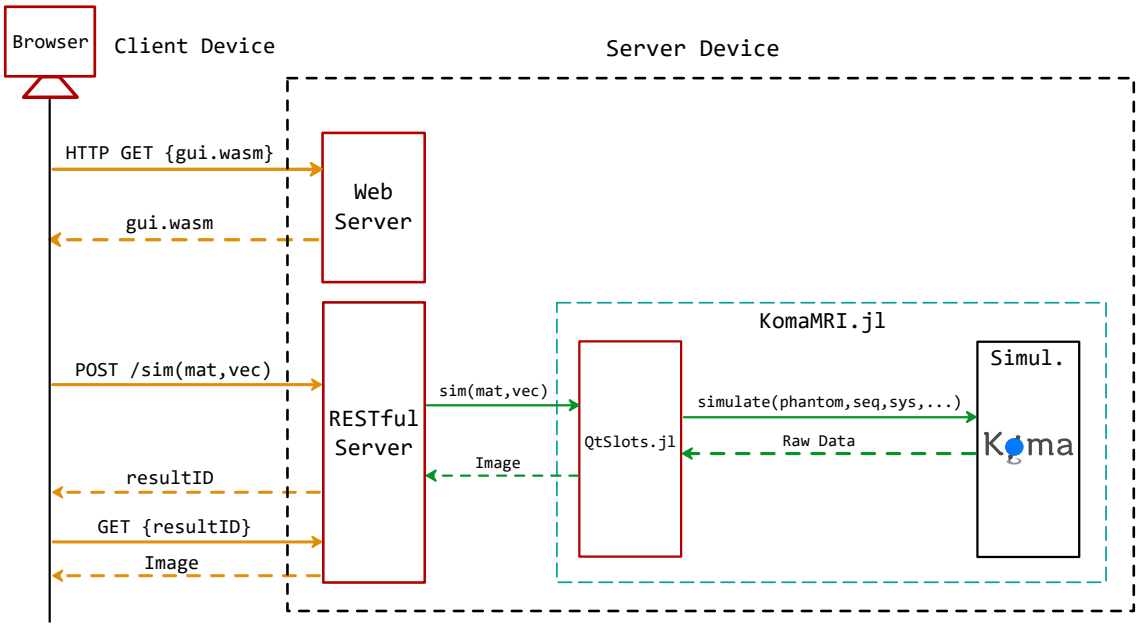


Figura F.1: Diagrama de funcionamiento de la aplicación web. Las flechas naranjas representan los comandos del protocolo establecido entre el cliente y el servidor (HTTP), mientras que las flechas verdes corresponden a las llamadas a las funciones de Julia

Bibliografía

[1] R. H. Caverly, "MRI fundamentals: RF aspects of magnetic resonance imaging (MRI)", *IEEE Microwave Magazine*, vol. 16, no. 6, pp. 20–33, 2015.

[2] D. Treceño-Fernández, J. Calabia-Del-Campo, M. L. Bote-Lorenzo, E. G. Sánchez, R. de Luis-García, and C. Alberola-López, "A Web-Based Educational Magnetic Resonance Simulator: Design, Implementation and Testing", *Journal of medical systems*, vol. 44, no. 1, pp. 1–11, 2019.

[3] C. Castillo-Passi, R. Coronado, G. L. Varela-Mattatall, C. Alberola-López, R. Botnar, and P. Irarrázaval-Mena, "KomaMRI.jl: An Open-Source Framework for General MRI Simulations with GPU Acceleration", *(enviado)*, 2022.

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

[5] T. Qt-Company, *Qt, Cross-platform software development for embedded & desktop*. 2022. dirección: `https://www.qt.io/` (visitado 11-07-2022).

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

[7] E. Moya, C. Alberola y F. Simmross, *Síntesis de imagen de resonancia magnética mediante GPU*. Trabajo de Fin de Grado, Universidad de Valladolid, 2017.

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

[9] H. D. Young y R. A. Freedman, *Física universitaria con física moderna*. Pearson Education, 2013.

[10] E.-S. H. Ibrahim, *Heart Mechanics: Magnetic Resonance Imaging—Advanced Techniques, Clinical Applications, and Future Trends*. CRC Press, 2017.

[11] C. Alberola y C. Castillo-Passi, «An overview of MRISim.jl», inf. téc., 2022. dirección: `https://es.overleaf.com/read/jjpbfkpcrdjh`.

[12] M. Jouda, «Innovative Concepts for the Electronic Interface of Massively Parallel MRI Phased Imaging Arrays», Tesis doct., 2016.

[13] D. Folio and A. Ferreira, "Two-Dimensional Robust Magnetic Resonance Navigation of a Ferromagnetic Microrobot Using Pareto Optimality", *IEEE Transactions on Robotics*, vol. 33, no. 3, pp. 583–593, 2017.

[14] C. B. Paschal and H. D. Morris, "K-space in the clinic", *Journal of Magnetic Resonance Imaging*, vol. 19, no. 2, pp. 145–159, 2004.

[15] J. Lafuente and L. Hernández, "Técnica de la Imagen por Resonancia Magnética", *Resonancia Magnética del Sistema Músculo-Esquelético*, pp. 9–28, 2016.

[16] Julia-Contributors, *Julia Documentation*. 2022. dirección: `https://docs.julialang.org/en/v1/` (visitado 14-09-2022).

[17] L. G. Hanson, "A Graphical Simulator for Teaching Basic and Advanced MR Imaging Techniques", *RadioGraphics*, vol. 27, no. 6, e27–e27, 2007.

[18] S. B. McKagan, K. K. Perkins, M. Dubson, *et al.*, "Developing and researching PhET simulations for teaching quantum mechanics", *American Journal of Physics*, vol. 76, no. 4, pp. 406–417, 2008.

[19] T. Hackländer and H. Mertens, "Virtual MRI: A PC-based simulation of a clinical MR scanner", *Academic radiology*, vol. 12, no. 1, pp. 85–96, 2005.

[20] G. Torheim, P. A. Rinck, R. A. Jones, and J. Kvaerness, "A simulator for teaching MR image contrast behavior", *Magnetic Resonance Materials in Physics, Biology and Medicine*, vol. 2, no. 4, pp. 515–522, 1994.

[21] T. Stöcker, K. Vahedipour, D. Pflugfelder, and N. J. Shah, "High-performance computing MRI simulations", *Magnetic resonance in medicine*, vol. 64, no. 1, pp. 186–193, 2010.

[22] R. Kose and K. Kose, "Blochsolver: A GPU-optimized fast 3D MRI simulator for experimentally compatible pulse sequences", *Journal of Magnetic Resonance*, vol. 281, pp. 51–65, 2017.

[23] C. G. Xanthis, I. E. Venetis, A. Chalkias, and A. H. Aletras, "Mrisimul: A GPU-based parallel approach to MRI simulations", *IEEE Transactions on Medical Imaging*, vol. 33, no. 3, pp. 607–617, 2013.

[24] F. Liu, R. Kijowski, and W. F. Block, "Mrilab: Performing fast 3D parallel mri numerical simulation on a simple PC", *ISMRM Scientific Meeting & Exhibition*, vol. 2072, 2013.

[25] T. Knopp and M. Grosser, "MRIReco.jl: An MRI reconstruction framework written in Julia", *Magnetic resonance in medicine*, vol. 86, no. 3, pp. 1633–1646, 2021.

[26] T. Qt-Company, *Qt Documentation*. 2022. dirección: `https://doc.qt.io` (visitado 05-07-2022).

[27] Kitware, *About CMake*. 2022. dirección: `https://cmake.org/overview/` (visitado 05-07-2022).

[28] A. Rossberg, *WebAssembly Specification*. W3C, 2022.

[29] Emscripten-Contributors, *Emscripten Documentation*. 2015. dirección: `https://emscripten.org/docs/` (visitado 30-08-2022).

[30] A. G. Schreyer and S. K. Warfield, "Surface Rendering", *3D Image Processing: Techniques and Clinical Applications*, pp. 31–34, 2002.

[31] O. Taubmann, M. Berger, M. Bögel, Y. Xia, M. Balda, and A. Maier, "Computed Tomography: An Introductory Guide", *Medical Imaging Systems*, pp. 147–189, 2018.

[32] W. Schroeder, K. Martin, L. Avila y C. Law, *The VTK user's guide*. Kitware, 2010.

[33] G. Bradski, "The openCV library.", *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, vol. 25, no. 11, pp. 120–123, 2000.

[34] P. Villacorta-Aylagas, I. Ferández-Arias, C. Castillo-Passi, *et al.*, "A Cross-Platform Editor and Simulator of Magnetic Resonance Imaging Sequences: Design and Implementation", *(enviado)*, 2022.

[35] I. Fernández-Arias, P. Villacorta-Aylagas, C. Castillo-Passi, *et al.*, "Simulación Paralela Multi-dispositivo de Secuencias de Imagen por Resonancia Magnética", *(enviado)*, 2022.

[36] D. Treceño-Fernández, J. Calabia-del-Campo, F. Matute-Teresa et al., «Magnetic Resonance Simulation in Education: Quantitative Evaluation of an Actual Classroom Experience,» *Sensors*, vol. 21, n.º 18, pág. 6011, 2021.

[37] C. J. Taylor y D. J. Kriegman, «Minimization on the Lie group SO (3) and related manifold», inf. téc., 1994.