Universidad de Valladolid

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

MÁSTER EN INGENIERÍA DE TELECOMUNICACIÓN

MASTER'S THESIS

Contributions to Advanced Magnetic Resonance Imaging Simulation: A Model for Dynamic Simulation and a Web Interface for Pulse Sequence Development and Visualization

AUTHOR:

Pablo Villacorta Aylagas

SUPERVISORS:

Carlos Alberola López, Manuel Rodríguez Cayetano

March 2025

Title:	**Contributions to Advanced Magnetic Resonance Imaging Simulation: A Model for Dynamic Simulation and a Web Interface for Pulse Sequence Development and Visualization**
Author:	**Pablo Villacorta Aylagas**
Supervisors:	**Carlos Alberola López, Manuel Rodríguez Cayetano**
Department:	**TSCeIT**

Committee

Chair:	**Juan Pablo Casaseca de la Higuera**
Secretary:	**Antonio Tristán Vega**
Member:	**Federico Simmross Wattenberg**
Alternate Chair:	**Miguel Luis Bote Lorenzo**
Alternate Secretary:	**Miguel Ángel Martín Fernández**
Alternate Member:	**Luis Miguel San José Revuelta**

Dates

Submission:	**8th March 2025**
Defence:	**26th March 2025**

Abstract

Magnetic Resonance Imaging (MRI) simulation is of particular interest due to its ability to recreate a technique that, despite being extremely suitable for many clinical situations, is expensive and not easily accessible to researchers and trainees. Over the last few years, numerous MRI simulators have emerged, both open-source and proprietary. Among them, KomaMRI stands out as the only open-source tool compatible with all Operating Systems, which includes a graphical user interface (GUI) and vendor-agnostic GPU support. Additionally, it is compatible with widely-used MRI community standards and is written in Julia, which enables efficient and extensible code.

In this context, two key areas have been identified for improving the usability and versatility of the simulator. First, the functionality for defining and simulating dynamic phantoms presents room for improvement, as it only allows motion to be described using analytical expressions. Second, in a previous Bachelor's Thesis [1], a desktop application for designing MRI sequences was developed. This application also shows potential for improvement, and its conversion to a web application was proposed as a future line of work.

This Master's Thesis thus addresses the dual objective of defining a novel dynamic phantom model within the KomaMRI simulator and creating an enhanced web-based application for the editing and simulation of pulse sequences.

For the first objective, the data structure of the simulator has been extended to include information about the motion of the phantom. To achieve this, the global motion of the model has been defined as a list of independent motions, which allows specifying the type, temporal behaviour, and affected spins for each of them. Additionally, simulation functions have been modified to incorporate the calculation of model displacements, and the KomaMRI visualization tool has been improved to allow for the temporal representation of dynamic phantoms. Finally, a new file format has been defined to facilitate the storage and sharing of these digital phantoms. All of this has been developed using the Julia programming language.

For the second objective, a full-stack development has been carried out, addressing both the front-end and the back-end, as well as the communication mechanisms between them. Specifically, the front-end includes an improved version of the previously developed sequence editor, a 3D visualization tool for the selected slice, and two additional panels: one for visualizing the temporal sequence diagram and the other for displaying simulation results. This implementation combines the Qt framework with web technologies such as HTML, JavaScript, and WebAssembly. The back-end, developed in Julia, includes an HTTP server with a REST API, the MRI simulator, and additional modules including the database and front-end files.

Experiments conducted with the dynamic phantom demonstrate the ease of defining and simulating dynamic anatomical models, while also offering reduced simulation times. Furthermore, the obtained results show a high degree of realism, both in demonstrative experiments and those which compare the enhanced version of KomaMRI with other contributions in the field of dynamic MRI simulation. As for the web-based sequence editor evaluation, it highlights its usefulness, interactivity and smoothness, also demonstrating the ability to design and simulate arbitrarily complex pulse sequences without the need for local installations.

The contributions of this work can be summarized as the enhancement of an MRI simulator with improved dynamic phantom simulation capabilities, the definition of a new file format for digital phantoms, and the development of a freely accessible web application for designing and simulating pulse sequences, which benefits both researchers and technical users.

Keywords

Magnetic Resonance Imaging, Simulation, Motion, Web Application, Pulse Sequence.

Resumen

La simulación de Imagen por Resonancia Magnética (MRI) resulta de especial interés debido a su capacidad para recrear una técnica que, a pesar de ser extremadamente apropiada para muchas situaciones clínicas, es costosa y poco accesible para investigadores y técnicos en formación. En los últimos años, han surgido multitud de simuladores de MRI, tanto de código abierto como propietarios. Entre ellos, KomaMRI destaca por ser la única herramienta de código abierto, compatible con todos los sistemas operativos, que cuenta con interfaz gráfica (GUI) y con soporte para GPU independiente del proveedor. Además, es compatible con estándares ampliamente utilizados en la comunidad y está escrito en Julia, lo cual permite un código eficiente y extensible.

En este contexto, se han identificado dos áreas clave para mejorar la usabilidad y versatilidad de este simulador. En primer lugar, la funcionalidad para definir y simular fantomas dinámicos presenta margen de mejora, ya que solo permite describir el movimiento de los modelos anatómicos mediante expresiones analíticas. En segundo lugar, en un Trabajo Fin de Grado previo [1], se desarrolló una aplicación de escritorio para la edición de secuencias de MRI, la cual también muestra posibilidades de mejora y cuya conversión a una aplicación web quedó planteada como línea de trabajo futuro.

Este Trabajo Fin de Máster, por tanto, enfrenta el doble objetivo de definir un nuevo modelo de phantom dinámico dentro del simulador KomaMRI y de crear una aplicación web mejorada para la edición y simulación de secuencias de pulsos.

Para el primer objetivo, se ha ampliado la estructura de datos del simulador para incluir en ella información sobre el movimiento del fantoma. Para ello, se ha definido el movimiento global del modelo como una lista de movimientos independientes, pudiendo definir su tipo, su comportamiento temporal, y el rango de espines afectados por cada uno de ellos. Asimismo, se han modificado las funciones de simulación para incorporar el cálculo de los desplazamientos del modelo y se ha mejorado la herramienta de visualización de KomaMRI para permitir la representación temporal de los fantomas dinámicos. Por último, se ha definido un nuevo formato de fichero que facilita el almacenamiento y la compartición de estos fantomas digitales. Todo ello ha sido desarrollado con el lenguaje de programación Julia.

Para el segundo objetivo, se ha llevado a cabo un desarrollo completo (*full-stack*), abordando tanto el *front-end* como el *back-end*, así como los mecanismos de comunicación entre ambos. Concretamente, el *front-end* incluye una versión mejorada del editor de secuencias previamente desarrollado, una herramienta de visualización 3D del corte seleccionado y dos paneles adicionales: uno para visualizar el diagrama temporal de la secuencia y otro para mostrar los resultados de las simulaciones. Esta implementación combina el *framework* Qt con tecnologías web como HTML, JavaScript y WebAssembly. Por su parte, el *back-end*, desarrollado en Julia, incorpora el servidor HTTP mediante una API REST, el simulador de MRI y módulos adicionales como la base de datos y los ficheros del *front-end*.

Los experimentos realizados con el fantoma dinámico evidencian la facilidad para definir y simular modelos anatómicos con movimiento, además de ofrecer tiempos de simulación reducidos. Asimismo, los resultados obtenidos muestran un alto grado de realismo, tanto en experimentos demostrativos como en aquellos que comparan la versión mejorada de KomaMRI con otras contribuciones en el campo de la simulación de MRI dinámica. Por otro lado, las evaluaciones del editor web de secuencias evidencian su utilidad, interactividad y fluidez, demostrando además la capacidad de diseñar y simular secuencias de pulsos arbitrariamente complejas sin necesidad de instalaciones locales.

Las contribuciones de este trabajo, por lo tanto, se resumen en la ampliación de un simulador de MRI con capacidades mejoradas para simular fantomas dinámicos, la definición de un nuevo formato de fichero para fantomas digitales, y el desarrollo de una aplicación web de uso libre para el diseño y simulación de secuencias de pulsos, que beneficia tanto a investigadores como a usuarios técnicos.

Palabras clave

Imagen de Resonancia Magnética, Simulación, Movimiento, Aplicación Web, Secuencia de Pulsos.

Acknowledgements

I would like to thank all the people who have supported me during this journey, both on a professional and personal level.

First and foremost, I sincerely thank my supervisors, Prof. Carlos Alberola López and Prof. Manuel Rodríguez Cayetano, for their invaluable guidance, support, and advice throughout this Master's Thesis. I would also like to express my gratitude to Prof. Federico Simmross Wattenberg for his unwavering assistance whenever I have needed it, as well as to all the faculty members of the Image Processing Laboratory (LPI) who have offered their help along the way.

I would also like to thank our collaborators from the Pontificia Universidad Católica de Chile, Prof. Pablo Irarrázaval, and especially Dr. Carlos Castillo Passi, for creating the foundational MRI simulator for this work and for their patience and tireless support, without which this project would not have been possible.

Our collaborators from the Department of Energetic Engineering and Fluid Mechanics at the University of Valladolid also deserve special mention. I am deeply grateful to Prof. José Sierra Pallares and Dr. Joaquín Anatol Hernández for providing the flow data and for all their support throughout this part of the project. Their willingness to help and constant availability have been incredible, and I really appreciate it.

I also want to extend my gratitude to Dr. Rosa María Menchón Lara, my colleague at LPI, for providing the cardiac data used in some experiments and for her unwavering support, not only in relation to this part of the project but throughout my entire work and time in the group.

Every member of the LPI deserves a mention. I am especially grateful to Elisa, Tino, and Irene, who have provided me with both professional and personal support and have shown me, through their experience, that it is possible to have fun while researching and doing science. Of course, my thanks also go to Patri, Inés, Rafa, Susana, Tomasz, Álvaro, Guillem, Óscar, Elena, Tuli, Sara, Raquel, Benjamín, Alessandra, Jorge, Biagio, Pablo, Rodri, Santi, Antonio, and Miguel Ángel.

No menos importantes han sido mis personas más cercanas. Gracias a mis padres, María José y Félix, y a mi hermana Lucía, por su apoyo incondicional y por ser quienes más habéis "sufrido" conmigo durante todo este camino. Gracias también a mis abuelos, Josefina y Rafael, por todo su respaldo y cariño. También han sido fundamentales mis amigos, que me han servido de soporte y vía de escape cuando más lo he necesitado.

Y gracias a Inés, por su paciencia infinita, por creer en mí incluso cuando yo dudaba, y por estar a mi lado y hacerme ver que lo realmente importante va mucho más allá de este trabajo. Esto no habría sido posible sin ti.

Mil gracias a todos y cada uno de vosotros,

Pablo.

Contents

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Objectives 2
- 1.3 Phases and Methods 3
- 1.4 Resources 4
  - 1.4.1 Hardware 4
  - 1.4.2 Software 4
  - 1.4.3 Data 5
- 1.5 Structure of the Master's Thesis 5

2 Fundamentals of MRI, Simulation and Technologies **7**
- 2.1 MRI Fundamentals 7
  - 2.1.1 Physical principles 7
  - 2.1.2 MR image formation and acquisition 14
  - 2.1.3 MRI Pulse Sequences 18
  - 2.1.4 Imaging Phantoms 26
- 2.2 MRI Simulation Principles 27
  - 2.2.1 Pulse Sequence 27
  - 2.2.2 Digital Phantom 27
  - 2.2.3 Solving the Bloch equations 27
  - 2.2.4 Incorporating flow data within MRI simulations 28
- 2.3 State of the Art in MRI Simulation 30
  - 2.3.1 JEMRIS 30
  - 2.3.2 CMRsim 32
  - 2.3.3 KomaMRI 34
  - 2.3.4 Numerical approach by T. Puiseux 39
- 2.4 Programming Technologies: Julia, Qt, VTK, and Web 40
  - 2.4.1 Julia 40
  - 2.4.2 Qt 42
  - 2.4.3 VTK 43
  - 2.4.4 Web Technologies 43

3 Methods: Analysis, Design and Implementation **46**
- 3.1 Dynamic Phantom Model 46
  - 3.1.1 Requirements Analysis 47
  - 3.1.2 Design considerations illustrated through key examples 48
  - 3.1.3 Phantom class extension and simulation functions 49
  - 3.1.4 4D Phantom visualization tool 61
  - 3.1.5 Phantom file format 62
  - 3.1.6 Motion-related resources and scripts 62
- 3.2 Web Sequence Editor Application 63
  - 3.2.1 Background 63
  - 3.2.2 Requirements Analysis 64
  - 3.2.3 General Operation of the Application 66
  - 3.2.4 Front-end Modules and Panels 66

3.2.5 Back-end Module . . . 71
3.2.6 Integration, Module Interaction, and Server-Side Compilation . . . 73

4 Results and Testing **75**
4.1 Dynamic Phantom Model . . . 75
4.1.1 Definition, Visualization and Storage of Dynamic Phantoms . . . 76
4.1.2 Demonstrative Evaluation . . . 78
4.1.3 Comparative Validation . . . 87
4.2 Web Sequence Editor Application . . . 100
4.2.1 Overview of the Application and GUI . . . 100
4.2.2 Interface Performance and Responsiveness . . . 100
4.2.3 Key Functionalities and User Interaction . . . 102

5 Conclusions and Future Work **105**
5.1 Conclusions . . . 105
5.2 Contributions . . . 106
5.3 Future work . . . 106

Appendices **108**
A Rotations and Spinors . . . 109
A.1 Rotation Matrices . . . 109
A.2 Euler-Rodrigues Rotation Formula . . . 109
A.3 Spinors . . . 110
A.4 Rotating reference frame . . . 111
B Background on tagging and phase constrast . . . 113
B.1 Solution of the Bloch equations in the presence of a gradient . . . 113
B.2 Tagging: pulses and gradients . . . 113
B.3 Phase Contrast: pulses and gradients . . . 114
C Flow data acquisition and processing . . . 116
D Phantom file format . . . 118
E Amdahl's Law . . . 120

Bibliography **126**

List of Figures

2.1 Spinning hydrogen proton. 7
2.2 $\boldsymbol{B}_0$ magnetic field influence on the spins alignment [1]. 8
2.3 Ampère's Law evaluated for two conductor geometries [19]. 9
2.4 Precession motion of a spin subjected to a static magnetic field [1]. 9
2.5 Effect produced by the application of an RF pulse on the net magnetization vector[21]. . 11
2.6 Net magnetization vector behaviour after turning off the 90º pulse. 11
2.7 T_1, longitudinal, or spin-lattice relaxation. 12
2.8 T_2, transversal, or spin-spin relaxation. 12
2.9 MR signal detection with a coil [21]. 13
2.10 Configuration of the magnetic gradient coil set of an MRI scanner [22]. 14
2.11 Slice selection along z [1]. 15
2.12 Spatial encoding using phase and frequency encoding gradients [21]. 16
2.13 (a) Raw data in th K-space. (b) Corresponding image [23]. 17
2.14 Sequence diagram of a typical spin echo [21]. 18
2.15 Sequence diagram of a typical gradient echo [21]. 19
2.16 Fast Spin Echo (FSE) sequence [26]. 20
2.17 Two-dimensional spoiled steady-state sequence [16]. 20
2.18 Two-dimensional b-SSFP sequence. 21
2.19 Sequence diagram of a GE-EPI and its corresponding K-space trajectory [1]. 21
2.20 Frames extracted from cine series [30]. 22
2.21 Diagram of a cine sequence in which the ECG triggering technique is applied. 22
2.22 Short axis tagging at the mid ventricular level covering the cardiac cycle (A-F) [32]. . . 23
2.23 A SPAMM sequence with two RF pulses, with flip angles θ_1 and θ_2, respectively. 23
2.24 Effect of bipolar gradients on a static set of spins compared to a set of spins moving along the gradient direction [33]. 24
2.25 Axial PC image through the heart [33]. 24
2.26 Trapezoidal bipolar gradients used in PC-MRI. 25
2.27 Time of Flight (TOF) effect and its resulting images. 26
2.28 Examples of phantoms used in MRI [36]. 26
2.29 JEMRIS simulation of an oscillating sphere inside a static ring. [5]. 31
2.30 KomaMRI GUI, displaying the 3D phantom viewer [4]. 35
2.31 Basic diagram of the KomaMRI simulator and organization of its code into sub-packages [4]. 35
2.32 Solution of the Bloch equations for one time step can be described by (1) a rotation and (2) a relaxation step [4]. 37
2.33 Diagram of the functions called to perform the simulation. 38
2.34 Evolution of the simulation time step over an arbitrary pulse sequence [37]. 39
2.35 (a) Diagram of the HDF5 structure. (b) Practical example of use [57]. 41
2.36 Operation diagram of a REST API. 45

3.1 Diagram of the structures and functions in `KomaMRIBase` and `KomaMRICore` involved in the new dynamic phantom model. 50
3.2 Operation of the `get_spin_coords` function in the context of the `simulate` function. . 52
3.3 Non-periodic `TimeCurve` with a single repetition. 53
3.4 Periodic `TimeCurve` with a single repetition. 54
3.5 Non-periodic `TimeCurve` with multiple repetitions. 54
3.6 Periodic `TimeCurve` with multiple repetitions. 55

3.7 Graphical representation of the time curves generated with the constructors in Code 3.8. 56
3.8 Spin span types. 56
3.9 Distribution of actions into two subgroups: `SimpleAction` and `ArbitraryAction`. . . . 57
3.10 (a) Reference frame used for rotation. (b) Schematic diagram of myocardial strain for diastole and systole. 59
3.11 Displacements calculation for an arbitrary action. 59
3.12 Spin state reset over a cylindrical phantom with fluid flow motion inside. 60
3.13 Application GUI developed in [1]. 64
3.14 General operation of the web application. 66
3.15 Front-end design of the application, structured into four modules. 67
3.16 Sketch of the part of the GUI corresponding to the sequence editor and its panel layout 68
3.17 Communication flow for rendering the sequence diagram. 70
3.18 Internal communication scheme between front-end components. 73

4.1 Cube phantom undergoing translational motion in the three axes. 76
4.2 Cube phantom undergoing rotational motion around y and z. 77
4.3 Cube phantom subset undergoing translational motion in the three axes. 77
4.4 Comparison of two brain phantoms with identical translational and rotational motions but different time spans. 78
4.5 Brain phantom undergoing translational motion, visualized using the interactive viewer of KomaMRI. 78
4.6 Two reconstructions from a brain acquisition undergoing translational motion in the x direction. 79
4.7 T_2 map of an XCAT dynamic heart phantom. 80
4.8 Six of the ten acquired frames during a 2D cine simulation over a dynamic heart XCAT phantom. 81
4.9 Cylindrical phantom used in the experiment. 81
4.10 Five of the 30 acquired phases, which reflect the steady state reached by the wall spins. 82
4.11 Comparison of two sets of longitudinal cine frames acquired at different flow velocities. . 83
4.12 Flow phantoms used in the PC-MRI experiments. 84
4.13 Results of the PC-MRI experiment performed on a user-made flow phantom. 85
4.14 Spin trajectory generation for a synthetic aorta CFD dataset at a certain FOV. 86
4.15 Results of the PC-MRI experiment performed on a flow aorta phantom. 86
4.16 Temporal evolution of the inner radius of the heart motion model [11]. 88
4.17 Five of the 100 acquired frames during a 2D cine simulation with tagging. 89
4.18 Diagram illustrating the execution in KomaMRI of the tagging experiment from [11] utilizing four GPUs. 90
4.19 Evolution of the execution times in KomaMRI of the tagging experiment from [11] as the number of GPUs increases. 91
4.20 Configuration of the Bloch solver validation under flow motion conditions. 91
4.21 Evolution of the three components of the magnetization along the z axis for several velocities imposed to the particles. 92
4.22 NRMSE values for the three magnetization components at different flow velocities and time step sizes. 93
4.23 1D phantom with flow motion along the z direction. 94
4.24 Temporal evolution of the magnetization of three distinct spins within a 20mm flow phantom. 95
4.25 Volume rendering of a stenotic U-bend CFD, visualized in Paraview. 96
4.26 Simulation results for the PC-MRI experiment of turbulent flow downstream of a stenotic U-bend. 98
4.27 Comparison of Ground Truth CFD velocities with those obtained from PC-MRI experiments using CMRsim and KomaMRI. 99
4.28 Final layout of the application, which is divided into panels according to the initial design. 100
4.29 Firefox Inspector's Network tab displaying the HTTP requests, responses, and their respective load times during the initial application load. 101
4.30 Application layout during initialization. 101

4.31 GUI panels displaying the configuration of a single-shot EPI sequence. 102
4.32 (a) Sequence diagram of the EPI sequence. (b) Simulation result. 102
4.33 GUI panels displaying the configuration of a Spin Echo (SE) sequence. 103
4.34 MRI simulation results based on the SE sequence created in the GUI. 103
4.35 File management drop-down menu, which allows the import and export of sequence and scanner configuration files. 104
4.36 3D visualization of the slices selected by the RF pulse of the `Ex` block. 104

C.1 Diagram of the methodology implemented to derive spin trajectories from arbitrarily complex flow data and import them into the KomaMRI simulator. 116

List of Code Snippets

2.1 KomaMRI `Phantom` structure definition. 36
2.2 Spins positions acquisition in KomaMRI. 38
3.1 Usage examples of the `NoMotion`, `Motion`, and `MotionList` structs. 50
3.2 `get_spin_coords` method for `Motion` input arguments. 51
3.3 `TimeCurve` structure definition. 53
3.4 Non-periodic `TimeCurve` with a single repetition. 53
3.5 Periodic `TimeCurve` with a single repetition. 54
3.6 Non-periodic `TimeCurve` with multiple repetitions. 54
3.7 Periodic `TimeCurve` with multiple repetitions. 55
3.8 Example usage of the `TimeRange` and `Periodic` functions. 55
3.9 Pseudocode for the new `run_spin_precession!` implementation. 60
3.10 KomaMRI motion-related adapts. 61
3.11 Fragment of the `plot_phantom_map` implementation. 62
4.1 Assignment of a translational motion to the phantom. 76
4.2 Assignment of a rotational motion to the phantom. 76
4.3 Assignment of a translational motion to a phantom subset. 77
4.4 Writing and reading a phantom to/from a `.phantom` file 78
4.5 Displacements acquisition and motion-corrected reconstruction. 79
4.6 Flow cylinder phantom generation. 82
4.7 Double flow cylinder phantom generation. 84
4.8 3D deformable cylinder phantom generation. 88
4.9 Flow motion generation methods for the validation experiment. 91

List of Tables

4.1 PD, T_1, and T_2 values used in the cylindrical phantom. 81

Acronyms

API	Application Programming Interface
AST	abstract syntax tree
BW	bandwidth
CD	Continuous Delivery
CFD	Computational Fluid Dynamics
CI	Continuous Integration
CMR	Cardiac Magnetic Resonance
CPU	Central Processing Unit
ECG	electrocardiogram
EPI	Echo Planar Imaging
ETL	Echo Train Length
FEG	Frequency Encoding Gradient
FID	Free Induction Decay
FIESTA	Fast Imaging Employing Steady-state Acquisition
FISP	Fast Imaging with Steady State Precession
FLASH	Fast Low Angle Shot
FOV	field of view
FRQ	Functional Requirements
FSE	Fast Spin Echo
GE	Gradient Echo
GPU	Graphics Processing Unit
GRASS	Gradient Refocused Acquisition in the Steady State
GT	Ground Truth
GUI	Graphical User Interface
LPI	Image Processing Lab
MOC	Meta-Object Compiler
MPI	Message Passing Interface
MR	Magnetic Resonance
MRA	MR Angiography
MRI	Magnetic Resonance Imaging
NFR	Non-Functional Requirements
NMR	Nuclear Magnetic Resonance
NRMSE	normalized root mean square error
ODE	Ordinary Differential Equation

PD	proton density
PC	Phase Contrast
PEG	Phase Encoding Gradient
QML	Qt Meta Language
RAM	Random Access Memory
RAS	Right-Anterior-Superior
REST	Representational State Transfer
RF	radiofrequency
SE	Spin Echo
SNR	Signal to Noise Ratio
SPAMM	SPAtial Modulation of Magnetization
SSFP	Steady-State Free Precession
SSG	Slice Select Gradient
TE	Echo time
TKE	turbulent kinetic energy
TOF	Time of Flight
TR	Repetition time
URI	Uniform Resource Identifier
$\mathbf{V_{ENC}}$	encoding velocity
VMR	Vascular Model Repository
VNC	Virtual Network Computing
vps	views per segment
VTK	Visualization Toolkit
XCAT	Extended Cardiac-Torso

Chapter 1

Introduction

Magnetic Resonance Imaging (MRI) is an imaging technique based on the Nuclear Magnetic Resonance (NMR) phenomenon. It provides detailed high-contrast images of body tissues and organs and it is customarily used by the medical community for diagnostic purposes. This technique differs from other imaging technologies based on X-rays in that MRI does not use ionising radiation and it is based on static and spatially varying magnetic fields and the application of electromagnetic energy at the frequency of radio waves [2]. It is a highly versatile tool, which makes it the most suitable option in many clinical situations, especially for soft tissue. However, operating an MRI system is challenging due to various factors [3], and the high cost makes learning and research hardly accessible directly on the equipments.

It is with this motivation that MRI simulators arise. Numerical simulations are an important tool for analyzing and developing new acquisition and reconstruction methods in MRI. Simulations allow researchers to isolate and study phenomena by removing unwanted effects, such as hardware imperfections, off-resonance, and others. Additionally, with the increasing use of Machine Learning models, simulation becomes even more relevant, because it can be used to generate synthetic data for training, or to construct signal dictionaries to infer quantitative measurements from the acquired data. Moreover, simulations are an excellent tool for education and training, as hands-on experience is a great way to assimilate the theoretical and practical components of MRI [4].

Over the last few years, a number of MRI simulators have been released, particularly those focused on research. Within this set of tools, several open-source options have been reported, such as JEMRIS [5], MRILab [6, 7], KomaMRI [4], or the recently released CMRsim [8], which specialises in motion and flow-related simulations. Regarding proprietary software options, MRISIMUL [9] —which later evolved into coreMRI [10]— is notable for incorporating high-performance and GPU-based simulations, including motion and flow [11]. Moreover, BlochSolver [12] offers another alternative for running simulations accelerated by the use of GPUs.

Among all these tools, KomaMRI stands out as the only one that meets the following four criteria: availability of a Graphical User Interface (GUI), vendor-agnostic GPU support, open-source —and extensible— code, and support for all Operating Systems. Furthermore, KomaMRI is compatible with some widely used MRI community standards such as Pulseq [13] and ISMRMRD [14]. Many of these features come from the fact that KomaMRI is written in the Julia programming language [15], which allows simple, readable and modular code to be generated and efficiently executed. For the work presented in this paper, this simulator is therefore an excellent starting point.

1.1 Motivation

As mentioned above, KomaMRI stands out for its extensibility and open-source nature, which facilitates continuous enhancement by the scientific community. These features encourage the exploration of new functionalities that can further enrich this MRI simulator. Within this context, two key areas have been identified that can significantly improve the usability and versatility of the simulator: a holistic approach to implementing dynamic phantoms and the development of a web-based sequence editor.

- While the dynamic phantom functionality was already implemented, it required the motion to be defined by analytical expressions. This approach could become cumbersome, particularly for complex or irregular motion patterns such as non-rigid motion or fluid flow, which are difficult to model analytically.
- Additionally, although the simulator includes 2D and 3D phantom visualization tools, it lacks the ability to visualize phantoms that change over time. This has motivated the addition of dynamic phantom visualization capabilities.
- Another important aspect concerning the dynamic phantom model is the lack of a standardised file format to store all the information related to the anatomical model. While an open and fairly standardised format for MRI sequences, i.e. Pulseq [13], is currently available, this is not the case for digital phantoms. This is therefore a motivation for the definition of a file format for the storage of phantoms, which facilitates data sharing and ensures experiment reproducibility.
- As for the web-based sequence editor, it will democratise access to advanced sequence design tools from any browser and it should make its integration into an on-line educational tool far easier. The creation of a desktop version of a sequence editor was the main objective of a previously presented Bachelor Thesis [1], in which the extension to a web application was mentioned as a future line of work. This Master's Thesis is therefore a natural continuation of the work previously carried out.
- Regarding the educational purpose of the sequence editor, user experiences and testing with healthcare professionals and MRI trainees have highlighted the need to include additional functionalities. These include the explicit definition of sequence parameters such as Echo time (TE) and Repetition time (TR), as well as greater flexibility in the definition of gradient shapes. The implementation of these features could significantly enhance the learning experience for these users.
- Finally, another future line presented in [1] was the improvement of the 3D visualization tool of the excited slices. In that thesis this tool was carried out with a technology that was not sufficiently efficient and, above all, quit unsuitable for the case of a client-server application.

1.2 Objectives

On the basis of the previous section, two main objectives can be identified for this work:

1. Definition and incorporation of a novel dynamic phantom model within the KomaMRI simulator. This will facilitate the definition of simple motions and will also enable the creation of any arbitrary complex motion. Overall, the model will provide users with a greater degree of flexibility in motion definition and generation. The following sub-objectives can be stated:
    - Extension of the KomaMRI phantom structure by including a field that contains all the information concerning the pieces of motion involved, their type, amount, and duration, among others.
    - Modification of the simulator core functions to allow phantom displacements to be calculated on-the-fly in a fast and efficient manner. This involves understanding the GPU-focused strategy behind the development of these functions and making the changes accordingly.
    - Enhancement of the phantom visualization tool included in KomaMRI to make it capable of displaying time-varying phantoms.

- Definition of a file format for digital phantom storage and sharing. Files with this format should contain all the information necessary to build a phantom: positions, contrast, off-resonance and motion, among others.

2. Creation of a client-server application that will host an improved version of the MRI sequence editor developed in the Bachelor Thesis [1]. It is worth noting that this enhanced web version will interact with the improved KomaMRI simulator, making use of the newly implemented features described in the first objective. The following secondary objectives are considered:

   - Design and implementation of the front-end, which will contain the improved versions of the sequence editor and the 3D visualization tool for the selected slices, as well as panels for the visualization of the sequence diagram and the simulation results.
   - Design and implementation of the back-end, which would include the server process and the remaining back-end modules, such as the MRI simulator and the database files.
   - Development of communication and interaction mechanisms between the front-end and back-end. This includes selecting the communication protocol between them and properly organizing the code into directories to ensure that the relevant files are accessible from the client and can be rendered on the front-end.

1.3 Phases and Methods

To achieve the aforementioned objectives, the work has been structured into a set of phases that apply to both objectives. While the overall framework remains the same, the specific details of each phase will vary depending on the objective. Hence, a common structure is followed, while we address the unique aspects of each objective in every stage:

1. **Learning stage**, focused on studying the fundamental concepts necessary for the development of this work. Specifically, a solid understanding of the basics of MRI is required, including its physical principles, imaging techniques, and the key components involved in the process, such as pulse sequences and phantoms. It will also be necessary to know the foundations of MRI simulation, as well as the principles and characteristics of the most prominent state-of-the-art simulators. In particular, for obvious reasons, a deep understanding of the operation of the KomaMRI simulator will be required, as well as familiarity with the workflow for contributing to its repository and documenting these contributions.

   Focusing on the second main objective, it will be crucial to learn some of the most important current web technologies, such as HTML, Javascript, Node or the emerging WebAssembly, as well as technologies related to REST APIs, which can communicate the front-end with the back-end of the application.

2. **Design and Implementation**: this stage begins with an initial analysis of the requirements associated to each of the objectives. Additionally, it includes technology selection, method planning and implementation. For the first objective, this will mean selecting the Julia packages to be used, developing an interaction diagram, and programming the functions that will satisfy this diagram.

   For the objective related to the web-based sequence editor, a prototype of the graphical interface will be developed, the appropriate web technologies will be selected and an implementation will be carried out in accordance with the design.

3. **Testing**: a combination of automated —Continuous Integration (CI)/Continuous Delivery (CD) pipelines— and manual testing is used to check the software for errors. This is particularly relevant since the developed code will be openly available in GitHub repositories, allowing both researchers and those interested in learning MRI to download it and use it. In addition, for the dynamic phantom, tests will be conducted to evaluate the ease of including and combining motions in the phantom. Benchmarking will also be carried out to assess performance, and the

accuracy of the results will be measured by comparing them with real acquisitions, as well as with results from other simulators considered as Ground Truth (GT). Regarding the web-based sequence editor, the tests will verify the proper functioning of all components and ensure that various sequences can be created correctly. Additionally, they will confirm that the sequences can be imported and exported in different formats, such as JSON and Pulseq.

4. **Documentation**: this stage is also crucial due to the points outlined in the previous phase: since the software is intended for widespread use, there must be clear installation guides and an accessible user manual that explains how to use the added functionalities.

5. **Maintenance**: issues reported by users are addressed, as well as those arising from updates or changes to any of the dependencies used in the software.

1.4 Resources

The following resources have been employed in the completion of this Master's Thesis:

1.4.1 Hardware

- Image Processing Lab (LPI) desktop computer with the following characteristics:
  - CPU: AMD Ryzen 7 5800X 8-Core 3.8 GHz.
  - GPU: NVIDIA Quadro RTX 4000 8 GB.
  - RAM: 32 GB.
- Remote access to the servers available in the Image Processing Laboratory. This access is possible using both ssh and a Virtual Network Computing (VNC) client. In addition, the file system of the computer used is located on these servers. Specifically, the server with the highest computational capacity, used for occasional performance testing, has the following characteristics:
  - CPU: AMD EPYC 7513 32-Core 2.6 GHz.
  - GPU: 4 x NVIDIA RTX A5000 24 GB.
  - RAM: 1 TB.
- Personal laptop used to remotely connect to servers and conduct local functionality and performance tests on hardware from other vendors (in this case, an Intel CPU/GPU)):
  - CPU: Intel Core i7-8550U 8-Core.
  - GPU: NVIDIA GeForce MX150 / Mesa Intel UHD Graphics 620 (KBL GT2).
  - RAM: 8 GB.
- Access to GitHub CI servers and the JuliaGPU Buildkite CI infrastructure, which enables remote testing across all operating systems and multiple CPU and GPU vendors. These servers are available for use thanks to KomaMRI being a project within the JuliaHealth[1] community.

1.4.2 Software

- Debian GNU/Linux 12 Operating System (in both desktop and laptop computers).
- Visual Studio Code, a source code editor developed by Microsoft for Windows, Linux, macOS and Web.

[1] https://github.com/JuliaHealth

- Furthermore, the following software will be needed:
  - Qt v6.6.
  - CMake.
  - Emscripten (emsdk) v3.1.37.
  - Julia v1.10
  - Node v20.11 and NPM

1.4.3 Data

- LPI has high quality anatomical models of the brain and myocardium. These models will be used as input to the simulator.
- In addition, collaborating members of the Department of Energy Engineering and Fluid Mechanics[2] have generated Computational Fluid Dynamics (CFD) flow data of some anatomical models, such as a realistic aorta or atrium, as well as simpler models such as a U-bend tube (see Appendix C).

1.5 Structure of the Master's Thesis

This document is divided into five chapters, beginning with this Introduction. As seen, this first chapter is structured into the following sections: motivations, objectives of the work, phases and methods, required resources, and document structure. The material that can be found in the rest of the chapters is the following:

Chapter 2 corresponds to the first phase described in Section 1.3 and focuses on the study of the fundamental concepts necessary for this work. Specifically, it explains the principles of MRI, with special emphasis on acquisition methods and motion-related MRI sequences, as these are particularly relevant to this project. The chapter then explores the foundations of MRI simulation, followed by a review of state-of-the-art simulation tools, highlighting how they implement motion when applicable. Among these, special attention is given to KomaMRI, the simulator used in this project, which is described in more detail. Finally, the chapter introduces the key programming tools and technologies relevant to this work, including the Julia programming language, the Qt framework, the VTK library, and various web-related technologies such as HTML, JavaScript, WebAssembly, and REST API concepts.

Chapter 3, in turn, corresponds to the second phase outlined in Section 1.3. It presents the methodology followed for the development of the two main projects that constitute the foundation of this work. These two projects, which may be considered independent of each other, are associated with the two objectives outlined in Section 1.2. Hence, this chapter is divided into two sections. The first focuses on the development of a dynamic phantom model for KomaMRI, while the second addresses the development of a Web-based application for the creation and simulation of MRI pulse sequences. Both sections include a contextualization, a requirements analysis, and considerations that motivate the chosen methodology. Additionally, each section breaks down the explanation of the methodology into separate sections, which allows for a modular perspective on the approach.

Chapter 4 presents the results of the work. As in the previous chapter, it is divided into two sections, each corresponding to one of the outlined objectives. For the dynamic phantom model, a series of experiments are presented. Some of these experiments are demonstrative, and show how the dynamic phantom can be used and configured, as well as how motion-related acquisitions, such as cardiac cine, Time of Flight, or Phase Contrast, can be performed. Other experiments, referred to as comparative, aim to validate the implemented functionality by replicating experiments conducted in

[2]Group of prof. José Benito Sierra-Pallares, located at Escuela de Ingenierías Industriales, Universidad de Valladolid.

reported contributions which use different and already validated simulation tools. The second section of this chapter presents the results of the web sequence editor from the user's perspective, starting with an overview of the application and subsequently evaluating each of its functionalities through practical cases.

Chapter 5 presents the main conclusions drawn from the project, highlights the contributions of the work that have resulted in several conference presentations, and outlines future lines of work that could be pursued during the upcoming academic and professional stages. These future lines of work can be summarized as improving performance, optimizing and solving issues within the dynamic phantom model, as well as adding functionalities and enhancing usability for the web sequence editor.

Finally, the **Appendices** include additional supplementary material which supports the content presented in the document. Specifically, Appendix A presents a set of concepts related to different ways of defining rotations, as well as the rotating reference frame often used in MRI studies. Appendix B provides theoretical background on the concepts of tagging and Phase Contrast in MRI, solving the corresponding equations for each case and deriving a final expression. Appendix C explains the method used to incorporate realistic flow data into the anatomical models implemented. Appendix D outlines the structure for the phantom file format developed. Finally, Appendix E contains a brief description of Amdahl's Law, which is useful for evaluating the improvement factor introduced by adding more processors to a given task.

Chapter 2

Fundamentals of MRI, Simulation and Technologies

2.1 MRI Fundamentals

2.1.1 Physical principles

a) Spins and Nuclear Magnetic Moments

Atomic nuclei with an odd atomic number possess an angular momentum $\boldsymbol{J}$ known as spin. In the classical vector model, this magnitude is defined as a vector representing the rotation of atomic nuclei around a given axis [16]. These rotating charged nuclei generate a magnetic field around them, known as the magnetic moment $\boldsymbol{\mu}$. The relationship between this magnetic moment and the spin is given by the equation:

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

where γ is the so-called gyromagnetic ratio, a physical constant characteristic of each type of nucleus. In our case, we are particularly interested in hydrogen nuclei (^{1}H), composed of a single proton, and abundantly present in the form of H_2O in biological tissues. Hydrogen nuclei have a gyromagnetic ratio $\frac{\gamma}{2\pi} = \bar{\gamma} = 42.58$ MHz/T. Figure 2.1 shows a rotating hydrogen nucleus representation.

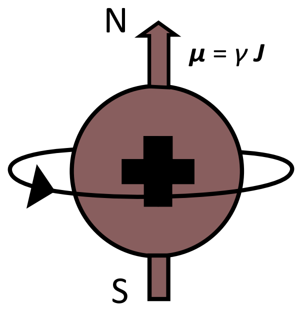


Figure 2.1: Spinning hydrogen proton. The proton exhibits a rotational motion around an axis and behaves similarly to a magnet with north and south poles [1].

As any other vector, the magnetic moment consists of a module and a direction:

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

The module is dependent on the so-called spin quantum number I, which is characteristic of each type of nucleus and must not be null in order to produce resonance. For hydrogen nuclei, $I = \frac{1}{2}$.

On the other hand, the direction $\hat{u}$ of the magnetic moment is inherently random in the absence of an external magnetic field, as it depends solely on the random thermal motion of the nuclei [17]. Consequently, in a tissue composed primarily of hydrogen nuclei, each magnetic moment will point in a different direction. Since these directions are random (and can be assumed to be equally probable), the spins will cancel each other out, and the total magnetic field of the system, known as net magnetization $\boldsymbol{M}$, will be zero.

However, under the application of a static magnetic field $\boldsymbol{B}_0$, hydrogen nuclei tend to align with the direction of this field, with each spin adopting one of two possible orientations:

- Parallel: same direction and orientation as $\boldsymbol{B}_0$ (by convention, this will be the $\hat{z}$ direction).
- Anti-parallel: same direction but opposite orientation to $\boldsymbol{B}_0$ $(-\hat{z})$.

While not all nuclei will align, and some will still cancel each other out, there will be a slight excess of protons that align with the magnetic field, resulting in a non-zero net magnetization parallel to the field [18]. This occurs because the parallel orientation corresponds to lower energy than the anti-parallel orientation, leading to more nuclei adopting the parallel alignment. Figure 2.2 illustrates the effect of the $\boldsymbol{B}_0$ field on the spin system. It is important to note that for the net magnetization vector to point in the direction of the field, not every nucleus needs to be aligned that way; it is enough that the overall contribution of the protons to the total magnetic field is greater in the direction parallel to the field. As a result, components in other directions cancel out, leaving only a net component aligned with $\boldsymbol{B}_0$.

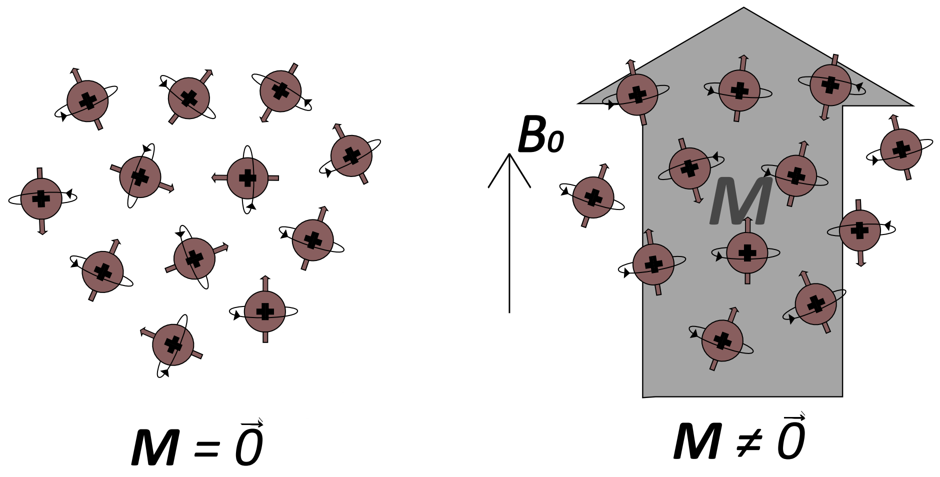


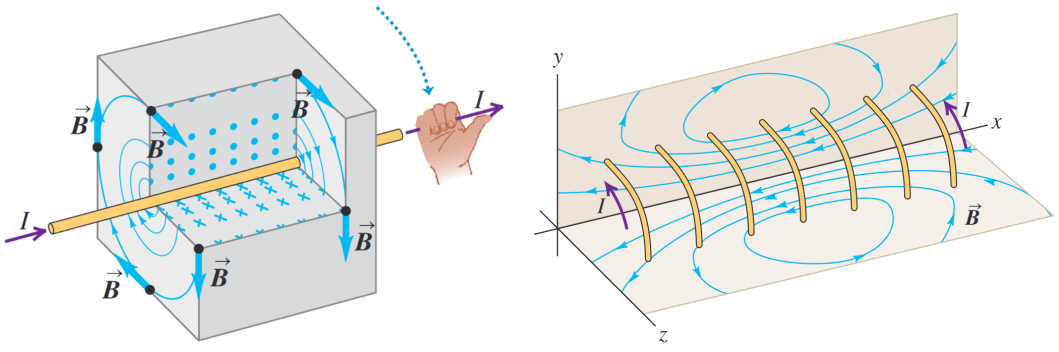
Figure 2.2: $\boldsymbol{B}_0$ magnetic field influence on the spins alignment [1].

To understand how the constant magnetic field $\boldsymbol{B}_0$ is generated, it is important to consider Ampère's Law, which states that the circulation of the magnetic field around a closed loop is proportional to the algebraic sum of the currents passing through the surface enclosed by that loop:

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

In the case of an infinite conducting wire, this results in magnetic field lines that are circular and concentric around the conductor (see Figure 2.3a). However, since we are aiming for straight field lines (at least within the region of interest), we need to generate the magnetic field not with a straight conductor, but with a solenoid. The magnetic field inside the solenoid will meet our requirements [1].

Figure 2.3b illustrates the basic working principle of a solenoid, within which the patient would be positioned.



(a) Magnetic field generated by a straight conductor.

(b) Magnetic field generated by a solenoid.

Figure 2.3: Ampère's Law evaluated for two conductor geometries [19].

b) Spin precession

In addition to the previously explained rotational motion, spins exhibit a second type of movement: precession. To understand this concept in simple terms, one can visualize a spin as a spinning top. A spinning top rotates around its axis, while the force of gravity simultaneously acts to pull it downward. The combined effects of gravity and the spinning motion result in the top precessing [18]. Similarly, spins also undergo precession, but in this case, it is not gravity acting on them; rather, it is the influence of the external magnetic field. Figure 2.4 illustrates this phenomenon.

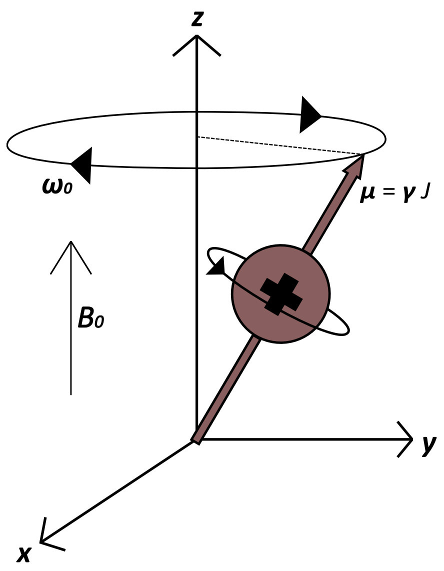


Figure 2.4: Precession motion of a spin subjected to a static magnetic field [1].

The frequency at which spins precess is given by the Larmor frequency ω, which is proportional to the magnetic field $\boldsymbol{B}_0$ [20]:

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

This proportionality is crucial for understanding the processes involved in the generation and acquisition of magnetic resonance images.

c) Net magnetization

As mentioned previously, the net magnetization vector $\boldsymbol{M}$ is composed of the sum of all individual microscopic magnetic moments:

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

where N_s is the total number of spins in the body, and (M_x, M_y, M_z) are the components of the net magnetization vector in the $\hat{x}$, $\hat{y}$, and $\hat{z}$ directions, respectively.

It has also been previously established that, under the influence of the static magnetic field $\boldsymbol{B}_0$, the only non-zero component of the net magnetization vector will be the one parallel to the field. Therefore, following the convention that $\boldsymbol{B}_0$ points in the longitudinal direction $\hat{z}$:

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

where the superscripts ^{0} indicate that only the magnetic field $\boldsymbol{B}_0$ is present. The symbol $N\uparrow$ represents the number of spins oriented in the parallel direction, while $N_\downarrow$ indicates the number of spins oriented in the anti-parallel direction. Additionally, $\bar{h}$ refers to the reduced Planck constant or Dirac constant [17]:

$$\bar{h} = \frac{h}{2\pi}, \qquad \textit{Plank constant} \rightarrow h = 6.33 \cdot 10^{-34} J \cdot s \tag{2.7}$$

d) RF excitation and flip angle

Despite all that has been discussed so far, it is still not possible to produce or acquire any magnetic resonance signal. This is due to the fact that the longitudinal component of the magnetization vector, although non-zero, is very small compared to $\boldsymbol{B}_0$, making it impossible to take measurements directly in that direction. Therefore, to correctly detect the magnetization, it is necessary for it to also be present in the transverse plane, meaning that the components M_x and M_y must be non-zero. To accomplish this, a second magnetic field, $\boldsymbol{B}_1(t)$, must be applied in the transverse direction, with a module given by [17]:

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

where $B_1^e(t)$ is the amplitude of the complex envelope of the field, ω_{rf} is the carrier frequency, and ϕ is the initial phase, which will be assumed to be zero. $B_1(t)$ must oscillate at a frequency equal to the Larmor frequency ($\omega_{rf} = \omega_0$) in order for resonance to occur and for efficient transfer of energy [18]. The precession frequency of hydrogen nuclei when, for example, $B_0 = 1.5, T$, is approximately 64 MHz, which falls within the radiofrequency range. For this reason, the field B_1 is commonly known as a radiofrequency (RF) pulse.

The application of this RF pulse will cause the net magnetization vector of the system $\boldsymbol{M}$ to move from its initial orientation, which is parallel to $\boldsymbol{B}_0$, and begin to acquire transverse components. The evolution of this process over time is described by the Bloch Equation, which accounts for the variation of $\boldsymbol{M}$ [20]:

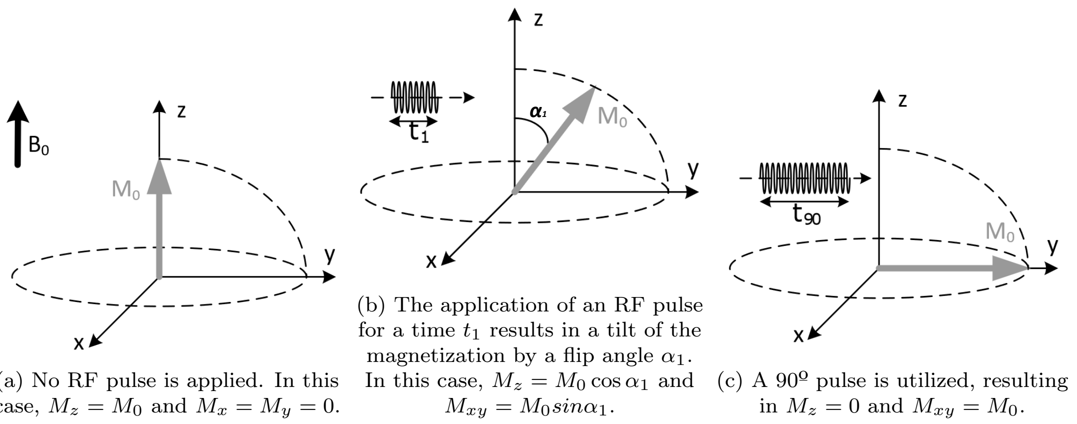
$$\frac{d\boldsymbol{M}}{dt} = \boldsymbol{M} \times \gamma\boldsymbol{B} - \frac{M_x\hat{x} + M_y\hat{y}}{T_2} - \frac{(M_z - M_z^0)\hat{z}}{T_1} \tag{2.9}$$

where M_z^0 is the equilibrium value of $\boldsymbol{M}$ only in the presence of the field $\boldsymbol{B}_0$. T_1 and T_2 are time constants that characterize the relaxation process of the spin system [17], and will be discussed later.

The application of an RF pulse of duration τ_p causes a rotation in the net magnetization vector, which becomes misaligned with the longitudinal axis by an angle α, known as *flip angle*:

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

This effect of the RF pulse over the magnetization vector is illustrated in Figure 2.5, in which M_0 can be interpreted as the module of the vector $\boldsymbol{M}$. Note that by adjusting $|B_1^e(t)|$ and τ_p, it is possible to achieve arbitrary flip angles. This explains the use of terms such as "90º pulse" or "180º pulse" to describe RF pulses that generate precisely these flip angles. Figure 2.5c shows an example of a 90º pulse.



(a) No RF pulse is applied. In this case, $M_z = M_0$ and $M_x = M_y = 0$.

(b) The application of an RF pulse for a time t_1 results in a tilt of the magnetization by a flip angle α_1. In this case, $M_z = M_0 \cos\alpha_1$ and $M_{xy} = M_0 sin\alpha_1$.

(c) A 90º pulse is utilized, resulting in $M_z = 0$ and $M_{xy} = M_0$.

Figure 2.5: Effect produced by the application of an RF pulse on the net magnetization vector[21].

If the duration of the RF pulse is assumed to be small compared to T_1 and T_2, Equation (2.9) can be simplified as:

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

Similarly, if the pulse is sufficiently short to allow its amplitude to be considered constant, (2.10) is reduced to:

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

Note that doubling the amplitude also doubles the flip angle.

e) Relaxation

If the RF pulse is turned off, the magnetic moments of the nuclei and the corresponding net magnetization $\boldsymbol{M}$ continue precessing around $\boldsymbol{B}_0$, at the Larmor frequency ω_0. Simultaneously, $\boldsymbol{B}_0$ begins again to tilt towards the alignment parallel to $\boldsymbol{B}_0$, finally reaching the thermal equilibrium condition it had before the RF excitation [21]. This effect, shown in Figure 2.6, produces a spiral trajectory.

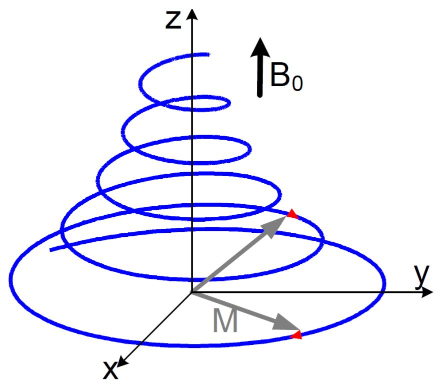
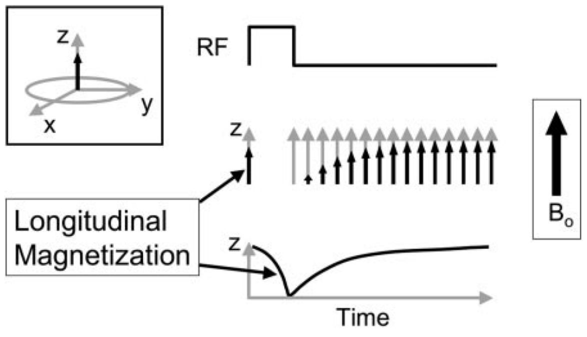


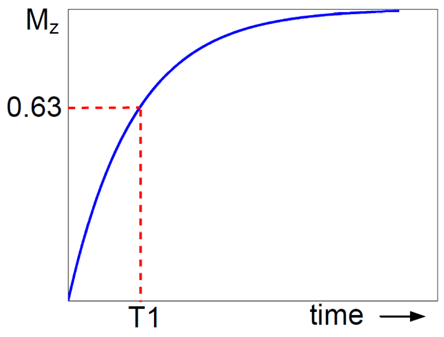
Figure 2.6: Net magnetization vector behaviour after turning off the 90º pulse. $\boldsymbol{M}$ starts tilting towards $\boldsymbol{B}_0$, while precessing at the Larmor frequency [21].

Relaxation process can be divided into three simultaneous sub-processes:

- **Free precession**: it is the precession movement of $\boldsymbol{M}$ around $\boldsymbol{B}_0$ [17].
- **T_1, longitudinal or spin-lattice relaxation**: it is caused by the spins exchanging energy with the surrounding environment. It causes the recovery of longitudinal magnetization [17] and is characterized by T_1, a tissue dependent parameter, but also dependent on the strength of the main magnetic field. An example of this effect is shown in the Figure 2.7a. More specifically, we can define T_1 as the time it takes for the longitudinal magnetization to reach 63% of its final value (see Figure 2.7b), assuming that a 90° RF pulse has been applied. The magnetization of tissues with different values of T_1 will grow again in the longitudinal direction at different rates [18].
- **T_2, transversal or spin-spin relaxation**: it is due to the loss of phase coherence in the spin precession. It results in the loss of the transverse components of $\boldsymbol{M}$ [17] and is characterised by T_2. Recall that, during the RF pulse, the protons start to precess together (go into "phase"). Immediately after the pulse, the protons are still in phase but begin to phase shift due to various effects, including spin-spin interaction. We can define T_2 as the time it takes for the transverse magnetization to fall to 37% of its original value. Figure 2.8 accounts for this effect.

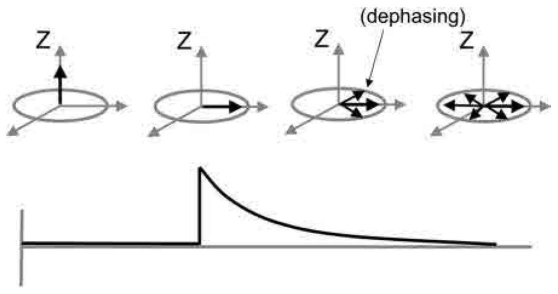


(a) Longitudinal relaxation (T_1) after applying a 90° pulse [18].

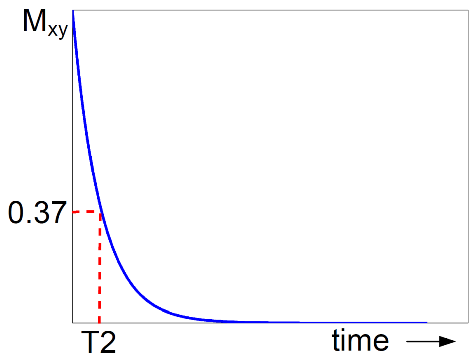


(b) T_1 definition [21].

Figure 2.7: T_1, longitudinal, or spin-lattice relaxation.



(a) Transverse relaxation (T_2) after applying a 90° pulse [18].



(b) T_2 definition [21].

Figure 2.8: T_2, transversal, or spin-spin relaxation.

Figures 2.7b and 2.8b have already shown that the net magnetization vector can be divided into two components: transverse and longitudinal:

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

where $\hat{u}$ is a unit vector with two components (u_x, u_y). The time evolution of M_z and M_{xy} after turning off the RF pulse is described by the following equations [16]:

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

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

where M_z^0 is the longitudinal magnetization just before the RF pulse and M_z^{0+} and M_{xy}^{0+} are the longitudinal and transverse magnetization just after the applying the pulse. For the 90° RF pulse case, M_z^{0+} would be zero, so the Equation (2.14) would be reduced to [21]:

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

It has been previously mentioned that the proton phase shift is not only due to spin-spin interactions (if only this effect were taken into account, we could still claim that the transverse relaxation time corresponds to T_2), but that there are other phenomena that can further reduce this time, such as:

- ΔB_0 inhomogeneities in the magnetic field
- The magnetic susceptibility
- The so-called chemical shift.

For now, none of these effects will be discussed in detail, they will simply be taken into account in the transverse relaxation process. When all four effects are considered, and not just the spin-spin interactions, the protons are dephased faster, so the transverse relaxation time decreases and is no longer called T_2, but T_2^*. In practice, and for simplicity, it is enough to take into account the inhomogeneities of the magnetic field to calculate T_2^*:

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

f) MR signal detection

As mentioned above, it is impossible to detect the longitudinal component of the magnetization due to its small value with respect to the $\boldsymbol{B}_0$ field. The only signal that can be detected and measured is the one coming from the transverse component M_{xy}. This is why radiofrequency pulses are applied, as it has been discussed above.

To detect this transverse magnetization, a coil with a magnetic field perpendicular to $\boldsymbol{B}_0$ is placed close enough to the body, as shown in Figure 2.9. The net magnetization will then generate a voltage induced in the coil, as stated in Faraday's Law. This voltage will be a sinusoid of frequency ω_0 and its amplitude will decrease as M_{xy} decreases, i.e., at a rate of T_2^*. This is the so-called Magnetic Resonance (MR) signal, commonly known as Free Induction Decay (FID) [21].

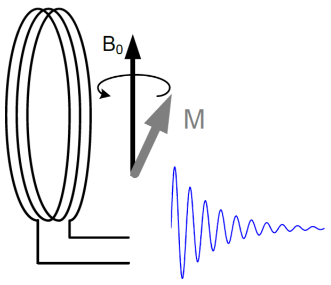


Figure 2.9: MR signal detection with a coil [21].

It should be noted that the signal received by the scanner will not provide information about T_2. Instead, the measurable quantity will be T_2^*. However, the true value of T_2 can be calculated by applying a sequence of RF pulses known as a spin echo.

2.1.2 MR image formation and acquisition

As mentioned earlier, the MR signal is an analogue voltage signal induced by the transverse component of the net magnetization, and it has a sinusoidal waveform with a frequency and amplitude that decay over time. However, this signal does not provide much information, especially for medical analysis.

On the other hand, an MR image (MRI) is a graphical representation of a two-dimensional matrix obtained by processing the MR signal through various stages such as demodulation, sampling, and inverse Fourier transforms. The MR image is what provides real information for medical diagnosis. The challenge in acquiring an MR image is being able to spatially locate the specific part of the body from which the signal is obtained. This spatial location is achieved through the use of gradients.

a) Gradients

The magnetic field gradients change the intensity of the magnetic field $\boldsymbol{B}_0$ in each of the three Cartesian dimensions (x, y, z), so that the Larmor frequency and magnetic moment of each spin depends on its position [2]. Each of the three gradients is applied with a different coil, so that each one can be switched on or off independently. Figure 2.10 shows how the coils that generate the gradients can be used in a real MRI system.

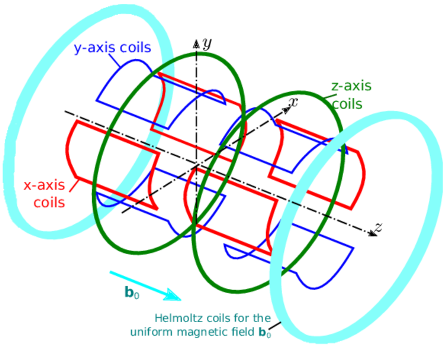

y-axis coils
y
z-axis coils
x
x-axis coils
z
$\mathbf{b}_0$
Helmoltz coils for the uniform magnetic field $\mathbf{b}_0$

Figure 2.10: Configuration of the magnetic gradient coil set of an MRI scanner [22].

These gradients, in addition to making the precession frequency of each spin dependent on its position, can be configured so that this dependence is also present in its phase. This allows for the location of each spin to be determined, for example, in the x dimension by examining its frequency and in the y dimension by analysing its phase. These techniques are referred to as frequency and phase coding and will be discussed in greater detail later on.

b) Excitation stage: Slice selection

In this stage, the RF pulse is activated, thereby initiating the aforementioned processes: spins excitation, emergence of the transverse component of the magnetization, relaxation, and so forth.

However, if instead of exciting all the spins in the body, only a specific slice is to be excited, a linear gradient —Slice Select Gradient (SSG) [21]— must also be activated at the same time as the RF pulse is activated. In this way, the resonance frequency of the spins will be dependent on their position, and only those spins whose precession frequency coincides with that of the RF pulse will be excited. [17]. A slice may be selected along any arbitrary direction by controlling the value taken by each of the three gradients (G_x, G_y, G_z) during excitation. Furthermore, if the RF pulse is given a certain bandwidth (BW), a slice with a thickness proportional to that width may be selected.

The simplest illustration is that of a slice perpendicular to the longitudinal direction (from head to toe), which only requires the activation of G_z. The net magnetic field in the direction of G_z is then:

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

Now, the activated RF (see Equation (2.8)) must have a center frequency:

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

Figure 2.11 shows this selection. We can see how G_z corresponds to the slope of the function. This means that the larger G_z is, the more sensitive the frequency is to variations along z and, therefore, the narrower the slice will be. For $z = 0$, the centre frequency of the pulse corresponds to the Larmor frequency ω_0. This point is known as the isocenter.

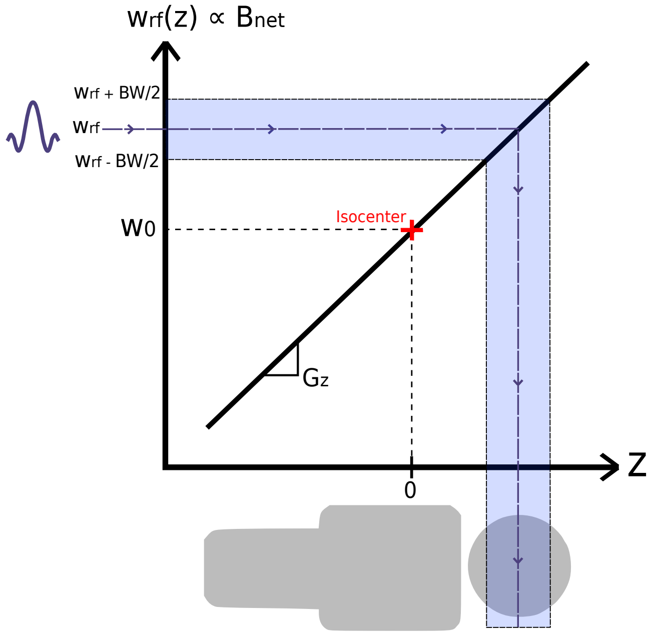


Figure 2.11: Slice selection along z [1].

Taking it now to the general case, suppose that an RF pulse of complex amplitude $B_1 = B_{1x} + jB_{1y}$ is applied for an interval Δt short enough to consider the amplitude constant. Suppose, furthermore, that some gradients $\mathbf{G} = (G_x, G_y, G_z)$ are applied and that there are B_0 inhomogeneities in the magnetic field. Then, the local, position-dependent magnetic field will be as follows:

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

Consequently, the effect produced by the pulse on the position of the point $\mathbf{r}$ is a rotation[1] of the magnetization vector at an angle ϕ around the $\mathbf{n}$ axis:

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

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

Thus, if gradients $\mathbf{G}$ are applied during excitation, the maximum transverse component will manifest in all points where $\mathbf{G}^T\mathbf{r} = 0$. This is the 3D equation of a plane whose normal vector is $\mathbf{G}$ and which

[1]The rotation in the Equation (2.21) follows the right hand rule, while the rotation in the Equation (2.10) follows the left-hand rule; this explains the negative sign in the Equation (2.21), which makes the rotations equivalent.

contains the point $(0,0,0)$ when $\Delta B_0(\mathbf{r}) = 0$. Although $\Delta B_0(\mathbf{r})$ characterises the inhomogeneities, it can also be explicitly chosen to be different from 0 in order to select a slice separate from the isocenter. If the pulse RF has frequency $\omega_{rf} = \omega_0 + \Delta\omega$, where $\omega_0 = \gamma B_0$, then $\Delta B_0(\mathbf{r}) = -\frac{\Delta\omega}{\gamma}$ and the excited slice will satisfy:

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

This is the equation of the plane perpendicular to the vector $\mathbf{G}$ which is separated from the origin (in the direction of $\mathbf{G}$) by a distance $\frac{\Delta\omega}{\gamma|\mathbf{G}|}$.

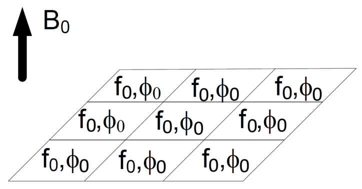
In addition, as mentioned above, the slice width will be a function, in essence, of the pulse bandwidth, ie., of the (temporal) bandwidth of the complex envelope $B_{1x}(\mathbf{r},t)+jB_{1y}(\mathbf{r},t) = B_1^e(\mathbf{r},t)e^{j\varphi(\mathbf{r},t)}$. The pulse shape can be approximated by short pulses of constant amplitude, so that the equations (2.21) and (2.22) can still be satisfied.

c) Acquisition stage: K-space trajectory

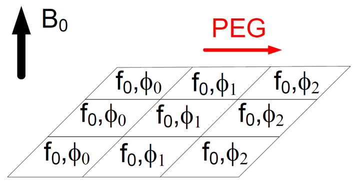
Once the slice is selected and excited, and just after deactivating the gradients and the RF pulse, all the spins in the slice will precess with approximately the same phase and frequency. This is when spatial encoding comes into play. This allows locating the signal produced by each spin in the two coordinates that define the excited plane. Spatial encoding is divided into two processes:

- **Phase encoding**: during this stage a Phase Encoding Gradient (PEG) is applied along one of the axes defining the excited plane (in the case of the simple example above, where a slice along z was selected, the PEG could be activated, for example, at y). During its application, the slice spins will precess at different frequencies depending on their position on the PEG axis, so they will also be out of phase. After deactivating the gradient, the spins will again all precess at the same approximate frequency, but now with different phases [21]. This phase difference can be quantified, allowing us to locate, in one of the two directions, the spatial origin of the received MR signal. Figure 2.12b shows the effect of this encoding.

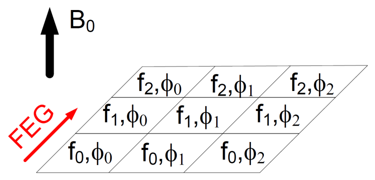
- **Frequency encoding**: applying the PEG makes spins precess at the same frequency but with different phases. Now, a Frequency Encoding Gradient (FEG) is applied along the third axis, which is mutually perpendicular to the SSG and PEG axes (in the case of the example, this third axis would correspond to x). In this case the spins, which are already precessing with different phases, also start to precess with different frequencies according to their position on the FEG axis. Accordingly, the signal is now ready to be recorded, as it contains both phase and frequency information. Because the signal is recorded during the application of the FEG, this Frequency Encoding Gradient is also referred to as the Readout Gradient [21]. Figure 2.12c shows the effect produced by this encoding.



(a) Selected slice before activating the spatial encoding gradients. All spins have the same frequency and phase.



(b) Applying a PEG causes the magnetic moments of each spin to vary in phase, depending on its position on the gradient axis.



(c) During the application of the FEG, the magnetic moments will show different frequencies together with the different phases due to the PEG.

Figure 2.12: Spatial encoding using phase and frequency encoding gradients [21].

If a PEG is applied in the y direction for a time T_{pe}, the position-dependent phase offset will be:

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

And the received signal of the whole object will be given by:

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

where $\rho(y)$ corresponds to the spin density of the object along y. After demodulating:

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

Now, applying a FEG in the x direction gives the position-dependent frequency:

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

So the received signal in this case would be:

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

After demodulating:

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

If terms k_x and k_y are defined such that:

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

And substituting in (2.29) and (2.26) respectively, it can be observed that there is a Fourier relation between the spatial variation of the spin density and the received signal:

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

This is known as K-space, and it constitutes the domain over which the original signal (raw data) obtained is distributed. In addition, the signal in K-space corresponds to the Fourier transform of the desired image, so it is sufficient to do the inverse transform of the K-space matrix to obtain the image. Figure 2.13 shows the example of an image and the K-space matrix from which it was obtained [23].

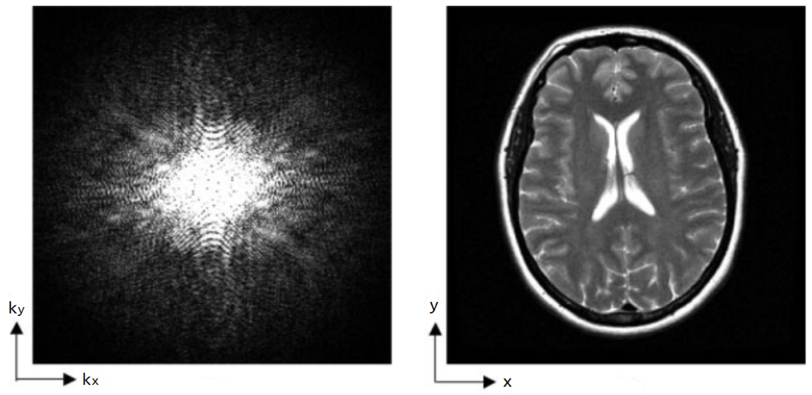


Figure 2.13: (a) Raw data in th K-space. (b) Corresponding image [23].

The K-space location (k_x, k_y) to which the data from any sampled point belong is governed by the cumulative effect of gradients from the time of excitation RF to the time when that sample is collected [23]. This means that, although the previous equations are useful to understand the analytical process of K-space formation, what is really useful is the fact that the application of gradients during the acquisition phase produces a displacement in K-space, conventionally as follows:

- Gradients activated in the vertical direction (y in the example) will correspond to the phase encoding gradients. Their application produces a vertical displacement through K-space.
- Gradients activated in the horizontal direction (x in the example) will correspond to the frequency encoding gradients. They allow moving horizontally through K-space and during their application the reading of the data will be performed.

Therefore, the combination of these two types of gradients will allow us to follow any arbitrary trajectory when acquiring the data. The gradients, together with the previous excitation(s), define what is known as the MRI pulse sequence.

2.1.3 MRI Pulse Sequences

A pulse sequence is a series of events comprising RF pulses, gradient waveforms, and data acquisitions. The purpose of the pulse sequence is to manipulate the magnetization in order to produce and acquire the desired signal. Pulse sequences play a central role in MR imaging [24].

a) Basic pulse sequences

Although there are numerous acronyms and variants, the basic pulse sequences fall into two main families: Spin Echo (SE) and Gradient Echo (GE).

- **SE sequence**: this sequence starts with a gradient SSG and a 90° RF pulse that excite a given slice. Then, a PEG is applied to induce phase differences in the magnetic moments. After a time equal to TE/2 a 180° RF pulse and the SSG are applied to correct the phase difference caused by the ΔB_0 [21] inhomogeneities. After a time equal to Echo time (TE), an echo signal is generated, which is read by applying a FEG. This process is repeated after a Repetition time (TR), modifying the gradient PEG with each repetition. The 180º pulse compensates for the phase shift produced by the ΔB_0, making the echo amplitude decay at a rate of T_2 instead of T_2^* [17]. There are variants such as the FSE, which obtains multiple echoes per 90° RF pulse, reducing acquisition time. Figure 2.14 shows the temporal diagram of a typical SE sequence.

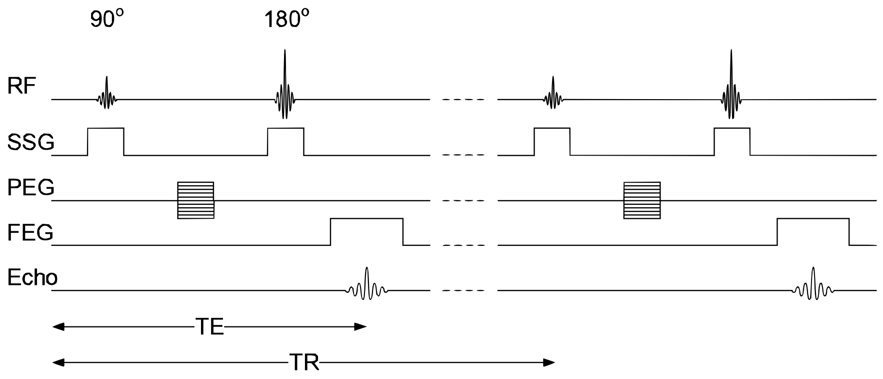


Figure 2.14: Sequence diagram of a typical spin echo [21].

- **Gradient Echo (GE) sequence**: in this sequence, an RF pulse is applied together with an SSG, followed by a PEG for phase encoding. Simultaneously, a negative readout gradient —a FEG— is activated to correct the dephasing that will occur during readout. Next, the FEG is inverted to refocus the magnetic moments and generate the MR signal echo. The key difference from Spin Echo is that Gradient Echo uses a negative readout gradient instead of a 180º RF pulse to refocus the magnetic moments [21]. Figure 2.15 shows a GE sequence diagram.

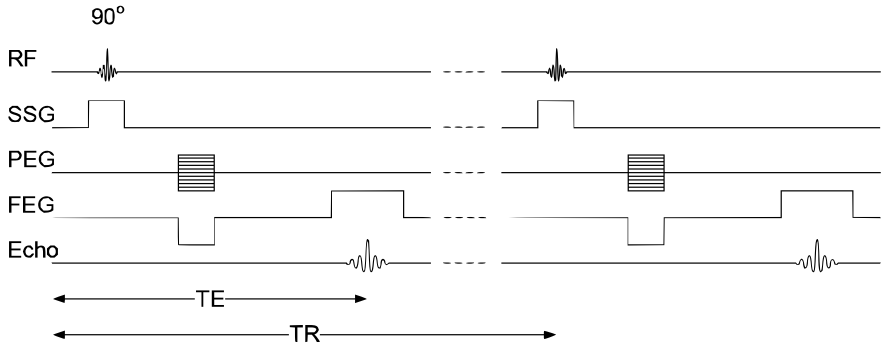


Figure 2.15: Sequence diagram of a typical gradient echo [21].

b) Fast pulse sequences

Although basic pulse sequences are essential and form the foundation of many of the complex sequences used today, one of their main drawbacks is their long acquisition times. This means that patients must undergo extended MRI sessions, which, while not dangerous, can be uncomfortable due to the need to remain still for long periods of time.

Additionally, there are several reasons why it is desirable to speed up scanning. First, shorter acquisition is less prone to motion artifacts. Related to this, a sequence that is fast enough can be acquired during breath-hold and thus yields images without respiratory artifacts [25]. Lastly, with faster MRI sequences, what were once considered motion artifacts can now be transformed into measurable signals that provide valuable information. For example, the beating of the heart or the flow through an artery, previously seen as sources of image degradation, can now be accurately measured to offer insights into physiological processes.

It is easy to derive that the total data acquisition time for a spin echo imaging experiment is [16]:

$$T_{acq} = N_{acq} \cdot N_{enc} \cdot \mathrm{TR} \tag{2.32}$$

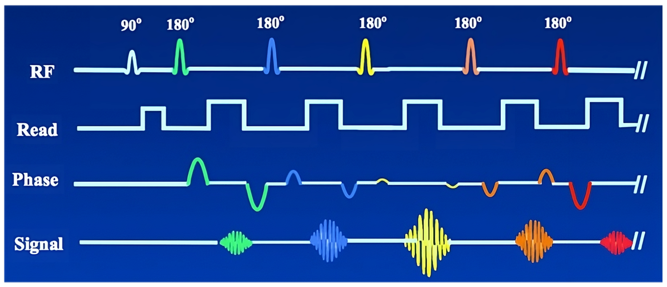
where N_{acq} is the number of signal acquisitions (or signal "averagings") for each encoded signal, N_{enc} is the number of "encodings", and TR is the time interval between two consecutive encoded signals. Thus, T_{acq} can be shortened by reducing any of the three factors, individually or simultaneously [16]. There are several strategies to achieve this:

- **Fast Spin Echo (FSE) sequence**: conceptually, this sequence is a simple extension of the basic spin echo method. Its operation is based on the generation of M echo signals per 90° excitation pulse. If these signals are encoded differently, M K-space lines will be generated for each excitation [16]. This term M is also known as Echo Train Length (ETL). Assume that a total of N_{enc} "encodings" are required to cover K-space. Then, the number of excitations necessary is given by [16]:

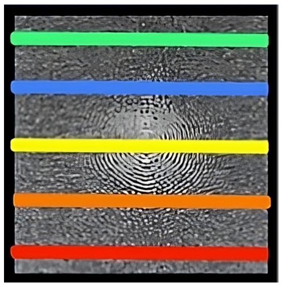
$$N_{ex} = \frac{N_{enc}}{M} \tag{2.33}$$

  Hence, a factor of M improvement in imaging speed is obtained over the conventional single-echo method. The value of M is limited by the T_2 value of the object being imaged [16]. Figure 2.16 demonstrates the generation of multiple echoes per excitation through the application of multiple 180° RF pulses per TR.

- **Fast Gradient Echo sequence**: this set of sequences operate in the regime of very short TRs. When a spin system is excited by a train of RF pulses with repetition time $\mathrm{TR} \ll T_2$, the spin system will reach a dynamic equilibrium, known as steady state. Based on how the transverse magnetization is handled after each excitation, these methods are conveniently grouped into two classes: spoiled steady-state and true steady-state methods [16]:



(a) FSE sequence diagram. Only one TR is displayed. The application of five 180° RF pulses at $t = n(\text{TE}/2)$ produces five echo signals at $t = n\text{TE}$, where $n = 1, .., 5$.



(b) Lines of K-space that are filled during the same TR.

Figure 2.16: Fast Spin Echo (FSE) sequence [26].

– **Spoiled Steady-State Imaging**: as the nomenclature suggests, methods in this class establish a steady-state longitudinal magnetization but destroy or "spoil" any residual transverse magnetization before a new RF pulse is applied. To achieve this, a spoiler gradient pulse is applied along the slice-selection direction. Specifically, the amplitude of the spoiler gradient is varied from one excitation to another in order to avoid coherent build up of the transverse magnetization [16]. Figure 2.17 shows the sequence diagram of a spoiled steady-state sequence, which is also known as Fast Low Angle Shot (FLASH) or spoiled GRASS (Gradient Refocused Acquisition in the Steady State).

  In addition to the gradient spoiling method, effective RF spoiling methods have also been developed, which incrementally step the phase of the RF pulses [16].

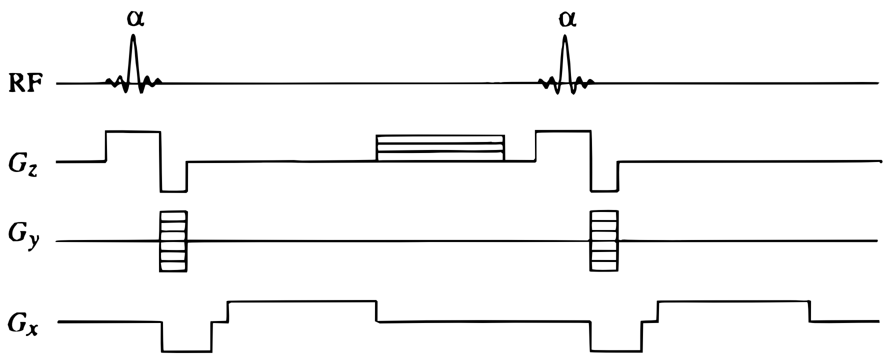


Figure 2.17: Two-dimensional spoiled steady-state sequence [16].

– **(True) Steady-State Imaging**: when TR is on the order of T_2 and no attempt is made to spoil the transverse magnetization, both the longitudinal and transverse magnetizations will reach a steady state. This phenomenon is called Steady-State Free Precession (SSFP) [16]. There are several types of true steady-state sequences, as well explained in [24, 27], the focus will be on the balanced steady-state free precession (b-SSFP) sequences, which are known as TrueFISP (Siemens) or FIESTA (GE). These represent the most commonly used on the SSFP sequences, since they are extremely fast.

  The b-SSFP sequence is designed to fully compensate for all gradient dephasing from one repetition to the next. In other words, for each gradient applied, a reverse gradient is applied at the end of the sequence to reverse its effects; this is also true for the RF pulse, which alternates between positive and negative flip angles in successive repetitions. Figure 2.18 shows the sequence diagram of a balanced SSFP sequence.

- **Echo Planar Imaging (EPI)**: EPI is an ultra-fast acquisition technique that captures multiple lines of K-space after a single excitation pulse. EPI is not a pulse sequence on its own, since

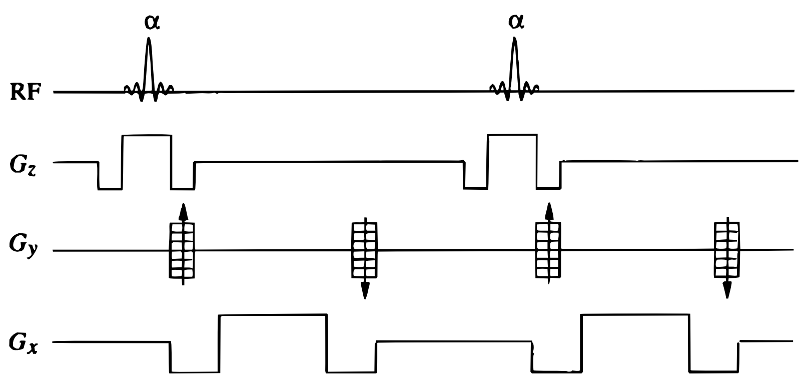


Figure 2.18: Two-dimensional b-SSFP sequence. All the gradients are balanced from cycle to cycle and the sign of the RF pulses alternates in every TR [16].

it needs to be combined with a prior excitation, which can be spin echo (SE-EPI) or gradient echo (GE-EPI). In its purest form (single-shot), all lines are acquired after one pulse, while in segmented versions (multi-shot), the acquisition is done in parts. In EPI, echoes are generated from the FID using alternating FEGs and PEGs. While it allows for very fast acquisitions, this sequence is susceptible to Signal to Noise Ratio (SNR) issues and artifacts due to the rapid decay of the FID at the rate of T_2^* [28]. Figure 2.19 shows an EPI sequence diagram as well as its corresponding K-space trajectory.

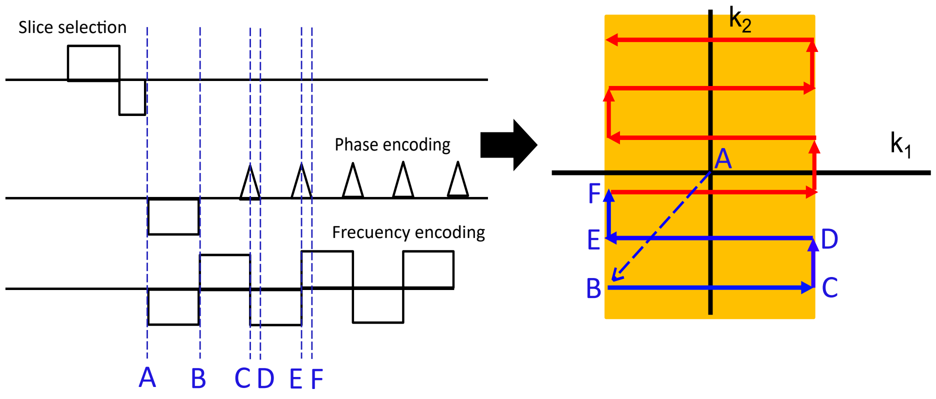


Figure 2.19: Sequence diagram of a GE-EPI and its corresponding K-space trajectory [1].

c) Cardiac cine, triggering and tagging

Cardiac Magnetic Resonance (CMR) **cine** sequences are an effective method for imaging the heart's contractile performance due to its high spatial and temporal resolution. This technique generates a movie-like representation of the heart, allowing a detailed visualization of its motion and function [29]. Figure 2.20 shows an example of a cardiac cine acquisition.

The base pulse sequence is a bright-blood technique, typically a balanced SSFP gradient echo, which generates high intravascular signal relative to other tissues due to the high T_2/T_1 ratio of blood. This sequence allows very short TR and TE values to be used, so multiple lines of K-space can be acquired during a single heartbeat.

With this type of sequence, it is not possible to obtain images for each cardiac phase within a single cardiac cycle. It is necessary to take advantage of the periodicity of the heart's motion pattern to acquire these images over several cycles. However, this heart's motion pattern is not perfectly

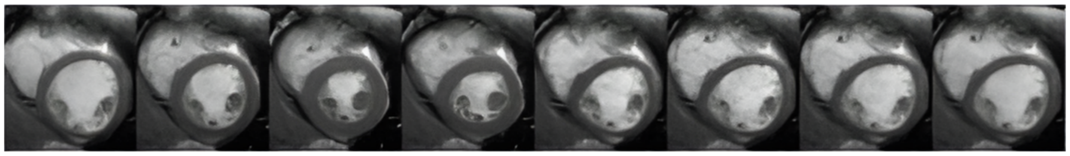


Figure 2.20: Frames extracted from cine series [30]. Eight frames of a short axis acquisition show the contractile function of the heart, which is evident in the fourth one.

periodic. There may be fluctuations in heart rate, arrhythmias, and other irregularities, which is why the technique known as **triggering** is employed.

Cardiac triggering synchronizes a pulse sequence to the cardiac cycle of a patient. Its goal is to acquire an entire set of K-space data at approximately the same portion of the cardiac cycle, even though the duration of the entire acquisition is longer than a single R-R interval [24]. The most commonly used triggering technique employs an electrocardiogram (ECG) signal, which is recorded simultaneously with the MRI acquisition to ensure synchronization.

Usually, the pulse sequence is triggered when the amplitude of the R wave reaches its peak voltage, which can be detected by setting a threshold [24]. Subsequently, during each R-R interval, one or more K-space lines are acquired for every cardiac phase. In this context, K-space lines are known as *views* and each cardiac phase is referred to as a *segment*. Assuming that K views are acquired for each segment during every R-R interval, and that the K-space has a total of M lines, M/K cardiac cycles will be needed to complete the acquisition. Figure 2.21 illustrates how a total of N segments (frames) are acquired, obtaining 2 views for each segment during every R-R interval.

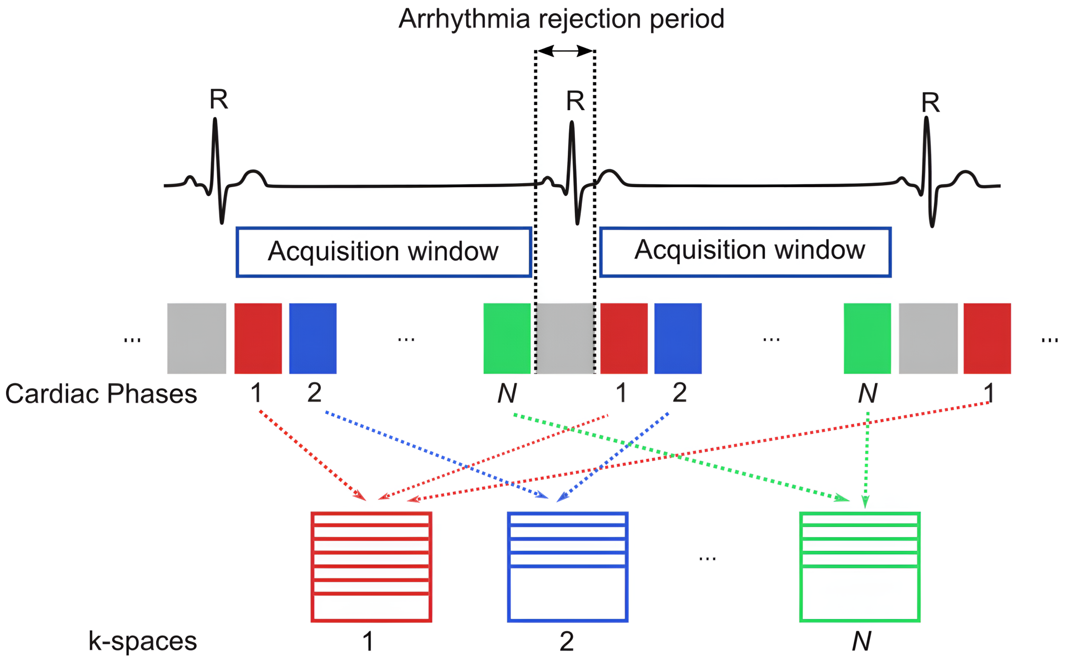


Figure 2.21: Diagram of a cine sequence in which the ECG triggering technique is applied. N segements (frames) are acquired, obtaining 2 K-views for each segement during every R-R interval [31].

An additional technique known as **tagging** can be applied over cine imaging to track myocardial motion in greater detail. RF tagging pulses are used to spatially label an image with a specified physical of physiological property. Most commonly, a tagging pulse places a series of parallel stripes or orthogonal grids on an image, which are known as tags [24]. Among other applications, the deformation of the tags can be used to track and measure the motion pattern of the myocardium, as can be seen in Figure 2.22.

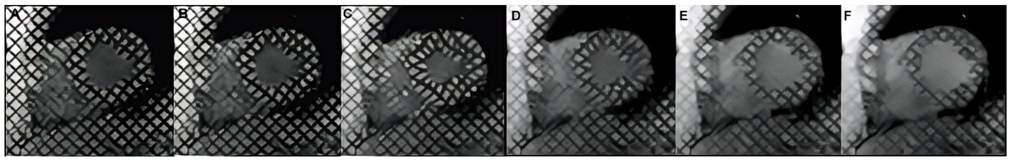


Figure 2.22: Short axis tagging at the mid ventricular level covering the cardiac cycle (A-F) [32].

The most basic tagging technique is known as SPAtial Modulation of Magnetization (SPAMM). In its simplest form, it consists of two consecutive RF pulses (e.g., rectangular pulses) with a gradient lobe sandwiched in-between (Figure 2.23). An optional spoiler gradient pulse is usually applied after the second RF pulse. This combination of pulses and gradients is applied immediately before the actual imaging sequence, generating a stripe pattern. These stripes are perpendicular to the tagging gradient direction. If a grid pattern is desired, it will be necessary to apply two consecutive SPAMMs, one with tagging gradients in the horizontal direction (e.g., x) and another in the vertical direction (e.g., y), before the imaging sequence. More details on how the tagging expressions are derived can be found in Appendix B.

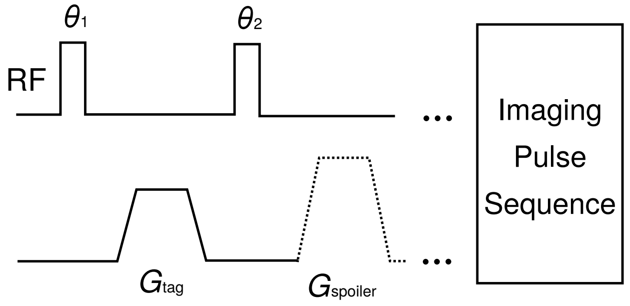


Figure 2.23: A SPAMM sequence with two RF pulses, with flip angles θ_1 and θ_2, respectively. A tagging gradient G_{tag} is placed between the RF pulses, and an optional spoiler gradient (dotted lines) $G_{spoiler}$ can be applied after the second RF pulse [24].

d) MR Angiography Imaging techniques

MR Angiography (MRA) is a specialized type of MRI that focuses specifically on imaging blood vessels. Some MRA methods produce images with suppressed (i.e., hypointense) blood signal, and are commonly referred to as dark blood methods. In contrast, bright blood methods enhance blood flow visibility, providing a clearer view of the vessels. The focus here is on the latter approach. Specifically, two non-contrast techniques will be described:

Phase Contrast (PC) MRI This method generates images of magnetization motion by applying flow-encoding gradients, which translate the velocity of the magnetization into the phase of the image. Typically, a bipolar gradient is used, because it produces a phase that is linearly proportional to velocity. The axis of the bipolar gradient determines the direction of flow sensitivity. Normally, the bipolar gradient is applied to only one axis a time, but by simultaneously applying them along two or more of the logical axes, flow sensitivity along any arbitrary axis can be achieved. PC pulse sequences are usually formed by adding the flow-encoding gradient lobes to a gradient echo sequence [24].

Figure 2.24 illustrates the effect of the bipolar gradient on spins. When applying a gradient to change precession frequencies, if an equal but opposite gradient is applied afterward, then the spins that are stationary will have no phase shift. However, those protons that are moving will undergo varying degrees of phase shift because their location along the gradient changes. This concept can be used to evaluate protons that are moving through a plane. Protons that are in plane (or in a selected 3D space) undergo a uniform phase shift during the first gradient application. Protons moving in the direction of the gradient will experience different field strengths as their positions vary and therefore undergo a different net phase shift. The opposite gradient is applied to the stationary protons and changes the phase of the protons moving in the direction of the gradient. When the signal phases are deconstructed, only the mobile protons will have a phase shift. When the phase shift is mapped visually, the velocity is directly displayed for the interpreter for qualitative assessment (see Figure 2.25) [33].

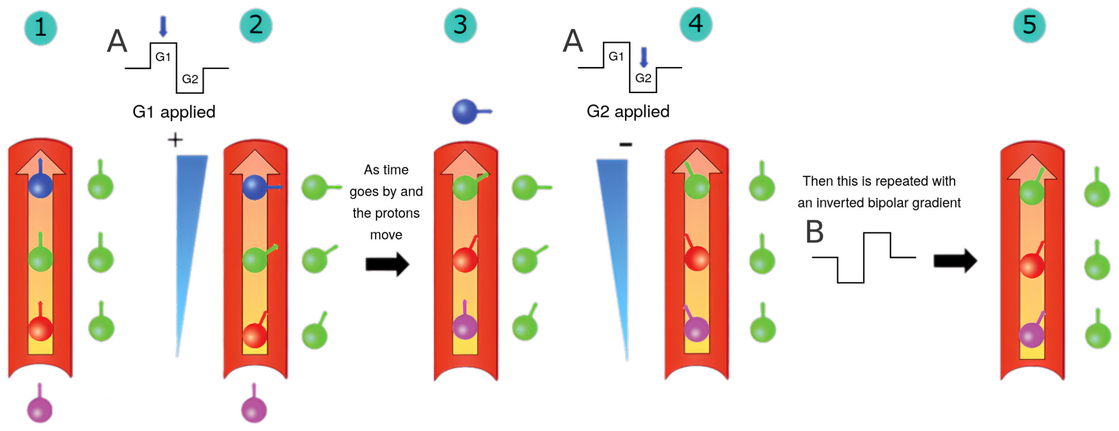

1
2
3
4
5
A
G1
G2
G1 applied
+
As time goes by and the protons move
G2 applied
–
Then this is repeated with an inverted bipolar gradient
B

Figure 2.24: Effect of bipolar gradients on a static set of spins compared to a set of spins moving along the gradient direction [33]. The A and B gradients represent the two possible polarities of the bipolar gradient. In PC-MRI, two separate acquisitions are performed: one using gradient A and the other using gradient B. Velocity information is then derived through a phase difference reconstruction [34].

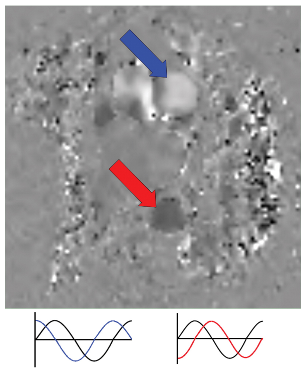


Figure 2.25: Axial PC image through the heart. Blood flow enters the plane in the pulmonary trunk (blue arrow) and exits it in the descending aorta (red arrow). A positive phase shift (blue sine wave) represents inflow, while a negative phase shift (red sine wave) represents outflow. Gray areas indicate no phase shift, white areas, a positive phase shift, and black areas, a negative phase shift [33].

As just mentioned, the degree of phase shift in PC imaging corresponds directly to the proton velocity in the direction of the gradient (Eq. (2.34)). However, because angles have a limited range of 360º —i.e., a phase shift of 405º is indistinguishable from one of 45º—, only a certain range of velocities can be mapped. Hence, in order to consider both positive and negative velocities symmetrically, the selected range of angles is chosen to be from -180º to 180º. Any angle measurements that exceed 180º in the positive or negative direction will demonstrate aliasing and not be displayed accurately [33].

To be more specific[2], let us define the phase shift ϕ produced by a gradient on a spin with constant velocity [34]:

$$\phi = 2\pi\bar{\gamma} \cdot m_1 \cdot v_{\parallel} + \phi_0 \tag{2.34}$$

[2]Further details can be found in Appendix B.3.

where $\bar{\gamma}$ is the gyromagnetic ratio in units of Hz/T, $v_{\|}$ is the spin velocity component in the parallel direction to the gradient vector, and m_1 is the first moment of the gradient. The term ϕ_0 represents all contributions to the MR phase which are not related to flow [34], and which can be neglected in this study case, with the exception of the phase offset produced by the RF pulse[3]. When the gradient waveform is toggled between two shapes A and B, as indicated in Figure 2.24, the change in first moment is given by $\Delta m_1 = m_{1A} - m_{1B}$, so that for each spin [34]:

$$\Delta\phi = \phi_A - \phi_B = 2\pi\bar{\gamma} \cdot \Delta m_1 \cdot v_{\|} \tag{2.35}$$

Two complete sets of raw image data are acquired, A and B. A phase difference reconstruction is then applied to the two raw data sets to obtain $\Delta\phi$, which is proportional to the velocity $v_{\perp}$ and whose sign determines the flow direction. As for the magnitude image M, it can be reconstructed from the average of M_A and M_B, or using exclusively one of the two acquisitions, M_A or M_B. Because γ and Δm_1 are known, one can quantitively extract the value of $v_{\perp}$ from Equation (2.35) [34].

Therefore, given a certain gradient, the aforementioned aliasing will occur when the velocity to be measured produces a phase difference $\Delta\phi$ outside the range of [-π, π]. The maximum velocity that can be measured before aliasing occurs is called the encoding velocity ($\mathrm{V_{ENC}}$), and its value can be derived from Equation (2.35):

$$\Delta\phi_{\max} = \pi = 2\pi\bar{\gamma} \cdot \Delta m_1 \cdot \mathrm{V_{ENC}} \quad \Rightarrow \quad \mathrm{V_{ENC}} = \frac{1}{2\bar{\gamma} \cdot \Delta m_1} \tag{2.36}$$

Since this value is inversely proportional to the strength of the gradient, $\mathrm{V_{ENC}}$ can be increased by decreasing the strength of the gradient and viceversa [33]. For the specific case of trapezoidal bipolar gradients (see Figure 2.26), it can be demonstrated that $\Delta m_1 = 2GT(T-\delta)$, where T is the long base of the trapezoid, G is its height and δ corresponds to the rise time. Consequently, these are the three design parameters when defining $\mathrm{V_{ENC}}$.

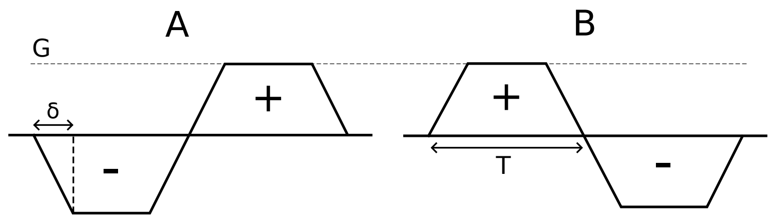


Figure 2.26: Trapezoidal bipolar gradients used in PC-MRI.

The trade-off is that the spatial resolution for small velocities degrades if $\mathrm{V_{ENC}}$ is too high [33]. Hence, the key consideration lies in estimating the maximum velocity expected to be measured and defining a $\mathrm{V_{ENC}}$ accordingly.

Time of Flight (TOF) TOF effects refer to signal variations which result from the motion of spins flowing into or out an imaging volume during a given pulse sequence. In particular, the effect known as flow-related enhancement [26] is a consequence of magnetic saturation, which is related to the steady state explained in Section 2.1.3b.

The tissues within the imaged volume reach a steady state when they are subjected to a continuous train of RF pulses with short TR. This means that the next RF pulse is applied before the longitudinal magnetization M_z has fully recovered from the preceding pulse. As a result, the maximum achievable longitudinal magnetization, and consequently the produced MR signal, are lower than that of tissues which were not initially present in the imaged volume. "Fresh" blood flowing into the imaged slab has not been subjected to these RF pulses and is therefore fully magnetized (unsaturated). When a bolus

[3]The phase offset produced by an RF pulse is determined by its envelope components, $B_1 = B_{1x} + jB_{1y}$. These components define the rotation axis for the magnetization vector. Thus, a purely real and positive envelope will produce a phase offset of $\pi/2$, while a purely imaginary and negative one will produce a zero offset.

of this fresh blood enters a slice and is subjected to its first set of RF pulses, its signal is significantly stronger than that of the tissues (and other blood) within the slab [26]. Figure 2.27 illustrates this phenomenon and its result in the acquired images.

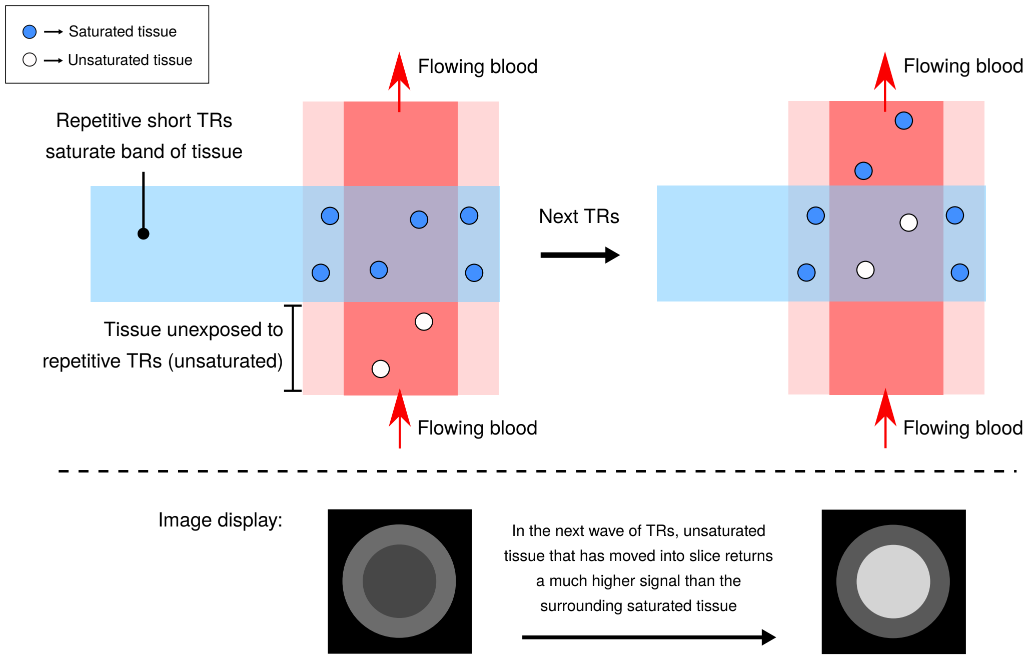
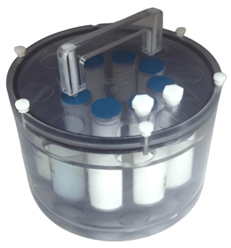

● → Saturated tissue
○ → Unsaturated tissue
Flowing blood
Flowing blood
Repetitive short TRs saturate band of tissue
Next TRs
Tissue unexposed to repetitive TRs (unsaturated)
Flowing blood
Flowing blood
Image display:
In the next wave of TRs, unsaturated tissue that has moved into slice returns a much higher signal than the surrounding saturated tissue

Figure 2.27: Time of Flight (TOF) effect and its resulting images.

2.1.4 Imaging Phantoms

To reproducibly and precisely characterize the ability of a medical imaging system to safely produce accurate images, it is useful to image objects which have properties similar to human tissues for the imaging method being tested, have geometries and compositions that are known exactly, and can remain in an exact location in the imaging system for indefinite periods of time. For this purpose, imaging phantoms are produced for every method (or modality) of medical imaging [35].

Today, imaging phantoms typically consist of materials —liquids, gels, semisolids, and/or solids depending on the tissue being mimicked and the imaging modality being evaluated— with desired properties and known geometries arranged in glass or plastic containers, sometimes also with glass or plastic inclusions of known dimensions [35]. Figure 2.28 shows two examples of phantoms used in MRI.


(a)

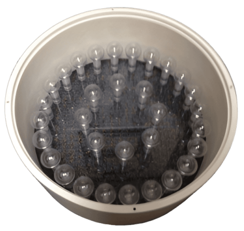

(b)

Figure 2.28: Examples of phantoms used in MRI [36].

2.2 MRI Simulation Principles

In the context of MRI simulation, there are two main paradigms which differ in their approach for generating MRI data. These two approaches may be referred to as analytic simulation and Bloch simulation. The former uses simplified mathematical models to describe the MR process and to obtain the resulting images. These models are based on closed-form expressions that relate tissue properties (such as T_1 and T_2) and MRI sequence parameters, quickly producing the expected image. As for the latter strategy, it employs numerical methods to directly solve the Bloch equations, providing highly accurate results.

According to [3], which presents an analytic MRI simulator, the use of direct expressions makes it necessary to impose the generation of artifacts such as noise, chemical shift or fold-over. However, more complex artifacts such as eddy currents or off-resonance are not so straightforward, so simulated images may not always be realistic. Therefore, while analytic simulators are effective for basic educational purposes due to their speed, they are not suitable for research environments that demand accurate and realistic results. Furthermore, the main drawbacks of Bloch simulators, specifically, their slowness and computational cost, are now being mitigated through the use of GPUs and optimized algorithms. In this context, the state of the art is in Bloch simulators, also referred to as numerical simulators, which will be the focus of this study. This section will cover the common fundamentals of these simulators.

2.2.1 Pulse Sequence

The concept of MRI pulse sequence in simulation does not differ from what was explained in Section 2.1.3. However, there are key differences in how sequences are stored, formatted, and modified between real MRI scanners and simulators. In real scanners, sequences are saved in proprietary file formats, often including calibrated parameters to match their specific hardware limitations. In simulators, sequences are generally stored in more flexible open formats like JSON, XML, or the specific format for MRI sequences, Pulseq. Many simulators also offer their own sequence design frameworks, which are compatible with that particular simulator and, in some cases, with Pulseq as well.

Pulseq [13] provides a hardware-independent, open-source programming environment that promotes rapid sequence prototyping, allowing sequences to be designed in high-level languages such as Matlab or Python, or through a GUI. The project also defines a specific file format (`.seq`) that facilitates the storage and sharing of sequences, enhancing compatibility and reproducibility across different simulation and scanner environments.

2.2.2 Digital Phantom

A digital phantom is a computational representation of a physical object, such as the human body or specific anatomical structures, utilized in simulations to replicate the characteristics and behaviours associated with actual MRI procedures. Instead of using a physical object (see Section 2.1.4) for testing, the digital phantom allows for virtual experiments.

This computational model is designed to encapsulate essential information regarding the spatial distribution and/or displacements of tissues, along with their MRI-relevant properties, including T_1 and T_2 relaxation times, proton density (PD), and off-resonance effects, among others.

2.2.3 Solving the Bloch equations

Numerical simulators are based on a classic description of MRI physics as described by the Bloch equations (see Eq. (2.9)) in the rotating reference frame[4]. Although some simulators use more general

[4]The rotating reference frame (see Appendix A.4) is a transformed coordinate system so observer rotates with spins at Larmor frequency. This simplifies the analysis of NMR phenomenon for several reasons, one of which is that it eliminates the contributions of the static magnetic field $\boldsymbol{B}_0$ to the total magnetic field vector $\boldsymbol{B}$.

equations, such as Bloch–Torrey to simulate diffusion and Bloch–McConnell to simulate chemical exchange [4], they all rely on the same principle.

The effective magnetic field at time t is modelled by[5]:

$$\boldsymbol{B}(\boldsymbol{r},t) = \begin{bmatrix} B_{1x}(t) \\ B_{1y}(t) \\ \mathbf{G}(t)^T \boldsymbol{r} + \Delta B_z(\boldsymbol{r},t) \end{bmatrix} \tag{2.37}$$

where $\Delta B_z(t)$ may include, depending on the simulator, off-resonance effects, such as chemical shift, non-linear gradients, eddy currents, or concomitant gradient fields [5, 37], among others. Some simulators may also consider multi-coil acquisitions and coil sensitivity maps, so B_{1x} and B_{1y} should be also position ($\boldsymbol{r}$) dependent in that case.

The matrix form of the Bloch Equation

$$\frac{d}{dt}\begin{pmatrix} M_x \\ M_y \\ M_z \end{pmatrix} = \begin{pmatrix} -\frac{1}{T_2} & \bar{\gamma}B_z & -\bar{\gamma}B_y \\ -\bar{\gamma}B_z & -\frac{1}{T_2} & \bar{\gamma}B_x \\ \bar{\gamma}B_y & -\bar{\gamma}B_x & -\frac{1}{T_1} \end{pmatrix}\begin{pmatrix} M_x \\ M_y \\ M_z \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \frac{M_0}{T_1} \end{pmatrix} \tag{2.38}$$

simplifies the calculations to be performed by the simulators. Specifically, it allows the temporal evolution of $\boldsymbol{M}$ to be calculated using matrix operations. This is especially useful for numerical methods, as optimized algorithms for linear systems can be applied. Moreover, it allows dividing the vector $\boldsymbol{M}$ into its three spatial components (M_x, M_y, M_z).

Solving this equation requires a previous time discretization stage, in which the total sequence duration (suppose a time interval $[0, T]$) is divided into N time steps of duration Δt[6]. Thus, at each time step, the aim is to calculate the evolution of $\boldsymbol{M}(t)$ from the value of the previous step, thereby determining $\boldsymbol{M}(t + \Delta t)$ from $\boldsymbol{M}(t)$. Therefore, for the n-th spin [8]:

$$\boldsymbol{M}_n(t + \Delta t) = T_n(\Delta t) \circ \Phi_n\big(t, \Delta t, \boldsymbol{r}_n(t)\big) \circ R\big(t, \Delta t, \boldsymbol{r}_n(t)\big) \circ \boldsymbol{M}_n(t) \tag{2.39}$$

where the $\circ$ operator represents the composition of transformations applied sequentially to the magnetization vector $\boldsymbol{M}_n(t)$. Specifically, $R(t, \Delta t, \mathbf{r}_n(t))$ is the rotating operation corresponding to RF application, $\Phi_n(t, \Delta t, \mathbf{r}_n(t))$ is the rotation operator corresponding to precession, and $T_n(\Delta t)$ is the relaxation operator [8]. The specific definitions of each of these terms depends on the simulator. Additionally, the method employed to solve this iterative equation may vary, with options including numerical techniques such as Runge-Kutta, operator splitting [38], and others.

Since Equation (2.39) represents the magnetization and, consequently, the signal produced by a single spin, the total magnetization at any given time instant is the sum of the contributions from all the spins within the digital phantom.

2.2.4 Incorporating flow data within MRI simulations

In order to simulate flow-related MRI techniques, it will be necessary for the flow data to be stored and included as part of the input arguments of the simulator. Specifically, these data should be inherent to the phantom. What follows is a description of how flow information is defined and obtained, and how such data can be incorporated into MRI simulations.

Fluid field specification For the description of the continuous fluid medium motion, two alternatives can be used. The first, attributed to Lagrange, identifies each point in the fluid by its initial

[5]Note that this is a more general form for Equation (2.20), as it now takes into account other effects in addition to $\boldsymbol{B}_0$ inhomogeneities.

[6]Δt does not have to be constant for all time steps. Some MRI simulators use N equal time steps, while others allow for variability in the length of each time step. This flexibility enables data to be collected with greater precision if required by the complexity of the sequence.

position $\boldsymbol{r}_0$, and tracks its evolution over time, along with the fluid properties associated with it. Thus, the trajectory function is given by

$$\boldsymbol{r}_T = \boldsymbol{r}_T(\boldsymbol{r}_0, t_0, t), \tag{2.40}$$

which determines the position at successive time instants t of the material point that was at position $\boldsymbol{r}_0$ at time t_0. The velocity and acceleration of this material point are as follows [39]:

$$\boldsymbol{v}(\boldsymbol{r}_0, t_0, t) = \frac{\partial \boldsymbol{r}_T}{\partial t}, \quad \boldsymbol{a}(\boldsymbol{r}_0, t_0, t) = \frac{\partial^2 \boldsymbol{r}_T}{\partial t} \tag{2.41}$$

Alternatively, the Eulerian perspective is the one used to describe fluid motion. Its focus is not on tracking individual material points of the fluid. Instead, fluid-mechanical quantities are defined at each point $\boldsymbol{r}$ within the fluid domain, fixed in relation to the reference frame, at every time instant. In this approach, position and time are treated as independent variables. The local fluid velocity $\boldsymbol{v}(\boldsymbol{r}, t)$ is taken as the fundamental variable. When $\boldsymbol{v} = \boldsymbol{v}(\boldsymbol{r})$ is independent of time, the motion is referred to as steady. On the other hand, when the fluid quantities are the same at every instant for any point in the fluid domain, the motion is called uniform. In this case, the velocity depends only on time, $\boldsymbol{v} = \boldsymbol{v}(t)$ [39].

Trajectories The material points trajectories are defined by the system of differential equations

$$\frac{d\boldsymbol{r}}{dt} = \boldsymbol{v}(\boldsymbol{r}, t), \tag{2.42}$$

which, along with the initial conditions $\boldsymbol{r}(t_0) = \boldsymbol{r}_0$, determine the trajectories

$$\boldsymbol{r} = \boldsymbol{r}_T(\boldsymbol{r}_0, t_0, t). \tag{2.43}$$

The intersection of the surfaces $F_1(\boldsymbol{r}, \boldsymbol{r}_0, t_0) = 0$ and $F_2(\boldsymbol{r}, \boldsymbol{r}_0, t_0) = 0$, obtained by eliminating t in Equation (2.43), is a curve that represents the path or trajectory traced by the material point in its motion [39].

CFD Computational Fluid Dynamics (CFD) are a branch of fluid mechanics that uses numerical analysis and data structures to analyze and solve problems that involve fluid flows. Computers are used to perform the calculations required to simulate the free-stream flow of the fluid, and the interaction of the fluid with surfaces defined by boundary conditions. This technique is used, among other purposes, to obtain velocity fields within the domain of a volume.

Specifically, the method involves discretizing a region of space by creating what is known as a spatial mesh[7], dividing the region into small control volumes. The discretized conservation equations are then solved within each of these volumes. In practice, this involves solving an algebraic matrix in each cell iteratively until the residual is sufficiently small. As a result, a discretized velocity mesh in both space and time is obtained, representing the velocity field within the geometry and following an Eulerian approach.

Flow MRI simulation CFD data must then be incorporated into the phantom, either in their Eulerian form or as Lagrangian trajectories. In either case, when it comes to simulation, the need to account for the dynamics of the spins leads to a significant increase in computational load [37].

In its traditional formulation, the Bloch equations (2.9) are defined for each isochromat[8], for which they are ordinary differential equations expressed as a Lagrangian formalism. Therefore, it is necessary to establish consistency between the formalism used for the CFD data and that of the Bloch equations.

[7]The spatial mesh can be either structured, where the elements are regular and uniform, or unstructured, consisting of irregular elements. The former are easy to generate and store but do not adapt well to complex geometries and may require many elements to capture fine details. Unstructured meshes, on the other hand, are more flexible and adaptable, but they come with increased complexity and overhead since they require more data and connectivity information.

[8]The term isochromat refers to a microscopic group of spins that resonate at the same frequency.

A classical approach often adopted in the literature consists in solving the Eulerian formulation of the Bloch equations [40, 41], where the CFD velocity $\boldsymbol{u}$ is used to transport the magnetization vector and a convection term is explicitly added to the time rate of change of the magnetization vector [37]:

$$\frac{d\boldsymbol{M}}{dt}(\boldsymbol{r},t) = \frac{\partial \boldsymbol{M}(t)}{\partial t} + \big(\boldsymbol{u}(\boldsymbol{r},t)\cdot\nabla\big)\boldsymbol{M}(t) \tag{2.44}$$

This approach has a relatively low computational cost since both the flow and Bloch equations are solved on the same fixed mesh without needing velocity interpolation. However, as explained in [37], it presents some disadvantages such as the lack of analytical solution or the rapid velocity changes compared with the time scale of the MR sequence, which make this approach less suitable for complex flow configurations.

An alternative approach consists in modelling the spin isochromats with Lagrangian particles —as in Equation (2.43)— using the CFD velocity to update each particle position. The Bloch equations can then be solved independently for each particle, with no spin-spin interaction [42–44]. Therefore, the computational load can be easily partitioned on multiple cores to accelerate the calculations. Nevertheless, a sufficient number of particles is required to accurately approximate the MR signal [37]. Furthermore, the need to store information about the position of each particle means that, in cases where high spatial and temporal resolution is required, the size of the files involved can become prohibitively large, particularly when simulating long-duration sequences such as 4D flow MRI. Handling these files, as well as interpolating the particles positions at the time instants required by the MRI sequence, can lead to unbearable computational costs [37].

The approach to addressing these flow-related challenges, particularly those associated with the second method, depends on the simulator employed.

2.3 State of the Art in MRI Simulation

This section provides a detailed explanation of those simulators that stand out in the state of the art, mainly, JEMRIS [5], CMRsim [8], KomaMRI [4], and an unnamed hybrid —numerical + analytic— approach [37] for efficiently simulating MRI acquisitions under realistic flow conditions. An explanation will be provided on how the MRI simulation is carried out, specifically focusing on how motion is incorporated within these simulations.

2.3.1 JEMRIS

General overview JEMRIS (Jülich Extensible MRI Simulator) [5] is currently the most widely used open source MRI simulation tool. Released in 2010, its development was driven by the desire to achieve generality of simulated three-dimensional MRI experiments reflecting modern MRI systems hardware. Until then, no MRI simulator had simultaneously addressed considerations such as parallel transmit and receive, off-resonance effects, nonlinear gradients, and arbitrary spatio-temporal parameter variations at different levels [5]. JEMRIS was developed in C++ to achieve usability, as well as performance. Among its features, it includes three MATLAB GUIs for interactively designing MRI sequences, defining coil configurations, and executing simulations. Additionally, it supports exporting the created sequences to Pulseq.

Operating principles JEMRIS follows the methodology outlined in Section 2.2.3 to tackle the Bloch equations. First, considering N coils, each with a position-dependent $\boldsymbol{B}_1$ field, Equation (2.37) becomes more specific [5]:

$$\boldsymbol{B}(\boldsymbol{r},t) = [\mathbf{G}(t)^T\boldsymbol{r} + \Delta B_z(\boldsymbol{r},t)]\,\hat{z} + \sum_{n=1}^{N}[\boldsymbol{B}_{1x}^{n}(\boldsymbol{r},t)\,\hat{x} + \boldsymbol{B}_{1y}^{n}(\boldsymbol{r},t)\,\hat{y}] \tag{2.45}$$

where $\Delta B_z(\boldsymbol{r},t)$ accounts for non-linear gradient terms, chemical shift, random mesoscopic frequency fluctuations, macroscopic field variations, and concomitant gradient fields [5].

Moreover, JEMRIS uses the formulation of the Bloch Equation (2.38) in cylindrical coordinates, as it is well suited for numerical implementation [5]:

$$\frac{d}{dt}\begin{pmatrix} M_r \\ \phi \\ M_z \end{pmatrix} = \begin{pmatrix} \cos\phi & \sin\phi & 0 \\ -\frac{\sin\phi}{M_r} & \frac{\cos\phi}{M_r} & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \left[\begin{pmatrix} -1/T_2 & \bar{\gamma}B_z & -\bar{\gamma}B_y \\ -\bar{\gamma}B_z & -1/T_2 & \bar{\gamma}B_z \\ \bar{\gamma}B_y & -\bar{\gamma}B_x & -1/T_1 \end{pmatrix} \cdot \begin{pmatrix} M_r\cos\phi \\ M_r\sin\phi \\ M_z \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ M_0/T_1 \end{pmatrix} \right] \quad (2.46)$$

Specifically, it relies on the CVODE [45] library, a highly optimized integrator for ordinary differential equations (ODEs). The methods used in CVODE are variable-order, variable-step multistep methods, allowing both the integration order and the step size to be dynamically adjusted for each time step. This ensures that the local error remains consistently below a predetermined threshold, resulting in extremely accurate solutions. However, the trade-off lies in processing times, which may not be as fast as they could be (see Figure 2b in [5]). In order to divide the problem into processes that can be executed in parallel and thus reduce computation times, JEMRIS uses CPU multithreading via the Message Passing Interface (MPI) library; nonetheless, the simulator does not include GPU support.

Motion In its original version [5], JEMRIS includes, by default, the capability to simulate dynamic models with rigid motion; that is, it only allows specifying a single trajectory for all spins, hence simulating the movement of a rigid sample. For instance, Figure 2.29, extracted from [5], shows the result of simulating a dynamic MRI sequence over a phantom that consists of a moving ball within a ring. This phantom is split into a static and a moving component, which are simulated separately; after the simulation, the results are combined by adding the two signals.

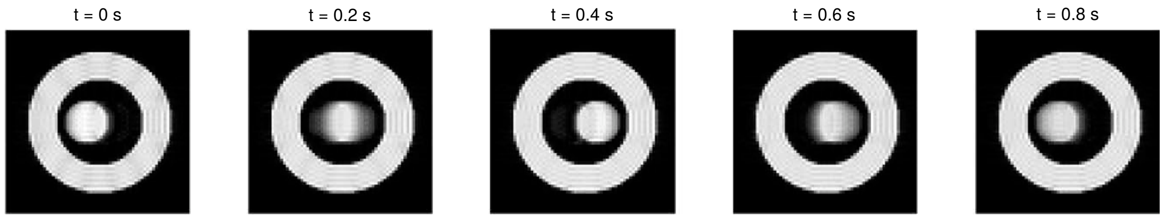


Figure 2.29: JEMRIS simulation of an oscillating sphere inside a static ring. [5].

Adding to this complexity is the fact that defining a dynamic phantom in JEMRIS requires two separate files. The first is an HDF5 (.h5) file, which contains three datasets: the `data` dataset stores the PD, T_1, T_2, T_2^*, and $\Delta\omega$ maps. The `offset` dataset is a three-element vector that specifies the phantom's global position offset along the x, y and z axes relative to the origin. Last, the `resolution` dataset is another 3x1 array which defines the spatial resolution in each dimension, i.e., the spin separation in x, y, and z.

The second file, either in ASCII (.dat) or HDF5 (.h5) format, contains the motion-related information. It describes the rigid motion of the phantom, in which translations and rotations are supported. For ASCII, it presents the following format, where each line provides the position and rotation of the phantom at successive time points:

```
Rot_origin_x   Rot_origin_y   Rot_origin_z

time_1   X_pos(t_1)  Y_pos(t_1)  Z_pos(t_1)  X_rot(t_1)  Y_rot(t_1)  Z_rot(t_1)

  .

  .

  .

time_n   X_pos(t_n)  Y_pos(t_n)  Z_pos(t_n)  X_rot(t_n)  Y_rot(t_n)  Z_rot(t_n)
```

Nowadays, and thanks to contributions [44, 46, 47] made after the original version of JEMRIS, the following motion-related functionalities are included:

- Flow MRI simulations [44, 46]. By extending the C++ code with a new *TrajectoryFlow* class, it is possible to specify a different trajectory for each spin, thus enabling the description of flow phenomena [46]. In particular, the application of a Lagrangian description of the flow was the most obvious approach considering the operating mode of JEMRIS. The results of this contribution confirm that any experiment involving both static tissues and complex flow data from CFD can be simulated. However, in many cases —and especially when dealing with a large number of spins or with complex sequences— simulation times become impractically large [44].
- Motion of deformable phantoms [47], which can be used for realistic simulations of cardiac MRI or respiration. The 4D Extended Cardiac-Torso (XCAT) [48] software was used to define both the anatomical maps and the motion pattern of the phantom. The definition of independent trajectories for each spin is made possible, once again, through the development of a JEMRIS extension. As before, the results are realistic. However, the drawback is a high simulation time, especially when a large number of spins are required or selective excitations are applied [47].

Advantages and limitations In light of this, JEMRIS demonstrates the distinct advantage of offering highly realistic simulations with the capability to incorporate all types of motion, closely aligning with actual MRI phenomena. Additionally, it supports multiple coils.

However, this level of realism also presents some limitations. Configuring motion is often complex and time-consuming, as each specific trajectory requires careful definition. Additionally, the high computational demands result in prolonged simulation times. These factors can restrict the practicality of JEMRIS for large-scale simulations.

2.3.2 CMRsim

General overview CMRsim [8] is a newly released framework that enables the simulation of CMR images using dynamic digital phantoms as input. Featuring an easy-to-use API written in pure Python, CMRsim is specifically designed to simplify the task of defining complex motions for the user, making this its primary strength. The framework also provides modularity for designing comparative simulation experiments, supported by an architecture that allows for the seamless integration of custom simulation modules. Furthermore, CMRsim promotes reproducibility through the use of versioned and documented open-source software, and it is readily scalable to support highly detailed digital phantoms. In terms of performance, competitive simulation speeds are achieved using TensorFlow for CPU/GPU acceleration.

Regarding compatibility with Pulseq, this tool supports the reading and subsequent simulation of files in this format. Furthermore, although it is not part of the simulator itself, the same research group that developed CMRsim also created CMRseq [49], a sequence design tool that allows for both reading and writing these file types.

The description of dynamic digital phantoms in CMRsim is based on the Lagrangian framework with non-interacting "massless" particles. Each particle n is defined by a time-dependent trajectory $\boldsymbol{r}_n(t)$ and a set of properties $\Gamma_n(\boldsymbol{r}_n, t) = \{M_0(\boldsymbol{r}_n, t), T_1(\boldsymbol{r}_n, t), T_2(\boldsymbol{r}_n, t)...\}$ including available magnetization M_0 and relaxation times T_1 and T_2 [8]. For practical purposes, the information of these fields is stored in NumPy arrays of length N_{spins} each.

In regard to particle trajectories, they are represented by means of the so-called *Trajectory Modules*, which provide an interface to query positions $\boldsymbol{r}(t)$ at any given time point t. This abstraction decouples motion implementation from MR signal simulation and therefore ensures modularity and extensibility [8].

Operating principles To perform simulations, CMRsim offers two paradigms: numerical and analytical solutions to the Bloch equations [8]. For the reasons outlined at the beginning of this section, the focus here will be on the former. Returning to Equation (2.39), it decomposes the problem into three components, described below. This technique, known as operator splitting [38], is used by CMRsim to solve the differential Bloch Equation.

1. The RF rotation operator is defined by the following matrix [8]:

$$R(t,\Delta t) = \begin{bmatrix} \cos^2\frac{\psi}{2} & \mathrm{e}^{i2\theta}\sin^2\frac{\psi}{2} & -i\mathrm{e}^{\mathrm{i}\theta}\sin\psi \\ \mathrm{e}^{-i2\theta}\sin^2\frac{\psi}{2} & \cos^2\frac{\psi}{2} & i\mathrm{e}^{-\mathrm{i}\theta}\sin\psi \\ -\frac{i\mathrm{e}^{-\mathrm{i}\theta}\sin\psi}{2} & \frac{i\mathrm{e}^{\mathrm{i}\theta}\sin\psi}{2} & \cos\psi \end{bmatrix} \tag{2.47}$$

   This matrix uses the the so-called spinors to carry out the rotation of the vector $\boldsymbol{M}$. The spinor rotation method is detailed in Appendix A.3, where Equation (A.12) directly corresponds to (2.47). It is also important to note that, when working with spinors, $\boldsymbol{M}$ is not explicitly defined by its three spatial components (M_x, M_y, M_z), but rather as $\boldsymbol{M} = \left[M_{xy}, M_{xy}^{*}, M_z\right]^T$. Regarding the angle ψ, it is calculated through integration with the trapezoidal rule (see Eq. (2.48)). θ is determined by the RF pulse angle at the midpoint of the simulated interval:

$$\psi(t,\Delta t) = \gamma|B_1(t+\Delta t) + B_1(t)|\frac{\Delta t}{2}, \qquad \theta(t,\Delta t) = \angle\left(\frac{B_1(t+\Delta t)+B_1(t)}{2}\right) \tag{2.48}$$

   $R(t,\Delta t)$ represents a rotation of the magnetization vector with respect to the transversal components of the net magnetic field, that is, $\boldsymbol{B}_{\mathbf{I}} = (B_x, B_y, 0)$. In other words, a rotation occurs around the axis defined by the direction of the RF pulse.

2. The subsequent precession of the particle is defined as [8]:

$$\Phi_n\big(t,\Delta t,\boldsymbol{r}(t)\big)\circ\boldsymbol{M}_n = \mathrm{diag}\left(\mathrm{e}^{\mathrm{i}\phi_n(t,\Delta t,\boldsymbol{r}(t))}, \mathrm{e}^{-\mathrm{i}\phi_n(t,\Delta t,\boldsymbol{r}(t))}, 1\right)\cdot\boldsymbol{M}_n \tag{2.49}$$

   where the precession phase $\phi_n(t,\Delta t,\boldsymbol{r}(t))$ depends on the phase-generating effects captured by dedicated submodules φ_s. All submodules have access to the gradient waveform, particle position, and all additional fields $\Gamma(\boldsymbol{r}(t),t)$. The precession phase hence is calculated as [8]:

$$\phi_n\big(t,\Delta t,\boldsymbol{r}(t)\big) = \gamma\Big(\mathbf{G}(t+\Delta t)\cdot\boldsymbol{r}^n(t+\Delta t)+\mathbf{G}(t)\cdot\boldsymbol{r}^n(t)\Big)\frac{\Delta t}{2}+\sum_s\varphi_s\Big(\mathbf{G}(t),\boldsymbol{r}^n(t),\Gamma\big(\boldsymbol{r}(t),t\big)\Big) \tag{2.50}$$

   where the first term is the integral of the product of gradients and positions, which results in the magnetic fields. This integral is solved by means of the trapezoidal rule. The second term accounts for the off-resonance effects.

   $\Phi_n\big(t,\Delta t,\boldsymbol{r}(t)\big)\circ\boldsymbol{M}_n$ then represents a rotation of the magnetization vector with respect to the longitudinal component of the net magnetic field, namely $\boldsymbol{B}_{\mathbf{II}} = (0,0,B_z)$, which is the component affected by the gradients.

3. Finally, the relaxation operator is defined as [8]:

$$T_n(\Delta t)\circ\boldsymbol{M}_n = \mathrm{diag}\left(\mathrm{e}^{-\frac{\Delta t}{T_2^n}}, \mathrm{e}^{-\frac{\Delta t}{T_2^n}}, \mathrm{e}^{-\frac{\Delta t}{T_1^n}}\right)\cdot\boldsymbol{M}_n + \begin{bmatrix}0 & 0 & 1-\mathrm{e}^{-\frac{\Delta t}{T_1^n}}\end{bmatrix}^T\cdot M_z^0 \tag{2.51}$$

For each sampling event, the signal $s_m(t)$ for a coil with spatial sensitivity $C_m(\boldsymbol{r})$ is computed as the sum over the transverse magnetization M_{xy}^n of all N particles [8]:

$$s_m(t) = \sum_{n=0}^{N} M_{xy}^n(t)\cdot C_m(\boldsymbol{r}^n(t))\cdot\mathrm{e}^{-\mathrm{i}\vartheta_{ACQ}(t)} \tag{2.52}$$

where the receiver phase $\vartheta_{ACQ}(t)$ is a simulation parameter.

A comprehensive explanation is provided in [8], where the process of incorporating effects such as off-resonance phase accumulation into the simulation —through $\varphi_s(\cdot)$ indicated above— is also discussed.

Motion It has been mentioned that the calculation of the trajectories is independent of the simulation. The following method is used inside the Bloch simulation loop in every integration step, before performing the actual simulation calculations. It takes the current positions $\boldsymbol{r}(t)$ as well as the temporal step size Δt and returns the updated positions $\boldsymbol{r}(t+\Delta t)$ and additional fields $\Gamma\big(\boldsymbol{r}(t+\Delta t),\ t+\Delta t\big)$ [8]:

$$f\big(\boldsymbol{r}(t), \Delta t\big) \rightarrow \{\boldsymbol{r}(t+\Delta t),\ \Gamma\big(\boldsymbol{r}(t+\Delta t),\ t+\Delta t\big)\}, \quad \text{where } \boldsymbol{r}(\cdot) \in \mathbb{R}^{N\times 3} \tag{2.53}$$

Although the basis is described in Equation (2.53), the implementation of this method depends on the specific *Trajectory Module*. Among others, of particular interest is the *FlowTrajectory* module, which accounts for flow-based motion by numerically integrating particle paths in spatially dependent velocity fields. Its method `increment_particles` moves the particles according to:

$$\boldsymbol{r}(t+\Delta t) = \boldsymbol{r}(t) + \Delta t \cdot \boldsymbol{v}(\boldsymbol{r}(t)) \tag{2.54}$$

where $\boldsymbol{v}(\boldsymbol{r}(t))$ corresponds to the velocity field at each point in the geometry. The velocity field corresponds to a structured mesh extracted from a VTK file, which can, for instance, be generated using CFD.

Therefore, two calculations need to be performed "on the fly" for each spin and each time step:

1. Spatial interpolation is used to obtain the velocity value at any point in space, even at locations that do not coincide with the nodes of the structured velocity mesh.
2. Once the velocity at the exact point in space is obtained, Equation (2.54) is solved, thus obtaining the updated particle position.

Advantages and limitations From all the above, it can be concluded that the advantages of CMRsim include its ease of defining and simulating complex movements, its simple installation, and its open-source, extensible nature. Additionally, it offers quite acceptable simulation speeds and supports multiple coils, further enhancing its capabilities.

On the other hand, CMRsim presents some limitations. For instance, the operator splitting approach in three parts[9] can increase numerical error, especially when off-resonance effects are present. Additionally, while simulation speeds are generally acceptable, they could be improved, as Python, even with TensorFlow, is not the fastest language, and on-the-fly trajectory calculations from CFD data also contribute to a lower performance. Regarding turbulent flow, it is recreated by combining the velocity field with a random component based on the Langevin [50] Equation[10], which simplifies the complex dynamics of turbulence by reducing it to a stochastic process. Finally, CMRsim only supports structured meshes. If it reads a VTK file containing an unstructured mesh, it must convert it into a structured format, which presents challenges when dealing with complex geometries.

2.3.3 KomaMRI

General overview KomaMRI [4] is a Julia package meant to simulate general MRI scenarios. It generates raw data by solving the Bloch equations. As it was mentioned in Chapter 1, this tool stands out by offering a wide set of capabilities that make it high versatile and accessible. It supports all major operating systems and provides a GUI (Figure 2.30) that makes it user-friendly for researchers across different levels of expertise[11]. Its vendor-agnostic GPU support facilitates accelerated simulations on any hardware, and its open-source nature allows modifying and building upon its codebase, promoting customization and collaboration. KomaMRI is also compatible with key MRI community standards,

[9] These three parts are well-defined in Equation (2.39). Recall that $R(t, \Delta t, \mathbf{r}_n(t))$ represents the rotation of the magnetization around $\boldsymbol{B}_{\mathbf{I}} = (B_x, B_y, 0)$, $\Phi_n(t, \Delta t, \mathbf{r}_n(t))$ corresponds to the rotation around $\boldsymbol{B}_{\mathbf{II}} = (0, 0, B_z)$, and $T_n(\Delta t)$ is the relaxation operator. Other simulators, such as KomaMRI, use a single rotation operator around $\boldsymbol{B} = (B_x, B_y, B_z)$, which reduces the operator splitting from three separate steps to only two steps.

[10] A specific turbulent flow example is shown in Section 4.1.3c.

[11] Further effort has been done in this Ms. Thesis with respect to the GUI (see Sections 3.2 and 4.2).

as it supports both reading and writing[12] Pulseq (`.seq`) [13] files, and export the simulation results to ISMRMRD (`.mrd`) [14] format. Furthermore, it comes with detailed documentation[13] that includes installation guides, practical examples, and comprehensive information about each of its functions. Many of these strenghts are due to the fact that it is written in the Julia programming language [15] The key features of Julia will be discussed in the following section of this document.

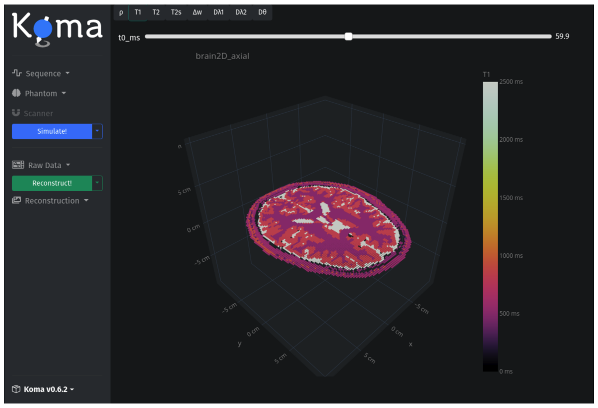


Figure 2.30: KomaMRI GUI, displaying the 3D phantom viewer [4].

Simulation pipeline and code structure Figure 2.31 shows the simulation pipeline followed by KomaMRI, which is divided in two steps: simulation and reconstruction. The latter is handled by a third-party Julia package called MRIReco [51], which is responsible for transforming the simulated raw signal into an image. As for the former, it requires three input arguments to perform the simulation: the scanner parameters, the phantom and the pulse sequence. Moreover, KomaMRI is structured into five sub-packages: `KomaMRIBase` for custom types and functions, `KomaMRICore` for simulation functions, `KomaMRIFiles` for file I/O, `KomaMRIPlots` for plotting, and `KomaMRI` as the main GUI. This modular design enhances maintainability and allows users to access only the necessary components. For instance, GUI users interact with the complete `KomaMRI` package, which integrates all sub-packages, while advanced users can directly utilize specific sub-packages such as `KomaMRICore` for simulation purposes.

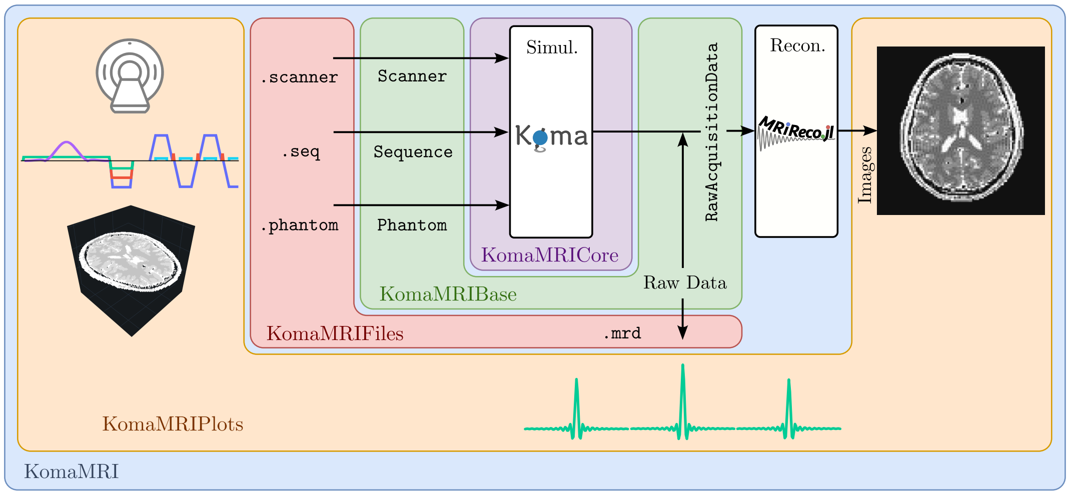


Figure 2.31: Basic diagram of the KomaMRI simulator and organization of its code into sub-packages [4].

[12]Pulseq file writing is still under development. Its progress can be tracked in GitHub Pull Request #284: https://github.com/JuliaHealth/KomaMRI.jl/pull/284.

[13]KomaMRI documentation is available at https://juliahealth.org/KomaMRI.jl.

Focusing on the digital phantom of KomaMRI, it is defined by the `Phantom` structure, the definition on which is shown in Code 2.1. Initial positions (`x`, `y`, `z`), contrast-related (`T1`, `T2`, `T2s`, ρ) information, as well as off-resonance effects (`Δw`) associated with each spin, are all stored in the form of Julia vectors. The `Phantom` structure also contains diffusion-related fields (`Dλ1`, `Dλ2`, `Dθ`), which are not currently used in the simulations. As for **motion**, `ux`, `uy`, and `uz` fields are three functions that allow for the analytical definition of the spin displacements in the three spatial directions, based on their initial position and the specific point in time. Regarding simulation principles, the solution of Equation (2.38) for a single spin is independent of the state of the other spins in the system, a key feature that enables parallelization [4], and that KomaMRI takes special advantage of.

```julia
@with_kw mutable struct Phantom{T<:Real}
    name::String = "spins"
    x   ::AbstractVector{T}
    y   ::AbstractVector{T} = zeros(size(x))
    z   ::AbstractVector{T} = zeros(size(x))
    ρ   ::AbstractVector{T} = ones(size(x))
    T1  ::AbstractVector{T} = ones(size(x)) * 1_000_000
    T2  ::AbstractVector{T} = ones(size(x)) * 1_000_000
    T2s ::AbstractVector{T} = ones(size(x)) * 1_000_000
    #Off-resonance related
    Δw  ::AbstractVector{T} = zeros(size(x))
    #Diffusion
    Dλ1 ::AbstractVector{T} = zeros(size(x))
    Dλ2 ::AbstractVector{T} = zeros(size(x))
    Dθ  ::AbstractVector{T} = zeros(size(x))
    #Motion
    ux  ::Function = (x,y,z,t)->0
    uy  ::Function = (x,y,z,t)->0
    uz  ::Function = (x,y,z,t)->0
end
```

Code 2.1: KomaMRI `Phantom` structure definition.

Operating principles The approach of KomaMRI to solve the Bloch Equation is to use a first-order operator splitting method, similar to CMRsim. However, in this case, the Bloch Equation is split not into three, but into a two-step process[14], rotation and relaxation, for each time step $\Delta t = t_{n+1} - t_n$[15] (Figure 2.32) [4]. Furthermore, to simplify calculations when possible, the pulse sequence is divided into two regimes: excitation and precession. The former is used when RF pulses are active, while the latter applies when no pulses are present.

The operation for each process (rotation and relaxation) and regime (excitation and precession) is specified as follows, with $M_{xy} = M_x + iM_y$ and $B_1 = B_{1x} + \mathrm{i}B_{1y}$.

1. The rotation[16] is described by [4]:

$$\frac{d\boldsymbol{M}^{(1)}}{dt} = \begin{bmatrix} 0 & \gamma B_z & -\gamma B_y \\ -\gamma B_z & 0 & \gamma B_x \\ \gamma B_y & -\gamma B_x & 0 \end{bmatrix} \boldsymbol{M}^{(1)} \tag{2.55}$$

   with initial condition $\boldsymbol{M}^{(1)}(t_n) = \boldsymbol{M}(t_n)$. This matrix is solved differently depending on the regime.

[14]Focusing again on Equation (2.39), in KomaMRI the elements corresponding to the RF rotation $R(t, \Delta t, \mathbf{r}_n(t))$ and precession $\Phi_n(t, \Delta t, \mathbf{r}_n(t))$ are combined under a single rotation operator.

[15]Note that Δt is not constant in KomaMRI. Different time step constraints will limit the time step to advance the Bloch equations depending on the events in the sequence (see Figure 2.33).

[16]As previously stated, the two rotation operators in CMRsim were around $\boldsymbol{B}_{\mathbf{I}} = (B_x, B_y, 0)$ and $\boldsymbol{B}_{\mathbf{II}} = (0, 0, B_z)$, respectively. In KomaMRI, the rotation is directly calculated around $\boldsymbol{B} = (B_x, B_y, B_z)$, thereby reducing the numerical error.

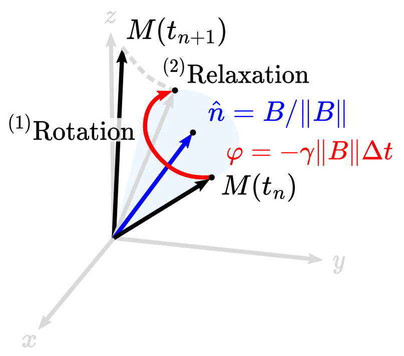


Figure 2.32: Solution of the Bloch equations for one time step can be described by (1) a rotation and (2) a relaxation step [4].

- The rotations during the excitation regime are also efficiently represented as spinors, as occurs in CMRsim. These spinors, denoted as α and β, are arranged in a rotation matrix $\boldsymbol{Q}$ [4], which is presented again here for convenience, although it is already included in Appendix A.3:

$$\boldsymbol{Q} = \begin{bmatrix} \alpha & -\beta^* \\ \beta & \alpha^* \end{bmatrix}, \quad \text{with } |\alpha|^2 + |\beta|^2 = 1 \tag{2.56}$$

Spinors can represent any three-dimensional rotation as [4]:

$$\alpha = \cos\left(\frac{\varphi}{2}\right) - \mathrm{i}\, n_z \sin\left(\frac{\varphi}{2}\right), \qquad \beta = -\mathrm{i}\, n_{xy} \sin\left(\frac{\varphi}{2}\right) \tag{2.57}$$

The rotation axis is defined by the effective magnetic field $\boldsymbol{B}$:

$$n_{xy} = B_1/||\boldsymbol{B}||, \quad n_z = B_z/||\boldsymbol{B}|| \tag{2.58}$$

and

$$\varphi = -\gamma||\boldsymbol{B}||\Delta t \tag{2.59}$$

is the phase accumulated due to $\boldsymbol{B}$. Then, the application of a spinor rotation to a magnetization element is described by the operation [4]

$$\begin{bmatrix} M_{xy}^+ \\ M_z^+ \end{bmatrix} = \boldsymbol{Q} \begin{bmatrix} M_{xy} \\ M_z \end{bmatrix} = \begin{bmatrix} 2\alpha^*\beta M_z + \alpha^{*2} M_{xy} - \beta^2 M_{xy}^* \\ (|\alpha|^2 - |\beta|^2) M_z - 2\,\mathrm{Re}(\alpha\beta M_{xy}^*) \end{bmatrix} \tag{2.60}$$

- For the precession regime, excitation fields are null ($B_x = B_y = 0$), so all the rotations are with respect to z. Therefore, they can be simply described with a complex exponential applied to the transverse magnetization [4]:

$$M_{xy}^+ = M_{xy}\mathrm{e}^{\mathrm{i}\varphi} \tag{2.61}$$

2. As for the relaxation step, it is described by[17] [4]:

$$\frac{d\boldsymbol{M}^{(2)}}{dt} = \begin{bmatrix} -\frac{1}{T_2} & 0 & 0 \\ 0 & -\frac{1}{T_2} & 0 \\ 0 & 0 & -\frac{1}{T_1} \end{bmatrix} \boldsymbol{M}^{(2)} + \begin{bmatrix} 0 \\ 0 \\ \frac{M_0}{T_1} \end{bmatrix} \tag{2.62}$$

with $\boldsymbol{M}^{(2)}(t_n) = \boldsymbol{M}^{(1)}(t_{n+1})$. Rewriting it by separating the transverse and longitudinal components of the magnetization, the typical relaxation equations —(2.14) and (2.15)— appear. The magnetization is thereby updated by:

$$\begin{bmatrix} M_{xy}^+ \\ M_z^+ \end{bmatrix} = \begin{bmatrix} \mathrm{e}^{-\Delta t/T_2} M_{xy} \\ \mathrm{e}^{-\Delta t/T_1} M_z + M_0\left(1 - \mathrm{e}^{-\Delta t/T_1}\right) \end{bmatrix} \tag{2.63}$$

Given all of this, the magnetization at the end of the time step is $\boldsymbol{M}(t_{n+1}) = \boldsymbol{M}^{(2)}(t_{n+1})$ [4].

[17]Note the equivalence with Equation (2.51) from CRMsim.

Performance and accuracy optimization The mechanisms used to optimize both accuracy and performance[18] are explained in detail in [4], and can be summarized in Figure 2.33. The sequence and the phantom can be divided into parts, thus taking advantage of both CPU multithreading and GPU computing features.

Motion The lowest-level functions, where the equations previously mentioned are implemented, are called `run_spin_excitation!` and `run_spin_precession!`, corresponding to each of the two possible regimes. These functions obtain, prior to the effective field and magnetization calculations, the positions of each spin for the specified time points. For this, the fields `ux`, `uy`, and `uz` of the phantom are used, as shown in Code 2.2. The resulting matrices, `xt`, `yt`, and `zt`, will have a number of rows equal to the number of spins and a number of columns corresponding to the time instants in which positions are evaluated.

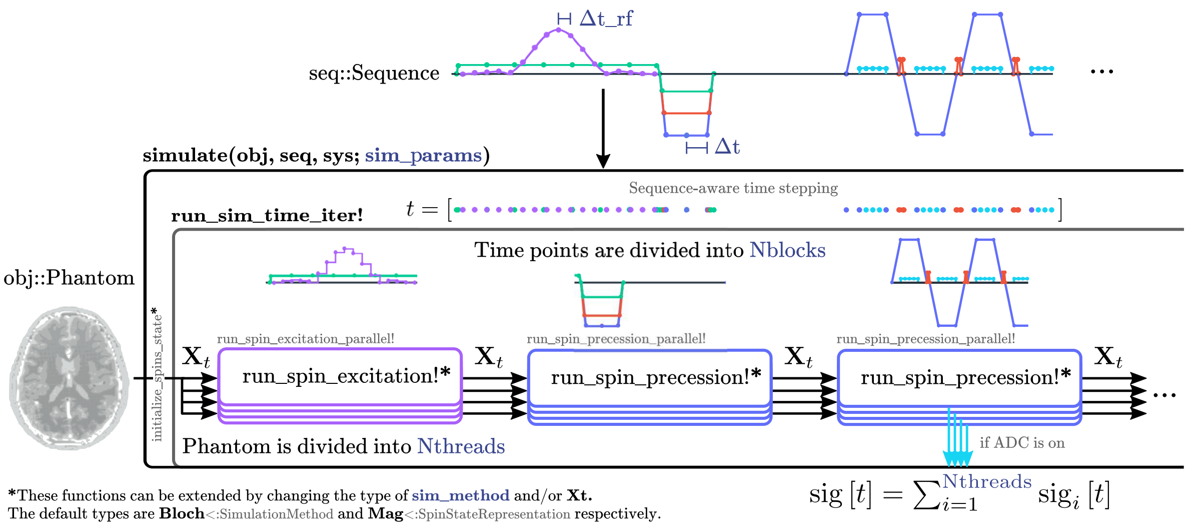

Δt_rf
seq::Sequence
Δt
simulate(obj, seq, sys; sim_params)
Sequence-aware time stepping
run_sim_time_iter!
$t = [\ldots]$
Time points are divided into Nblocks
obj::Phantom
initialize_spins_state*
run_spin_excitation_parallel!
run_spin_precession_parallel!
run_spin_precession_parallel!
$\mathbf{X}_t$
run_spin_excitation!*
run_spin_precession!*
run_spin_precession!*
Phantom is divided into Nthreads
if ADC is on
$\mathrm{sig}\,[t] = \sum_{i=1}^{\mathrm{Nthreads}} \mathrm{sig}_i\,[t]$
*These functions can be extended by changing the type of sim_method and/or Xt.
The default types are Bloch<:SimulationMethod and Mag<:SpinStateRepresentation respectively.

Figure 2.33: Diagram of the functions called to perform the simulation. The sequence is discretized and its time points are then divided into `Nblocks` to reduce the amount of memory used. The phantom is divided into `Nthreads`. If a readout is carried out, the simulator will add the signal contributions of each thread to construct the acquired signal `sig[t]` [4].

Advantages and limitations The advantages and strengths of this simulator have already been detailed throughout this section. As for areas of improvement, the developers of this tool are well aware of these limitations and are either currently implementing solutions or have them in mind for the future. To be more specific, KomaMRI currently assumes constant magnetic properties over time, limiting the simulation of dynamic contrast-enhanced imaging techniques [4]. Additionally, it lacks support for multiple coil[19] simulations and does not account for T_2^* decay[20]. Regarding motion, the use of analytical functions to define it makes it unnecessarily difficult to model simple movements and restricts the definition of more complex motion patterns. This last limitation is, as stated in Chapter 1, the primary focus of this work.

[18] In addition to what is discussed in [4] regarding performance, the latest advancements have further improved simulation times and memory allocations, which are detailed in https://juliahealth.org/KomaMRI.jl/stable/explanation/4-gpu-explanation/. The methods used focus on optimizing the functions `run_spin_excitation!` and `run_spin_precession!` for both CPU and GPU. The CPU implementation prioritizes memory conservation, and makes extensive use of pre-allocation for the simulation arrays. The GPU implementation, among other optimizations, employs kernel-based methods that are highly optimized.

[19] https://github.com/JuliaHealth/KomaMRI.jl/issues/150.

[20] https://github.com/JuliaHealth/KomaMRI.jl/issues/62.

```
function run_spin_excitation!(p::Phantom{T}, seq::DiscreteSequence{T},
    sig::AbstractArray{Complex{T}}, M::Mag{T}, sim_method::SimulationMethod
    ) where {T<:Real}
    #Simulation
    for s ∈ seq #This iterates over seq, "s = seq[i,:]"
        #Motion
        xt = p.x .+ p.ux(p.x, p.y, p.z, s.t)
        yt = p.y .+ p.uy(p.x, p.y, p.z, s.t)
        zt = p.z .+ p.uz(p.x, p.y, p.z, s.t)
        (...)
    end
    (...)
end

function run_spin_precession!(p::Phantom{T}, seq::DiscreteSequence{T},
    sig::AbstractArray{Complex{T}}, M::Mag{T}, sim_method::SimulationMethod
    ) where {T<:Real}
    #Simulation
    #Motion
    xt = p.x .+ p.ux(p.x, p.y, p.z, seq.t')
    yt = p.y .+ p.uy(p.x, p.y, p.z, seq.t')
    zt = p.z .+ p.uz(p.x, p.y, p.z, seq.t')
    (...)
end
```

Code 2.2: Spins positions acquisition in KomaMRI.

2.3.4 Numerical approach by T. Puiseux

What is presented in [37] is not a full MRI simulation framework like the previous three[21], but rather a numerical approach designed to efficiently simulate time-resolved 3D phase-contrast MRI (or 4D Flow MRI) acquisitions under realistic flow conditions. To address the problems presented at the end of Section 2.2.4, a method is proposed to perform CFD/MRI simulations "on the fly", i.e., advancing the particle positions and computing the resulting NMR signal during the calculation, without storing the particle trajectory history [37].

The CFD-MRI simulation procedure of this approach stands out due to aspects such as its time discretization (see Figure 2.34), which is aware both of the MR sequence and the CFD time steps, its particle seeding method or its spoiling modelling [37]. Moreover, and more with regard to this work, the fluid velocity —i.e., the CFD— is dynamically calculated for each time step, and then interpolated to the particles. It should be noted, as also shown in Figure 2.34, that the time steps for CFD differ from the time steps of the sequence, being the latter considerably shorter than the former. Within each CFD iteration of time Δ_{CFD}, the fluid velocity is kept constant, so this CFD calculation is not needed for every numerical advancement of the Bloch equations.

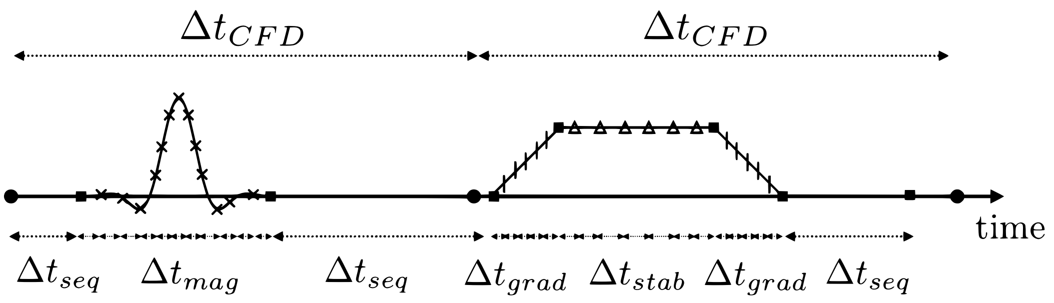


Figure 2.34: Evolution of the simulation time step over an arbitrary pulse sequence (RF and gradient) [37].

[21]Note that, although it is the latest one presented in this document, [37] was published before [4] and [8].

With respect to the solution of the Bloch equations, a semi-analytic solution was implemented. The Bloch equations are solved numerically using a fourth-order Runge-Kutta scheme (RK4) during the RF excitations and analytically whenever the particles experience relaxation or encoding gradient events. A detailed explanation is provided in [37].

This approach is of particular relevance and is included in this document as it was one of the first to implement this hybrid method for solving the Bloch equations[22]. More importantly, it is the only one to have tested such complex 4D flow MRI simulations. However, the simulation times for these experiments were considerably long, around 20 hours for 1.1 million particles.

With regard to the accuracy of the results, [37] presents a study on the influence of the spin density and the spatial resolution, showing that accuracy improves with higher spin density up to a plateau around 40 particles/voxel, and higher spatial resolution —smaller voxel sizes— also leads to an improvement, though to a lesser degree.

2.4 Programming Technologies: Julia, Qt, VTK, and Web

The following section represents a selection of tools and programming technologies that are of particular interest in the context of this work.

2.4.1 Julia

Julia [15] is a high-level, dynamic programming language developed by MIT, used for computations in various fields such as physics, biology, engineering, mathematics and finance. It is designed to be both extensible and composable, featuring a flexible package ecosystem. One of its key advantages is its capability to efficiently solve complex numerical problems involving large datasets, something that other languages in its category struggle with unless supplemented by compiled code written in a secondary language, such as C or C++.

a) Julia features

The Julia language has distinguishing features, which aim to achieve a balance between user needs and computational efficiency [15]. Notable among them are:

- **Optional data type definitions**: in statically-typed languages like C or Fortran, all data types must be explicitly defined to enable static checks during compilation, resulting in excellent performance. In contrast, dynamic languages often omit type definitions, increasing programmer productivity but leading to reduced performance, as the compiler lacks type information essential for generating optimized code [15]. Julia type system allows for the optional specification of types, giving programmers the flexibility to either omit type definitions or include them to enhance performance when needed.
- **Multiple Dispatch**: this feature enables selecting the appropriate function implementation based on the data types of all its input arguments [15]. It allows for the definition of multiple functions with the same name that process data differently according to its arguments type. Many of built-in Julia functions leverage this capability, providing efficient and intuitive type-specific behaviour. For instance, the operator $*$ multiplies when applied to numeric arguments, but performs concatenation when used with strings.
- **Meta-programming**: in Julia, meta-programming refers to the ability of a program to manipulate its own code during execution. This is achieved by writing Julia code that processes and modifies other Julia code before it is executed. The process involves two main stages: first,

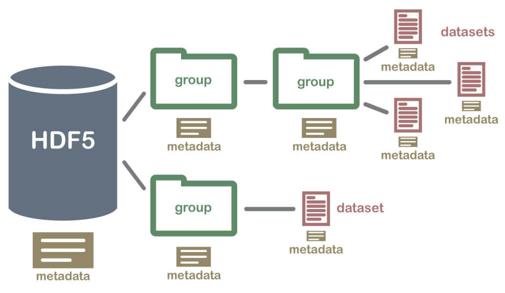
[22]Another simulator that proposed different solutions depending on the RF activation was MRISIMUL [9].

the raw code is parsed into an abstract syntax tree (AST), and then it is executed. Metaprogramming allows access to the AST, enabling code transformations. This facilitates the generation of more complex expressions or the optimization of code before execution, all while maintaining the performance of native Julia code.

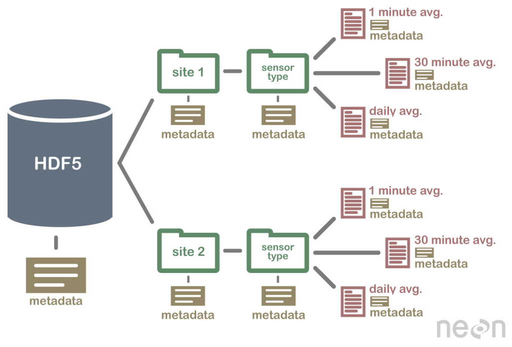
b) Modules and Packages

In Julia, code is organized into modules and packages. Among all the packages listed in the Julia registry, two have already been referenced in this document [4, 51]. Additionally, it is worth mentioning a set packages of particular relevance to this project:

- **Interpolations.jl** [52] implements a variety of interpolation schemes for the Julia Language. It has the goals of ease-of-use, broad algorithmic support, and exceptional performance, supporting interpolant usage on GPU via broadcasting.
- **KernelAbstractions.jl** [53] is a package that allows writing GPU-like kernels targetting different execution backends. It intends to be a minimal and performant library that explores ways to write heterogeneous code.
- **Functors.jl** [54] and **Adapt.jl** [55], when used together, enable seamless conversion of the data types within the internal elements of complex and nested data structures, while preserving the external wrappers and their types. This functionality is particularly useful for adjusting data precision within Julia structures or converting data into GPU-compatible formats.
- **HDF5.jl** [56] is an HDF5 interface for the Julia language. HDF5 stands for Hierarchical Data Format v5, and is a file format and library for storing and accessing data, commonly used for scientific data. In HDF5, a “group” is analogous to a directory, a “dataset” is like a file. This format also uses “attributes” to associate metadata with a particular group or dataset. Figure 2.35 illustrates this file structure. Furthermore, HDF5 is specifically designed for efficiently storing large amounts of heterogeneous data, ensuring fast and straightforward read and write operations.



(a)



(b)

Figure 2.35: (a) Diagram of the HDF5 structure. (b) Practical example of use [57].

- **PlotlyJS.jl** [58] is a Julia interface to the Plotly.js visualization library, which enables the creation and visualization of interactive 2D and 3D plots, along with statistical graphs, providing robust infrastructure for displaying these plots across various frontends and exporting them for external use.
- **Oxygen.jl** [59] is a micro-framework built on top of the HTTP.jl library that is designed to simplify the process of building web applications in Julia. It is heavily inspired by the FastAPI [60] Python library. It provides key features for creating and managing Application Programming Interfaces (APIs), such as straightforward routing, JSON serialization/deserialization, support

for HTTP methods, and middleware chaining, which are essential for handling API requests and responses. Moreover, it supports both static and dynamic file hosting, allowing serving static assets such as images, CSS and JavaScript files, as well as dynamic content generated by the API, such as JSON responses or templated pages.

2.4.2 Qt

Qt [61] is a framework oriented towards the development of cross-platform GUIs. It is not a programming language in itself but rather a set of classes, methods, functions, and libraries written in C++. A preprocessor, the Meta-Object Compiler (MOC), is used to extend the C++ language. Before the compilation step, the MOC analyzes the source files written in C++ with Qt extensions and generates standard C++ source files from them.

Qt is divided into modules that group various functionalities and tools according to the required purpose. These modules allow developers to include only the parts of the framework that are relevant to their application.

a) Signals and Slots

In user interface programming, when the state of an object changes, it is often necessary to notify another object. Ideally, objects of any type should be able to communicate seamlessly. For instance, when a user clicks a close button, the window's `close()` function should be triggered automatically [62].

While other technologies rely on callbacks, Qt uses a more flexible mechanism called signals and slots. A signal is emitted when a specific event occurs, and any `QObject` can have predefined or custom signals. Slots are functions executed in response to signals, allowing structured and efficient event handling [62].

b) The Qt Quick module and QML

The Qt Quick module is designed for creating smooth and dynamic cross-platform user interfaces. It is the standard library for writing applications in QML.

Qt Meta Language (QML) is a multi-paradigm language used for developing applications with a high degree of dynamism. In QML, each of the application building blocks must be declared along with their properties. Its syntax initially resembles a mix of HTML and CSS, as its structure is quite similar to a markup language. QML includes an event handling system that uses JavaScript, which can be used to control the entire behaviour and logic of the user interface. Note the distinction between the program core logic, which resides in C++, and the interface logic, which handles user interaction events and is managed with JavaScript.

c) Compilation of applications based on the Qt framework

Among the build systems supported by Qt, qmake and CMake stand out.

- **qmake**, created by the Qt company itself, provides a project-oriented build system to manage the process of building applications, libraries, and other components. With qmake, projects are described using a project file (`.pro`), which is then used to generate standard files such as the Makefile. The project files typically contain a list of source and header files, general configuration information, and any application-specific details, such as a list of libraries to link against or additional include paths.

  This system has advantages, such as its high compatibility with Qt and its simple syntax. However, its limited documentation and, more importantly, its exclusivity to Qt, make qmake less suitable for large projects that combine multiple technologies.

- **CMake** [63] is a cross-platform code generation and automation tool designed to be used with native build environments, such as GCC or MinGW. CMake uses simple configuration files, named CMakeLists.txt, to generate standard files (e.g., Makefile in Unix) that are used by the compiler.

  CMake offers advantages such as a greater availability of information and documentation, and a better compatibility with other languages and technologies. Nevertheless, it also has some drawbacks, such as its syntax complexity, its detailed control over the build process or its error handling system, which make it more difficult to learn.

2.4.3 VTK

The Visualization Toolkit (VTK) [64] is an open-source, freely available software system for 3D computer graphics, modeling, image processing, volume rendering, scientific visualization, and 2D plotting. It supports a wide variety of visualization algorithms and advanced modeling techniques, and it takes advantage of both threaded and distributed memory parallel processing for speed and scalability, respectively. VTK is designed to be platform agnostic. This means that it runs just about anywhere, including on Linux, Windows, and Mac, Web and mobile devices. The core functionality of VTK is written in C++ to maximize efficiency. This functionality is wrapped into other language bindings to expose it to a wider audience.

Building upon VTK's powerful foundation, VTK-m [65] extends these capabilities to better support emerging processor architectures, such as multi-threaded CPUs, GPUs, and high-performance computing (HPC) systems. VTK-m specifically focuses on fine-grained concurrency and parallelism, which is essential for efficiently processing and visualizing data at extreme scales. By providing abstract models for both data representation and execution, VTK-m enables the same algorithms to be optimized across diverse hardware architectures, enhancing performance without requiring extensive code modifications.

2.4.4 Web Technologies

a) HTML

HTML (HyperText Markup Language) is the most basic building block of the Web and defines the meaning and structure of web content. "Hypertext" refers to links that connect web pages to one another, either within a single website or between websites. It uses "markup" to annotate text, images, and other content for display in a Web browser [66].

This language provides a wide range of structural elements, which are set off from other text in a document by "tags", consisting of the element name surrounded by "<" and ">". The name of an element inside a tag is case-insensitive. That is, it can be written in uppercase, lowercase, or a mixture. [66]. HTML elements include multiple levels of headings (`<h1>` to `<h6>`), paragraphs (`<p>`), and various types of lists, such as ordered (`<ol>`) and unordered lists (`<ul>`). It also supports embedding images (`<img>`) and videos (`<video>`), as well as linking content through hyperlinks (`<a>`).

Other technologies besides HTML are generally used to describe a web page's appearance/presentation (CSS) or behaviour (JavaScript) [66].

b) JavaScript

JavaScript (JS) is a lightweight interpreted programming language with first-class functions[23]. While it is most well-known as the scripting language for Web pages, many non-browser environ-

[23] A programming language is said to have First-class functions when functions in that language are treated like any other variable. For example, in such a language, a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable [66].

ments also use it, such as Node.js, Apache CouchDB and Adobe Acrobat. JavaScript is a prototype-based[24], multi-paradigm, single-threaded, dynamic language, supporting object-oriented, imperative, and declarative styles [66].

JavaScript's dynamic capabilities include runtime object construction, variable parameter lists, function variables, dynamic script creation (via `eval`), object introspection (via `for...in` and `Object` utilities), and source-code recovery (JavaScript functions store their source text and can be retrieved through `toString()`).

Node.js Node.js [67] is an open-source and cross-platform JavaScript runtime environment that allows developers to build server-side and network applications with JavaScript [66]. It runs on the V8 JavaScript engine, and executes JavaScript code outside a web browser.

Node.js enables developers to use JavaScript not only for client-side scripting, but also for server-side programming. This unification of client and server code under one language is called the "JavaScript anywhere" paradigm. By using JavaScript on both ends, Node.js helps streamline the development process, and improves efficiency. The main benefit is that it allows for dynamic web page content generation on the server before sending the page to the user's browser. This is particularly useful in modern web applications where real-time updates and interactive features are necessary [66].

The code in Node.js is structured in modules and packages, allowing for the modularity and scalability of the applications. To handle all existing packages in Node.js, the primary tool is **npm** (Node Package Manager), a package manager that is downloaded and bundled alongside Node.js[25]. Its command-line client `npm` can be used to download, configure and create packages for use in Node.js projects. Packages hosted on npm are downloaded from its public registry[26] [66].

VTK.js VTK.js [68] is not a wrapper, but a complete rewrite of the VTK framework using plain JavaScript, allowing for interactive 3D visualizations directly in the browser. It can be easily installed as a JavaScript package via npm. VTK.js, just like the original VTK, is particularly effective for visualizing volumetric medical data, like MRI acquisitions, allowing for a detailed exploration of anatomical structures from various perspectives. Additionally, it supports interactive cutting planes, enabling users to slice through the volume dynamically.

c) WebAssembly

WebAssembly [69] is an open standard developed by the World Wide Web Consortium (W3C), defining a portable binary code format for programs which are executed inside a Web browser. It also defines its corresponding textual assembly language, and interfaces to facilitate interactions between such programs and their hosting environment. WebAssembly code runs in a low level virtual machine, which mimics the functionality of the many microprocessors on which it can be executed. This standard is gaining interest due to the desire to run more computationally intensive code in browsers. Users are increasingly spending more time in browsers and less time using local applications. This trend has created an urgency to remove barriers to running a wide range of programs directly within the browser, thus motivating WebAssembly conception. It is possible to convert a variety of programming languages to WebAssembly. However, for each case, a different tool or compiler is required. Examples of languages that support compilation to WebAssembly include C/C++, Rust, C#, and Go.

[24] Prototype-based programming is a style of object-oriented programming in which classes are not explicitly defined, but rather derived by adding properties and methods to an instance of another class or, less frequently, adding them to an empty object [66].

[25] Although it is included in Node.js, npm can be used not only when developing server-side applications but also when developing JavaScript for the browser.

[26] NPM registry available at https://registry.npmjs.org/.

Compilation of a C/C++ module to WebAssembly When writing C++ code for execution in a web browser, the following tools are required:

- Web browser with WebAssembly support, such as Firefox, Chrome, Safari or Edge.
- C/C++ compiler for WebAssembly: the one most commonly used is Emscripten [70].
- C/C++ compiler, such as GCC on Linux or MinGW on Windows.
- Simple local web server, to host and serve the files generated by the compilation process.

Using these tools, WebAssembly compilation can be achieved with commands like `emcc`, which generates both a `.wasm` file and an `app.js` file that loads the `.wasm` and acts as a bridge between WebAssembly and the browser. Additionally, an `index.html` file is needed to load the JavaScript file, allowing access through the browser.

d) REST APIs

A REST API is an Application Programming Interface (API) that follows the design principles of the Representational State Transfer (REST) architectural style. REST defines a set of rules and guidelines about how web APIs should be built [71].

API An API is a set of definitions and protocols for building and integrating application software. It acts as a mediator between the users or clients and the resources or web services they want to get. It allows web developers to share resources and information while maintaining security, control, and authentication [71].

REST REST is a set of architectural constraints, not a protocol or a standard. API developers can implement REST in various ways. When a client makes a request via a RESTful API, it transfers a representation of the resource state to the requester or endpoint. This information is delivered via HTTP in formats such as JSON, HTML, XML, or plain text, with JSON being the most widely used due to its readability for both humans and machines [71].

The most important field in the header of HTTP requests sent by clients is the Uniform Resource Identifier (URI), which identifies the resource on the server to which an HTTP method is applied. These methods include GET (retrieve information), POST (send new data), PUT (update an existing resource), and DELETE (remove a resource). Figure 2.36 shows the operation diagram of a REST API.

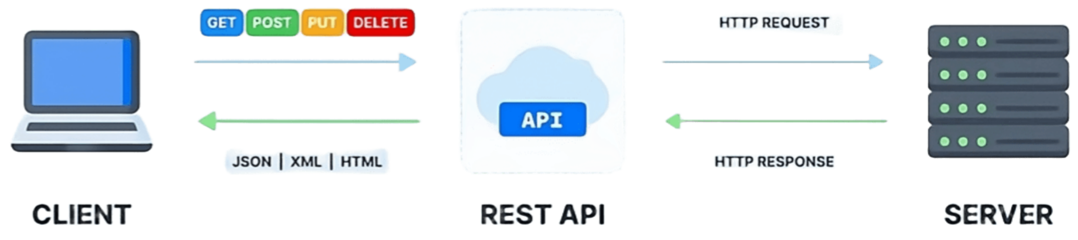


Figure 2.36: Operation diagram of a REST API.

Chapter 3

Methods: Analysis, Design and Implementation

This Chapter outlines the methods used in the development of the two main projects that form the basis of this work. Therefore, it is divided into two sections. The first focuses on the development of a dynamic phantom model for KomaMRI. Regarding the second, it is dedicated to the design and implementation of a web MRI sequence editor.

Each section starts with an analysis of the requirements, which can be divided into two categories, namely, functional and non-functional requirements. Functional Requirements (FRQ) outline the features, capabilities, and behaviour of the system, specifying what the system should do and directly impacting its functionality for users. In contrast, Non-Functional Requirements (NFR) define the system quality attributes, performance and operational constraints. They emphasize the effectiveness and efficiency in executing its functions, thereby they have a great influence on overall user satisfaction and experience.

Additionally, the chosen technologies are presented for each section, along with the corresponding design and implementations details.

3.1 Dynamic Phantom Model

The implementation of a dynamic phantom model is directly aligned with the first main objective presented in Section 1.2 (page 2). In order to achieve it, an extension of KomaMRI is proposed. This simulator has been chosen as the starting point of this work for the reasons exposed in Section 2.3, which place it ahead of other existing MRI simulation tools. These reasons can be summarized as its extensibility, ease of use and, above all, its performance, that make it a highly versatile and promising tool.

Specifically, we propose an extension of the phantom structure and the functions responsible for handling its motion in simulations. Additionally, an enhancement to the phantom visualization tool in KomaMRI is proposed, allowing for the visualization of its temporal evolution, and thus its motion. This section also outlines the methods for developing a specific file format to store both static or dynamic phantoms, along with the functions implemented in the simulator to read and write such files. All modifications, including the generation of documentation, will be implemented using the Julia programming language, by means of the packages described in Section 2.4.1b. Furthermore, the complete code described in this section and all its previous versions are available on the KomaMRI GitHub repository[1].

[1]KomaMRI GitHub repository: https://github.com/JuliaHealth/KomaMRI.jl.

3.1.1 Requirements Analysis

a) Functional Requirements (FRQ)

- **FRQ-1-01**
  The definition of simple rigid motions, such as translation and rotation, must be straightforward and expressible with only a few parameters, specifically the axis along which the motion takes place and the magnitude of the applied motion. Similarly, the definition of some non-rigid motions, such as cardiac contraction and expansion, must be simple and equally parametrizable.
- **FRQ-1-02**
  It must be possible to incorporate any type of non-rigid motion into the phantom, regardless of its complexity. This includes motions that are not necessarily parametrizable, and which may be imported from external sources, such as XCAT.
- **FRQ-1-03**
  It must be possible to endow the phantom with flow motions, which may have been imported from real data or obtained through Computational Fluid Dynamics (CFD).
- **FRQ-1-04**
  The time span of the motion must be controllable, either by defining an initial and final time point or by providing information about the period and temporal asymmetry factor in the case of periodic motions. Furthermore, if greater temporal precision is required, it should be possible to design an arbitrary time curve to accurately define the behaviour of the motion over time.
- **FRQ-1-05**
  The user must be able to select which regions of the anatomical model are affected by the motion, in the case that the entire model is not intended to be affected. This must be possible through the definition of a “spin span”, which will contain the spins of the model that are subjected to the motion.
- **FRQ-1-06**
  The dynamic phantom model must allow combining an arbitrary number of motions, each with potentially different nature, duration, and spin span.
- **FRQ-1-07**
  The defined phantom, along with its included motions, must be viewable through a 4D (3D + time) visualization tool. The temporal evolution of the phantom must be controllable through a horizontal slider.
- **FRQ-1-08**
  All phantom data must be storable in files with a specific format, enabling phantom sharing and ensuring the reproducibility of experiments.
- **FRQ-1-09**
  A set of motion-related MRI sequences, such as Phase Contrast (PC), Time of Flight (TOF), and cardiac cine, must be available in the simulator for its use over the created dynamic phantoms.

b) Non-Functional Requirements (NFR)

- **NFR-1-01**
  The simulation software must maintain consistent functionality, regardless of the type of motion included. Both with and without motion, the calculation of the anatomical model positions will be performed separately and independently before solving the Bloch equations.
- **NFR-1-02**
  The logic and computations used to resolve particle positions within the phantom must be entirely transparent to the user. This ensures that users can define, adjust, and simulate motions without requiring in-depth knowledge of the underlying computational processes.

- **NFR-1-03**
  Simulations without any motion must retain the same speed and memory allocation efficiency as prior versions. The inclusion of motion capabilities must not degrade the performance of static simulations.
- **NFR-1-04**
  While simulation time and memory allocation may increase when dynamic models are simulated, this increase must be kept as minimal and controlled as possible, thus maintaining better performance than current state-of-the-art simulators. To achieve this, the functions responsible for resolving the positions of the phantom must be optimized and seamlessly integrated with the existing simulator functions, ensuring efficient use of resources and minimal impact on overall performance.
- **NFR-1-05**
  The simulation system must support fast execution on both CPUs and GPUs, ensuring compatibility with a wide range of hardware configurations. Optimizations should leverage the computational power of GPUs without sacrificing compatibility for CPU-only environments.
- **NFR-1-06**
  While using less powerful hardware may slow down simulations, it must never prevent them from running. The system should adapt to varying hardware capabilities to ensure that simulations remain accessible, even on lower-end machines.

3.1.2 Design considerations illustrated through key examples

This section presents several use cases and examples which align with the previously presented functional requirements and have been borne in mind in the design of the phantom model. These use cases have been identified by adopting the perspective of a potential user and considering possible motion-related scenarios that might be significant and valuable to simulate. Each of these scenarios presents a unique aspect that justifies certain design decisions. The following use cases are presented:

- **Multiple simultaneous or sequential motions**, which can be useful for simulating real-world scenarios such as:
  - A brain of a patient moving inside the scanner, which would generate, for example, simultaneous translation and rotation motions. Alternatively, these two motions could be defined sequentially in order to simulate the patient first moving their head to one side and then rotating it.
  - Myocardial motion, where contraction is combined with rotation or torsion. Additionally, a periodic rigid translation motion could be incorporated to simulate the breathing of the patient. All these motions should occur simultaneously.

  These scenarios bring the need to define the phantom motion as a collection of simpler, elemental movements that can be configured independently. It must be possible to define the time intervals during which these movements occur, enabling them to take place either simultaneous or sequentially.
- **Periodic and pseudo-periodic motions**, which can account for some common motions in the body, such as the previously mentioned myocardial motion or breathing-induced displacements. In both cases, the motion pattern may not be perfectly periodic. For instance, cardiac motion can exhibit variations such as changes in heart rate or arrhythmias. Similarly, respiratory motion may be irregular due to factors such as the patient's physiological or emotional state (e.g., anxiety) or involuntary variations that result in differing durations for inhalation and exhalation over time.

  The need for this case is the capacity to design the displacement curve of the phantom for one period, and extend it over time for an arbitrary number of repetitions. Furthermore, each repetition must be compressible or expandable in time to account for the arrhythmic patterns previously mentioned.

- **Flow and diffusion motions**, which can present significant challenges in terms of modelling, parametrization, and storage, especially when the anatomical model is complex. Examples of scenarios which involve such motions include simulating blood flow through the aorta or the diffusion phenomenon of water molecules within the brain.

  For flow motion, a convenient approach would be to store the motion information as a set of independent trajectories, one for each spin, which would avoid the need of a closed-form analytical expression to define these motions. Additionally, this would allow for the definition of any arbitrary motion, regardless of its complexity. Finally, this method can also be useful for incorporating diffusion-related motion when, for instance, sampling trajectories of the diffusion propagator are available.

An important consideration is that, regardless of the specific motion model, there may be a need to apply motion to only a specific part of the phantom, while keeping the rest of the spins static.

Taking all of the above into account, some additional considerations for the design of the dynamic phantom model are:

- If the phantom includes more than one motion, they will be stored in a list. Each motion in it will be an independent instance and will contain information about its type, the spatial directions in which it occurs, the spins it affects, and its temporal behaviour. The latter will be defined by a time curve, which will arbitrarily specify how the motion progresses in time, from its initial to its final state.
- Time curves will contain both the information of absolute time instants as well as information on how the samples of the trajectory will be read at each time instant. This allows the user, by defining only a forward trajectory, to create forward and backward motion as well as periodicity and pseudo-periodicity.
- To meet the need for both simple, parametrized motion definitions (such as translation and rotation) and the inclusion of arbitrarily complex movements, the model will include two types of motions, referred to as actions: *simple* and *arbitrary*. Simple actions will be defined by means of parameters —with, generally speaking, an explicit dependence with time— while arbitrary actions will be described through independent trajectories for each spin. During simulation, the displacements calculations for each type of action will be slightly different from each other but, in all cases, they will remain transparent to the end user.

3.1.3 Phantom class extension and simulation functions

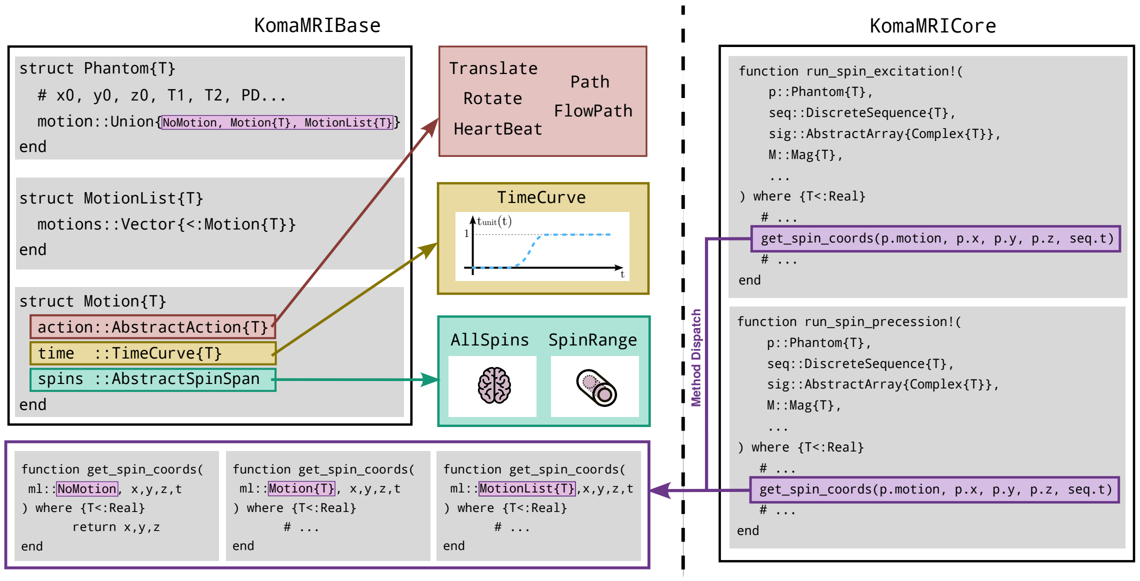
To accomplish requirements **FRQ-1-01** to **FRQ-1-06**, as well as all non-functional requirements, it is necessary to refer back to code 2.1, which shows the phantom structure definition of KomaMRI v0.8[2]. Since the phantom displacements are currently stored in the fields `ux`, `uy`, and `uz` as functions, and considering the disadvantages associated with this approach (see Section 2.3.3), it will be necessary to replace them with a `motion` field that encapsulates the motion information in a simpler and more straightforward manner for the user. Figure 3.1 presents a diagram of this code change and the subsequent modifications in `KomaMRIBase` and `KomaMRICore`.

Specifically, the `motion` field is required to be one of the following three predefined data structures, designed for this purpose. Code 3.1 provides examples of how to use these three structures to define the motion of a phantom.

- `NoMotion` is the default type, assigned to static phantoms. Its corresponding structure contains no fields.
- `Motion`: this structure contains information about a basic motion, understood as the combination of an action, a time curve, and a spin span. These three fields will be described in more detail later, along with a description of all the possible values each can take.

[2]v0.8 is the latest KomaMRI version prior to the implementation of the changes described within this document.

- `MotionList` is a structure that contains a single field called `motions`, which is a vector of instances of type `Motion`. This approach allows for the assignment of as many motions as needed to the phantom, with each motion being independent in terms of type, duration, and spin span. This design makes it possible to define both sequential and simultaneous concatenations of motions over time, based on the specific time values of each motion added to the list. Similarly, the range of spins affected is, as mentioned, independent for each motion in the list.



```
struct Phantom{T}
  # x0, y0, z0, T1, T2, PD...
  motion::Union{NoMotion, Motion{T}, MotionList{T}}
end

struct MotionList{T}
  motions::Vector{<:Motion{T}}
end

struct Motion{T}
  action::AbstractAction{T}
  time   ::TimeCurve{T}
  spins  ::AbstractSpinSpan
end
```

```
function get_spin_coords(
 ml::NoMotion, x,y,z,t
) where {T<:Real}
       return x,y,z
end
```

```
function get_spin_coords(
 ml::Motion{T}, x,y,z,t
) where {T<:Real}
       # ...
end
```

```
function get_spin_coords(
 ml::MotionList{T},x,y,z,t
) where {T<:Real}
       # ...
end
```

```
function run_spin_excitation!(
    p::Phantom{T},
    seq::DiscreteSequence{T},
    sig::AbstractArray{Complex{T}},
    M::Mag{T},
    ...
) where {T<:Real}
   # ...
   get_spin_coords(p.motion, p.x, p.y, p.z, seq.t)
   # ...
end
```

```
function run_spin_precession!(
    p::Phantom{T},
    seq::DiscreteSequence{T},
    sig::AbstractArray{Complex{T}},
    M::Mag{T},
    ...
) where {T<:Real}
   # ...
   get_spin_coords(p.motion, p.x, p.y, p.z, seq.t)
   # ...
end
```

Figure 3.1: Diagram of the structures and functions in `KomaMRIBase` and `KomaMRICore` involved in the new dynamic phantom model. The three fields of the `Motion` structure are highlighted: `action`, `time`, and `spins`, each of which can be of different types, illustrated in the red, yellow, and green boxes, respectively. Additionally, the core functions call the `get_spin_coords` function, which has three different methods depending on whether its input argument is of type `NoMotion`, `Motion`, or `MotionList`.

```julia
# Load phantom example:
phantom = brain_phantom2D()

# 1. NoMotion. Static phantom:
phantom.motion = NoMotion()

# 2. Motion:
phantom.motion = Motion(Translate(0.0, 0.1, 0.2), TimeRange(0.0, 1.0), AllSpins())

# 3. MotionList:
phantom.motion = MotionList(
    Motion(Translate(0.0, 0.1, 0.2), TimeRange(0.0, 1.0), AllSpins()),
    Motion(Rotate(0.0, 0.0, 45.0), Periodic(1.0, 0.5), SpinRange(1:1000))
)
```

Code 3.1: Usage examples of the `NoMotion`, `Motion`, and `MotionList` structs.

As shown in Figure 3.1, each field of the `Motion` structure must be of a specific type. Specifically, the `time` field must be an instance of the composite type (i.e., structure) `TimeCurve`, while the `spins` and `action` fields must be of a type that inherits from the abstract types[3] `AbstractSpinSpan` and `AbstractAction`, respectively. Each of these fields, along with their corresponding types, will be described in the following sections.

[3]In Julia, an abstract type serves as a superclass in a type hierarchy, organizing and classifying concrete types without the need of being instantiable. It provides a common foundation for sharing methods and promoting flexibility and extensibility in programming.

Consequently, when running the simulation, it will no longer be necessary to call `ux`, `uy`, and `uz` from the `run_spin_excitation!` and `run_spin_precession!` functions, as it was shown in Code 2.2. Instead, a new function, `get_spin_coords`, has been defined to compute the spin positions based on the new `motion` field along with the initial positions of each spin and the corresponding time instants. These positions, as before, will be contained in three matrices, `xt`, `yt`, and `zt`, each with a number of rows equal to the number of spins and a number of columns corresponding to the time instants in which positions are evaluated.

Taking advantage of the method dispatch feature in Julia, we defined a different `get_spin_coords` method for each of the aforementioned structures. This implies that, depending on the input argument type, a specific method of the function will be invoked. Specifically, when the `motion` field is `NoMotion`, the function will simply return the initial positions of each spin. When it is `Motion`, it will compute the positions at each time instant by evaluating the `action`, `time`, and `spins` fields, which will be explained below. Finally, if the `motion` field type is `MotionList`, the displacements caused by each of the individual motions will be evaluated, and their contributions will be summed to obtain the final computed positions.

For simplicity, let us consider the `get_spin_coords` method for `Motion`[4] input arguments, illustrated in Code 3.2. In this implementation, the three aforementioned fields of the `Motion` structure are utilized through calls to functions that also employ method dispatch. Accompanying the code, Figure 3.2 offers a more detailed and clearer representation of how each of these functions is invoked.

```julia
function get_spin_coords(m::Motion{T}, x, y, z, t) where {T<:Real}
    ux, uy, uz = x .+ 0*t, y .+ 0*t, z .+ 0*t # buffers for displacements
    t_unit = unit_time(t, m.time)
    idx = get_indexing_range(m.spins)
    displacement_x!(@view(ux[idx, :]), m.action, x[idx], y[idx], z[idx], t_unit)
    displacement_y!(@view(uy[idx, :]), m.action, x[idx], y[idx], z[idx], t_unit)
    displacement_z!(@view(uz[idx, :]), m.action, x[idx], y[idx], z[idx], t_unit)
    return x .+ ux, y .+ uy, z .+ uz
end
```

Code 3.2: `get_spin_coords` method for `Motion` input arguments.

- `unit_time`: this function is responsible for interpolating and, therefore, giving a continuous behaviour to the `TimeCurve` structure. Although it will be detailed in Section 3.1.3a, this structure fundamentally consists of two fields in the form of arrays, `t` and `t_unit`, which represent discrete instances of absolute time and transformed time, respectively. Specifically, the variable `t_unit` ranges within the interval [0,1], where, in essence, 0 means *beginning of trajectory* and 1 *end of trajectory*. Hence, the mapping between these two variables —carried out by function `unit_time` and parameterized by the `time` field of the `Motion` structure— indicates whether the trajectory should be followed either forward or backward: a positive derivative in this function means forward motion, i.e., follow the trajectory as defined, and a negative derivative means a backward path.

  The `unit_time` function and the `TimeCurve` structure operates conceptually similar to the so-called animation curves, commonly found in software for video editing, 3D scene creation, or video game development. These curves allow for the timing of transitions to be controlled independently of the magnitude or other characteristics of the motion. To compute the output time corresponding to each input time, the `unit_time` function uses the **Interpolations.jl** package to perform a piecewise linear interpolation on the curve defined by the `time` field. In its simplest form, this curve contains only the start and end times of the transition, but it can include an arbitrary number of intermediate points.

[4]When the input argument is a `MotionList`, the `get_spin_coords` implementation is essentially the same as when the input is a `Motion`, except that it is necessary to iteratively calculate the displacements produced by all the motions in the `MotionList`, in the correct order.

- `get_indexing_range`: the input argument of this function is the `spins` field of the motion, which defines the "spin span", that is, the set of spins that are affected by the motion. It returns a range in the appropriate format for use as an index range, allowing efficient manipulation of position and displacement arrays for the relevant spin elements. Spin span types will be presented in Section 3.1.3b.
- `displacement_x!`, `displacement_y!`, and `displacement_z!`: these functions calculate the displacements caused by the `action` field of the motion along the x, y, and z directions, respectively. They take as input the action, the initial positions of the spins, and the time vector at which the displacements should be evaluated. Rather than returning an array of displacements, they update the existing arrays `ux`, `uy`, and `uz`[5], allowing the accumulated effect of all the motions involved to be tracked in the case that there is more than one, as it occurs with `MotionList`. Action types and their corresponding `displacement!` implementations will be explained in Section 3.1.3c.

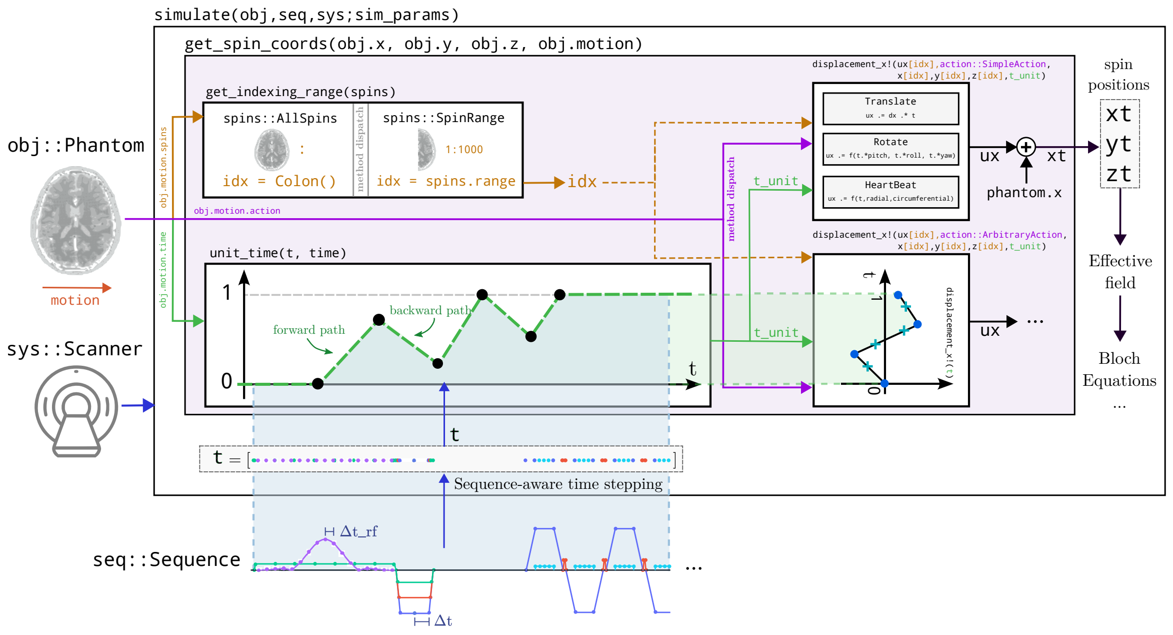


Figure 3.2: Operation of the `get_spin_coords` function (purple rectangle) in the context of the `simulate` function. It can be observed that three functions are involved: `unit_time` transforms the absolute time vector, extracted from the sequence, into a time vector that ranges from 0 to 1, in which positive slopes indicate forward motion and negative slopes indicate backward motion. `get_indexing_range` returns the range of spins affected by the motion in a suitable format for indexing. Last, `displacement!` functions compute the displacements caused by the motion's action, within the specified spin range, and at the transformed time instants. The calculated displacements are added to the initial spin positions to determine the absolute positions at each time instant. These positions are then used to compute the effective magnetic field at each location, which is used to solve the Bloch equations.

The following points provide a detailed explanation of each field in the motion structure and, specifically, of their corresponding structures.

a) TimeCurve

The `time` field defines the temporal behaviour of the motion and must be an instance of the `TimeCurve` structure, illustrated in Code 3.3. As its name implies, this structure represents a temporal

[5]Note that now `ux`, `uy`, and `uz` no longer refer directly to the motion functions, but to the displacement matrices resulting from their computation.

or animation curve, a concept which was discussed earlier. The curve is mainly defined by two vectors: `t` and `t_unit`, which represent the horizontal (x axis) and vertical (y axis) components, respectively.

Additionally, the `TimeCurve` structure contains two more fields, independent of each other: `periodic` is a Boolean that indicates whether the time curve should be repeated periodically; `periods` contains as many elements as repetitions are desired in the time curve. Each element specifies the scaling factor for that repetition. This enables the representation of pseudo-periodic repetitive patterns with varying durations, such as those observed in cardiac arrhythmias.

```julia
struct TimeCurve{T<:Real}
    t::AbstractVector{T}
    t_unit::AbstractVector{T}
    periodic::Bool
    periods::Union{T,AbstractVector{T}}
end
```

Code 3.3: `TimeCurve` structure definition.

To illustrate the functionality of `TimeCurve` and enhance the understanding of both the structure itself and the `unit_time` function, four examples of `TimeCurve` instances are presented below. These examples correspond to the four possible combinations of values for the `periodic` and `periods` fields. In the figures accompanying these examples, the black dots represent the (`t`, `t_unit`) pairs, while the dashed blue line shows the output produced by the `unit_time` function for each case:

Non-periodic motion with a single repetition This is the simplest case, where the phantom starts from a resting state, performs the motion, and ends in a final state, where it will remain indefinitely. To achieve this, when an input time instant falls outside the interval in which motion occurs, a flat extrapolation is performed. If the time instant is to the left of the interval, the extrapolated value is set to the first value of `t_unit`, and if it is to the right, it is set to the last value of `t_unit`. Code 3.4 and Figure 3.3 provide an example of a `TimeCurve` with these characteristics.

```julia
timecurve = TimeCurve(t=[.0, .2, .4, .6], t_unit=[0, 0.2, 0.5, 1.0])
```

Code 3.4: Non-periodic `TimeCurve` with a single repetition.

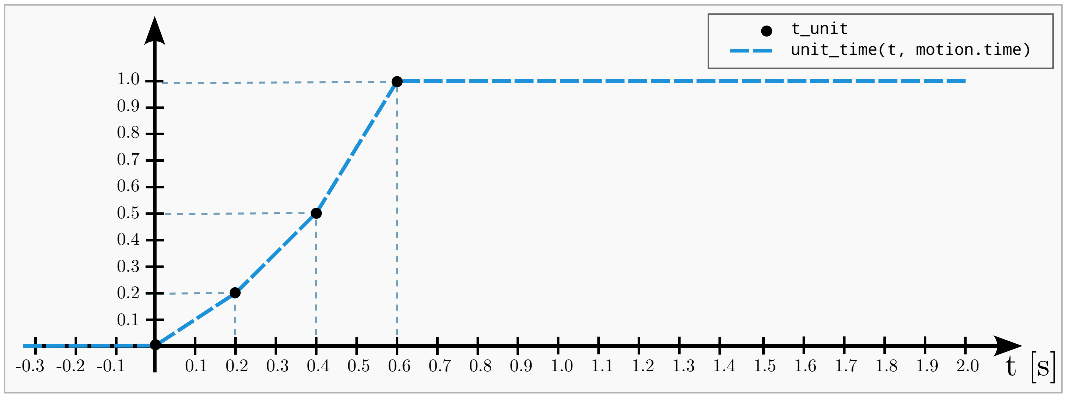


Figure 3.3: Non-periodic `TimeCurve` with a single repetition. A flat extrapolation is applied outside the time interval defined by the curve.

Periodic motion with a single repetition This represents a typical periodic motion, where the shape of the period is determined by the vectors `t` and `t_unit`. By setting the argument `periodic=true` in the `TimeCurve` constructor call, the `unit_time` function performs periodic extrapolation instead of flat extrapolation, seamlessly repeating the same pattern for both positive and negative time directions. Code 3.5 and Figure 3.4 illustrate this case.

```
timecurve = TimeCurve(t=[.0, .2, .4, .6], t_unit=[0, 1.0, 1.0, 0], periodic=true)
```

Code 3.5: Periodic `TimeCurve` with a single repetition.

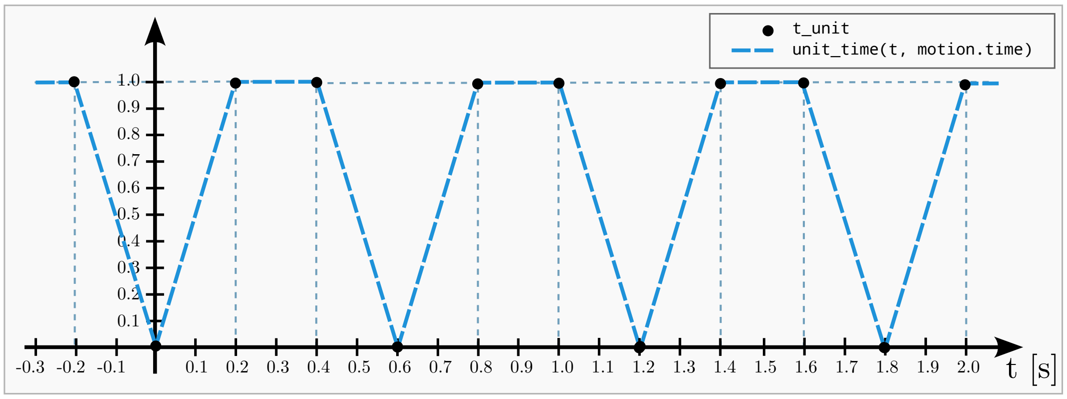
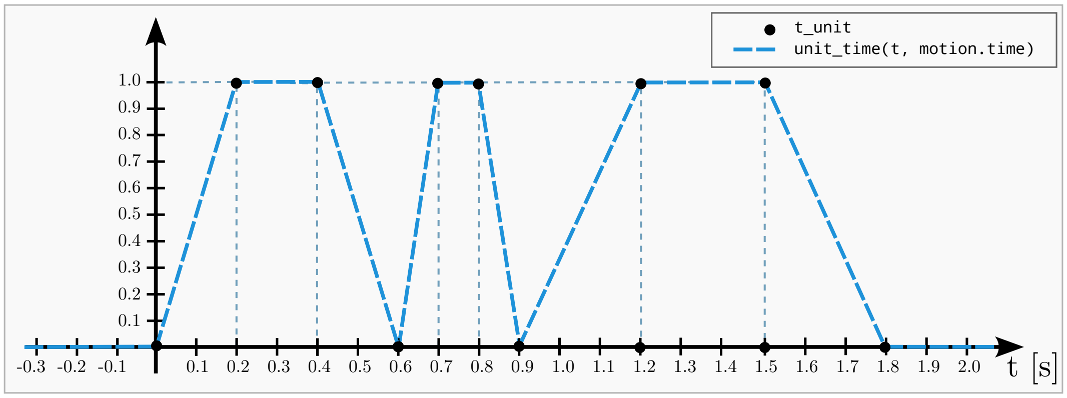


Figure 3.4: Periodic `TimeCurve` with a single repetition. A periodic extrapolation is applied outside the time interval defined by the curve.

Non-periodic motion with multiple repetitions In this motion, the pattern specified by `t` and `t_unit` is repeated a limited number of times. In this case, the `TimeCurve` constructor includes an additional input argument, `periods`, provided as a vector with as many elements as repetitions are desired. Each element of this vector acts as a scaling factor for the time axis, allowing each repetition to be individually stretched or compressed. This flexibility makes it possible to define non-uniform or arrhythmic patterns. Code 3.6 and Figure 3.5 provide an example of this scenario.

```
timecurve = TimeCurve(t=[.0, .2, .4, .6], t_unit=[0, 1.0, 1.0, 0], periods=[1.0, 0.5, 1.5])
```

Code 3.6: Non-periodic `TimeCurve` with multiple repetitions.



,

Figure 3.5: Non-periodic `TimeCurve` with multiple repetitions. A flat extrapolation is applied outside the time interval defined by the curve.

Periodic motion with multiple repetitions The most complex case arises when the `periodic` argument is set to `true` and the `periods` vector contains more than one element. In this scenario, a finite set of scaled repetitions is repeated indefinitely. An example of this configuration is shown in Code 3.7 and Figure 3.6.

```
timecurve = TimeCurve(t=[.0, .2, .4, .6], t_unit=[0, 1.0, 1.0, 0],
              periods=[1.0, 0.5, 1.5],  periodic=true)
```

Code 3.7: Periodic `TimeCurve` with multiple repetitions.

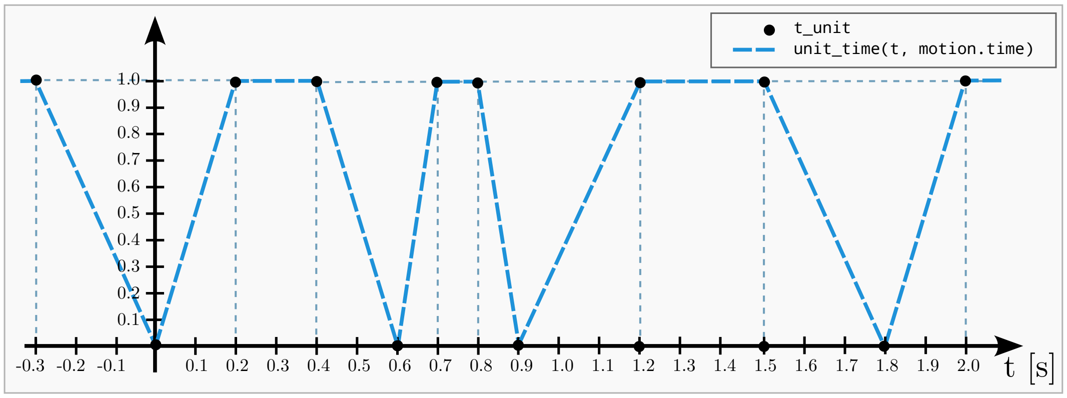


Figure 3.6: Periodic `TimeCurve` with multiple repetitions. A periodic extrapolation is applied outside the time interval defined by the curve.

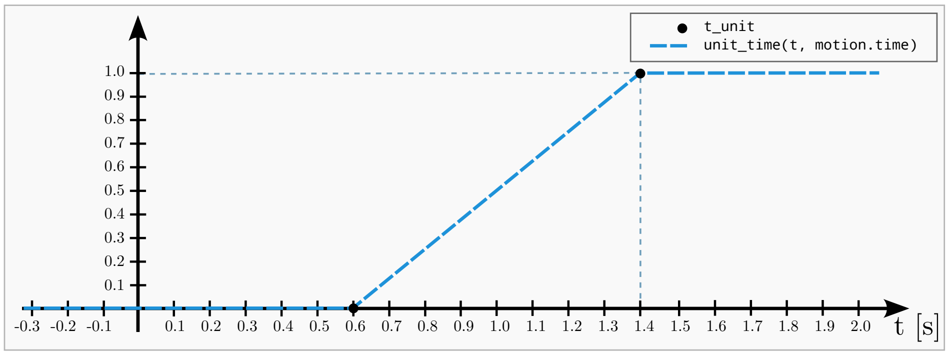
Additionally, two custom `TimeCurve` constructors have been implemented to streamline the process of defining motions with simple time spans. An example of how these constructors are used, along with their corresponding graphical representations, can be found in Code 3.8 and Figure 3.7, respectively.

In essence, these constructors can be seen as simple functions that only require two arguments. However, internally, these functions still generate a `TimeCurve` structure, but with some predefined fields. The two constructors are as follows:

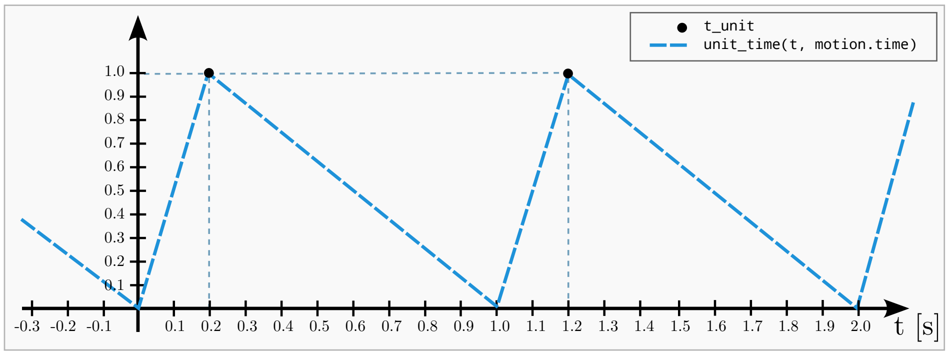
- `TimeRange`: this function defines a time interval, with start and end times. The transformation applied to the time axis is linear within the range (`t_start`, `t_end`) and constant outside of it, as illustrated in Figure 3.7a.
- `Periodic`: this function is designed to handle time intervals that repeat periodically. In this case, we have to assure that the phantom recovers its original shape at the end of each period. The period will comprise a deformation interval and a recovery interval and, generally speaking, these intervals lengths may not coincide. This is solved by defining a the `asymmetry` parameter, for which the value 0.5 means symmetry, i.e, the rise and fall times are equal, each comprising half of the `period`, while lower values imply left skewness and higher values right skewness. An example of such motion patterns are those of the heart, where the duration of systole is typically shorter than that of diastole. Figure 3.7b illustrates this time transformation.

```
# TimeRange
timerange = TimeRange(t_start=0.6, t_end=1.4)
# Periodic
periodic = Periodic(period=1.0, asymmetry=0.2)
```

Code 3.8: Example usage of the `TimeRange` and `Periodic` functions.



(a)



(b)

Figure 3.7: Graphical representation of the time curves generated with the constructors in Code 3.8. (a) `TimeRange` curve. (b) `Periodic` curve.

b) AbstractSpinSpan

The type of the `spins` field must inherit from `AbstractSpinSpan`. This field specifies which regions or tissues of the phantom are affected by the motion. The information is provided as a range of indices corresponding to the affected spins. Figure 3.8 illustrates the two types of spin span:

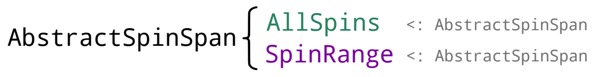


Figure 3.8: Spin span types.

- `AllSpins`: this is the default type, used to specify that the motion should affect all the spins of the phantom.
- `SpinRange`: this structure is used to select a certain range and number of spins. Its only field, `range`, is a collection of indices that allows accessing, by means of array indexing, to only the desired spins within the phantom. This is particularly useful when different parts of the anatomical model need to have different types of motion. More specifically, one possible use case is modelling the flow of a fluid, such as blood, within a static cavity.

c) AbstractAction and action types

The `action` field contains information about the motion itself, specifically, the type of motion —e.g., translation, rotation, etc.— and its magnitude. This magnitude can also be interpreted as the resulting displacement or the final state achieved after the motion has taken effect. As previously stated, the `action` field must be an instance of a structure inheriting from the `AbstractAction` type.

However, two intermediate abstract types have been introduced, both extending `AbstractAction`. These types classify actions into two distinct categories, referred to as "simple" and "arbitrary" actions. Figure 3.9 illustrates, by means of a tree diagram, the different actions and how they organize under these abstract types.

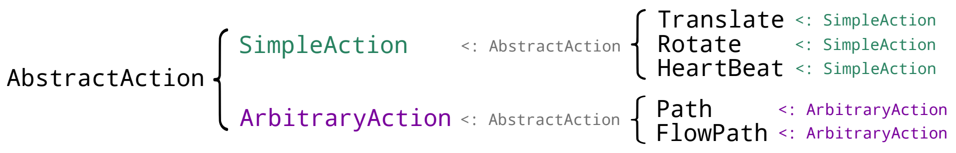

AbstractAction
SimpleAction <: AbstractAction
Translate <: SimpleAction
Rotate <: SimpleAction
HeartBeat <: SimpleAction
ArbitraryAction <: AbstractAction
Path <: ArbitraryAction
FlowPath <: ArbitraryAction

Figure 3.9: Distribution of actions into two subgroups: `SimpleAction` and `ArbitraryAction`.

Next, we will outline the actions that have been defined along with their respective implementations of the `displacement!` function, which include the formulas used in each action to compute displacements at each point in time. In all these formulas the variable `t_unit`, which is obtained by

```
t_unit = unit_time(t, motion.time)
```

will be expressed, for simplicity, as

$$t_u(t;\mathbf{\Theta})$$

where $\mathbf{\Theta}$ comprises the parameters that define the `TimeCurve` and t is the Sequence-aware time stepping indicated in Figure 3.2.

Simple actions These include all actions whose motion can be described using a finite set of well-defined parameters. These actions are relatively simple in terms of both mathematical implementation and representation, as they can be easily modelled with a closed-form expression. This means that the motion associated with these actions can be described using a mathematical formula that directly relates the input parameters to the resulting transformation. The simple actions that have been defined, and their corresponding `displacement!` implementations, are:

- `Translate`: this structure defines a linear translation. Its fields are the displacements in the three axes: `dx`, `dy`, and `dz`. This motion is independent of the initial positions of the spins, and the displacements are directly calculated as:

$$ux(t) = dx \cdot t_u(t;\mathbf{\Theta}), \qquad uy(t) = dy \cdot t_u(t;\mathbf{\Theta}), \qquad uz(t) = dz \cdot t_u(t;\mathbf{\Theta}) \tag{3.1}$$

- `Rotate` defines a rotation in the three axes[6]: x (`pitch`, γ), y (`roll`, β), and z (`yaw`, α). For this rotation, the Right-Anterior-Superior (RAS) orientation [72] is followed, and the rotations are applied counter-clockwise, following the right-hand rule. Figure 3.10a illustrates this reference frame. The center of rotation is dynamically set at the current position of the point that was at position $(0, 0, 0)$ at the beginning of the experiment.

  Rotations along these axes are determined using a rotation matrix for each axis (see Appendix A.1), and combining them into a single resulting matrix:

$$\begin{aligned} R(t) &= R_z(\alpha)\cdot R_y(\beta)\cdot R_x(\gamma) \\ &= \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix} \\ &= \begin{bmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma \end{bmatrix} \end{aligned} \tag{3.2}$$

[6]The symbol γ in this section is used with the meaning of an angle expressed in radians. It should not be confused with the gyromagnetic constant.

where γ, β, and α are time-dependent angles obtained from the final values `pitch`, `roll`, and `yaw`:

$$\gamma(t) = \texttt{pitch} \cdot t_u(t; \mathbf{\Theta}), \qquad \beta(t) = \texttt{roll} \cdot t_u(t; \mathbf{\Theta}), \qquad \alpha(t) = \texttt{yaw} \cdot t_u(t; \mathbf{\Theta}) \tag{3.3}$$

Therefore, the displacements are calculated as:

$$\begin{bmatrix} ux(t) \\ uy(t) \\ uz(t) \end{bmatrix} = R(t) \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} - \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{3.4}$$

- `HeartBeat`: it defines a heartbeat-like motion, characterized by three types of strain: circumferential, radial, and longitudinal. Figure 3.10b illustrates these strains in the myocardium. These parameters range from -1 to 1, as they must represent both contraction —from -1 to 0— and expansion —from 0 to 1—.

  To calculate the spin displacements associated with the strains, it is useful to convert the x, y, and z vectors to cylindrical coordinates:

$$r = \sqrt{x^2 + y^2}, \qquad \theta = \arctan\left(\frac{y}{x}\right), \qquad z = z \tag{3.5}$$

  Now, let $\mathbf{r}$ represent the vector consisting of the r values of the spins within the heart. In this manner, the displacements of each spin in the radial direction can be determined as:

$$\begin{aligned} \Delta_{\text{circumferential}} &= \texttt{circumferential\_strain} \cdot \max(\mathbf{r}) \\ \Delta_{\text{radial}} &= -\texttt{radial\_strain} \cdot (\max(\mathbf{r}) - r) \end{aligned} \tag{3.6}$$

  Therefore, the total time-dependent displacement for each spin in the radial direction will be:

$$\Delta_r(t) = (\Delta_{\text{circumferential}} + \Delta_{\text{radial}}) \cdot t_u(t; \mathbf{\Theta}) \tag{3.7}$$

  Since the conversion of the longitudinal component between Cartesian and cylindrical coordinates is straightforward, it is sufficient to multiply this component by the longitudinal strain factor.

  Thus, converting back to Cartesian coordinates, the displacements in each direction are:

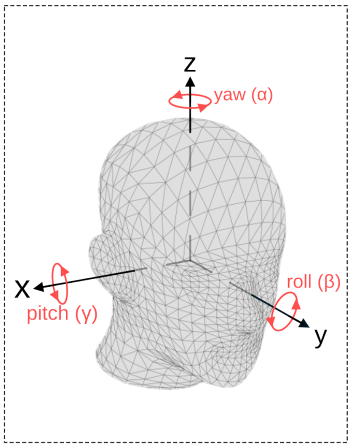
$$\begin{aligned} ux(t) &= \Delta_r(t) \cdot \cos(\theta) \\ uy(t) &= \Delta_r(t) \cdot \sin(\theta) \\ uz(t) &= \texttt{longitudinal\_strain} \cdot z \cdot t_u(t; \mathbf{\Theta}) \end{aligned} \tag{3.8}$$

Arbitrary actions Rather than describing motion through a set of parameters, arbitrary actions enable the direct specification of the trajectory of each spin. These trajectories consist of a series of positions at specific points in time. The exact timing of these points depends on the previously mentioned time span; for instance, if the trajectories include N_t time points and the time span extends from 0 to T, the time points will be evenly spaced between 0 and T, with $\Delta t = \frac{T}{N_t - 1}$.

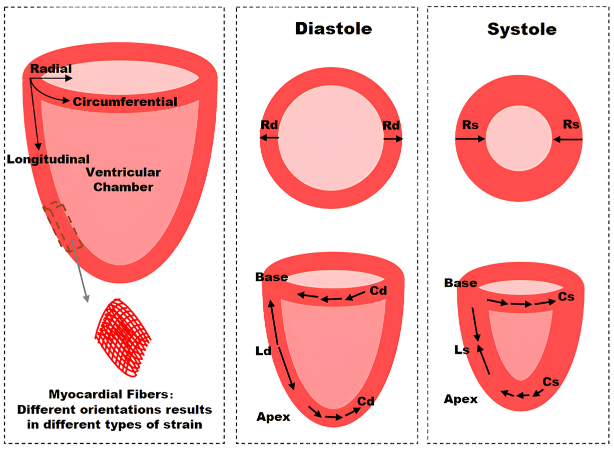
Two types of arbitrary actions have been defined: `Path` and `FlowPath`. Both share three common fields —`dx`, `dy`, and `dz`— which correspond to the aforementioned trajectories, and differ only in the presence of an additional field in `FlowPath`, called `spin_reset`, whose purpose will be explained later.

Both types also share the same implementation of the `displacement!` functions, which rely on temporal interpolation of the trajectories at the time points of interest. This interpolation uses the matrices `dx`, `dy`, and `dz` as input, where each matrix has a number of rows equal to the number of dynamic spins and a number of columns corresponding to the discrete time points that define the trajectory, which can be referred to as nodes. Figure 3.11 illustrates this operation. Specifically, a piecewise linear interpolation method is employed, evaluated at the time points specified by the sequence `seqd`.

To include flow and diffusion dynamics, `FlowPath` extends the `Path` type by incorporating the `spin_reset` field. In order to understand the functionality and purpose of this field, it is important to consider some preliminary points that, although they may have been inferred so far, have not yet been explicitly explained in this document:



(a)



(b)

Figure 3.10: (a) Reference frame used for rotation. (b) Schematic diagram of myocardial strain for diastole and systole. Three types of strain are defined along longitudinal, circumferential, and radial, respectively [73].

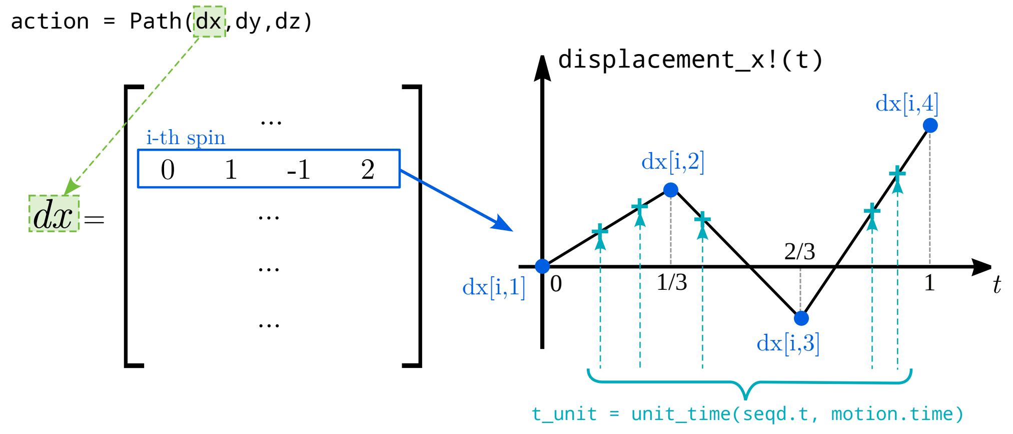
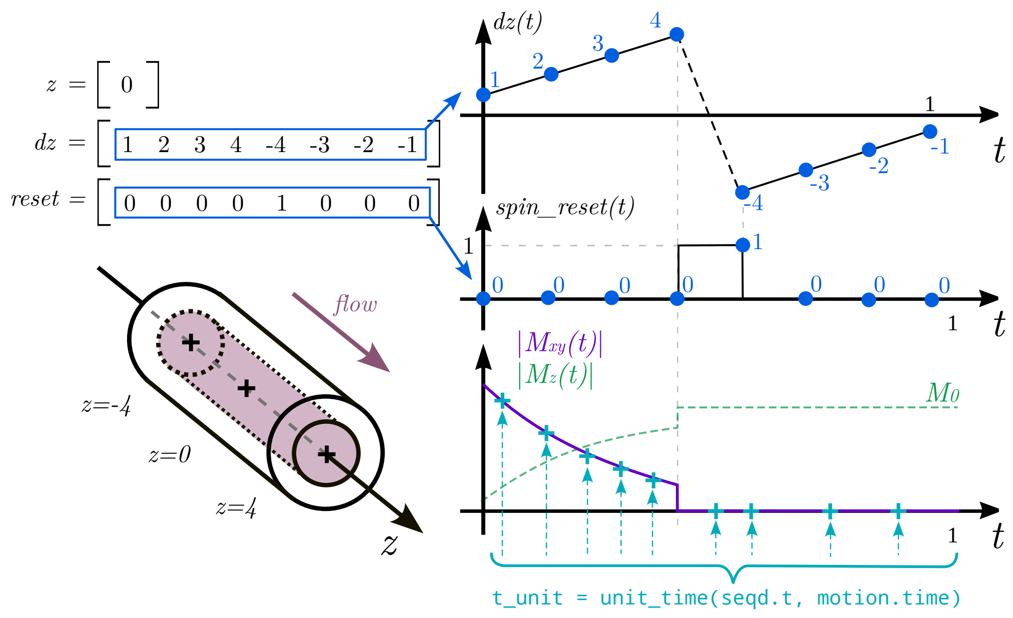


Figure 3.11: Displacements calculation for an arbitrary action. For simplicity, only `dx` for the i-th spin is shown. Spin displacements are calculated by means of a piecewise linear interpolation. The variable `seqd.t` represents the discretized sequence time points in which the displacements are interpolated (referred to in Figure 3.2 as *Sequence-aware time stepping.*)

1. The phantom in KomaMRI must always maintain a constant number of spins during the simulation, as it is composed of fixed-size vectors.
2. Spins can be classified based on their motion:
   (a) Static spins: those that do not move during the entire simulation time.
   (b) Dynamic spins: those that exhibit motion, and can be divided in:
      i. Tissue spins (e.g., myocardium): these are always present within the simulated imaging volume. They move relative to an initial position.
      ii. Flow spins (e.g., blood): these exhibit fluid flow motion through a vessel, meaning that some spins will enter and some will exit the imaged volume. Spins that undergo diffusion can also be accounted for in this class.
3. In order to satisfy both points 1 and 2(b)ii simultaneously, it is necessary to remap the position of the spins once they exit the volume. When a spin exits through an exit conduit, it must be immediately reinjected through an entry conduit. This introduces a discontinuity in its motion pattern.

4. In addition to remapping its position, the magnetization of the spin must be reset to its initial state:
$$M_z = M_0, \qquad M_{xy} = 0, \tag{3.9}$$
so that it no longer contributes to the acquired signal and can be interpreted as a "new" spin.

The `spin_reset` field is therefore a flag matrix which accounts for spins leaving the volume and being remapped to another entry position. It must have the same size as the `dx`, `dy`, and `dz` matrices. Hence, when simulating, it must also be interpolated in the same way, at the time steps specified by the sequence. However, this interpolation is simpler than the one performed with the displacement matrices, as we are only working with boolean values. Specifically, a piecewise constant interpolation is applied, always assigning the value at the next node. Figure 3.12 accounts for this spin state reset.



$z = [\,0\,]$
$dz = [\,1\ 2\ 3\ 4\ \text{-}4\ \text{-}3\ \text{-}2\ \text{-}1\,]$
$reset = [\,0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\,]$
flow
z=-4
z=0
z=4
z
dz(t)
1
2
3
4
-4
-3
-2
-1
1
t
spin_reset(t)
1
0
0
0
0
1
0
0
0
1
t
|Mxy(t)|
|Mz(t)|
M0
1
t
t_unit = unit_time(seqd.t, motion.time)

Figure 3.12: Spin state reset over a cylindrical phantom with fluid flow motion inside. The `z`, `dz` and `spin_reset` arrays are displayed for one flow spin with initial position `z + dz[1] = 1`. The spin moves along the flow direction, and upon reaching the exit boundary (`z=4`), it is remapped to the entry boundary (`z=-4`), setting the corresponding element in `spin_reset` to 1. The piecewise constant interpolation applied to the `spin_reset` array ensures that magnetizations are reset precisely at the moment the jump begins.

Unlike displacements, these calculations cannot be performed in advance. Instead, they must be performed after computing the magnetizations as they involve resetting the magnetizations for each spin that exhibits a discontinuity in its motion pattern. For this reason, another function, `outflow_spin_reset!`, has been defined in `KomaMRICore` to perform the calculations described above. Thanks to method dispatch, this function is only executed when the phantom includes a motion with `FlowPath`. Code 3.9 presents pseudocode for the implementation of `run_spin_precession!`, which includes calls to `get_spin_coords` and `outflow_spin_reset!`.

```
function run_spin_precession!(p::Phantom, seq, signal, mag)
    xt, yt, zt = get_spin_coords(p.motion, p.x, p.y, p.z, seq.t') # Motion
    Bz   = (...) # Effective field
    φ    = (...) # Rotation
    mag .= (...) # Mxy precession and relaxation, and Mz relaxation
    outflow_spin_reset!(mag, seq.t', p.motion) # Reset Spin-State (Magnetization)
    signal .= sum(mag.xy[:, findall(seq.ADC)]; dims=1) # Acquired signal
    return nothing
end
```

Code 3.9: Pseudocode for the new `run_spin_precession!` implementation.

Note that both the spin trajectories and the `spin_reset` matrix must be provided by the user when designing the phantom. For this reason, third-party software might be useful for generating these data, including displacement information and its discontinuities. Appendix C outlines the methods used by our collaborators[7] in this work to obtain, process, and integrate flow data into KomaMRI.

To implement the concepts described throughout this Section 3.1.3, various of the aforementioned Julia packages were utilized:

Functors.jl and **Adapt.jl** packages have been used to adapt data precision of the motion-related structures and transferring them, when necessary, to the GPU. The combination of the `fmap` and `adapt` functions allows the programmer to recursively adapt each field of the objects being transferred. Additionally, the `adapt` function needs to be overloaded in the cases that the input argument is a `MotionList`, which is achieved through the `adapt_storage` function:

```
adapt_storage(backend::KA.GPU, xs::MotionList) = MotionList(gpu.(xs.motions, Ref(backend)))
adapt_storage(T::Type{<:Real}, xs::MotionList) = MotionList(paramtype.(T, xs.motions))
```

Code 3.10: KomaMRI motion-related adapts.

For the interpolations involving arbitrary actions, **Interpolations.jl** has been employed. Specifically, the `GriddedInterpolation` structure leverages the need for performing N_{spins} independent 1D interpolations by combining them into a single 2D grid interpolation, which is more efficient. Additionally, the GPU support provided by this package allows for further optimization by taking advantage of this easily parallelizable grid structure and the function broadcasting feature.

Despite this, the motion-related functions are not yet fully optimized, and performance could be further improved by following strategies already applied in some functions of `KomaMRICore`, such as implementing separate versions for CPU and GPU, and for the latter, optimizing them at a lower level using **KernelAbstractions.jl**.

3.1.4 4D Phantom visualization tool

The `KomaMRIPlots` sub-package defines multiple functions for visualizing the elements involved in the simulation process. These are the phantom, the sequence, the K-space trajectory, and the results, both in raw signal form and as an image. All these functions are implemented by means of the **PlotlyJS.jl** package. Specifically, the `plot_phantom_map` function allows for the visualization of the spin parameter maps (T_1, T_2, T_2^*, PD) within the phantom, as shown in the example in Figure 2.30. These maps can be visualized in both 2D and 3D, with the choice being left to the user.

To incorporate the time dimension into the visualizations, capturing the temporal evolution of dynamic phantoms and thus addressing requirement **FRQ-1-07**, it is necessary to generate multiple scenes —i.e., frames— at different time points. For this purpose, a call to the aforementioned `get_spin_coords` function has been included within the implementation of `plot_phantom_map`. The set of time points at which this function is resolved is determined, by default, by the time span parameters of each motion included in the phantom. For instance, a motion with a `TimeRange(0.0, 1.0)` will yield two frames: one resolved at $t = t_{start} = 0.0$, and another at $t = t_{end} = 1.0$.

Referring to the implementation, Code 3.11 illustrates how each of the generated frames is stored in the `traces` variable. A horizontal slider is incorporated, enabling control of the time variable and displaying a specific element from this collection based on the slider's position. Lastly, two additional keyword arguments have been included into `plot_phantom_map`:

- `max_spins`: maximum number of displayed spins. For performance reasons, this is set by default to 20.000 spins.

[7]Group of prof. José Benito Sierra-Pallares, located at Escuela de Ingenierías Industriales, Universidad de Valladolid.

- `time_samples`: number of intermediate time samples between the endpoints defined by the motion time span. Default value is 0.

```julia
t = process_times(obj.motion)
# (...)
traces = GenericTrace[]
# (...)
l = Layout(;title=obj.name*": "*string(key)) # Create Layout
# (...)
for i in 1:length(t) # Add frames into the traces collection
    push!(traces, scatter3d( # For a 2D plot, use scattergl instead of scatter3d
        x=(x[:,i])*1e2,
        y=(y[:,i])*1e2,
        z=(z[:,i])*1e2,
        mode="markers",
        visible=i==1,
        showlegend=false))
end
# (...)
l[:sliders] = [attr( # Define the slider and its events
    visible=length(t) > 1,
    pad=attr(l=30, b=30),
    steps=[
        attr(
        label=round(t0*1e3),
        method="update",
        args=[attr(visible=[fill(false, i-1); true; fill(false, length(t) - i)])]
        )
        for (i, t0) in enumerate(t)
    ],
    currentvalue_prefix="t = ",
    currentvalue_suffix="ms",
)]
```

Code 3.11: Fragment of the `plot_phantom_map` implementation.

3.1.5 Phantom file format

In order to accomplish requirement **FRQ-1-08**, a new phantom (`.phantom`) file format has been defined. It is based on the HDF5 standard, and its complete structure is presented in Appendix D.

Regarding the modifications in the simulator, the `read_phantom` and `write_phantom` functions have been added into the `KomaMRIFiles` sub-package, thereby providing I/O functionalities:

- `read_phantom`: it reads a (`.phantom`) file and creates a `Phantom` instance from it.
- `write_phantom`: it writes a (`.phantom`) file from a `Phantom` instance.

Both functions have been implemented using the **HDF5.jl** package, which provides directives for handling such files. The complete implementation of these functions, along with all the other additions, is available in the KomaMRI GitHub repository.

3.1.6 Motion-related resources and scripts

To test all the contributions outlined in the previous sections and to provide users with practical resources for the simulator, several scripts have been developed. Thus, requirement **FRQ-1-09** is accomplished. The scripts generate sequences, phantoms, simulation pipelines and validation tests, and their contents and results are presented in Section 4.1.

3.2 Web Sequence Editor Application

The development of a client-server web application is aligned with the second main objective presented in Section 1.2 (page 3). This application aims to provide an improved web-based version of the MRI sequence editor introduced in [1], incorporating the enhancements proposed as future work in that study and integrating the improved KomaMRI simulator.

To achieve this, a web architecture is proposed, which consists of a front-end for user interaction and a back-end responsible for managing requests and interfacing with the simulator. The front-end we propose consists of a set of components —which we will refer to as modules—, each implemented using a distinct programming technology. These include an improved sequence editor, a sequence diagram viewer, an enhanced 3D visualization tool for selected slices, and a viewer for the resulting simulated images. With respect to the back-end, it will manage communication between the front-end and the KomaMRI simulator, executing the necessary simulations and returning the corresponding results.

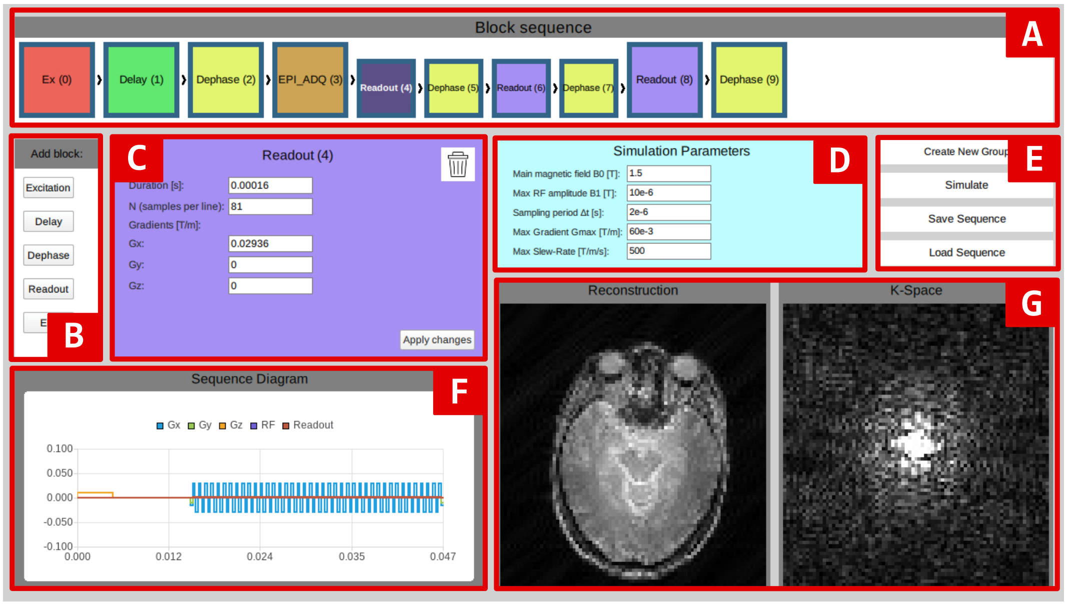
This section first introduces the application developed in [1], which serves as background for this work. After presenting the requirements, it then describes the general operation of the application, the technologies and strategies used for each module, and the methods employed for their communication. All the code discussed in this section is available in a GitHub repository[8].

3.2.1 Background

Figure 3.13 shows the GUI of the desktop application developed in [1], which serves as a reference for this section of the work. The sequence panel A stands out by occupying the entire width of the application window; this is intended as a way to attract the user attention to the sequence, which is the main constituent of the whole process. In addition, it eases its complete visualization, without the need of trimming or drastically reducing the number of visible blocks; these blocks can be freely moved, modified and deleted. Panel B provides the library of available blocks, which can be added to the sequence by simply clicking one of the panel buttons. Panels C and D contain, respectively, the block configuration and the scanner parameters. As for the former, clicking on a given block will display its configuration menu from which related parameters can be easily set; as for the latter, scanner parameters are directly tunable. Panel E is intended for sequence management; simple operations such as sequence loading and saving are available. In addition, the user is allowed to create a block group; groups are intended for providing repetition capabilities without the need of explicitly writing the repeated blocks. The button to launch the simulation is also allocated here for convenience. Finally, the lowermost panels, namely, panels F and G, are intended for visualization; specifically, panel F will display the sequence pulse diagram while panel G will show the simulation result, both in image and k spaces [74].

User testing has revealed some limitations in this application, which have guided the design considerations and the establishment of requirements (see Section 3.2.2) for this part of the work. While the GUI and its corresponding back-end provide essential functionalities, certain aspects can be improved. First, greater flexibility is needed when defining gradient shapes and intensity. Additionally, the application does not support the definition of global variables or sequence parameters such as TE, TR, ETL, or FOV, among others. These parameters, commonly found in other console-based simulators [3, 5, 6, 10], are particularly valuable when the tool is used for educational purposes in training technicians. These users typically do not need to configure each individual sequence block but instead require a more comprehensive overview, similar to the interface found in real MRI scanner consoles. Lastly, the 3D slice visualization tool, though functional, lacks interactivity and efficiency. Moreover, since it opens in a separate window, it is not ideal for a client-server application.

[8]WebMRISeq GitHub repository: https://github.com/pvillacorta/WebMRISeq/tree/tfm.



Duration [s]: 0.00016
N (samples per line): 81
Gradients [T/m]:
Gx: 0.02936
Gy: 0
Gz: 0

Main magnetic field B0 [T]: 1.5
Max RF amplitude B1 [T]: 10e-6
Sampling period Δt [s]: 2e-6
Max Gradient Gmax [T/m]: 60e-3
Max Slew-Rate [T/m/s]: 500

Figure 3.13: Application GUI developed in [1].

3.2.2 Requirements Analysis

Since this part of the work builds upon a previously developed application, the main requirements established in [1] remain essential. These requirements define the core functionality of the original system and provide the basis for the enhancements introduced in this work. For completeness, we summarize below the original requirements, identified with the prefix **OR**. Thus, the new requirements presented in this document focus exclusively on improvements and additional features beyond the original implementation.

a) Functional Requirements (FRQ)

Original Functional Requirements (OR-FRQ)

- **OR-FRQ-01**
  A set of basic building blocks for an MRI pulse sequence, including RF pulses, gradients, and readout windows, must be ready to use out of the box [74].

- **OR-FRQ-02**
  The application must allow users to create custom MRI sequences by arranging these blocks, with the ability to add, modify, move, and delete them using keyboard arrows or drag-and-drop.

- **OR-FRQ-03**
  The application must support grouping blocks into composite units, which behave like individual blocks. Operations such as create, move, and delete must also apply to groups.

- **OR-FRQ-04**
  Users must be able to define scanner settings, including the main magnetic field strength, maximum RF pulse intensity, minimum sampling period, maximum gradient strength, and maximum slew-rate.

- **OR-FRQ-05**
  The GUI must include additional panels, such as a time sequence diagram visualizer, a 3D slice viewer, and a simulator launcher to execute the created sequence and display the resulting simulated image.

Functional Requirements (FRQ)

- **FRQ-2-01**
The gradients in the sequence must by default have a trapezoidal shape, with the ability to configure parameters such as the initial delay, rise and fall times, flat top duration, and amplitude. Additionally, as outlined in future work in [1], there must be a `step` parameter that allows for adjusting the gradient amplitude between each TR.
- **FRQ-2-02**
The sequence editor must include a panel that allows users to define global variables, which can then be used to set and adjust the sequence parameters.
- **FRQ-2-03**
The sequence editor must include a panel that allows users to define a sequence description in the form of a text string, where they can provide information about any relevant details of the sequence.
- **FRQ-2-04**
The application must allow exporting and saving the created sequences (and scanner parameters) as files, as well as importing and loading them in a format that is readable by both the sequence editor itself, the KomaMRI simulator, and the user.
- **FRQ-2-05**
Both the sequence diagram and the 3D slice viewer must be interactive and connected to the sequence editor, so that they reflect any changes made in the editor.

b) Non-Functional Requirements (NFR)

Original Non-functional Requirements (OR-NFR)

- **OR-NFR-01**
The system must be fully functional as a stand-alone desktop application.
- **OR-NFR-02**
Communication between the technologies used must be seamless for the user, both in terms of programming and usability, as well as in terms of response times.
- **OR-NFR-03**
The chosen technologies must allow the system to be later converted into a web application while maximizing code reuse from the desktop version.

Non-functional Requirements (NFR)

- **NFR-2-01**
The developed web application must provide all previously described functionalities without requiring users to install any software on their computer. All features should be accessible directly through the browser without additional configurations or installations.
- **NFR-2-02**
The back-end of the application must be hosted on a server capable of handling various client requests. Clients should be able to request files related to the GUI, the sequence diagram, or the results of MRI simulations processed on the server, ensuring seamless communication between the server and the client interface.
- **NFR-2-03**
The application must feature an enhanced, aesthetically pleasing, and user-friendly interface, improving upon the visual design and usability of the original version [1]. This upgrade should significantly enhance the overall user experience.

3.2.3 General Operation of the Application

The general operation of the application is illustrated in Figure 3.14 and is based on a client-server architecture, specifically a REST architecture. The front-end runs in the browser, where user interactions generate HTTP requests that are sent to the server. This server, acting as an intermediary between the client and back-end resources —such as the MRI simulator, database, or front-end files— is a REST API. It follows the typical request-response model: clients send requests, and the server processes them and returns appropriate responses.

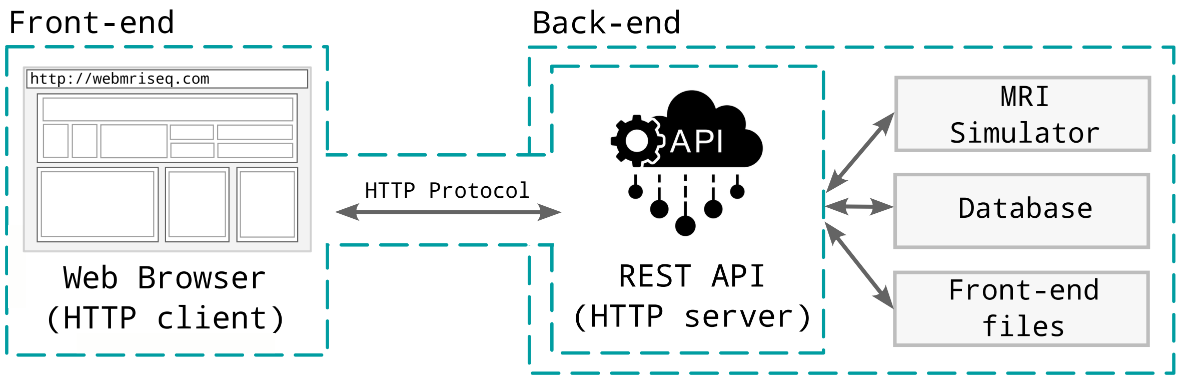


Figure 3.14: General operation of the web application. The web browser is responsible for both rendering the front-end and acting as an HTTP client to communicate with the server. The HTTP server, in turn, provides a REST API to the web browser for accessing the back-end functionalities.

The detailed operation of the front-end and back-end is explained in Sections 3.2.4 and 3.2.5, respectively.

3.2.4 Front-end Modules and Panels

As stated at the beginning of this section, the front-end design of the application is structured into four main modules, as shown in Figure 3.15. The first and primary module is the MRI sequence editor, which not only provides the core editing functionality, but also integrates controls for launching simulations and invoking graphical features. The remaining three modules serve as visualization panels, which display the results of these graphical functionalities —the sequence diagram and the selected 3D slice— as well as the simulation output. In the initial version of the application [1], these three visualization panels were implemented as QML (Qt) items. However, this work takes a different approach by extracting them from the main Qt interface and implementing them as independent web components. This shift allows for greater flexibility in selecting libraries and technologies better suited to their specific requirements. All the front-end files are located within the `/frontend` directory of the repository[9].

The following sections provide a detailed overview of each module, including the selected technologies for its development, its design, the panels it comprises, and the methods followed for its implementation.

a) Graphical Sequence Editor

Selected technologies Qt has been chosen as the core framework for this part of the work, not only to maintain continuity with its initial implementation [1], but also for its ability to create cross-platform GUIs with a native appearance and usability. Its compatibility with WebAssembly ensures execution

[9]See https://github.com/pvillacorta/WebMRISeq/tree/tfm/frontend.

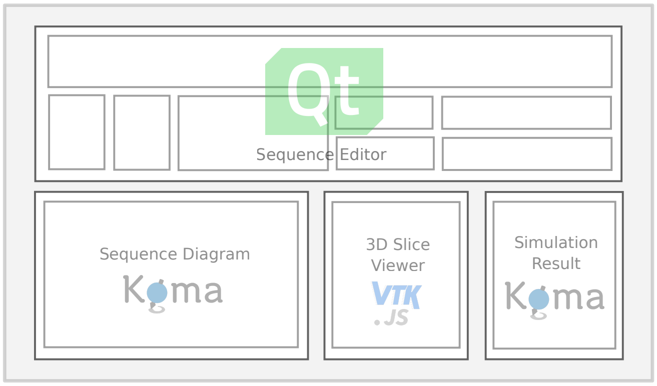


Figure 3.15: Front-end design of the application, structured into four modules. Each module, implemented with a different technology, may consist of a single panel (as in the three lower modules) or be further divided into multiple panels, as seen in the upper MRI sequence editor module.

in modern web browsers, so the transition from a desktop to a web application is nearly seamless[10]. In addition to Qt, two additional C++ libraries have been selected: `emscripten` and `nlohmann/json`[11]. The former facilitates communication between the C++ interface and the web components of other modules, written in JavaScript, while the latter is used for handling files and data structures in JSON format, and will be included in the project through `vcpkg`, the C++ package manager.

Regarding the Qt module, both Qt Quick and Qt Widgets have been used in this project. Qt Quick has been selected for most of the GUI because of its flexibility and intuitive QML syntax. Moreover, Qt Widgets was also integrated for specific functionalities, such as using concrete classes like `QFileDialog` to handle file selection, ensuring that both dynamic GUI and system-level features could be effectively combined.

As the build system for this module, CMake was chosen due to the advantages outlined in Section 2.4.2c. In addition, the C++ compiler used will be Emscripten, as it is specifically designed for compiling to WebAssembly.

Design As previously mentioned, some of the graphical functionalities of the application will be implemented using technologies other than Qt. Therefore, given that our starting point is the one presented in Section 3.2.1, the first necessary step is to remove panels F and G (Figure 3.13) from this module. On the other hand, in order to address requirements **FRQ-2-01** to **FRQ-2-04**, it will be necessary to add new panels or to extend the functionalities of the existing ones.

Specifically, Figure 3.16 shows the design of the part of the GUI corresponding to the sequence editor and its panel layout. At the top, the toolbar contains drop-down menus for file handling —creation, loading, and saving—, as well as for plotting functionalities. Panels A-D remain consistent with the previous implementation [1]: panel A provides a schematic overview of the sequence being created by the user through the concatenation of blocks; panel B allows the addition of predefined blocks to the sequence; panels C and D enable the configuration of parameters for individual blocks and for the scanner, respectively. Regarding panel E, it is designed for the creation of block groups and their storage. Panel F allows the user to select a phantom from a library and run the simulation on it. The simulation is carried out on KomaMRI, which runs on the server, as will be explained later. The addition of panel G is one of the most significant updates, as it enables the definition of global variables. Each text field in the interface —whether in the block parameters, scanner settings,

[10]The migration to the web was not entirely straightforward, not due to Qt itself, but rather because of the other technologies involved, which were more related to the back-end, and the way they interacted. The following sections describe the methodology and approach taken to successfully transform the application into a fully functional web version.

[11]JSON C++ library: https://github.com/nlohmann/json.

or within the variable panel itself— can now be configured as an expression containing the names of the defined global variables. This functionality enhances flexibility by allowing dynamic adjustments across various settings in the application. Finally, panel H consists of an editable text field where the user can input a description for the sequence.

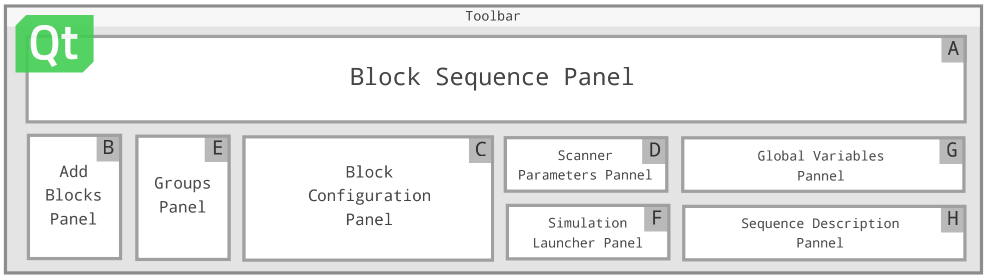

Toolbar
Qt
Block Sequence Panel
A
Add Blocks Panel
B
Groups Panel
E
Block Configuration Panel
C
Scanner Parameters Pannel
D
Global Variables Pannel
G
Simulation Launcher Panel
F
Sequence Description Pannel
H

Figure 3.16: Sketch of the part of the GUI corresponding to the sequence editor and its panel layout

Implementation The source code and files for this part of the GUI are stored in the directory `frontend/src/seqEditor` of the repository[12]. The most notable content of this directory includes the following:

- `main.cpp`: in this file, the Qt application is initialized from the `main` function by defining it as an instance of the `QApplication` class. Within this application, the QML files are loaded using the `load` function, with `Main.qml` as the entry point. Additionally, the `backend`[13] is integrated through the `setContextProperty` function.

- `backend.cpp`: this file contains the implementation of the `Backend` class, including the definition of its signals, slots, and auxiliary functions. The `backend` serves a dual purpose: first, it enables the communication between QML components and the C++ part of the application through signals and slots (see Section 2.4.2a); second, it allows for the interaction between the Qt-based sequence editor and the remaining web components of the application, written in JavaScript. This latter purpose is achieved using the `EM_JS` function of the `emscripten` library, which enables declaring JavaScript functions from inside a C/C++ file [70].

  The `Backend` class defines two signals: `uploadSequenceSelected` and `uploadScannerSelected`. These are emitted from the C++ side when the user wants to load a sequence or a scanner, respectively. The signals include as an argument the file path[14] where the sequence or scanner file has been saved. The QML side then receives the signals and processes the loaded information to display it in the GUI in the appropriate format.

  As for the `Backend` slots, they are invoked by the user when an event is triggered on the QML side:

  – Functions `getUploadSequence` and `getUploadScanner` open a `QFileDialog`, allowing the user to select a JSON file which contains the sequence or scanner information. After importing the data, they format it in a way that is readable by QML, store it in a temporary `.qml` file, and emit either the `uploadSequenceSelected` or `uploadScannerSelected` signal, depending on the case, along with the file path where the temporary QML file has been stored.

  – Functions `getDownloadSequence` and `getDownloadScanner` serve the opposite purpose of previous functions. They are designed to download the sequence or scanner created in the GUI. The exported file is in JSON format.

[12]See `https://github.com/pvillacorta/WebMRISeq/tree/tfm/frontend/src/seqEditor`.

[13]This `backend` object should not be confused with the back-end concept mentioned earlier, which refers to the entire application's logic, rather than just this Qt part.

[14]This path does not correspond to the local path where the client's file is stored but rather to a relative path within WebAssembly's sandboxed environment.

  - Functions `plotSequence`, `plot3D`, and `simulate` are responsible for sending the sequence and scanner data to the JavaScript layer[15] of the application. This layer then interacts with the server —specifically with KomaMRI— for the first and third functions, and with VTK.js for the second function. Thus, it is the JavaScript side that communicates directly with these services, not the C++ slots. For further details, refer to Section 3.2.6.

- **QML Files (`/qml`)**: These files define all the elements of the sequence editor's GUI, including their layout, interactions, and responses to user actions. The main file, `Main.qml`, initializes the application window and seamlessly integrates all interface components defined in separate QML files. Additionally, this main file defines two key elements:

  - Global[16] data structures, used to store relevant information for the GUI throughout the application's runtime. These structures exist only while the application is running; once the GUI window is closed, they are discarded. They serve as temporary storage for elements such as the constructed sequence, user-defined global variables, scanner parameters, and created block groups. Each of these structures is stored within an instance of the QML object `ListModel`, which organizes data in a list format.
  - JavaScript functions, which handle user-triggered events —such as button clicks, hovering over specific areas, or moving blocks— and manipulate the aforementioned global data structures accordingly. These functions can create, update, delete, and export data while also managing its presentation, such as expanding elements, displaying details, or collapsing them for a clearer interface.

  The specific details of these QML files, which include the JavaScript functions implementation and the global data structures definition, can be found in the previously mentioned GitHub repository, along with the rest of the code.

- `CMakeLists.txt`: this file defines the CMake configuration for building the C++/Qt project. It sets up the project to use Qt6 components —`Quick`, `QuickControls`, and `Widgets`— as well as the `nlohmann/json` library. Additionally, it defines the project structure, specifying source files and QML resources, and ensures proper linking of the required Qt and JSON libraries.

b) Sequence Diagram Viewer

Selected technologies Due to several factors, such as the unification of technologies and its suitability for our specific needs, KomaMRI itself has been chosen as the tool for drawing and displaying the temporal sequence diagram. Specifically, the `KomaMIPlots` sub-package has been used, which relies on `PlotlyJS.jl` to visualize instances of KomaMRI structures, such as the sequence, phantom, K-space trajectory, and simulated images, among others.

Additionally, the JavaScript Fetch API [66] has been chosen to handle HTTP requests between the client and the server. As outlined in Section 3.2.3, the server acts as a bridge between the front-end and the back-end, where KomaMRI is hosted.

Design and Implementation This module consists of a single panel displaying the sequence diagram generated by `KomaMRIPlots`. Figure 3.17 is a specialized view of Figure 3.14, focusing on the interactions related to the sequence diagram. It illustrates the communication between the client and server, detailing both the request and response flows.

On the client side, the `EM_JS` directive in C++/Qt enables calling JavaScript functions from WebAssembly. The function `plot_seq` is invoked with a JSON file containing the complete sequence data from the GUI. This JSON file is then sent from JavaScript to the REST server via a POST request to the `plot` endpoint. These request flows are represented with blue arrows in the figure.

[15] By "JavaScript layer", we refer to the files that constitute the core of the application's front-end; this should not be confused with the JavaScript code used in QML for event handling.

[16] They are global within the QML scope

On the server side, `KomaMRIPlots` —specifically, `PlotlyJS.jl`—processes the request and generates an interactive[17] HTML plot. This HTML file is sent back as the server's response, which the front-end embeds into the sequence diagram panel using an iframe. This response flow is depicted with green arrows, completing the communication cycle and ensuring seamless integration of the sequence visualization.

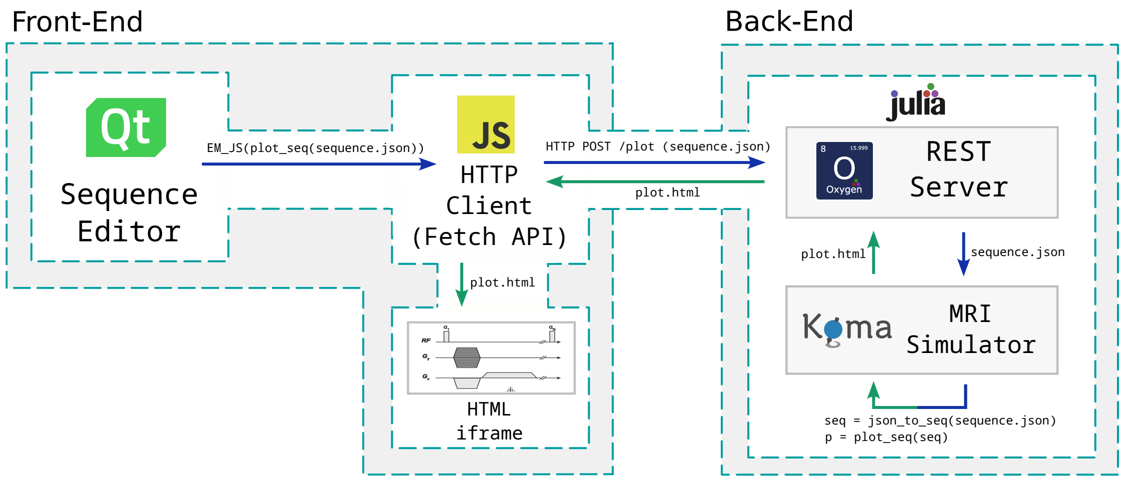


Figure 3.17: Communication flow for rendering the sequence diagram. Blue arrows represent client-side requests: the C++/Qt layer invokes JavaScript functions using `EM_JS`, which then sends the sequence data as a JSON file to the REST server via a POST request. Green arrows represent server responses: `KomaMRIPlots` generates an interactive HTML plot, which is returned to the client and embedded in the front-end using an iframe.

c) 3D Slice Viewer

Selected technologies The JavaScript library VTK.js, along with the `image-io` package from the `itk-wasm` project, have been chosen to develop the 3D visualization module for the selected slice. These libraries form an ideal framework for our needs, offering interactive 3D visualizations that do not require extreme detail, all within a lightweight, simple, and smooth viewer which can be embedded in the web browser. Additionally, the Webpack library is used to bundle JavaScript modules and manage assets, ensuring the application operates efficiently by optimizing this part of the front-end code. All these libraries can be installed using the npm package manager from Node.js.

Design and Implementation The 3D visualization module consists of a single panel displaying the phantom volume loaded from a NIfTI file, along with the slice selected by the MRI sequence, which is also part of that volume. The visualization is achieved using three orthogonal slice planes[18], as this reduces the computational load on the client side by avoiding rendering the full volume. Additionally, the slice selected by the MRI sequence is shown as another slice plane, which results in four planes being displayed in total: three orthogonal planes plus the selected one.

The JavaScript code for this part has been included in the `index.js` file located in the `/frontend/src` directory. Here, the NIfTI volume data is read and converted into VTK images using the `niftiReadImage` and `convertItkToVtkImage` methods from `itk-wasm`. To render the slice planes, the `vtkPlane`, `vtkImageResliceMapper`, and `vtkImageSlice` classes from VTK.js are used. Additionally, the `vtkImplicitPlaneRepresentation` class is employed to draw the contour of the selected plane and its normal vector. The detailed code is available in the GitHub repository.

[17] Along with the details in Sections 3.2.4c and 3.2.4d, this enables us to meet requirement **FRQ-2-05**.

[18] As outlined in the viewer developed by [3], this approach follows a similar methodology.

d) Simulation Result Viewer

Selected technologies As mentioned in Section 3.2.4b, KomaMRI includes functions that utilize PlotlyJS.jl to generate interactive plots. Therefore, `KomaMRIPlots` will also be employed for visualizing the images which result from the simulations. The process will follow the same approach as with the sequence diagram, which means that the JavaScript Fetch API will also be used to make HTTP requests to the server.

Design and Implementation Figure 3.17 can also be used as a reference for understanding this part of the system, as both the sequence viewer and the simulation results viewer are entirely analogous. The only difference is that, in this case, the server must receive not only the sequence information but also the phantom and scanner data, enabling it to perform the simulation and return the corresponding results. These results are provided as an HTML file which contains the plot generated by the `plot_image` function from `KomaMRIPlots`.

3.2.5 Back-end Module

The back-end of the application, as it has been shown in Figures 3.14 and 3.17, consists of an HTTP server and additional processes responsible for generating the necessary data. The HTTP server acts as an intermediary between the front-end and the rest of the back-end. It receives requests from the client, forwards them to the appropriate internal components, and returns the generated results. Notably, the entire back-end module has been implemented in Julia, and its source files can be found in the `/backend` directory of the repository[19].

This section first describes the HTTP server and then delves into the back-end resources involved in data generation, specifying the Julia modules used in each part and detailing its implementation.

a) HTTP Server: REST API

Selected technologies The Oxygen.jl [59] framework was chosen to implement the REST API that functions as the HTTP server. Additionally, the SwaggerMarkdown.jl package is used to generate pseudo-automatic documentation for the API methods, following the OpenAPI [75] specification.

Design and Implementation For the design of the HTTP server, a REST API (see Section 2.4.4d) has been defined to handle incoming HTTP requests from clients. In this architecture, the REST API and the HTTP server are unified in the same process. Once the HTTP request reaches the API, it directly calls Julia functions to interact with the back-end resources, bypassing HTTP for internal processing. This approach ensures that communication with the back-end occurs efficiently within the server process, without needing additional HTTP calls.

The implementation of this REST API is done in the `RESTserver.jl` file, where all the API methods are defined. Each method is associated with a specific resource, i.e., a unique URI. Specifically, the implemented methods are documented[20] and categorized into three sections: web content rendering, simulation, and plotting:

- The web content methods are the following:
  - `GET /`: this method, when directed to the root resource, responds with a 301 (Moved Permanently) HTTP status, redirecting the client to the `/app` URI.

[19] See `https://github.com/pvillacorta/WebMRISeq/tree/tfm/backend`.

[20] API documentation available in: `https://petstore.swagger.io/?url=https://raw.githubusercontent.com/pvillacorta/WebMRISeq/refs/heads/tfm/docs/api.yaml#/`.

- `GET /app`: this method serves the necessary files for rendering the front-end. It includes `index.html`, which acts as an entry point, as well as the WebAssembly files resulting from the sequence editor compilation, and the Webpack build files. All of these files are stored in the `/frontend/dist` directory. More details about this process can be found in Section 3.2.6.
- `POST /simulate`: This method starts a new simulation by receiving information about the sequence, scanner, and phantom from the client. The server generates a unique simulation ID and responds with HTTP status 202 (Accepted). The `Location` header in the response provides the URI where the client can later retrieve the simulation result: `/simulate/{simulationID}`.
- `GET /simulate/{simulationID}`: this method allows the client to retrieve the simulation result once the process is completed. This result, as stated in Section 3.2.4d, is provided as an HTML file which contains a plot with the simulated image. If the simulation is still running, the server responds with a 303 (See Other) HTTP status and includes in the `Location` header the URI where the client can track the progress of the simulation: `/simulate/{simulationID}/status`.
- `GET /simulate/{simulationID}/status`: this method provides real-time updates on the simulation status. It returns -1 if the simulation has not yet started, a value between 0 and 99 indicating the simulation progress percentage, 100 if the simulation is finished but the reconstruction is still ongoing, and 101 if the reconstruction has finished.

• The API method to generate the sequence diagram plot is the following:
  - `POST /plot`: this method generates a plot of the sequence diagram, which is created based on the sequence data provided in the body of the POST request. The sequence diagram, as outlined in Section 3.2.4b, is also provided as an HTML file.

On the client side, it is essential to handle the responses for each of these requests. The client needs to process the response content, ensuring appropriate actions are taken based on the data received. When necessary, the information should be displayed on the screen in the correct format. For example, when a simulation is ongoing, progress updates should be displayed, and upon completion, the result should be shown in a meaningful way to the user.

b) Back-end Resources: MRI Simulator, Database and Front-end Files

Selected technologies The Julia modules used to manage the other elements of the back-end are KomaMRI.jl, which handles MRI simulation and generates sequence and simulated image plots, and JSON3.jl, used for managing files and data structures in JSON format.

Design and Implementation The back-end resources consist of a set of Julia functions that can be invoked by the API to retrieve data and generate responses, which are then sent back to the client. These functions handle key operations such as MRI simulation execution, status tracking, and visualization generation, as well as retrieving the necessary front-end files stored in the `frontend/dist` directory (see Section 3.2.6).

Specifically, the back-end Julia functions are defined in two files:

• `ServerFunctions.jl`: this file contains the definition of all functions directly called by the API methods:
  - `render_html`: reads an HTML file and returns it as an HTTP response with a default status code of 200 and a content-type header set to HTML.
  - `json_to_sequence` and `json_to_scanner`: Convert JSON data from the client, corresponding to the sequence and scanner, into their respective `Sequence` and `Scanner` objects from KomaMRI.

  – MRI simulation functions: `sim` takes JSON representations of the sequence and scanner, a string referencing the phantom to be used, and the path of a temporary file for communication between back-end processes. It first converts JSON data into `Sequence` and `Scanner` objects using the functions above, then generates the phantom and runs the MRI simulation. Finally, it reconstructs the resulting signal using the `recon` function.

- `Sequences.jl`: this file defines two functions which correspond to specific sequence blocks in the front-end: `GRE` and `EPI`. They generate a `Sequence` object configured for either a Gradient Echo (GE) or Echo Planar Imaging (EPI) sequence.

3.2.6 Integration, Module Interaction, and Server-Side Compilation

This section provides a global perspective on the application, focusing on module interactions, compilation processes, and file organization. It details the communication methods between modules and sub-modules —previously described in other sections— and how they integrate to ensure seamless operation. Additionally, it examines the compilation process, specifying which parts are generated on the server and which on the client. Finally, it outlines the structure of source files and compiled outputs, designed to clearly separate development files from distribution files.

The application is designed so that all repository files, including both back-end and front-end components, reside on the server machine. The client, i.e., the end user, only requires a web browser to access and use all functionalities. Thus, both the `/backend` and `/frontend` directories must be present on the server machine. Additionally, for the front-end, this server must have Qt, CMake, Emscripten, and Node.js installed to handle compilation and execution requirements. On the other hand, for the back-end, a Julia installation is necessary on the server, as previously noted.

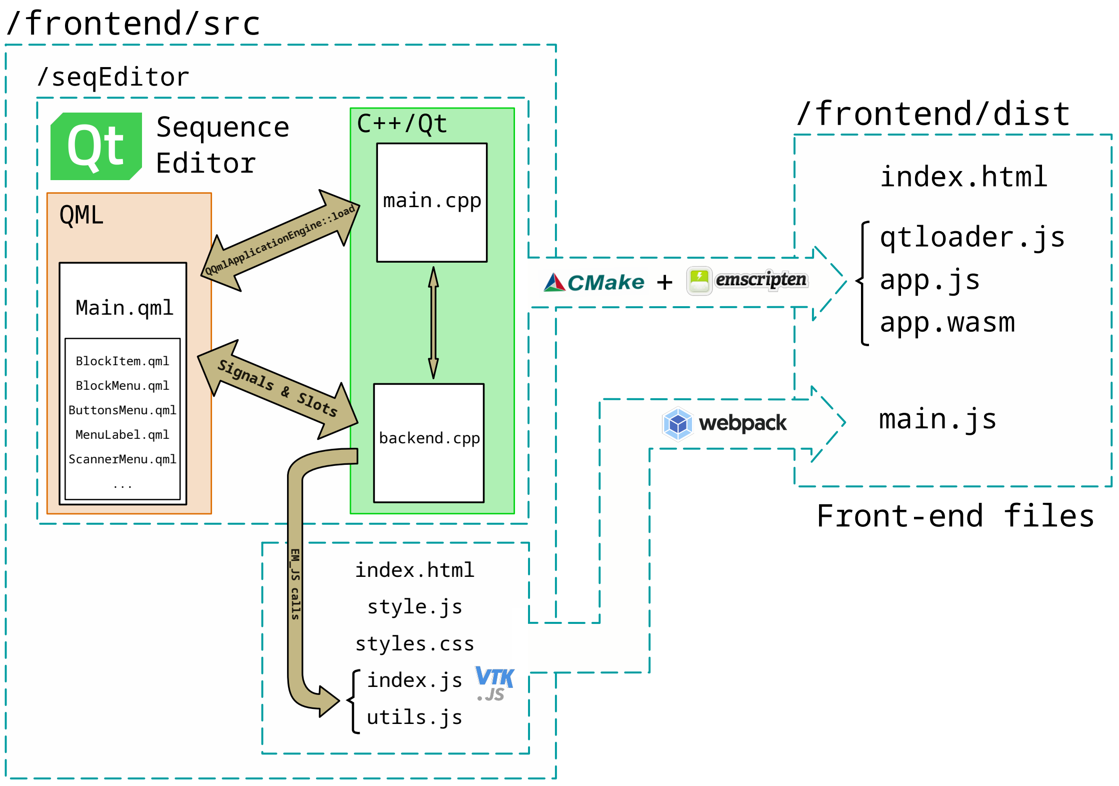


Figure 3.18: Internal communication scheme between front-end components. The `/frontend/dist` directory is referred to as "Front-end files" in Figure 3.14, and serves as the destination for compiled assets. The server accesses this directory to deliver the necessary files to the client, ensuring proper front-end rendering.

Figure 3.18 illustrates the structure of the `/frontend` directory, which is organized into two primary subdirectories: `/src` and `/dist`. The `/src` folder contains the files and structure detailed in Section 3.2.4, with brown arrows showing the communication between files and technologies. Meanwhile, the `/dist` folder, named “distributable”, holds the files compiled from `/src` to be shared by the server. Different compilation methods are used: CMake and Emscripten for the Qt-based sequence editor, and Webpack for JavaScript files.

Chapter 4

Results and Testing

This Chapter presents the results obtained during the development of the two main components of this Thesis. Accordingly, as with the previous chapter, this one will be divided into two sections, corresponding to the dynamic phantom model and the Web sequence editor, respectively.

Each section follows a different approach when presenting the results, suited to the nature of the two components. For the dynamic phantom model, the focus is on a structured evaluation which first demonstrates the capabilities of the implementation with practical examples, and then performs a more analytical evaluation comparing accuracy, performance and other relevant metrics with specific experiments from other MRI simulation contributions. For the Web interface, the focus is on a practical validation approach, which aligns with software engineering practices. Instead of scientific comparisons, this section focuses on usability testing, practical use cases, and validation of the functionalities in real-world scenarios.

4.1 Dynamic Phantom Model

To evaluate all the components related to the dynamic phantom model developed within KomaMRI, this section demonstrates how users can interact with the implemented functionalities, how realistic motion-related experiments can be performed, and how the results of these and other experiments validate the accuracy of the dynamic model. First, Section 4.1.1 verifies the ability to define, visualize, and store dynamic phantoms through simple pilot experiments. Beyond these basic validations, a series of experiments were conducted, structured around two main evaluation approaches:

1. **Demonstrative Evaluation**. Presented in Section 4.1.2, this approach showcases the flexibility and ease of defining and simulating dynamic phantoms by recreating motion-related MRI acquisitions—such as cardiac cine, TOF, and PC-MRI—which are commonly performed on real systems. Rather than direct comparisons with other tools, these evaluations focus on demonstrating the model's capabilities. This work has led to the publication of three conference communications [76–78].

2. **Comparative Validation**. Covered in Section 4.1.3, this approach assesses the accuracy and performance of the implemented model. Three specific experiments were conducted, which compare the dynamic phantom model both qualitatively and quantitatively against previously existing MRI simulation tools.

As in the previous chapter, all the scripts used to obtain the results have been included in a public GitHub repository[1] and may also be part of the official KomaMRI documentation[2].

[1]`https://github.com/pvillacorta/KomaMotionExperiments`.

[2]`https://juliahealth.org/KomaMRI.jl`.

4.1.1 Definition, Visualization and Storage of Dynamic Phantoms

This section aims to verify three key aspects of the functionalities implemented in Section 3.1. First, it verifies the ability to define and combine motions within the digital phantom, a feature which directly results from Section 3.1.3. Simultaneously, it assesses the performance and utility of the enhanced dynamic phantom visualization tool, developed as described in Section 3.1.4. Finally, it demonstrates how the defined dynamic phantoms can be stored and loaded using the `.phantom` file format, which was outlined in Section 3.1.5. Several tests have been performed, where different types of motion have been added to the phantom and visualized at various time frames using the enhanced KomaMRI viewer.

In all the cases, the initial phantom is a hollow cube with a side length of 1 mm and a spin spacing of 10 µm, centered at the origin with its faces aligned with the coordinate axes. Each face of the cube is assigned a different T_1 value to facilitate the visualization of the motions.

a) Translation motion

As shown in Code 4.1, a translational motion has been added, with components of 0.5, 0.6 and 0.7 mm along the three spatial directions, respectively. This motion, which lasts for 1 second, has been defined to affect the entire phantom. Figure 4.1 illustrates the phantom at three different time points. The bottom slider allows users to position the phantom at the desired time point and observe its exact position at that moment. Notably, without the improvements made to the phantom viewer, it would not be possible to visualize its temporal evolution. This is mentioned here as it is the first section where this feature is showcased, but it naturally applies to all subsequent sections as well.

```
obj.motion = Translate(5e-4, 6e-4, 7e-4, TimeRange(0.0, 1.0), AllSpins())
```

Code 4.1: Assignment of a translational motion to the phantom.

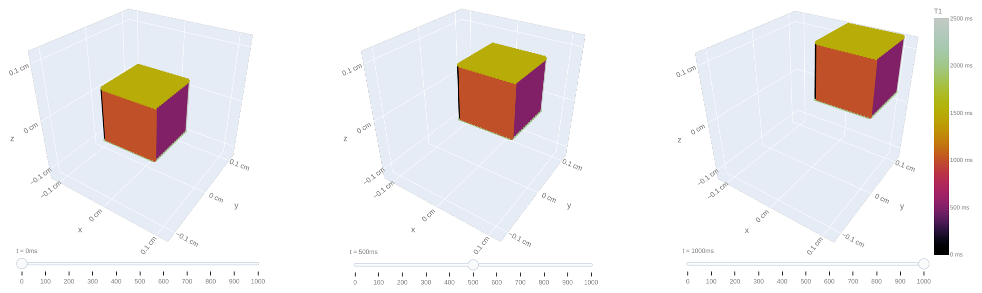


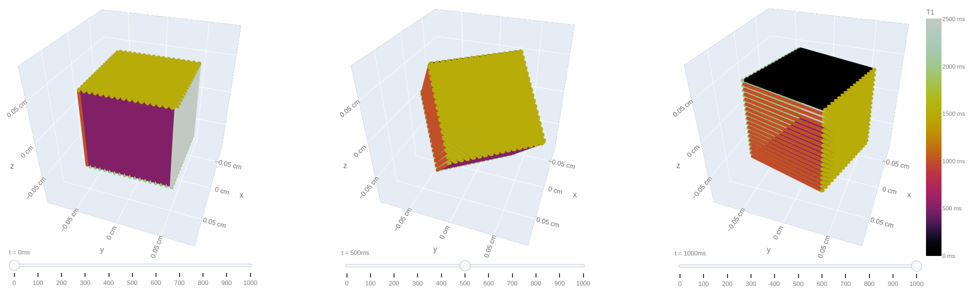
Figure 4.1: Cube phantom undergoing translational motion in the three axes. Three time instants are displayed. (a) $t = 0$. (b) $t = 500$ ms. (c) $t = 1$ s.

b) Rotation motion

Code 4.2 adds a rotational motion to the phantom, with 90º and 75º angles around the y and z axes, respectively. This motion also lasts for 1 second and affects all the spins in the phantom. Figure 4.2 shows the effect of this motion at three time points.

```
obj.motion = Rotate(0.0, 90.0, 75.0, TimeRange(0.0, 1.0), AllSpins())
```

Code 4.2: Assignment of a rotational motion to the phantom.



(a) (b) (c)

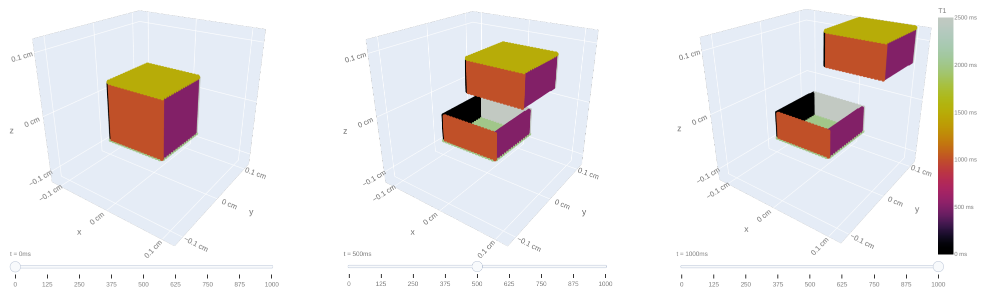
Figure 4.2: Cube phantom undergoing rotational motion around y and z. Three time instants are displayed. (a) $t = 0$. (b) $t = 500$ ms. (c) $t = 1$ s.

c) Addition of motion to a phantom subset

As shown in Code 4.3, it is possible to assign the motion to a specific part, rather than to the entire phantom. This is done by means of the **SpinRange** structure, where the range of the spin indices to be affected has to be specified. Figure 4.3 shows the result of a translation motion applied to the upper half of the phantom, which consists of a range of spins approximately from 30,000 to 60,000.

```
obj.motion = Translate(5e-4, 6e-4, 7e-4, TimeRange(0.0, 1.0), SpinRange(30000:length(obj)))
```

Code 4.3: Assignment of a translational motion to a phantom subset.

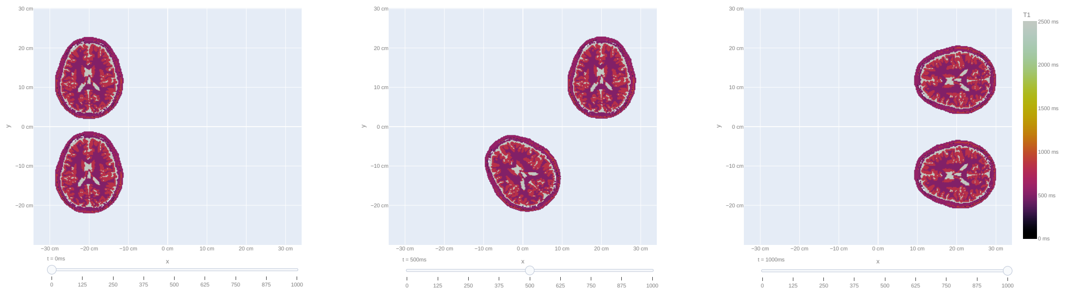


(a) (b) (c)

Figure 4.3: Cube phantom subset undergoing translational motion in the three axes. Three time instants are displayed. (a) $t = 0$. (b) $t = 500$ ms. (c) $t = 1$ s.

d) Motion combination

Motions can be freely added to the phantom, each defined by its type, time span, and range of spins. This enables motions to either overlap in time, influencing the phantom simultaneously, or occur in succession, with one motion transitioning into the next. Both scenarios are supported, allowing for complex combinations of effects on different spin ranges and over varying time spans. Figure 4.4 illustrates two brain phantoms subjected to the same translational and rotational motions but with different time spans. In the upper phantom, the translation is applied from 0 to 0.5 seconds, while the rotation spans from 0.5 to 1 second. In the lower phantom, both motions share the same time span, from 0 to 1 second. Additionally, this example demonstrates how phantom addition operates.



(a) (b) (c)

Figure 4.4: Comparison of two brain phantoms with identical translational and rotational motions but different time spans. Three time instants are displayed. (a) $t = 0$. (b) $t = 500$ ms. (c) $t = 1$ s.

e) Saving and Loading Phantoms

The implemented `write_phantom` and `read_phantom` functions allow users to easily store and retrieve phantoms from files with the `.phantom` extension. Code 4.4 illustrates the two simple lines required to save and load a phantom, respectively.

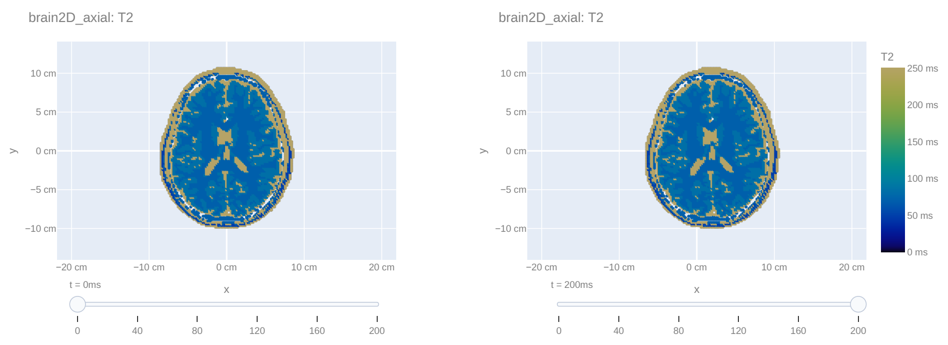
```
write_phantom(obj, "example.phantom")
obj2 = read_phantom("example.phantom")
```

Code 4.4: Writing and reading a phantom to/from a `.phantom` file

4.1.2 Demonstrative Evaluation

a) Simulation of patient motion and motion-corrected reconstruction

Examinating the effects of patient motion during an MRI acquisition can be both useful and insightful. This experiment simulates translational motion during a brain MRI scan to evaluate its influence on the acquisition process. Specifically, the starting point is the 2D example phantom provided by KomaMRI (`brain_phantom2D`), to which a translational motion of 2 cm along the x axis has been added by means of a `Motion` instance including a `Translate` action. The motion duration is set to 200 ms, resulting in a velocity of 0.1 m/s. Figure 4.5 illustrates the phantom at the initial and final time instants of the motion.



(a) (b)

Figure 4.5: Brain phantom undergoing translational motion, visualized using the interactive viewer of KomaMRI. T_2 map is displayed. (a) Initial time point ($t = 0$), with the phantom positioned at the origin $(0, 0, 0)$. (b) Final time point ($t = 200$ ms), after the phantom has translated 2 cm along the x axis.

Regarding the pulse sequence, we have chosen an EPI sequence with an acquisition duration of 184 ms, comparable to the motion duration. Thus, the phantom will move during the whole acquisition, and motion-induced artifacts will be noticeable in the reconstructed image (Figure 4.6a). The severity of the artifacts can vary depending on the acquisition duration and the K-space trajectory.

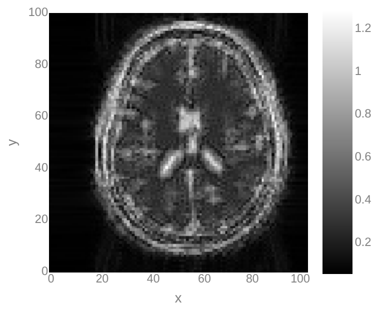
Motion-corrected Reconstruction In order to correct for the motion-induced artifacts, a motion-corrected reconstruction can be performed. This can be achieved by multiplying each sample of the acquired signal $S(t)$ by a phase shift $\Delta\phi_{\text{corr}}$ proportional to the displacement $\boldsymbol{u}(t)$ [79]:

$$S_{\text{MC}}(t) = S(t) \cdot \mathrm{e}^{i\Delta\phi_{\text{corr}}} = S(t) \cdot \mathrm{e}^{i2\pi\boldsymbol{k}(t)\cdot\boldsymbol{u}(t)} \tag{4.1}$$

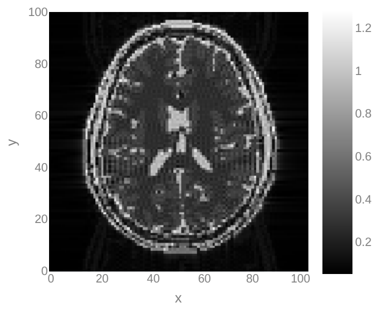
In practice, an estimation or measurement of the motion would be necessary before performing a motion-corrected reconstruction. However, for this example, the displacement functions $\boldsymbol{u}(\boldsymbol{r}, \mathrm{t})$ defined by `obj.motion` are used directly. Since translation is a rigid motion there is no position dependence, i.e., $\boldsymbol{u}(\boldsymbol{r}, t) = \boldsymbol{u}(t)$, and the required displacements can be obtained by calculating $\boldsymbol{u}(\boldsymbol{r} = 0, t = t_{\text{adc}})$. The necessary phase shift for each sample can now be obtained and applied to the acquired signal to correct its phase. Code 4.5 contains the lines of code that perform these operations and Figure 4.6b displays the result of the motion-corrected reconstruction. This complete example has been included in the KomaMRI documentation[3].

```
sample_times = get_adc_sampling_times(seq1)
displacements = hcat(get_spin_coords(obj.motion, [0.0], [0.0], [0.0], sample_times)...)
_, kspace = get_kspace(seq1)
ΔΦ = 2π*sum(kspace .* displacements, dims=2)
acq1.kdata[1] .*= exp.(im*ΔΦ)
image2 = reconstruction(acq1, reconParams)
```

Code 4.5: Displacements acquisition and motion-corrected reconstruction.



(a)



(b)

Figure 4.6: Two reconstructions from a brain acquisition undergoing translational motion in the x direction. (a) Original image with motion-induced artifacts. (b) Motion-corrected reconstruction.

b) Cardiac cine over an XCAT phantom

The phantoms used for simulations can be imported from external sources, such as the Extended Cardiac-Torso (XCAT) [48] model. In this experiment, a dynamic heart phantom with non-rigid motion is generated, and a cine sequence is applied to it for simulation.

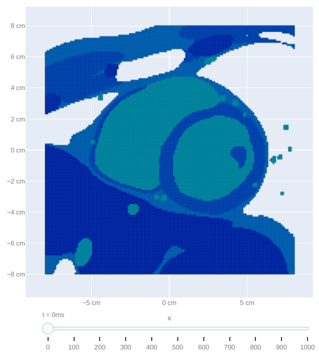
XCAT was used[4] to generate 3D images which correspond to the parametric maps of the heart across five phases distributed throughout a cardiac cycle. Specifically, the image which corresponds

[3] https://juliahealth.org/KomaMRI.jl/stable/tutorial/05-SimpleMotion/

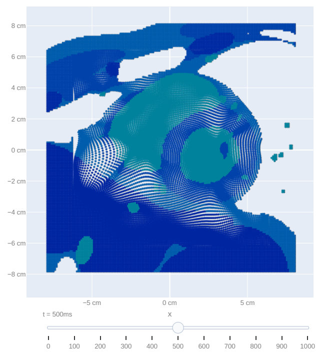
[4] Thanks to Rosa María Menchón-Lara for obtaining the XCAT cardiac maps.

to phase 1 was acquired and, for the remaining phases, the displacements of each voxel were obtained from the displacement vectors provided by XCAT. Then, by establishing a one-to-one voxel-to-spin correspondence, and knowing the voxel width Δx, the parametric maps can be reshaped into vectors, thus filling the initial position (`x`, `y`, `z`) and contrast-related (`T1`, `T2`, `T2s`, ρ) fields of the `Phantom` structure in KomaMRI. Regarding the displacements, they are stored in a `Path` structure as three matrices: `dx`, `dy`, and `dz`. Each matrix has N_{spins} rows and N_{phases} columns, where N_{phases} corresponds to the five cardiac phases in this case. The heart rate was set to 60 bpm.

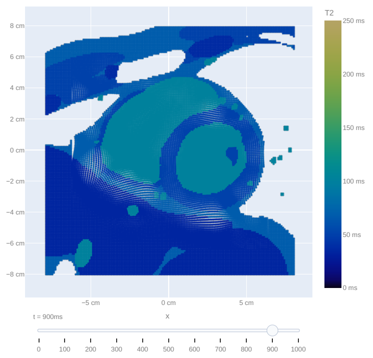
Figure 4.7 illustrates the resulting phantom in KomaMRI at three different time instants within the cardiac cycle. As the cycle progresses, the spins gradually deviate from their evenly spaced, mesh-like pattern. This effect is particularly pronounced during end-systole ($t \simeq 0.5$s), where the displacements relative to the initial diastolic positions reach their maximum. As a result, regions with spin accumulation emerge, alongside areas with reduced spin density.



(a)



(b)



(c)

Figure 4.7: T_2 map of an XCAT dynamic heart phantom. Three time instants are displayed. (a) $t = 0$, end-diastole phase. Since there is a direct voxel-to-spin correspondence and no motion has occurred yet, the spins are distributed as in the XCAT image, forming a structured mesh. (b) $t = 0.5s$, end-systole phase. Displacements cause the spins to change positions and the spins are no longer equally spaced, leading to regions of spin accumulation and depletion. (c) $t = 0.9s$. Moments before the end of the cycle, the spins are once again on track to form a regular mesh pattern.

The original XCAT images had a field of view (FOV) of 16 x 16 x 16 cm and a voxel size of 1 mm. This corresponds to matrices with 4,096,000 elements each. To reduce the number of elements —i.e., spins— and prevent unnecessary simulation overload, a 1 cm-high subset along the z dimension was extracted from each matrix. Only elements with non-zero PD were included in the phantom. This process resulted in a total of 179,688 spins.

The 2D cine acquisition consisted of 10 cardiac phases with a matrix size of 128 x 128 pixels. The sequence used was a b-SSFP with a TR of 100 ms, a flip angle of 4º, and 1 views per segment (vps) (see Section 2.1.3c). FOV was set to 15 cm. The simulation was completed in 80 seconds using the desktop computer described in Section 1.4. Notice that the number of simulated segments is independent of the number of frames that the phantom consists of.

Figure 4.8 presents 6 out of the 10 acquired frames. For the frames 4-8, which correspond to systole, an increase in the image intensity is observed in regions where spins accumulate, leading to a relative decrease of the intensity in the rest of the image. These intensity peaks occur primarily inside both blood pools, as they are the regions where most spins accumulate. Consequently, the combined contributions of all spins to the MR signal are higher within these areas. This effect is not realistic, as blood is an incompressible fluid, and this phantom model incorrectly treats it as compressible. This gives rise to future lines of work focused on how to extract more realistic phantoms and spin trajectories from image datasets, such as those produced by XCAT.

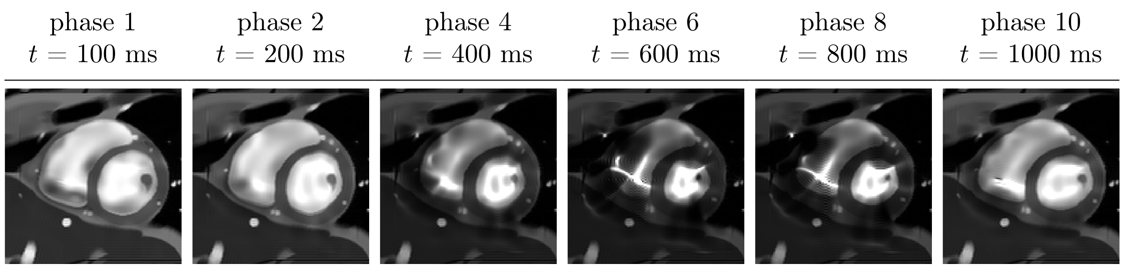


Figure 4.8: Six of the ten acquired frames during a 2D cine simulation over a dynamic heart XCAT phantom. In the systole frames (4–8), intensity increases in regions with spin accumulation, particularly within the blood pools.

c) Time of Flight (TOF) acquisition over a user-defined flow phantom

To demonstrate and evaluate the flow functionality, this experiment performs a TOF cine acquisition over a user-defined flow phantom. The phantom, shown in Figure 4.9, is modelled as a vertical cylindrical tube, i.e., a tube with its main axis located along the z axis, with outer radius of 10 mm, and an inner radius of 4.5 mm. The tube length is 40 mm. Spins on the walls are static while a continuous, steady, and uniform flow moves in the positive z direction. This is achieved by means of a `Periodic` time span, in which the particles that leave the tube from the upper boundary are reinjected back through the lower boundary. The flow velocity has been chosen so that a spin that enters the tube at the beginning of a period reaches the upper boundary at the end of the period. The spins separation has been set to 0.3 mm, so that the overall number of spins is 466,722. These spins were assigned T_1, T_2, and PD values as indicated in Table 4.1. The parameter values for the two regions of the phantom were chosen to be quite similar, ensuring that while a slight contrast exists between the two regions due to these differences, the predominant contrast arises from the flow effects.

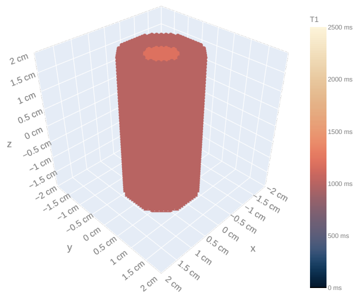


Figure 4.9: Cylindrical phantom used in the experiment.

	PD	T1 (ms)	T2 (ms)
tissue	1.0	1000	42
blood	0.9	1200	92

Table 4.1: PD, T_1, and T_2 values used in the cylindrical phantom. Two distinct tissues are identified, referred to as "tissue" and "blood", corresponding to the phantom walls and the flow circulating within it, respectively.

Code 4.6 illustrates the generation of the phantom. The code for the definition of the grid has been omitted. Once the grid is created, variable `bl` is a boolean 3D array that indicates which spins are located within the inner radius, i.e., within the hollow part of the tube. Similarly, the variable `ts` is another boolean 3D array that indicates which spins belong to the outer static tissue. Variable `dz` is initialized with the absolute positions of the spins along the simulation of the period. This lets us define the `spin_reset` array which contains for every spin the time position at which the spin leaves the tube through its upper boundary. The last two sentences that modify `dz` reinject the spins that leave the tube and then the initial spins positions are subtracted so that `dz` contains displacements, as expected. Finally, the `Phantom` structures are defined; as can be observed, the flow was specified by means of the `FlowPath` structure with a `Periodic` time span. The complete phantom, `obj`, is then obtained from the sum of the `tissue` and `blood` phantoms.

A cine sequence based on a b-SSFP was used with a TR of 30 ms and a flip angle of 50º. The cine consisted of 30 frames, with a FOV of 40 mm and a matrix size of 100 x 100 pixels.

```
# Geometrical parameters and grid generation:
R = 10e-3;    r = 4.5e-3;    L = 4e-2;    Δx = 3e-4
x, y, z = (...) # 3D cartesian grid, based on the previous parameters
bl = Bool.( x.^2 .+ y.^2 .<= (r)^2)
ts = Bool.((x.^2 .+ y.^2 .<= (R)^2) - (x.^2 .+ y.^2 .<= (r)^2))
# Displacements and spin_reset matrices:
Nt = 500;    d_max = L;    v = 4e-2 # 4 cm/s
dx = dy = zeros(length(z[bl]), Nt)
dz = z[bl] .+ cumsum(d_max/Nt .+ zeros(1,Nt), dims=2)
spin_reset = dz .> d_max/2
for i in 1:size(spin_reset, 1)
    idx = findfirst(x -> x == 1, spin_reset[i, :])
    if idx !== nothing
        spin_reset[i, :]  .= 0
        spin_reset[i, idx] = 1
    end
end
dz[dz .> d_max/2] .-= d_max
dz .-= z[bl]
# Phantom creation:
tissue = Phantom(
    name="Tissue", x=x[ts], y=y[ts], z=z[ts],
    T1=fill(1.0,length(x[ts])), T2=fill(0.042,length(x[ts])), PD=fill(1.0,length(x[ts])))
blood = Phantom(
    name="Blood", x=x[bl], y=y[bl], z=z[bl],
    T1=fill(1.2,length(x[bl])), T2=fill(0.092,length(x[bl])), PD=fill(0.9,length(x[bl])),
    motion=FlowPath(dx, dy, dz, spin_reset, Periodic(d_max/v, 1.0)))
obj = tissue + phantom
```

Code 4.6: Flow cylinder phantom generation.

Figure 4.10 presents the results of this TOF experiment. Two acquisitions, which correspond to separate simulations, were performed with different orientations. Each of the two simulations took around 5 minutes when performed on the desktop computer described in Section 1.4. The first row of the table corresponds to axial slices in the xy plane, while the second row represents a longitudinal acquisition in the zx plane. Neither plane is offset from the origin, and as a result, they intersect along the x axis. The TOF effects encountered in 2D GE sequences are noticeable and increase as the frames progress. These effects are caused by the "fresh" spins reinject into the bottom entry of the phantom, which present a higher signal intensity than the static, saturated spins.

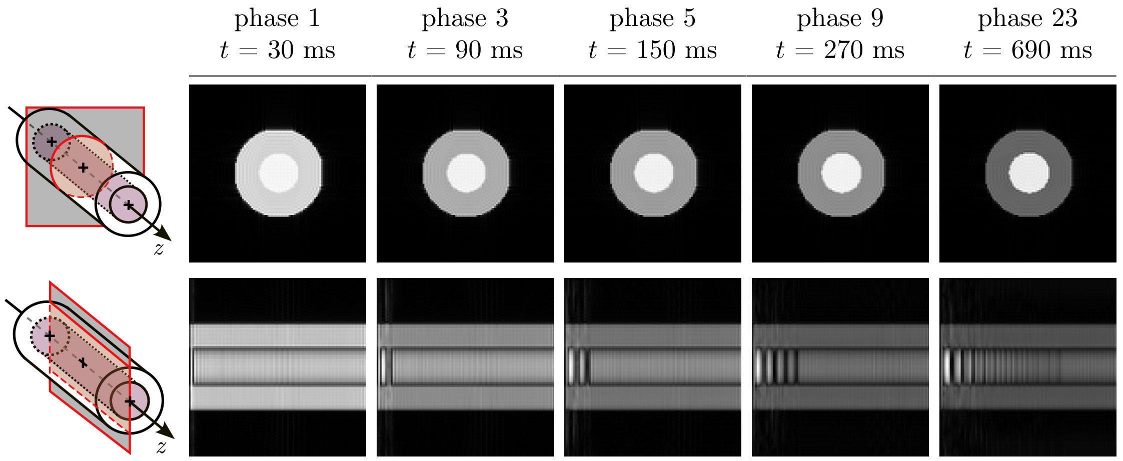


Figure 4.10: Five of the 30 acquired phases, which reflect the steady state reached by the wall spins. These spins emit less signal compared to the fresh spins injected through the lower entry of the phantom, which correspond to the left side of the frames in the second row. The flow velocity is 4 cm/s.

This phenomenon is particularly evident in the axial frames. In the selected slice, the flow spins within the central region are renewed with each TR. In contrast, the spins in the phantom walls undergo repeated excitations and eventually reach a steady state, limiting their maximum signal compared to that of the flow spins. As a result, the flow appears progressively brighter as the frames advance.

For the longitudinal slices, it can be observed how the flow moves from left to right. In the initial phase, both the walls and the flow exhibit similar image intensity, and the contrast between these two tissues arises solely from the differences in the T_1, T_2, and PD parameters. As phases progress, both the wall and flow spins that were already present in the selected slice during previous TRs eventually reach a steady state. Therefore, only the fresh spins which enter from the left emit a more intense MR signal.

Two additional effects can be observed in the longitudinal slice. The first is the appearance, within the flow region, of a pattern of vertical bands alternating between higher and lower intensity. This effect is caused by the alternation of the RF pulse phase in the b-SSFP sequence, which occurs every TR to maintain the coherence of the transverse magnetization and keep a constant accumulated phase. The RF phase alternation, combined with the motion, generates periodic changes in signal intensity, which manifest as alternating bands. The second effect involves artifacts related to the flow motion in the readout direction. These artifacts appear as vertical lines, perpendicular to the flow, which distort and blur the obtained images. Both the intensity of these artifacts and the width of the aforementioned bands increase with higher flow velocity (Figure 4.11) and longer TRs.

This experiment, and particularly the cine in the longitudinal slice, is not reproducible in a real-world scenario, as any conduit through which a fluid circulates must be part of a closed circuit. In such a case, exciting a plane parallel to the flow would result in the excitation of all spins within that plane[5], not just those within the FOV. The fact that this experiment can only be replicated in MRI simulators, specifically in KomaMRI, makes this tool a valuable and essential resource for the observation and understanding of effects such as TOF.

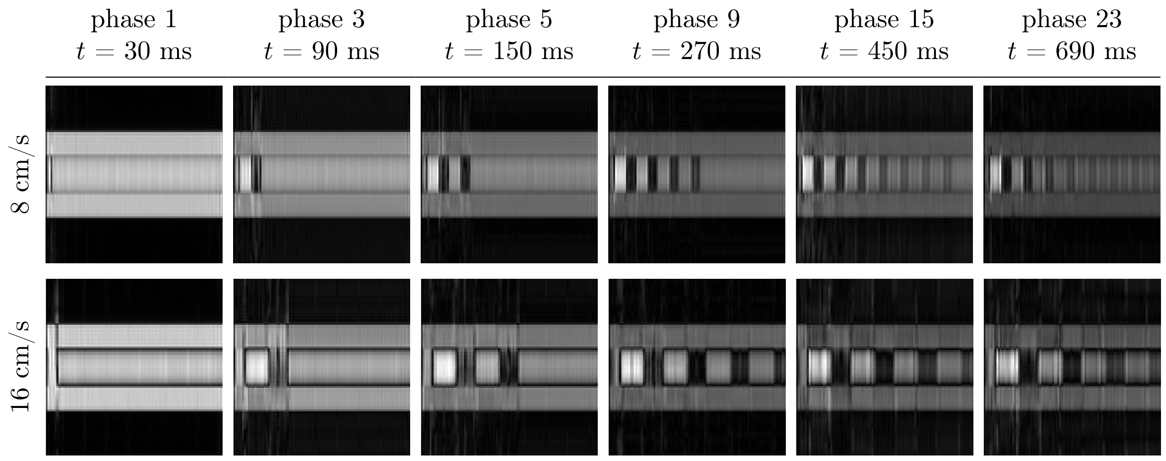


Figure 4.11: Comparison of two sets of longitudinal cine frames acquired with the same sequence used in Figure 4.10. Both the width of the bands and the intensity of the motion-related artifacts increase with the flow velocity.

d) Phase Contrast (PC) imaging of a user-made phantom and a realistic aorta

As stated in Section 2.1.3d, PC imaging encodes flow velocity, making it proportional to the intensity of the acquired images. Two PC-MRI experiments, performed in KomaMRI, are presented below. The first experiment aims to evaluate the feasibility of PC acquisitions in the simulator. To this end, and for simplicity, a user-made flow phantom was used. Regarding the second experiment, its goal is

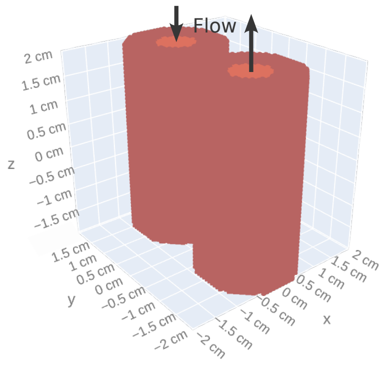
[5]Actually, not all spins in the plane would be excited, but rather those influenced by the sensitivity profile of the RF coil. These profiles never exhibit discontinuities or abrupt changes but instead feature smooth transitions, making it impossible to define a clear boundary between excited and non-excited spins.

to ensure that PC-MRI remains effective when handling a more complex and realistic scenario, while also showcasing how these complex anatomies, along with their motion-related information, can be defined in the simulator. The phantom model used in this case is an aorta.

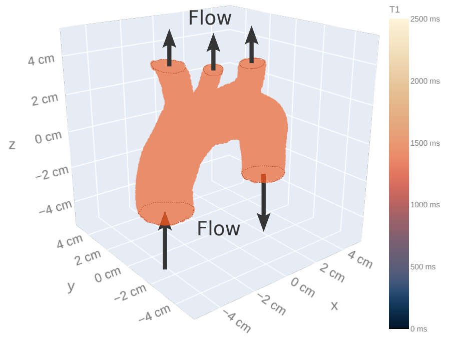
User-defined flow phantom experiment Figure 4.12a illustrates the phantom used in the first experiment, derived from the one described in Section 4.1.2c. It consists of two vertical cylindrical tubes through which a continuous flow moves along the z axis with a velocity of 10 cm/s, in opposite directions. Code 4.7 demonstrates how this new phantom can be created from the `obj` phantom generated in Code 4.6. Specifically, two copies of the original object are made, and each is shifted along the y axis by a distance equal to the cylinder's outer radius, plus a small offset. Then, a 180º rotation around the y axis is applied to the second copy, by means of a `Rotate` structure, reversing the direction of the flow. As a result, the copy located at positive y values has flow in the -z direction, while the copy at negative y values has flow in the +z direction. The total phantom size is 2 x 466,722 = 933,444 spins.

```
obj1 = copy(obj)
obj2 = copy(obj)
obj1.y .-= R + 1e-3
obj2.y .+= R + 1e-3
obj2.motion = MotionList(obj2.motion, Rotate(0.0, 180.0, 0.0, TimeRange(0.0,  1e-7)))
obj = obj1 + obj2
```

Code 4.7: Double flow cylinder phantom generation.



(a)



(b)

Figure 4.12: Flow phantoms used in the PC-MRI experiments. The black arrows indicate the direction of the flow. (a) User-made phantom. (b) Aorta phantom.

The pulse sequence used is a gradient echo, with a bipolar gradient added between each RF pulse and the corresponding dephasing and readout gradients. These PC bipolar gradients have been configured to achieve a V_{ENC} of 15 cm/s. The TE and TR times were set to 10 ms and 60 ms, respectively, with a flip angle of 15º. The FOV is 45 mm, with a matrix size of 100 x 100 pixels.

The results of this experiment are displayed in Figure 4.13. Note that, as explained in Section 2.1.3d, two separate acquisitions (A and B) have been required to obtain a resulting phase image ($\Delta\phi$) ranging from -π to π. These two acquisitions differ only in the polarity of the bipolar pulses in the sequence. Each acquisition took 3.5 minutes on the desktop computer described in Section 1.4. As for the magnitude image, it was obtained by means of the magnitude average of both acquisitions. The phase information in the ϕ_A and ϕ_B images ranges from 0 to π rad, with a phase offset[6] of $\pi/2$. For image ϕ_A, spins moving towards negative z have a phase between 0 and $\pi/2$, while spins moving

[6]The $\pi/2$ phase offset is caused by the real, positive RF pulse, which induces a rotation of the magnetization vector around the x axis.

towards positive z present a phase between $\pi/2$ and π. For image ϕ_B, the opposite occurs: spins moving towards negative z have a phase between $\pi/2$ and π, while spins moving towards positive z have a phase between 0 and $\pi/2$. By obtaining the phase difference image $\Delta\phi = \phi_A - \phi_B$, the aforementioned $\pi/2$ phase offset is cancelled out. Spins moving towards negative z acquire a phase between -π and 0, while spins moving towards positive z now have a phase ranging from 0 to π.

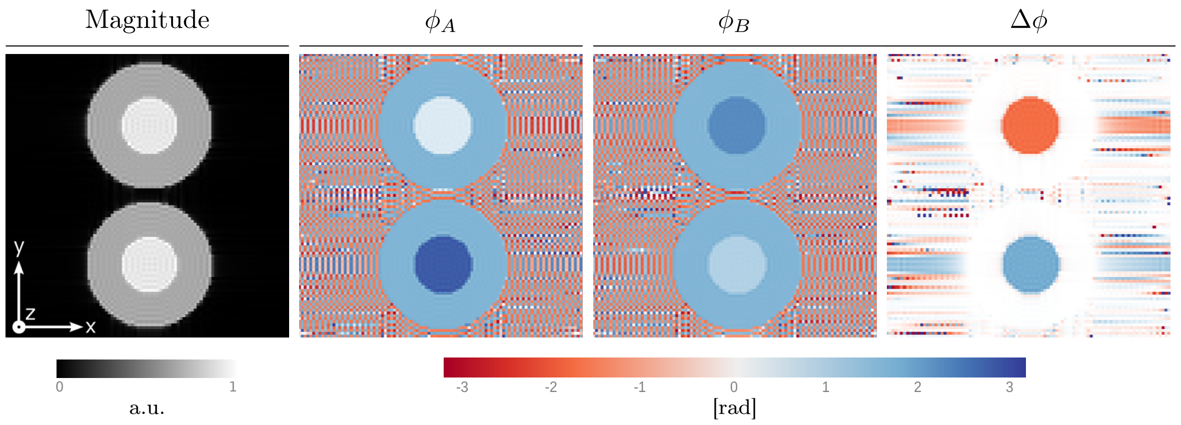


Figure 4.13: Results of the PC-MRI experiment performed on a user-made flow phantom. The magnitude image was obtained by averaging the magnitudes from acquisitions A and B, while the phase image $\Delta\phi$ comes from subtracting the phases ϕ_A and ϕ_B. Flow velocity is 10 cm/s and V_{ENC} is 15 cm/s.

These results demonstrate how flow information can be captured using the simulator. The phase difference ($\Delta\phi$) image clearly shows the flow direction and velocity, with a phase range from -π to π. This experiment highlights the potential of KomaMRI for the simulation of PC-MRI techniques, and provides a solid foundation for studies with more complex anatomies.

Aorta phantom experiment The second experiment performs a PC acquisition on an aorta phantom. The goal is to acquire two specific slices of the aorta: an axial slice, perpendicular to the ascending and descending sections, and a sagittal slice, capturing the aorta's profile. For each of these two slices, the objective is to generate three flow velocity maps, which correspond to the x, y, and z directions.

The phantom used is shown in Figure 4.12b. The anatomical and flow information for this aorta is extracted from the Vascular Model Repository (VMR)[7], and the trajectories of each spin in the phantom are obtained following the methodology outlined in Appendix C. A random seeding of 2,000,000 particles, i.e., spins, is then performed throughout the volume of interest (Figure 4.14), and their trajectories are calculated iteratively for each time step using VTK-m. These trajectories are resolved between t = 0 and t = 1 s, with a time step of 0.5 ms. From these trajectories, one out of every twenty positions has been stored in the resulting `.phantom` file, ensuring sufficient temporal resolution while keeping the file size manageable, around 3 GB. As a result, we obtain a total of 100 positions per particle. Additionally, for each time step and each particle, a position check is performed to ensure it remains within the volume of interest. If a particle is found outside this FOV, it means that the particle has exited through one of the exit conduits —the descending aorta, carotid arteries or subclavian arteries— and needs to be reseeded in the inlet conduit —the ascending aorta— in order to maintain a constant number of particles over time. Furthermore, a flag is used to track the precise time instant at which each particle is reseeded.

All this information concerning spin trajectories and their corresponding reset flags, along with the phantom-specific data —such as T_1, T_2, PD, etc.— is stored in a `.phantom` file, which can is read in KomaMRI and then converted into a `Phantom` structure. T_1, T_2, and PD have been set to 1400 ms, 200 ms and 1, respectively. The trajectories in x, y, and z, and the reset flags have been stored as four matrices, each with 2,000,000 x 100 elements, and then encapsulated within a `FlowPath` structure

[7]Vascular Model Repository, available in https://www.vascularmodel.com/.

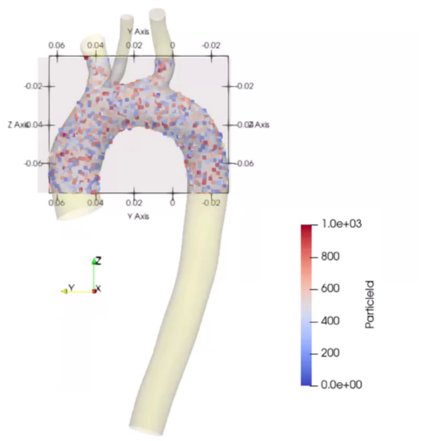


Figure 4.14: Spin trajectory generation for a synthetic aorta CFD dataset at a certain FOV.

(defined in Section 3.1.3c).

The pulse sequence used in this case is a GE-EPI multi-shot with an ETL of 16, TE and TR times of 16 ms and 60 ms, respectively. PC bipolar gradients are applied after each RF pulse, configured to achieve a V_{ENC} of 50 cm/s. The matrix size is 128 x 128, with a FOV of 110 mm. For the axial slice, the flip angle is set to 50º, and the RF pulse frequency offset is -2kHz, which results in the excited slice being displaced along the negative z axis from the point (0, 0, 0). For the sagittal slice, the flip angle is 5º, with a frequency offset of 0.8 kHz which displaces the selected slice along the positive y axis.

Since six $\Delta\phi$ images —which are obtained by subtracting ϕ_A and ϕ_B— need to be generated, a total of 12 simulations were conducted. Each simulation took 4 minutes on the desktop computer described in Section 1.4.

Figure 4.15 presents the results of the experiment. Note how the axes have changed with respect to Figure 4.14, as a 90º rotation has been applied to the phantom before the MRI simulation, around

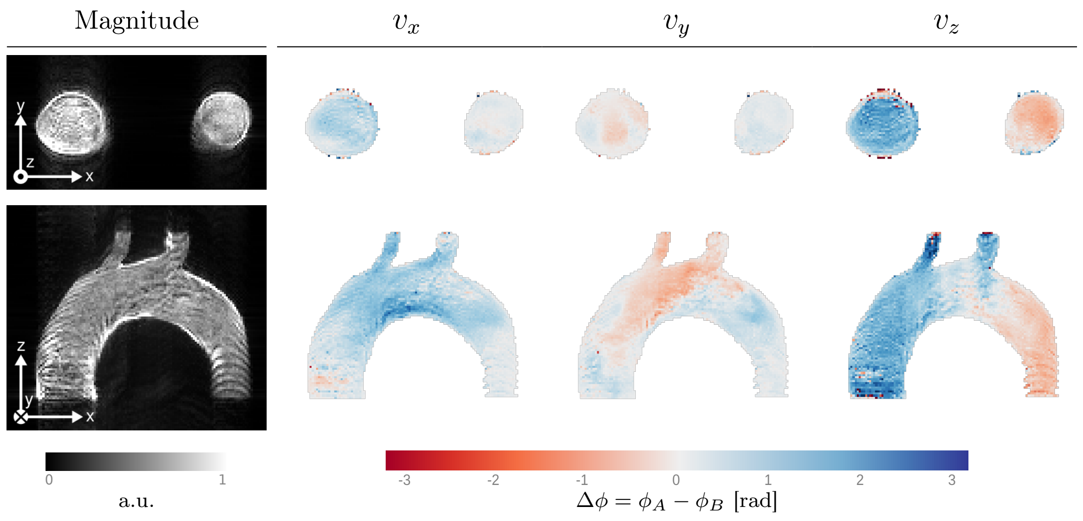


Figure 4.15: Results of the PC-MRI experiment performed on a flow aorta phantom. In the magnitude image, especially in the sagittal slice, the increased brightness and contrast at the edges of the aorta indicate the accumulation of spins along the vessel walls. In the phase images, the colour represents the flow direction in each axis, while the intensity reflects its magnitude relative to the V_{ENC} of 50 cm/s (i.e., with respect to $\Delta\phi = \pi$). Accordingly, the v_z component exhibits the highest values, while the v_y component shows the smallest values.

the z axis, and following the right-hand rule. The first row contains the axial images, which have been cropped to include only the significant region. The second row displays the sagittal images. Regarding the columns, the first one shows the magnitude images, obtained from an acquisition with no velocity encoding[8]. The next three columns show the phase images $\Delta\phi$ that result from encoding the flow velocity in the x, y, and z directions, respectively. Remember that the choice of which velocity component is encoded depends on the direction of the PC bipolar gradients. Furthermore, a mask has been applied to these six phase images, which excludes all pixels where the magnitude image does not exceed a certain intensity threshold.

In the magnitude images, the increased brightness and contrast at the vessel edges, particularly in the sagittal slice, highlight the accumulation of spins along the aortic walls. In the phase images, the colour indicates the flow direction for each velocity component, while the intensity reflects the relative magnitude of the flow with respect to the V_{ENC} of 50 cm/s. Among the velocity components, v_z exhibits the largest values, whereas v_y shows the smallest, consistent with the flow geometry of the phantom.

4.1.3 Comparative Validation

a) Myocardial tagging over a user-defined phantom

The work of Tecelao et al. [80] presents a cardiac deformation model for the myocardium, which was later used by Xanthis et al. [11] in a cardiac MRI simulation experiment. This experiment has been replicated in KomaMRI in order to compare the results with those of [11], both in terms of image similarity and execution times.

The digital phantom described in [11] has also been used in our experiment to maintain consistency in the setup. It is a homogeneous cylindrical 3D object, with a size of 100 mm x 100 mm x 80 mm. The inner diameter is 50 mm and the isochromat volume size is 0.25 mm x 0.25 mm x 1 mm, resulting in a total of 7,632,792 spins. T_1 and T_2 times were set to 900 ms and 50 ms, respectively [11].

For the deformation motion, the mathematical model presented in [80] was applied on the cylinder. This model accounts for longitudinal and radial contraction, axial torsion and rigid body displacement [11], and defines the spin positions in every point of the cylinder by means of the following analytical expressions in cylindrical coordinates[9] [80]:

$$r = \sqrt{r_i^2 + \frac{(R^2 - R_i^2)}{\lambda}} \tag{4.2}$$

$$\theta = \phi R + \Theta + \gamma Z + \varepsilon \tag{4.3}$$

$$z = \omega R + \lambda Z + \delta \tag{4.4}$$

where (r, θ, z) indicate the position of tissue in the deformed state and (R, Θ, Z) represent its position prior to the deformation. R_i and r_i are the inner radii of the cylinder before and after the deformation, respectively. λ refers to the longitudinal contraction, ϕ to the $R-\Theta$ shear, γ to the axial torsion, ε to the rigid body rotation, ω to the $R-Z$ shear, and δ to the rigid body displacement [11]. The values of these parameters have been chosen to represent human heart motion. Specifically, according to [80], the values are as follows: $\lambda = 1$, $\phi = 0.556$, $\gamma = 0.6$, $\varepsilon = 18.334$, $\omega = 0.278$, and $\delta = 4.167$ [11].

The time evolution of the phantom, from its undeformed to its deformed state, is periodic with a period of 0.8 s. The profile is similar to that shown in Figure 4.16, extracted from [11], which illustrates the evolution of the inner radius of the cylinder as the period progresses.

Code 4.8 demonstrates the generation, in KomaMRI, of the aforementioned digital myocardium model. First, the parameters that define the geometry are specified. As in Code 4.6, the lines for the definition of the Cartesian grid from these parameters have been omitted. Next, the arrays `R`, `Θ`, and `Z`,

[8]The magnitude images will be nearly identical regardless of whether velocity encoding is applied or not. The only difference will be in the phase images.

[9]γ is here used as an angle expressed in radians; it has nothing to do with the gyromagnetic constant.

which correspond to the initial spin positions in cylindrical coordinates, are generated. After defining the deformation model parameters, Equations (4.2) to (4.4) are applied to the initial positions, yielding the final position arrays `r`, θ, and `z`. These are then converted back to Cartesian coordinates, from which the initial positions are subtracted to obtain displacement vectors instead of absolute positions. The matrices `dx`, `dy`, and `dz` are constructed by the combination of a column of zeros, which represents the null initial displacements, with the column vector of final displacements. Finally, an instance of the `Phantom` structure is created, whose `motion` field is composed of a `Path` structure which contains the three matrices `dx`, `dy`, and `dz`, along with a `TimeCurve` that replicates the time profile observed in Figure 4.16.

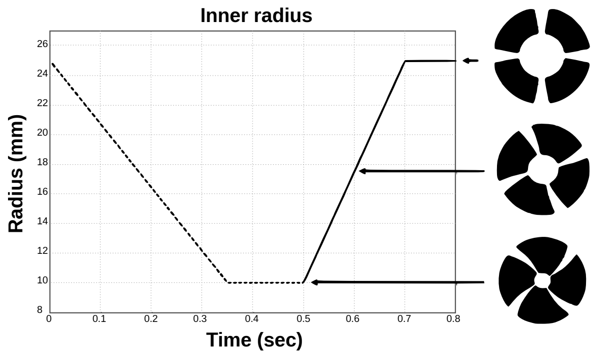


Figure 4.16: Temporal evolution of the inner radius of the heart motion model. Systole is represented by the dashed line and diastole by the solid line. The phases of the heart model are shown on the right, with artificial white radial tags placed to highlight the deformation of the heart during the cardiac cycle [11].

```
# Geometrical parameters and grid generation:
D  = 100e-3;  d  = 50e-3;   L  = 80e-3
Δx = 0.25e-3; Δy = 0.25e-3; Δz = 1e-3
Ro = D/2;     Ri = d/2;     ri = 10e-3
x, y, z = (...) # 3D cartesian grid, based on the previous parameters
# Cylindrical coordinates:
R = sqrt.(x .^ 2 + y .^ 2);   Θ = atan.(y, x);   Z = z
# Deformation model parameters:
λ = 1;   ω = 0.278;   δ = 4.167e-3
Φ, Γ, ϵ = [0.556e3, 0.6e3, 18.334] .* π/180
# Apply the deformation analytical expressions:
r = sqrt.(ri^2 .+ (R .^ 2 .- Ri^2) ./ (λ))
θ = Φ .* R .+ Θ .+ Γ .* Z .+ ϵ
z = ω .* R .+ λ .* Z .+ δ
# Back to cartesian coordinates:
Δxo = r .* cos.(θ) .- x
Δyo = r .* sin.(θ) .- y
Δzo = z .- z
# dx, dy and dz matrices:
Nspins = length(r)
dx = hcat(zeros(Nspins), Δxo)
dy = hcat(zeros(Nspins), Δyo)
dz = hcat(zeros(Nspins), Δzo)
# Phantom creation
obj = Phantom(
    name="3D Ring", x=x, y=y, z=z, T1=T1, T2=T2, ρ=ρ,
    motion = Motion(
        action=Path(dx, dy, dz),
        time=TimeCurve(t=[.0, .35, .5, .7, .8],t_unit=[0, 1.0,  1.0, 0, 0],periodic=true)))
```

Code 4.8: 3D deformable cylinder phantom generation.

As with the digital phantom, the pulse sequence used in the experiment from [11] has been replicated. It consists of a b-SSFP cine sequence with a flip angle of 40º and a TR of 8 ms. The acquisition is segmented along the cardiac cycle —0.8 s—, with a number of phases and vps of 100 and 1, respectively. Therefore, with a K-space matrix of 128 × 128 pixels, 128 cardiac cycles are required to obtain the complete cine. The FOV is 15 cm, and the slice thickness is 10 mm.

Additionally, tagging pulses have been added before each set of views, coinciding with the beginning of each cardiac cycle. Specifically, we have added a 1-1 SPAMM sequence, which consists of two RF pulses with a three-lobe sinc shape, a flip angle of 45º, and a duration of 3 ms each. The intermediate gradient in the x direction has a rectangular shape, a duration of 0.6 ms, and an intensity of 4 mT/m [11]. As for the crusher gradients applied after the second RF pulse, they have been defined with a duration of 1.2 ms and 20 mT/m strength.

In terms of results, Figure 4.17 shows five of the 100 acquired cine frames with the specified sequence and myocardium model. These frames demonstrate the temporal evolution of the contractility of the heart model based on the tagging that has been applied along the readout direction. Contrast reduction due to T_1 relaxation can also be observed throughout the cardiac cycle [11].

The first row of the table displays the images generated by MRISIMUL, directly extracted[10] from Figure 9 of [11]. The second row contains the frames obtained from the KomaMRI experiment. The similarity between both sets of frames is evident. Any minor differences, such as the spacing between tagging lines, the phantom size, or the rotation angle, can be attributed to slight deviations in the definition of the phantom and the sequence, which are not significant and do not suggest any difference in accuracy between the two simulators.

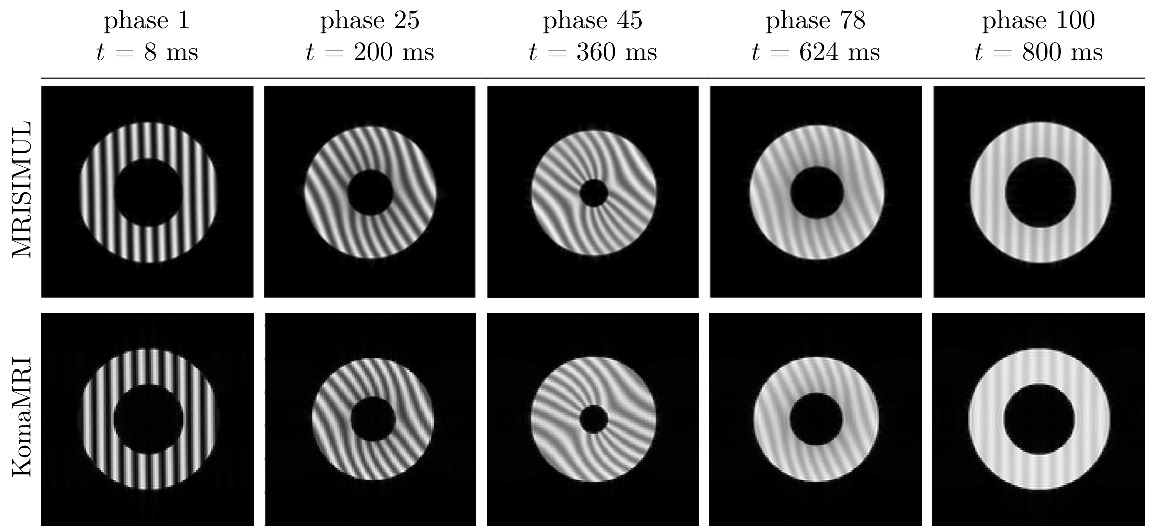


Figure 4.17: Five of the 100 acquired frames during a 2D cine simulation with tagging. The results obtained in KomaMRI (lower row) are almost identical to those extracted from [11] (upper row).

This specific experiment has been carried out on the server computer described in the second item of Section 1.4.1, which is equipped with four high-performance GPUs. Leveraging KomaMRI's capability to perform distributed simulations[11], the same experiment has been conducted under four different configurations: using from 1 GPU up to 4 GPUs. This has been done to observe the reduction in execution times as the number of devices increases.

The diagram in Figure 4.18 illustrates the experiment conducted with four GPUs. Due to the independent spin property, where each spin in the system is independent from the rest, the phantom spins

[10]It is worth noting that the phase numbers —1, 20, 36, 62, 80— shown in the original figure from [11] do not directly align with the phase numbers —1, 25, 45, 78, 100— used in this work, which are based on a consistent scaling of the cardiac cycle. To convert between these indices, simply multiply our frame number by 0.8 or divide the phases in [11] by this factor (which represents the duration of the cardiac cycle in seconds).

[11]See: https://juliahealth.org/KomaMRI.jl/stable/how-to/4-run-distributed-simulations/.

can be partitioned into separate simulations, with the results subsequently recombined. Specifically, the phantom is partitioned into as many parts as there are available devices, four in this case. This approach allows a separate Julia worker process to be assigned to every device, with each simulating a different portion of the object. If necessary, these phantom parts are further subdivided into smaller units that do not exceed the defined maximum number of spins per GPU —one million spins—, a value set to prevent GPU memory overflow.

That said, and referring again to the figure, each GPU performs two sequential simulations: one with 1,000,000 spins and another with 908,198 spins. While these two simulations are executed sequentially on each individual GPU, they are carried out in parallel across all GPUs. As a result, the total execution time of the entire simulation is determined by the slowest GPU, as it will take the longest to complete both simulations.

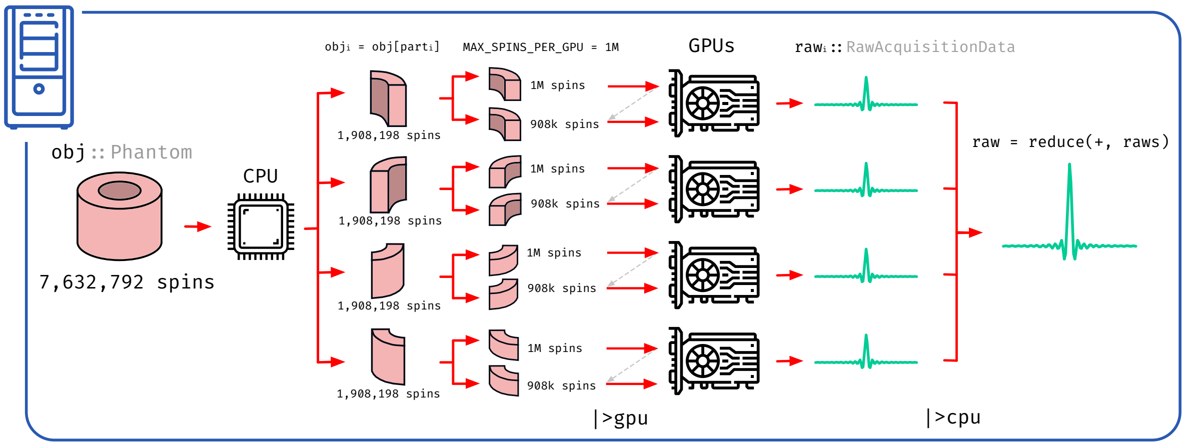


Figure 4.18: Diagram illustrating the execution in KomaMRI of the tagging experiment from [11] utilizing four GPUs. The cylindrical phantom is partitioned into four parts, one for each GPU. Each part is further subdivided into two smaller portions to ensure the defined maximum spin limit of one million is not exceeded. Consequently, each GPU is responsible for performing two simulations. The diagonal gray arrows indicate that, on each GPU, the first subpart of one million spins is simulated first, and only after this simulation is completed, the second subpart of 908,000 spins is sent to the GPU for simulation. The results can then be fetched asynchronously by the main process and combined to produce a final signal.

As previously mentioned, the experiment was conducted four times, incrementally increasing the number of GPUs used. Figure 4.19 shows the execution times for each case and illustrates how these times evolve as the number of GPUs increases. For the case of a single GPU, the execution time is comparable to the approximately 7 hours reported in [11], considering the performance differences between the GPUs used in each experiment.

For the remaining cases, as the number of GPUs increases, a reduction in execution times can be observed, although not in a linear manner. This non-linear reduction can be attributed to two main factors. First, there is the communication overhead between GPUs, which arises from the need for synchronization and data sharing between GPUs, and is closely linked to the earlier point that the total execution time of the entire simulation is limited by the slowest GPU. As more GPUs are utilized, the probability increases that one GPU will become a bottleneck, even if all GPUs have similar performance characteristics.

The second factor is Amdahl's Law (see Appendix E), as not all parts of the computation can be parallelized. The sequential portions of the process impose a limit on the potential speedup, and as more GPUs are added, the relative impact of the sequential part becomes more significant.

b) Validation of the Bloch solver under flow motion conditions

The validation experiment originally proposed in [81] and later conducted in [37, 82] has been replicated to confirm the accuracy of the solutions of the Bloch equations under flow motion conditions.

Number of GPUs	Execution time
1 GPU	7 h 42 min
2 GPUs	4 h 35 min
3 GPUs	3 h 6 min
4 GPUs	2 h 17 min

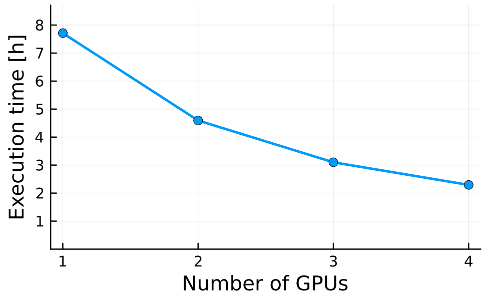
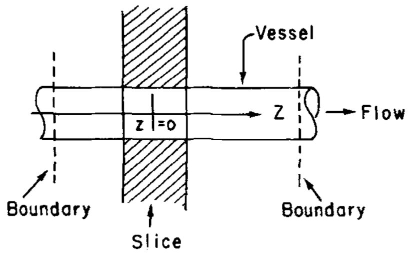
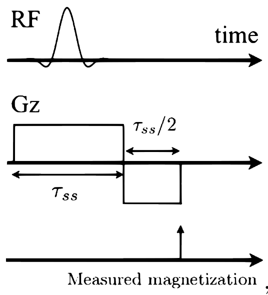


Figure 4.19: Evolution of the execution times in KomaMRI of the tagging experiment from [11] as the number of GPUs increases. The graph shows a nonlinear decrease in execution times.

The phantom designed for this experiment consists of a 1D segment along the z direction (Figure 4.20b), populated with an evenly spaced set of spins. A simple 90º slice-selection sequence, thoroughly described in [81], has been simulated and compared with the results obtained in [82]. The magnetization has been recorded at the end of the rewinder gradient, as marked by the arrow in Figure 4.20b [37]. This experiment has generated three magnetization profiles (M_x, M_y, M_z), which have been analyzed along the length of the phantom in the z axis for input particle velocities ranging from 0 to 200 cm/s.



(a)



(b)

Figure 4.20: Configuration of the Bloch solver validation under flow motion conditions. (a) 1D phantom along the z direction. A transaxial slice through a vessel is assumed so that flow is perpendicular to the slice [81]. (b) 90º slice selective excitation sequence simulated [37].

This flow motion can be replicated in our KomaMRI implementation through three distinct —yet equivalent— methods using `Translate`, `Path`, or `FlowPath`. The implementation is available in Code 4.9. As can be observed the first method is limited to one line of code.

```
seq = (...) # Sequence configuration in Yuan et al. 1987
## Phantom
Nspins = 400
v = 80e-2  # m/s <--------- VELOCITY
Lt = 30e-3 # 30mm
dzf = v * dur(seq)
obj = Phantom(x=zeros(Nspins), z=collect(range(-Lt/2-dzf, Lt/2-dzf, Nspins)))
# Method 1: Translate
obj.motion = Translate(0.0, 0.0, dzf, TimeRange(t_start=0.0, t_end=dur(seq)), AllSpins())
# Method 2: Path
Nt = 500
dx = dy = zeros(Nspins, Nt)
dz      = zeros(Nspins) .+ (dzf/(Nt-1) .* collect(0:(Nt-1))')
obj.motion = Path(dx, dy, dz, TimeRange(0.0, dur(seq)), AllSpins())
# Method 3: FlowPath
spin_reset = collect(Bool.(zeros(Nspins, Nt)))
obj.motion = FlowPath(dx, dy, dz, spin_reset, TimeRange(0.0, dur(seq)), AllSpins())
```

Code 4.9: Flow motion generation methods for the validation experiment.

The code creates the variable `obj`, which is a phantom with spins evenly located within a distance equal to the segment length. The phantom is leftwards-shifted on the amount of `dz` —which equals the total spin displacement during the sequence— so that the spins are evenly located with the symmetry indicated in Fig. 4.20a at readout time. Then, the three methods indicated to define the flow are invoked.

Figure 4.21 presents the results of the experiment. The central column displays the magnetization profiles obtained with the KomaMRI implementation. These profiles demonstrate excellent agreement with the results from [82], shown in the leftmost column.

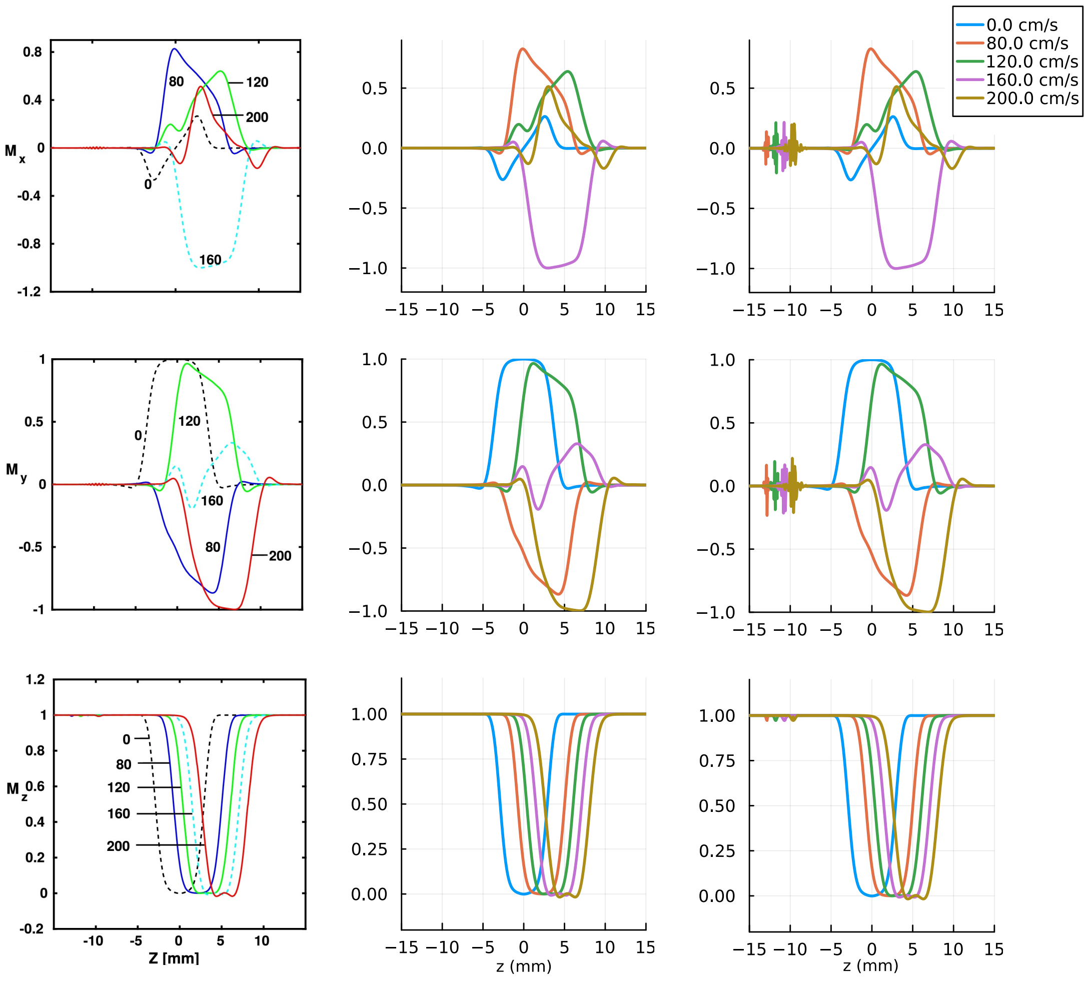
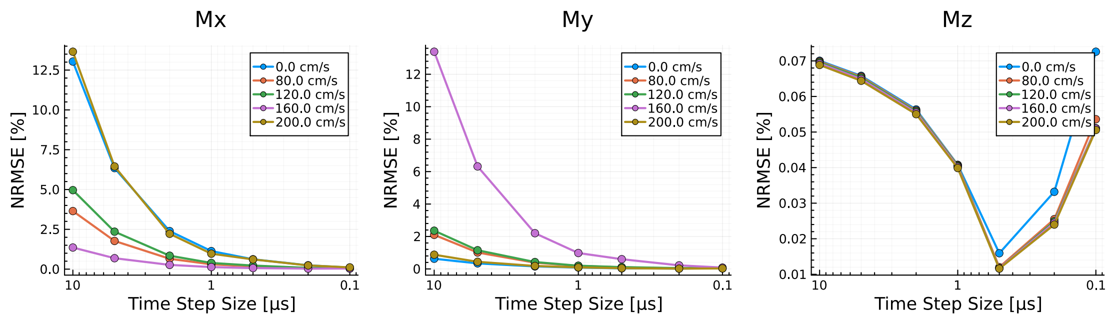


Figure 4.21: Evolution of the three components of the magnetization along the z axis for several velocities imposed to the particles. (a) Results extracted from [82]. (b) Magnetization profiles from the replicated experiment in KomaMRI. (c) Magnetization profiles when the phantom domain is limited to the boundaries at -15 and 15 mm, and the magnetization state of the reinjected spins is reset.

Additionally, for the same experiment and velocity set, a quantitative comparison was performed using the DifferentialEquations.jl [83] suite, specifically its OrdinaryDiffEq.jl package, which provides solvers and tools for ordinary differential equations. To ensure a highly accurate reference solution, the fifth-order Runge-Kutta method was employed. A total of 50 spins were simulated, and various time step sizes[12] ranging from 10^{-5} to 10^{-7} s were tested in KomaMRI. Figure 4.22 presents a convergence

[12]Note that, for simplicity, the same time step size was used both when the RF pulse is active and when it is not, despite the fact that KomaMRI allows them to be defined separately. While a smaller time step would generally be more critical when the RF pulse is active, it could have been relaxed when it is inactive.

analysis of the normalized root mean square error (NRMSE) for these time step sizes and velocity sets. Although certain combinations —M_z for v = [0, 200] cm/s and in M_y for v = 160 cm/s— yield higher errors, they decrease in all cases as the time step is reduced, reaching an acceptable value below 1% for $\Delta t \simeq 10^{-6}$ s. Regarding M_z, the error scale is around 100 times smaller than for M_x and M_y, which makes the convergence less apparent. However, this is not significant, as the error remains very low across all time step sizes. The minimum error of approximately 0.012% is obtained for $\Delta t = 10^{-7}$ s.



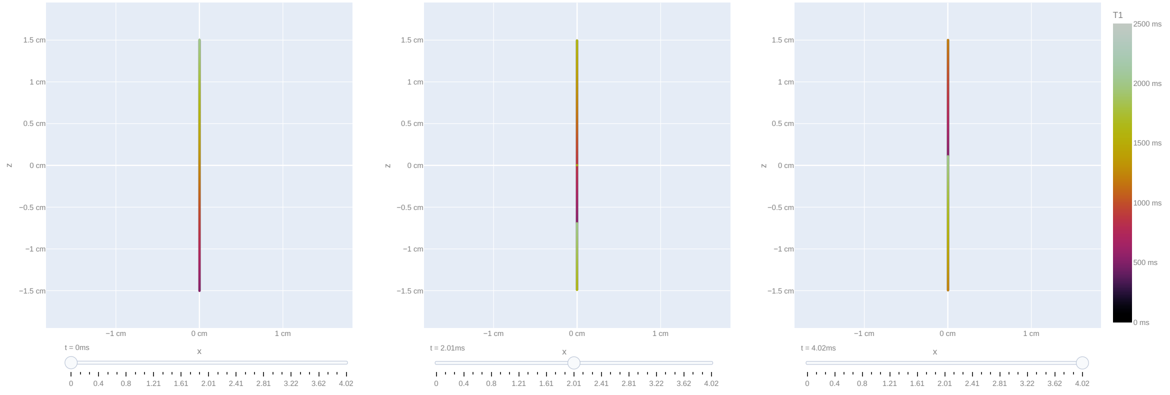
(a) (b) (c)

Figure 4.22: NRMSE values for the three magnetization components at different flow velocities and time step sizes. These error values were obtained by comparing the KomaMRI motion implementation with the DifferentialEquations.jl Bloch solutions.

As mentioned above, in this experiment the 1D segment composed by spins does not start in a centered position about at $z = 0$. Consequently, it does not initially lie within the defined phantom boundaries, between -15 and 15 mm. However, this is not a concern, as these boundaries are purely abstract, and the spins can, in fact, occupy any position in space. That said, there are cases where it is particularly advantageous to keep all the spins within a specific volume of interest and to reinject those that exit the boundaries back into an entry position or region. This approach is especially useful in flow simulations, where a large number of spins need to be tracked over long periods. Reinjecting exiting particles reduces the number of spins required to simulate flow MRI while it still produces realistic results.

This reinjection approach was previously implemented in Section 4.1.2c and has also been tested in the current validation experiment. For this purpose, the phantom was initially defined within the range of -15 to 15 mm, with the same uniform rectilinear motion in the positive z direction. Then, at every n-th discrete time instant $t_n = n \cdot dur(\text{sequence})/(N_t - 1)$, it will be checked whether any spin has exceeded z = 15 mm. If this occurs, the affected spins will be remapped to the entry point at z = -15 mm. Additionally, the `spin_reset` flag will be activated for each spin during its remapping process, which is not instantaneous and lasts for $\Delta t_n = t_n - t_{n-1}$. As a result, the spins will neither be excited nor contribute to the MR signal while being remapped, that is, while they are "jumping" back to the entry point. Given all the above, it is necessary to use the `FlowPath` structure to define this motion pattern. Figure 4.23 illustrates the 1D phantom and shows how the spins that exceed the upper boundary are remapped to the lower boundary.

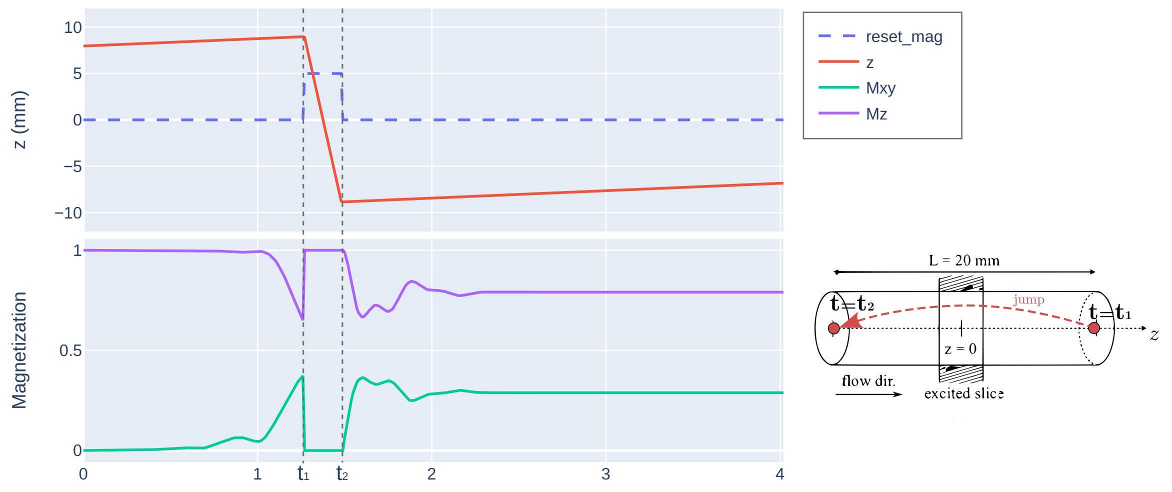
The magnetization profiles obtained can be observed in the right column of Figure 4.21. They are identical to the central column profiles, except for the rapid, low-amplitude and sinc-shaped oscillations that appear on the left side of each of the transverse magnetization profiles. The longitudinal components also show a small negative blip, but with even smaller amplitude. These perturbations arise from spins that are reinjected, with their jump ending while the RF pulse is still active. At the moment the jump ends, the spin is considered "fresh", with a magnetization state shown in Equation (3.9). This means that the longitudinal component will be fully aligned, and the effect of the RF pulse on it will cause a rotation which will lead to a higher transverse magnetization than expected. In the original problem, the equivalent spin is not exactly in these conditions, since, although it was initially located at $z <$ -15 mm, the selective RF pulse still has a slight effect on its magnetization state.



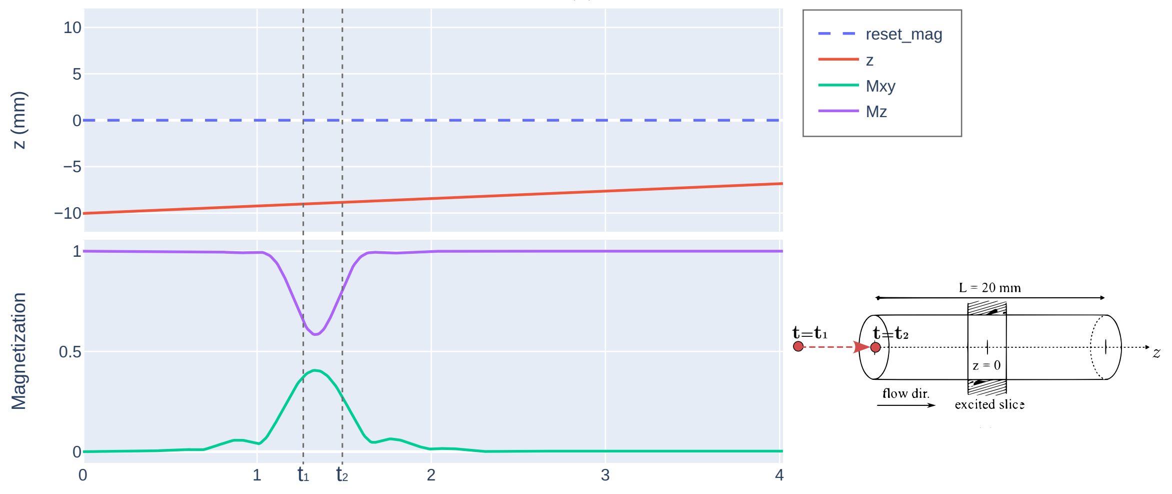
(a) (b) (c)

Figure 4.23: 1D phantom with flow motion along the z direction. The `FlowPath` structure has been used to define the jumps of spins that exit through the upper boundary and are remapped to the lower boundary. For illustrative purposes, a different T_1 value has been assigned to each spin to visualize the reinjection, and a velocity of 300 cm/s has been defined, although this is lower in the actual experiments.

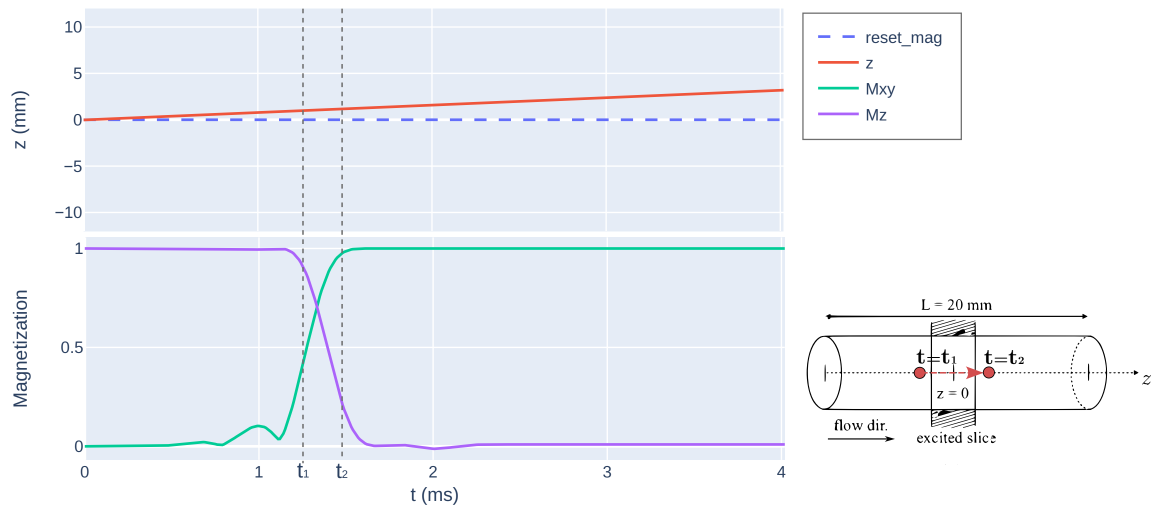
Figure 4.24 illustrates this effect through the temporal evolution of the magnetization of three distinct spins. They demonstrate how the perturbations observed in Figure 4.21c are not due to issues in the simulations but rather to the way the flow phantom is modelled. In this specific case, it is important to note that the phantom is too short to contain particle reinjection without introducing adverse effects. Therefore, care should be taken when designing flow phantoms, and a minimum length should be considered, which will be conditioned by the geometry and the flow velocity, among others.



(a)



(b)



(c)

Figure 4.24: Temporal evolution of the magnetization of three distinct spins within a 20mm flow phantom. (a) Spin that starts its jump at $t = t_1$ and finishes at $t = t_2$. (b) Equivalent spin that coincides in position with the previous one at $t = t_2$. (c) Spin located in the selected slice.

c) Turbulent flow with velocity encoded spoiled Gradient Echo

In order to evaluate the accuracy and computational performance of MRI flow simulations in KomaMRI, we have replicated the turbulent flow V_{ENC} experiment described in the CMRsim paper [8]. Specifically, this experiment involves several velocity-encoded spoiled Gradient Echo (GE) acquisitions, performed on a stenotic U-bend digital phantom. The starting point, illustrated in Figure 4.25, is a CFD file in VTK format, which contains the velocity fields within the geometry and has been extracted from the CMRsim repository[13].

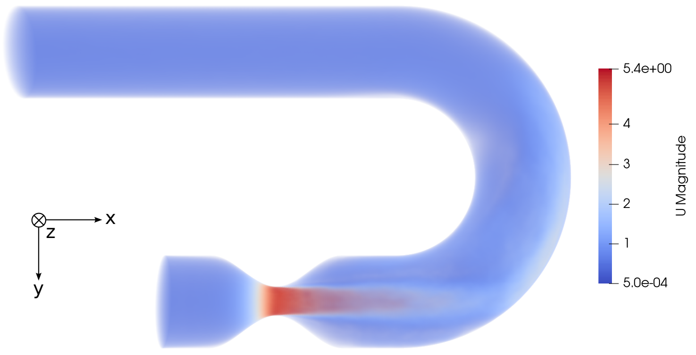


Figure 4.25: Volume rendering of a stenotic U-bend CFD, visualized in Paraview.

To ensure consistency, we have used the same desktop computer[14] to conduct the experiment with both CMRsim and KomaMRI. We have first run the experiment using CMRsim. Both the installation of this simulator and the execution of the experiment have been extremely straightforward, due to the comprehensive documentation[15] provided with the software and its example notebooks, which are also available in the CMRsim repository[16].

The experiment consists of seven acquisitions: one unweighted reference without V_{ENC} and six velocity-encoded acquisitions in different directions, with encoding values of (550, 50, 250, 50, 100, 50) cm/s along the directions ([0, 0, 1], [0, 0, 1], [0, 1, 0], [0, 1, 0], [1, 0, 0], [1, 0, 0]). The sequence parameters were TR = 10 ms, TE = 5 ms, and a flip angle of 15°. Imaging resolution was set to 2 × 2 mm with a FOV of 22.2 × 14.2 cm. The temporal grid size was set to 10 µs. The target density for both initial seeding and reseeding was $5/\text{mm}^3$, resulting in a total of 1.5 million particles [8].

Returning to Section 2.3.2, it can be recalled that the trajectories of each particle, i.e., of each spin, are calculated on-the-fly thanks to the `increment_particles` method of the corresponding *Trajectory Module* used in the simulation. Equation (2.54) showed the implementation of this function for the *FlowTrajectory* module. However, in this experiment, the *TurbulentTrajectory* module is used. Its `increment_particles` function is almost identical, except for the fact that it adds a direct temporal dependence on the velocity:

$$\boldsymbol{r}(t+\Delta t) = \boldsymbol{r}(t) + \Delta t \cdot \boldsymbol{v}(\boldsymbol{r}(t),\ t+\Delta t) \tag{4.5}$$

This temporal variability is introduced by a turbulent component $\boldsymbol{u}(t)$, which is computed by solving the Markov chain integration of the modified classical Langevin Equation [8]:

$$\boldsymbol{u}(t+\Delta t) = \boldsymbol{u}(t)\mathrm{e}^{\frac{-\Delta t}{\tau(\boldsymbol{r}(t))}} + \boldsymbol{\zeta}(\boldsymbol{r}) \cdot a(\boldsymbol{r})\sqrt{1-\mathrm{e}^{\frac{-2\Delta t}{\tau(\boldsymbol{r}(t))}}} \tag{4.6}$$

[13]CMRsim repository: https://gitlab.ethz.ch/ibt-cmr/mri_simulation/cmrsim/-/tree/master.

[14]Desktop computer described in Section 1.4.1.

[15]CMRsim documentation: https://people.ee.ethz.ch/~kozerkes/cmrsim/latest/index.html.

[16]The CMRsim example notebooks, which include the specific experiment conducted in this study, are available at: https://gitlab.ethz.ch/ibt-cmr/mri_simulation/cmrsim/-/blob/master/notebooks.

where $a(\boldsymbol{r}) \in \mathbb{R}^{3x3}$ is the Lund transformation based on the Cholesky decomposition of the Reyolds stress tensor [8], $\tau(\boldsymbol{r})$ is the Lagrangian integral time scale, and $\boldsymbol{\zeta}(\boldsymbol{r})$ follows a three-dimensional normal distribution $\mathcal{N}^3(0,1)$. Accordingly, considering $\mathbf{U}(t)$ as the mean velocity extracted from the spatial interpolation of the velocity VTK mesh, the total velocity vector update per step follows as [8]:

$$\boldsymbol{v}(\boldsymbol{r}(t),\ t) = \mathbf{U}(t,\ \boldsymbol{r}(t)) + \boldsymbol{u}(t) \tag{4.7}$$

The odd-numbered rows of Figure 4.26 show the results of this CMRsim experiment. Note how the axes have changed with respect to Figure 4.25, as a rotation has been applied to the phantom before the MRI simulation. From top to bottom, the images of magnitude, phase, phase difference with respect to the reference, and the turbulent kinetic energy (TKE) coefficient are displayed. This last coefficient is extracted from the magnitude images as follows:

$$\text{TKE} = \ln\left(\frac{\text{reference magnitude image}}{\text{magnitude image}}\right) \tag{4.8}$$

For the phase difference images, multiple phase wraps are present when the V_{ENC} is too low (see the explanation of aliasing in Section 2.1.3d). With a larger V_{ENC}, phase wrapping is not present, and a jet can be seen. Furthermore, the FEG readout gradients are parallel to the flow direction, the zero-V_{ENC} case shows a jet structure in the phase image. At higher V_{ENC} values, the TKE images show minimal structure; however, at lower V_{ENC}, turbulence around the jet becomes more apparent [8].

The total duration of this CMRsim experiment, which includes both the trajectory advection, the MRI simulation, and the reseeding for all seven V_{ENC} acquisitions, was 53 minutes.

For the replication of the experiment in KomaMRI, the most notable difference compared to CMRsim lies in the method used for generating the phantom and computing its spin trajectories. Whereas CMRsim calculates the trajectories on the fly while simulating MRI, our method pre-calculates these trajectories and stores them in a `.phantom` file, as explained in Section 3.1.3c and Appendix C. Once this file is generated, it can be loaded into KomaMRI, allowing the MRI simulation to be performed without the need to recompute the trajectories each time. Hence, when comparing with CMRsim, it is important to consider the total time as the sum of two components: the trajectory calculation time and the MRI simulation time. These are independent but both must be included to ensure a fair comparison.

The particle trajectory solver, developed with VTK-m, takes as input the CFD file in VTK format, described at the beginning of this section, as well as a seed file containing the initial positions of the 1.5 million particles, which are randomly and uniformly distributed throughout the volume. The script has been configured to solve the trajectories for a duration of 650 ms, comparable to the duration of the GE sequence used. The time step size has been set to 0.5 ms, resulting in a total of 1300 time steps. However, only one out of every five time steps has been stored in the phantom file, resulting in a total of 260 positions per spin. The total execution time for this script was 12 minutes and 40 seconds.

Regarding the MRI simulation, the phantom of 1.5 million spins was divided into chunks of 200,000 spins, which were sequentially sent to the GPU for simulation. The total simulation time for all seven V_{ENC}s was 7 minutes.

Therefore, the total time, combining trajectory computation and MRI simulation, amounts to 19 minutes and 40 seconds, representing a reduction of approximately 60% compared to CMRsim. However, this measurement is based on isolated simulations, and a more comprehensive benchmarking is expected to yield even greater time differences.

The even-numbered rows of Figure 4.26 show the results of the experiment with KomaMRI. At first glance, a strong similarity can be observed between these images and those produced by CMRsim, particularly in the phase images (rows 3 to 6). Regarding the magnitude images, a certain degree of particle accumulation can be observed along the outer edge of the curve, resulting in increased image intensity in that area. In contrast, darker regions appear in the central part of the tube bend, indicating a lower particle density in that region.

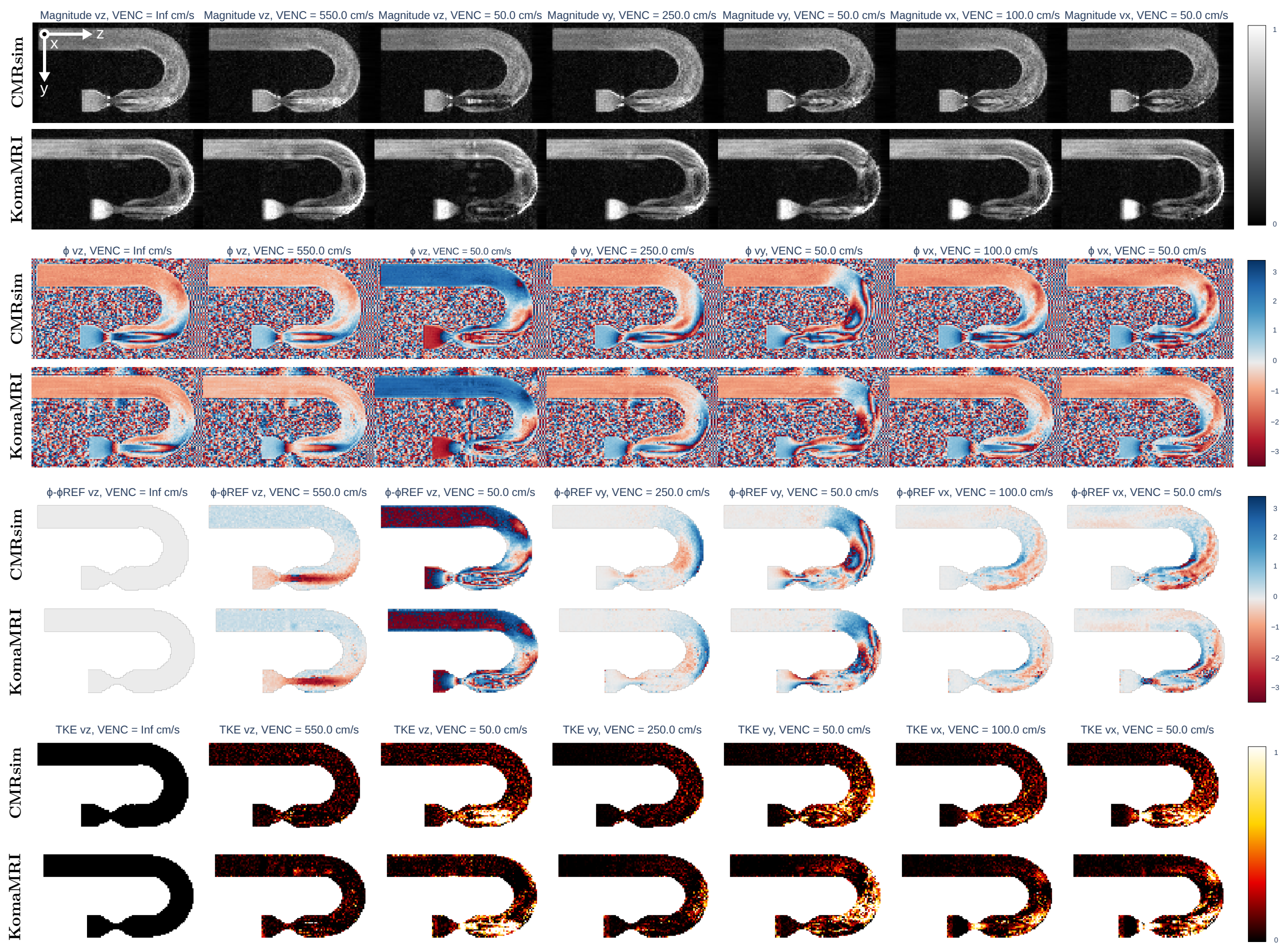
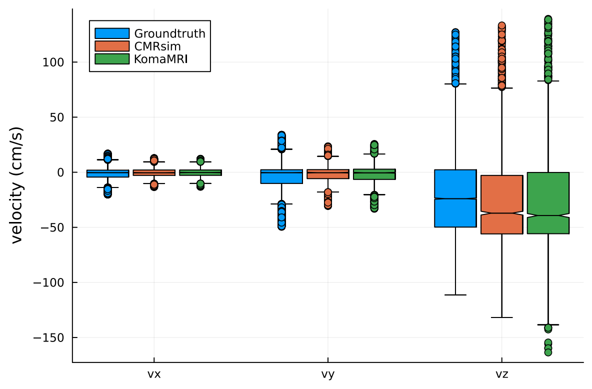

Magnitude vz, VENC = Inf cm/s
Magnitude vz, VENC = 550.0 cm/s
Magnitude vz, VENC = 50.0 cm/s
Magnitude vy, VENC = 250.0 cm/s
Magnitude vy, VENC = 50.0 cm/s
Magnitude vx, VENC = 100.0 cm/s
Magnitude vx, VENC = 50.0 cm/s
CMRsim
KomaMRI
ϕ vz, VENC = Inf cm/s
ϕ vz, VENC = 550.0 cm/s
ϕ vz, VENC = 50.0 cm/s
ϕ vy, VENC = 250.0 cm/s
ϕ vy, VENC = 50.0 cm/s
ϕ vx, VENC = 100.0 cm/s
ϕ vx, VENC = 50.0 cm/s
ϕ-ϕREF vz, VENC = Inf cm/s
ϕ-ϕREF vz, VENC = 550.0 cm/s
ϕ-ϕREF vz, VENC = 50.0 cm/s
ϕ-ϕREF vy, VENC = 250.0 cm/s
ϕ-ϕREF vy, VENC = 50.0 cm/s
ϕ-ϕREF vx, VENC = 100.0 cm/s
ϕ-ϕREF vx, VENC = 50.0 cm/s
TKE vz, VENC = Inf cm/s
TKE vz, VENC = 550.0 cm/s
TKE vz, VENC = 50.0 cm/s
TKE vy, VENC = 250.0 cm/s
TKE vy, VENC = 50.0 cm/s
TKE vx, VENC = 100.0 cm/s
TKE vx, VENC = 50.0 cm/s

Figure 4.26: Simulation results for the PC-MRI experiment of turbulent flow downstream of a stenotic U-bend. From top to bottom, the images of magnitude, phase, phase difference, and turbulent kinetic energy (TKE) are shown for different V_{ENC}s and directions. The odd-numbered rows correspond to the experiment conducted in CMRsim, while the even-numbered rows correspond to the experiment replicated in KomaMRI.

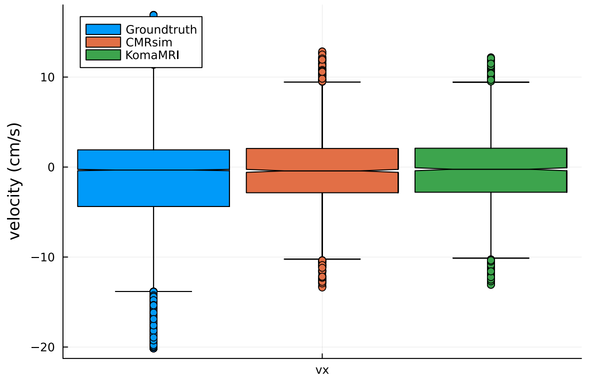
These differences, more pronounced in the magnitude images but also affecting the clarity of the phase images, are largely due to the method used to compute the particle trajectories in each case. As previously mentioned, our approach pre-computes the trajectories with a constant time step size, which requires a large number of time steps when high temporal resolution is needed throughout the entire experiment. In contrast, CMRsim dynamically adjusts the time step size according to the sequence events, providing higher resolution in time intervals where it is most needed, such as during RF pulse activations or readout periods.

Therefore, our approach benefits from requiring only a single pre-computation of the trajectories. However, its drawback is that the temporal nodes of these trajectories are fixed and do not dynamically adjust to the timing of pulse sequence events. This can lead to spin accumulation, which may affect the accuracy of the simulated images. Nevertheless, as previously mentioned, this issue can be mitigated by reducing the time step size.

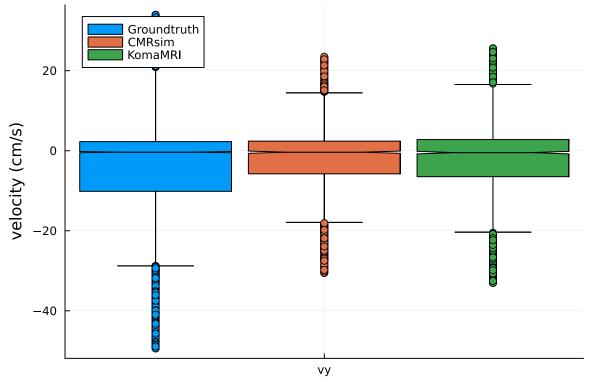
As a quantitative comparison, Figure 4.27 presents the boxplots of velocities retrieved from the phase difference images, for both CMRsim and KomaMRI, along with the boxplot corresponding to the velocity values directly extracted from the CFD, which will be referred to as the Ground Truth (GT). The differences between the two simulators are almost non-existent, although some minor discrepancies can be observed when compared to the GT. Specifically, the Kruskal-Wallis test [84] performed between the three groups —GT, CMRsim and KomaMRI— yields a p-value smaller than 0.05, which indicates that the null hypothesis of equality between the three groups is rejected. However, when conducting the Mann-Whitney U test [85] between CMRsim and KomaMRI, the p-values obtained for the velocities in the three spatial directions, are 0.2, 0.35, and 0.35, respectively. Hence, the null hypothesis, which posits no shift between the distributions of velocity samples obtained from both simulators, is accepted.



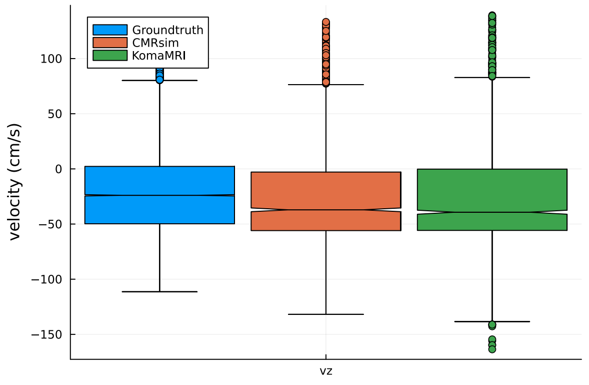
(a)



(b)



(c)



(d)

Figure 4.27: Comparison of Ground Truth CFD velocities with those obtained from PC-MRI experiments using CMRsim and KomaMRI. The results from both simulators are highly consistent. However, some discrepancies with the GT are present, reflecting the limitations of velocity estimation from phase images in planes parallel to the flow direction. (a) Boxplots of velocity in all three directions. (b) Boxplots of the velocity component in the x direction. (c) Boxplots of the velocity component in the y direction. (d) Boxplots of the velocity component in the z direction.

The main cause of the difference between the GT velocities and those obtained from the simulated images lies in the method used to derive the velocity from the phase images. In this experiment, the readout gradients are parallel to the flow direction, which result in a phase shift proportional to the velocity of the jet —even in the absence of V_{ENC}—. In real PC-MRI acquisitions, the imaging plane is typically chosen to be perpendicular to the flow direction to avoid these undesired phase shifts [86]. Hence, the observed differences are an inherent consequence of the acquisition setup rather than a limitation of the simulators, as the same effect would occur in a real MRI acquisition.

4.2 Web Sequence Editor Application

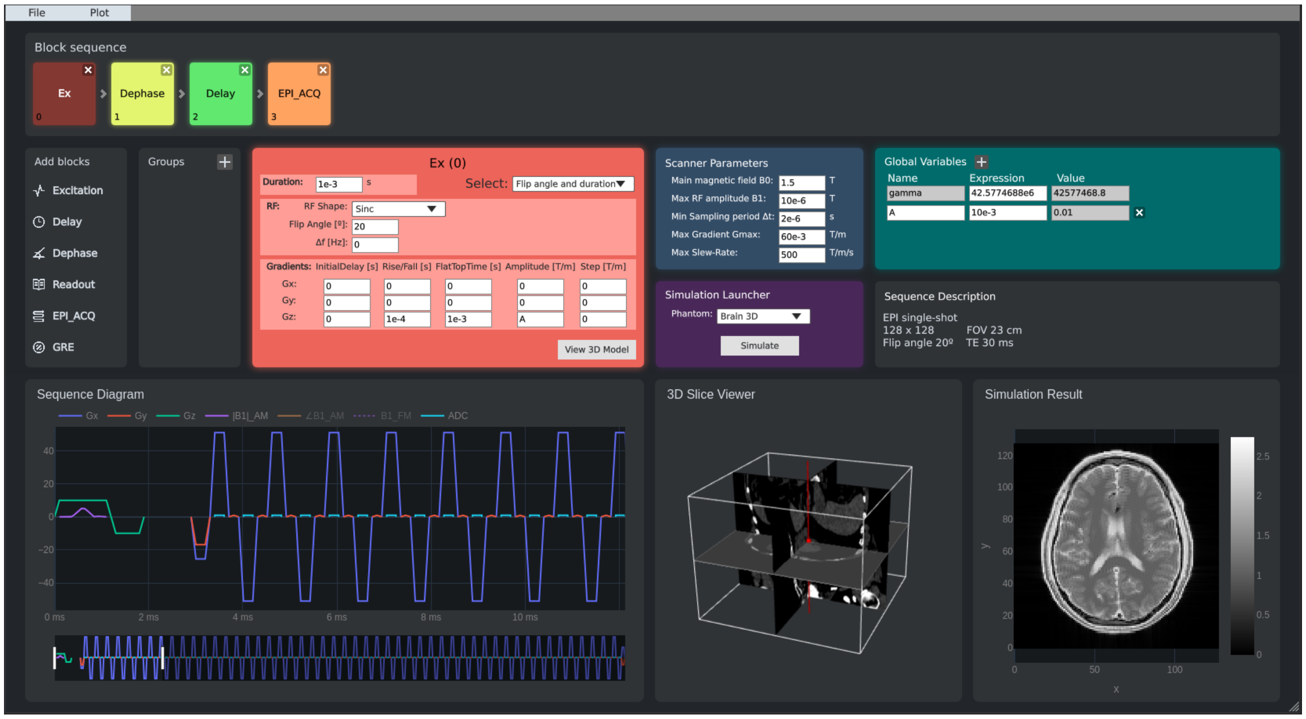
The validation of the Web-based Sequence Editor tool focuses on evaluating its usability, functionality, and integration within the overall system. This section begins with an overview of the final application and its GUI, outlining its main components. Then, aspects such as initial load time, interface responsiveness, and fluidity during user interactions —such as dragging blocks or manipulating graphical viewers— are evaluated. Last, key functionalities such as sequence creation and edition, file import and export, and simulation and plotting tools are tested through practical demonstrations.

It is worth noting that this part of the work has resulted in two contributions to national conferences [74, 87].

4.2.1 Overview of the Application and GUI

Figure 4.28 displays the final layout of the application, which is divided into panels. This structure mirrors the design outlined in Figure 3.15. The interface is designed to occupy the entire available screen space, although it adjusts to fit the browser window size. This layout, while not a replica, more closely resembles a real MRI scanner console than the previous version [1] of the tool.

The panel-based layout allows the application to be intuitive for both trainee technicians, who require a user experience closer to that of a real MRI scanner console, and researchers, who need to design and test their pulse sequences in a quick and intuitive manner. Thus, the application offers a user-friendly interface for beginners while providing the necessary tools for advanced tasks.



Ex (0)

Duration: 1e-3 s Select: Flip angle and duration

RF: RF Shape: Sinc
Flip Angle [º]: 20
Δf [Hz]: 0

Gradients:	InitialDelay [s]	Rise/Fall [s]	FlatTopTime [s]	Amplitude [T/m]	Step [T/m]
Gx:	0	0	0	0	0
Gy:	0	0	0	0	0
Gz:	0	1e-4	1e-3	A	0

Scanner Parameters

Main magnetic field B0:	1.5	T
Max RF amplitude B1:	10e-6	T
Min Sampling period Δt:	2e-6	s
Max Gradient Gmax:	60e-3	T/m
Max Slew-Rate:	500	T/m/s

Global Variables

Name	Expression	Value
gamma	42.5774688e6	42577468.8
A	10e-3	0.01

Simulation Launcher

Phantom: Brain 3D

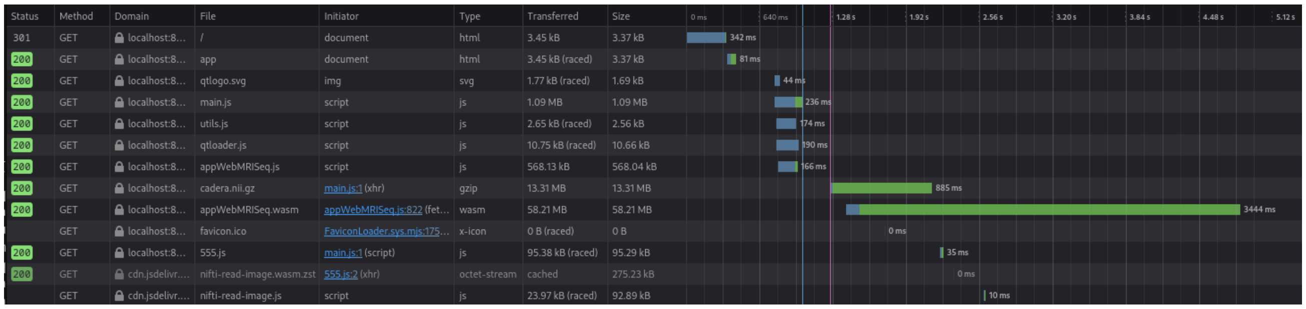
Sequence Description

EPI single-shot
128 x 128 FOV 23 cm
Flip angle 20º TE 30 ms

Figure 4.28: Final layout of the application, which is divided into panels according to the initial design.

4.2.2 Interface Performance and Responsiveness

Figure 4.29 displays a screenshot from the Firefox Inspector tool, specifically the Network tab, which shows the HTTP requests and responses made during the initial application loading. This visual representation is included to evaluate the initial load time of the web-based sequence editor when it is first opened in the browser. As can be seen, the two resources with the longest transfer times are the NIfTI file —0.8 seconds—, which contains the example volume (`cadera.nii.gz`) for the 3D viewer, and the WebAssembly (`.wasm`) file —3.4 seconds—, which includes the compiled sequence editor tool.



Status	Method	Domain	File	Initiator	Type	Transferred	Size	
301	GET	localhost:8...	/	document	html	3.45 kB	3.37 kB	342 ms
200	GET	localhost:8...	app	document	html	3.45 kB (raced)	3.37 kB	81 ms
200	GET	localhost:8...	qtlogo.svg	img	svg	1.77 kB (raced)	1.69 kB	44 ms
200	GET	localhost:8...	main.js	script	js	1.09 MB	1.09 MB	236 ms
200	GET	localhost:8...	utils.js	script	js	2.65 kB (raced)	2.56 kB	174 ms
200	GET	localhost:8...	qtloader.js	script	js	10.75 kB (raced)	10.66 kB	190 ms
200	GET	localhost:8...	appWebMRISeq.js	script	js	568.13 kB	568.04 kB	166 ms
200	GET	localhost:8...	cadera.nii.gz	main.js:1 (xhr)	gzip	13.31 MB	13.31 MB	885 ms
200	GET	localhost:8...	appWebMRISeq.wasm	appWebMRISeq.js:822 (fet...	wasm	58.21 MB	58.21 MB	3444 ms
	GET	localhost:8...	favicon.ico	FaviconLoader.sys.mjs:175...	x-icon	0 B (raced)	0 B	0 ms
200	GET	localhost:8...	555.js	main.js:1 (script)	js	95.38 kB (raced)	95.29 kB	35 ms
200	GET	cdn.jsdelivr....	nifti-read-image.wasm.zst	555.js:2 (xhr)	octet-stream	cached	275.23 kB	0 ms
	GET	cdn.jsdelivr....	nifti-read-image.js	script	js	23.97 kB (raced)	92.89 kB	10 ms

Figure 4.29: Firefox Inspector's Network tab displaying the HTTP requests, responses, and their respective load times during the initial application load.

These loading times, particularly for the `.wasm` file, may initially seem somewhat long. However, they are manageable, as they occur only during the initial application launch, and are similar to the loading times of modern desktop applications. Additionally, during the WebAssembly loading phase, the application provides feedback to the user, displaying a loading message and the Qt logo, as shown in Figure 4.30. This helps to manage expectations, reassuring users that the system is actively loading and not stalled.

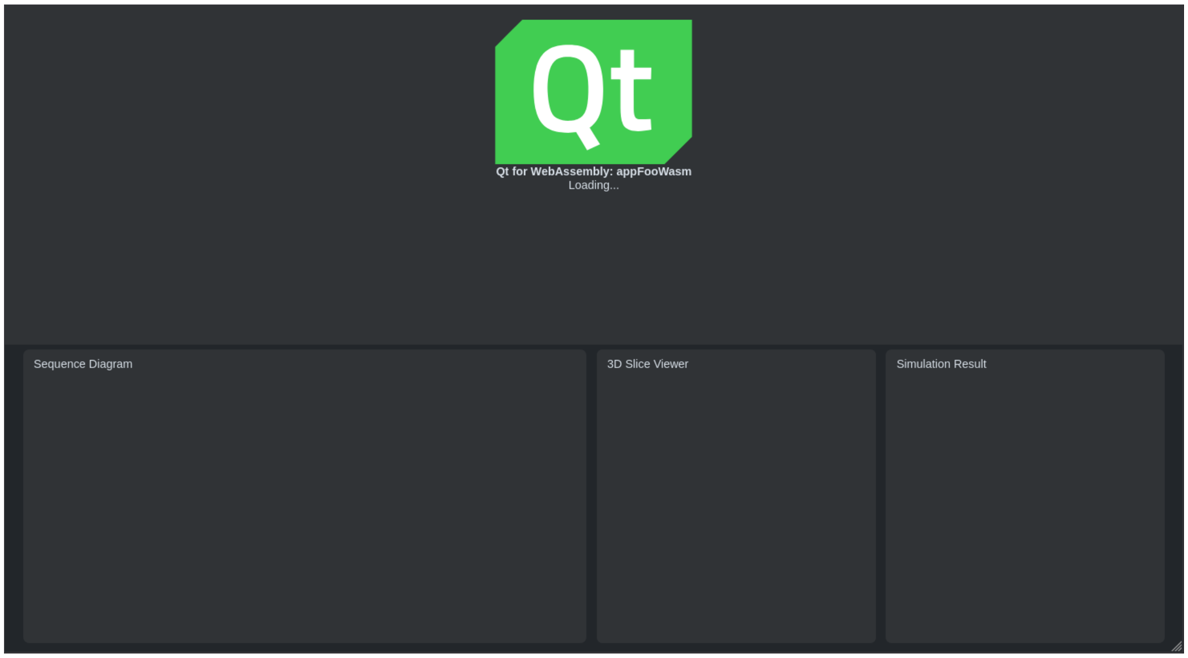


Figure 4.30: Application layout during initialization. The Qt logo and the accompanying text indicate that the application is in the process of loading.

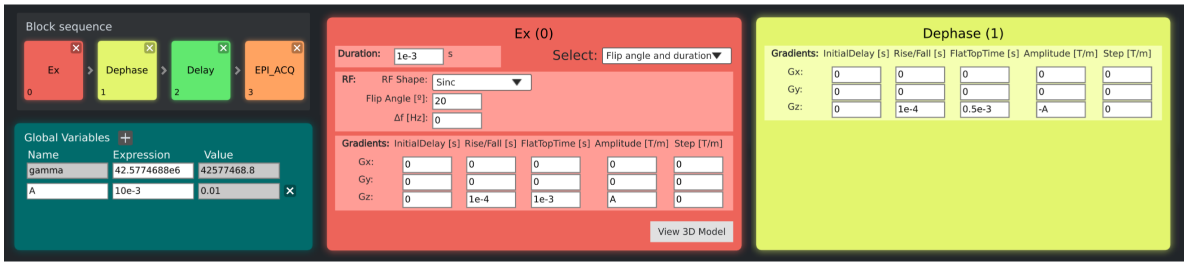
Once the initialization process is complete and the loading times outlined above have passed, the application provides a highly responsive user experience. Interactions feel smooth, with sequence blocks moving seamlessly and controls being intuitive. Actions such as plotting the sequence diagram or running an MRI simulation exhibit low response times. In the case of MRI simulations, execution times depend on complexity; however, a progress bar continuously updates the user on the simulation status, ensuring clarity and a smooth experience.

4.2.3 Key Functionalities and User Interaction

a) Sequence Creation, Editing, and Simulation

This section replicates the workflow of a user when designing and simulating sequences on a brain phantom. To demonstrate the application's versatility, two different sequences are created, covering key steps such as adding and arranging blocks, grouping them, defining global variables, and adjusting block parameters accordingly.

EPI Sequence The simplest case for the user is an EPI sequence, built using the predefined `EPI` block available in the application. Figure 4.31 displays the key GUI panels used to configure a single-shot EPI sequence by concatenating four blocks. The corresponding temporal diagram is shown in Figure 4.32a. Each block in the sequence can be mapped to elements in the diagram as follows: the `Ex` block corresponds to the sinc-shaped RF pulse and the simultaneous SSG in z; the `Dephase` block adds a negative z gradient immediately after the RF pulse; the `Delay` block introduces a 1 ms "empty" interval; finally, the `EPI_ACQ` block executes a 100×100 point EPI acquisition, which fully covers K-space. In the global variables panel, in addition to the default variable `gamma` ($= \bar{\gamma}$), the variable `A` is defined. This variable is used in the configuration panels of the `Ex` and `Dephase` blocks to set the amplitude of the z gradients. Modifying the value of `A` updates both blocks simultaneously, eliminating the need to adjust them individually. Figure 4.32 shows the sequence diagram which corresponds to the EPI sequence and the resulting image obtained from the simulation of this sequence on a brain phantom.


Block sequence

Ex	Dephase	Delay	EPI_ACQ
0	1	2	3

Global Variables

Name	Expression	Value
gamma	42.5774688e6	42577468.8
A	10e-3	0.01

Ex (0)

Duration: 1e-3 s

Select: Flip angle and duration

RF: RF Shape: Sinc

Flip Angle [º]: 20

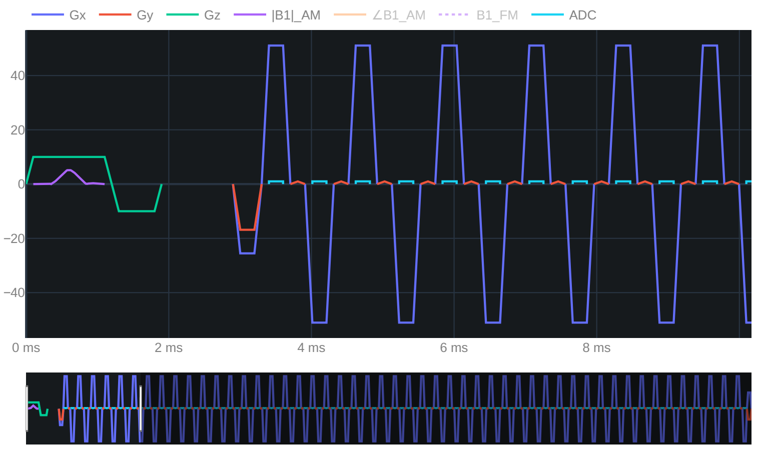
Δf [Hz]: 0

Gradients:	InitialDelay [s]	Rise/Fall [s]	FlatTopTime [s]	Amplitude [T/m]	Step [T/m]
Gx:	0	0	0	0	0
Gy:	0	0	0	0	0
Gz:	0	1e-4	1e-3	A	0

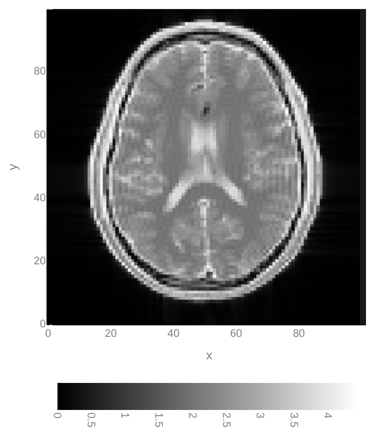
Dephase (1)

Gradients:	InitialDelay [s]	Rise/Fall [s]	FlatTopTime [s]	Amplitude [T/m]	Step [T/m]
Gx:	0	0	0	0	0
Gy:	0	0	0	0	0
Gz:	0	1e-4	0.5e-3	-A	0

Figure 4.31: GUI panels displaying the configuration of a single-shot EPI sequence. In the top left, the sequence block arrangement is shown. Below it, the global variables panel is displayed. To the right, the configuration panels for the `Ex` and `Dephase` blocks are presented, respectively.



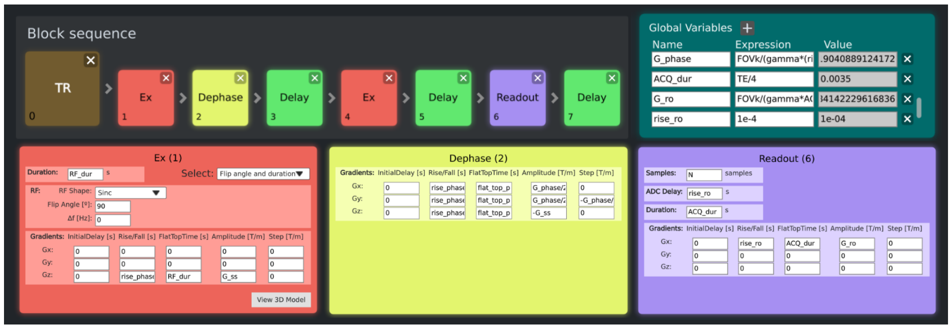
(a)



(b)

Figure 4.32: (a) Sequence diagram of the EPI sequence. (b) Simulation result.

Spin Echo (SE) Sequence The next case is a basic SE sequence, used to evaluate the influence of TE and TR parameters on the final image contrast. Figure 4.33 shows the most relevant GUI panels for configuring the sequence. The first key aspect to highlight is the existence of a block group called `TR`, which contains all blocks that repeat every TR. Although not visible in the figure, the number of repetitions for this group has been set to 64, which corresponds to the number of K-space lines to be acquired. The two `Ex` blocks represent the 90º and 180º RF pulses, which are characteristic of SE sequences. The figure displays the configuration panel for the first one. The `Dephase` block applies gradients in all three directions: in x and y to position the readout pointer at the top-right corner of K-space, and in z to compensate for the effect of the SSG. The `Readout` block acquires a single K-space line of 64 points, while simultaneously activating a FEG in the x direction. Finally, the `Delay` blocks introduce timing delays to ensure compliance with the predefined TE and TR values.



Global Variables

Name	Expression	Value
G_phase	FOVk/(gamma*(ri	.9040889124172
ACQ_dur	TE/4	0.0035
G_ro	FOVk/(gamma*A(	34142229616836
rise_ro	1e-4	1e-04

Ex (1)

Duration: RF_dur s Select: Flip angle and duration

RF: RF Shape: Sinc; Flip Angle [º]: 90; Δf [Hz]: 0

Gradients:	InitialDelay [s]	Rise/Fall [s]	FlatTopTime [s]	Amplitude [T/m]	Step [T/m]
Gx:	0	0	0	0	0
Gy:	0	0	0	0	0
Gz:	0	rise_phase	RF_dur	G_ss	0

Dephase (2)

Gradients:	InitialDelay [s]	Rise/Fall [s]	FlatTopTime [s]	Amplitude [T/m]	Step [T/m]
Gx:	0	rise_phase	flat_top_p	G_phase/2	0
Gy:	0	rise_phase	flat_top_p	G_phase/2	-G_phase/
Gz:	0	rise_phase	flat_top_p	-G_ss	0

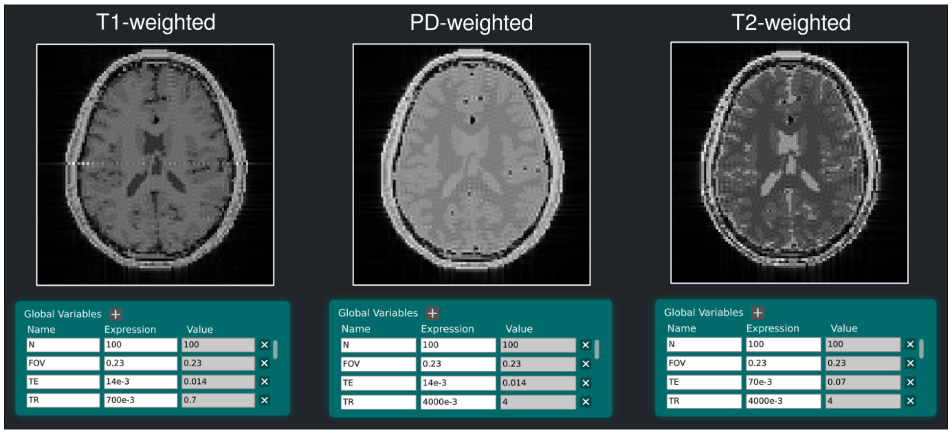
Readout (6)

Samples: N samples; ADC Delay: rise_ro s; Duration: ACQ_dur s

Gradients:	InitialDelay [s]	Rise/Fall [s]	FlatTopTime [s]	Amplitude [T/m]	Step [T/m]
Gx:	0	rise_ro	ACQ_dur	G_ro	0
Gy:	0	0	0	0	0
Gz:	0	0	0	0	0

Figure 4.33: GUI panels displaying the configuration of a SE sequence. The top left panel shows the sequence block arrangement, while the top right panel displays the list of defined global variables. The bottom panels contain the configuration settings for the `Ex` (1), `Dephase` (2), and `Readout` (6) blocks, respectively.

In this case, multiple global variables have been defined and referenced within each block's configuration. This ensures that any change in one of these variables affects the entire sequence. Figure 4.34 shows the results of simulating the SE sequence in three different experiments, where TE and TR values were adjusted. These variations alter the image contrast[17], and allow the acquisition of T_1, T_2, or PD-weighted images.



Global Variables (T1-weighted)

Name	Expression	Value
N	100	100
FOV	0.23	0.23
TE	14e-3	0.014
TR	700e-3	0.7

Global Variables (PD-weighted)

Name	Expression	Value
N	100	100
FOV	0.23	0.23
TE	14e-3	0.014
TR	4000e-3	4

Global Variables (T2-weighted)

Name	Expression	Value
N	100	100
FOV	0.23	0.23
TE	70e-3	0.07
TR	4000e-3	4

Figure 4.34: MRI simulation results based on the SE sequence created in the GUI. The bottom panels display the global variables of the sequence, with a particular focus on TE and TR, which were adjusted to obtain T_1, PD, and T_2-weighted images, respectively.

[17]Table 7.1 of [16] shows the appropriate TE and TR values for an SE sequence to obtain different image weightings. The terms "short", "long", and "appropriate" are used in comparison to the T_1 and T_2 times of the imaged tissues.

b) File import and export

The application allows users to export and import both sequence files and scanner configuration files. As shown in Figure 4.35, sequences can be saved either in JSON format or as a QML ListModel, ensuring compatibility with different workflows. To facilitate usage, the repository includes example JSON files compatible with the application, available in the `/examples` directory[18]. Regarding the reading and writing of Pulseq files, this functionality remains pending, as the goal is to leverage KomaMRI's existing capabilities[19].

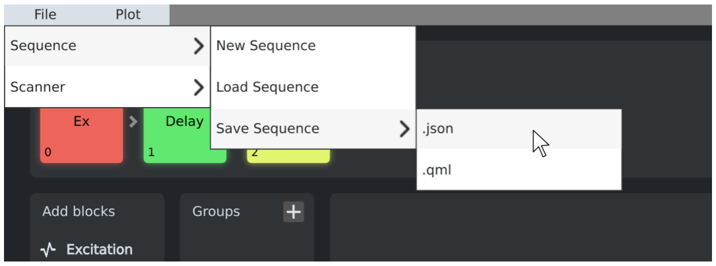
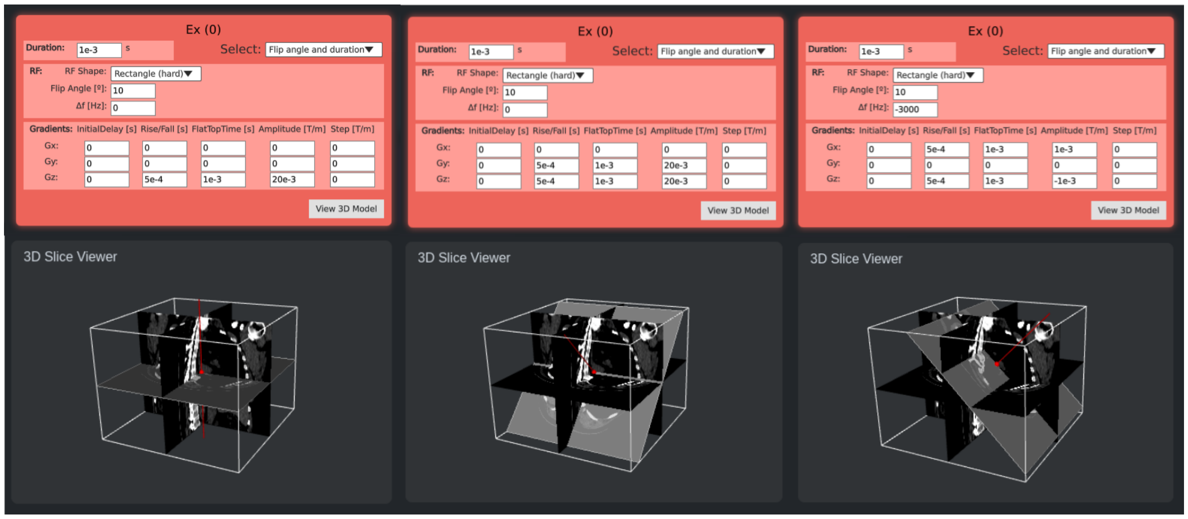


Figure 4.35: File management drop-down menu, which allows the import and export of sequence and scanner configuration files. This menu is located in the top toolbar of the application.

c) Plotting tools

In addition to the sequence diagram and simulated image viewers, the application includes a 3D visualizer implemented with VTK.js, which allows the visualization of the slice selected by the RF pulse of an `Ex` block in a volume. Figure 4.36 shows three configuration examples for the SSG gradients that are activated simultaneously with the RF pulse. In the first case, the xy plane is selected. In the second and third cases, oblique planes are chosen relative to the three orthogonal planes. In the third case, a frequency shift is added to the pulse to move the center of the plane away from the isocenter. The "View 3D Model" button updates the visualizer with the gradient information from the panel.


Ex (0)

Duration: 1e-3 s Select: Flip angle and duration

RF: RF Shape: Rectangle (hard)
Flip Angle [º]: 10
Δf [Hz]: 0

Gradients:	InitialDelay [s]	Rise/Fall [s]	FlatTopTime [s]	Amplitude [T/m]	Step [T/m]
Gx:	0	0	0	0	0
Gy:	0	0	0	0	0
Gz:	0	5e-4	1e-3	20e-3	0

Ex (0)

Duration: 1e-3 s Select: Flip angle and duration

RF: RF Shape: Rectangle (hard)
Flip Angle [º]: 10
Δf [Hz]: 0

Gradients:	InitialDelay [s]	Rise/Fall [s]	FlatTopTime [s]	Amplitude [T/m]	Step [T/m]
Gx:	0	0	0	0	0
Gy:	0	5e-4	1e-3	20e-3	0
Gz:	0	5e-4	1e-3	20e-3	0

Ex (0)

Duration: 1e-3 s Select: Flip angle and duration

RF: RF Shape: Rectangle (hard)
Flip Angle [º]: 10
Δf [Hz]: -3000

Gradients:	InitialDelay [s]	Rise/Fall [s]	FlatTopTime [s]	Amplitude [T/m]	Step [T/m]
Gx:	0	5e-4	1e-3	1e-3	0
Gy:	0	0	0	0	0
Gz:	0	5e-4	1e-3	-1e-3	0

Figure 4.36: 3D visualization of the slices selected by the RF pulse of the `Ex` block. Three configuration examples are displayed, with the slices selected over a hip volume.

[18] See https://github.com/pvillacorta/WebMRISeq/tree/tfm/examples.

[19] As mentioned in Section 2.3.3, Pulseq file support is already present in the KomaMRI; however, the writing functionality is still under development: https://github.com/JuliaHealth/KomaMRI.jl/pull/284.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This Master's Thesis aimed to achieve two main objectives: (1) the **development of a dynamic phantom model** for the simulation of both simple and complex motions within an MRI simulator, and (2) the **creation of a web-based sequence editor** for designing and testing pulse sequences in an interactive and user-friendly environment. Both objectives shared a common starting point: the KomaMRI simulator, which was extended with new functionalities for the first objective, and whose existing features were leveraged for the second.

Regarding the **dynamic phantom model**, the methodology for its development involved extending the phantom structure of the KomaMRI simulator, modifying its simulation functions, enhancing its phantom visualization tool, and defining a new file format for the storage and sharing of digital phantoms. The design considerations and requirements were based on practical examples, and after the development phase, a series of experiments were conducted, including demonstrative tests and comparisons with prior simulation studies. These experiments confirm the accuracy and functionality of the developed dynamic model.

While the results validate the proposed approach, there are still aspects that could be improved to enhance the realism of the model and its efficiency. For instance, the way cardiac deformation is modelled and the impact of pre-calculating flow trajectories could be refined to better handle complex motion patterns. Additionally, execution times naturally increase with dynamic simulations compared to static ones. These aspects are discussed in Section 5.3.

As for the **web-based sequence editor**, the methodology for its development involved designing and implementing both the front-end and the back-end, as well as ensuring seamless integration and interaction between them. The primary motivations for this part of the work stemmed from the limitations and future work outlined in [1], which were presented in this document as background. Thus, the requirements for this part of the work were based on the limitations of the former, and the chosen programming technologies enabled the continuation of the line of work established in the aforementioned Bachelor's Thesis. As a result, we have seen practical use cases that confirm the utility of the tool, as well as its interactivity, responsiveness, and smoothness.

Some areas for improvement remain, particularly regarding multi-user concurrency and the ability to design more complex pulse sequences, such as radial and spiral acquisitions or arbitrary gradient waveforms and RF pulses. These aspects are further discussed in Section 5.3.

The evaluation of both developments has demonstrated that the initial design requirements have been successfully met, thereby indicating that the two main objectives of the Master's Thesis have also been achieved. In summary, the work presented has extended an existing MRI simulator with enhanced motion capabilities and created a freely accessible sequence design tool, both of which contribute to the broader MRI research and education community, as detailed in the following section.

5.2 Contributions

The main contributions of this Master's Thesis are as follows:

- The KomaMRI simulator, already powerful in its own right, has been enhanced with a new functionality that further improves it. It is now possible to define both simple motions, through a few parameters, and arbitrarily complex motions, through trajectories. This adds a high level of flexibility to the tool, and makes it suitable for both educational and research environments.
- Additionally, the improvement in the phantom visualization tool of KomaMRI has made the task of defining and visualizing dynamic phantoms much simpler and more intuitive.
- The ability to define arbitrarily complex motions has enabled research into complex MR phenomena, such as diffusion. In this regard, collaborators in the group have already contributed[1] to research using the functionalities described in this document.
- This contribution has also led to the definition of a new `.phantom` file format, which follows the HDF5 standard and allows for the storage and sharing of phantoms, thus enhancing the reproducibility of experiments.
- The web-based sequence design application provides the MRI community with a free-use and accessible tool that eliminates the need for local installations. This tool provides an environment for users to design and test pulse sequences, and serves both as a research and as an educational tool: on the one hand, users can explore and configure very specific details of the sequence, such as the duration of each block, or the shape and strength of the gradients, which makes the tool ideal for researchers. On the other hand, users can load pre-configured sequences, bypassing the specific details of each block and focusing on more general parameters like TE, TR, or flip angle, among others. This scenario is more suitable for users with a more technical background.
- The graphical functionalities added to the web interface allow users to precisely observe the effect of the SSG gradients on the anatomical model used for simulation. Hence, slice selection from the developed interface is no longer a blind task, but rather one guided by the graphical tool.
- Finally, although this document initially mentioned the independence between the two parts of the work, their integration ultimately results in a complete environment for sequence editing and simulation. This unified platform enables the execution of all kinds of acquisitions, both static and dynamic, with the ability to visualize the results, such as videos for cardiac cine. Some of these functionalities are identified as future lines of work (see Section 5.3), where a full integration of both contributions will be explored.

As mentioned in the relevant results sections, these contributions have led to the submission of several communications to both national [74] and international [76–78, 87, 88] conferences. Furthermore, both the dynamic phantom model and the web-based sequence editor are planned to be submitted for publication as journal articles.

5.3 Future work

In light of the discussions in Sections 5.1 and 5.2, and considering aspects that remain open or potential improvements identified through the use of the implemented functionalities, several points have been identified that could serve as the basis for future lines of work.

Concerning the **dynamic phantom model**, the future lines of work are as follows:

[1]See the contribution by Justino R. Rodríguez-Galván at: `https://juliahealth.org/KomaMRI.jl/stable/tutorial/06-DiffusionMotion/`.

- **Improve motion modelling**. Enhance the realism of simulated deformations, specifically for cardiac motion. This could be achieved by developing alternative approaches that go beyond treating deformation as individual spin motions derived from XCAT maps.
- **Optimize storage and memory management**. As also discussed in [8], pre-calculating trajectories from a CFD may produce extremely large and unmanageable trajectory files, specially for long simulations involving many spins and complex flow patterns. The challenge lies in the development of more efficient strategies, not only for storing trajectories but also for computing them. The decision, for instance, between pre-computing trajectories or calculating them on-the-fly, has a significant impact and must be carefully analyzed.
- **Improve performance**. Developing and implementing optimization strategies to mitigate the increase in execution times associated with dynamic phantom simulations is desirable. One potential solution could involve adopting a kernel-based approach, using the `KernelAbstractions` and `AcceleratedKernels` packages, to implement the functions that handle displacement calculations. This approach could provide more efficient computation and reduce processing time.
- There are additional aspects of the code that could be improved, such as using the HDF5-based JLD2.jl library instead of HDF5.jl for storing phantoms[2], or using vectors of `Motion` instances instead of encapsulating them within a `MotionList` structure[3]. These improvements are internal, they focus on optimization and code readability, and are intended to enhance the code's clarity and facilitate its extensibility.
- Finally, it could be considered to add more types of motion, more complex diffusion models, or even reintroduce an `Action` that allows defining motion as an analytical function, as was initially implemented. This could be useful in certain scenarios.

As for the **web-based sequence editor**, the following future lines of work have been identified:

- **Extend the block library**. Introduce additional block types in the editor, such as blocks that directly implement SE sequences or blocks for EPI acquisitions with non-cartesian trajectories, including radial and spiral patterns.
- **Extend the phantom library**. Expand the collection of phantoms available for simulation. In addition, enable users to upload, visualize, and simulate phantoms created locally on their computers.
- **Optimize the back-end for improved user experience**. Address the concurrency issue by implementing user authentication methods and establishing a mapping between simulation IDs and their associated users. Proper distributed computing management should be implemented by associating each Julia process with its corresponding user.
- Finally, **improve the dynamic MRI simulation capabilities**. On the GUI side, this involves allowing users to both configure the phantom and its motion, design cine sequences, and view the resulting cine simulation in the viewer, either as videos or animations. This final area of future work will enable a seamless integration between the two components developed in this Master's Thesis.

[2]See: `https://github.com/JuliaHealth/KomaMRI.jl/issues/544`.

[3]See: `https://github.com/JuliaHealth/KomaMRI.jl/issues/480`.

Appendices

Appendix A. Rotations and Spinors

A.1. Rotation Matrices

The rotation matrices around the canonical axes are shown below. Typically, the rotations occur in the rotating reference frame (see Appendix A.4). However, to avoid overloading the notation, apostrophes will be omitted unless the context makes them necessary. These matrices represent rotations around the axes following the left-hand rule. The matrices are:

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

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

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

A.2. Euler-Rodrigues Rotation Formula

Rotation around an axis other than the canonical ones requires a preliminary alignment of the rotation axis with one of the canonical axes. In 3D, this becomes even more complex for an arbitrarily oriented rotation vector, as it would involve preliminary rotations about two axes, followed by the desired rotation, and the subsequent reverse rotations about the same two axes.

This can, however, be alternatively achieved through the use of the Rodrigues rotation formula. If $\mathbf{v}$ is a vector in $\mathbb{R}^3$ and $\mathbf{k}$ is a unit vector which describes a rotation axis, Rodrigues formula provides the expression for the rotated vector $\mathbf{v}_{\text{rot}}$ when $\mathbf{v}$ is rotated by an angle θ around $\mathbf{k}$ following the right-hand rule:

$$\mathbf{v}_{\text{rot}} = \mathbf{v} + (\mathbf{k} \times \mathbf{v}) \sin\theta + (\mathbf{k} \cdot \mathbf{v})(1 - \cos\theta)\mathbf{k} \tag{A.4}$$

The axis vector $\mathbf{k}$ can be expressed as the cross product $\mathbf{a} \times \mathbf{b}$, where $\mathbf{a}$ and $\mathbf{b}$ are two non-zero vectors that define the plane of rotation. The angle θ is measured from $\mathbf{a}$ towards $\mathbf{b}$. Letting α denote the angle between $\mathbf{a}$ and $\mathbf{b}$, θ and α are not necessarily equal but are measured in the same direction. The unit axis vector can then be written as:

$$\mathbf{k} = \frac{\mathbf{a} \times \mathbf{b}}{|\mathbf{a} \times \mathbf{b}|} = \frac{\mathbf{a} \times \mathbf{b}}{|\mathbf{a}||\mathbf{b}| \sin\alpha}$$

Equation (A.4) can be expressed as the product of the initial vector $\mathbf{v}$ by the rotation matrix $\mathbf{R}(\mathbf{k}, \theta)$ [89]:

$$\mathbf{v}_{\text{rot}} = \mathbf{R}(\mathbf{k}, \theta)\mathbf{v} = \left(\mathbf{I} + (\sin\theta)\mathbf{K} + (1 - \cos\theta)\mathbf{K}^2\right)\mathbf{v} \tag{A.5}$$

where $\mathbf{I}$ is the 3×3 identity matrix, and $\mathbf{K}$ is the cross-product matrix of the unit rotation axis $\mathbf{k} = (k_x, k_y, k_z)$:

$$\mathbf{K} = \begin{bmatrix} 0 & -k_z & k_y \\ k_z & 0 & -k_x \\ -k_y & k_x & 0 \end{bmatrix} \quad \rightarrow \quad \mathbf{k} \times \mathbf{v} = \mathbf{K}\mathbf{v}$$

Expanding the expression, the Rodrigues rotation matrix (SO(3)) takes the following form [89]:

$$\mathbf{R} = \begin{bmatrix} cos\theta + u_x^2(1 - cos\theta) & u_x u_y(1 - cos\theta) - u_z sin\theta & u_x u_z(1 - cos\theta) + u_y sin\theta \\ u_y u_x(1 - cos\theta) + u_z sin\theta & cos\theta + u_y^2(1 - cos\theta) & u_y u_z(1 - cos\theta) - u_x sin\theta \\ u_z u_x(1 - cos\theta) - u_y sin\theta & u_z u_y(1 - cos\theta) + u_x sin\theta & cos\theta + u_z^2(1 - cos\theta) \end{bmatrix} \tag{A.6}$$

A.3. Spinors

A third alternative to represent 3D rotations is the use of *spinors*. These come from quaternion algebra (SU(2)), and provide a more compact and computationally efficient way to represent rotations. Let M_{xy} denote the transverse component of the vector[4] and M_z the longitudinal component. Thus, the following matrix is defined [90]:

$$\mathbf{V}_{SU} = \begin{bmatrix} M_z & M_{xy}^* \\ M_{xy} & -M_z \end{bmatrix}$$

The rotation matrix $\mathbf{R}_{SU}$ is now defined as:

$$\mathbf{R}_{SU} = \begin{bmatrix} \alpha & -\beta^* \\ \beta & \alpha^* \end{bmatrix}$$

where both α and β are, in general, complex scalars that must satisfy the condition $|\alpha|^2 + |\beta|^2 = 1$ for the matrix to qualify as a true rotation matrix. These parameters are known as Cayley-Klein parameters or *spinors*. Based on these, the rotation is defined as follows:

$$\mathbf{V}_{SU,+} = \mathbf{R}_{SU}\mathbf{V}_{SU}\mathbf{R}_{SU}^H \tag{A.7}$$

where H denotes the conjugate transpose. After performing the operation, the result is:

$$\mathbf{V}_{SU,+} = \begin{bmatrix} M_{z+} & M_{xy+}^* \\ M_{xy+} & -M_{z+} \end{bmatrix}$$

with

$$M_{xy+} = 2\alpha^*\beta M_z + (\alpha^*)^2 M_{xy} - \beta^2 M_{xy}^* \tag{A.8}$$

$$M_{z+} = \left(|\alpha|^2 - |\beta|^2\right) M_z - \alpha\beta M_{xy}^* - \alpha^*\beta^* M_{xy} = \left(|\alpha|^2 - |\beta|^2\right) M_z - 2\Re\{\alpha\beta M_{xy}^*\} \tag{A.9}$$

If two rotations are applied in series, the operation can be written as (eliminating the subscript SU for simplicity):

$$\mathbf{R}_2\mathbf{R}_1\mathbf{V}\mathbf{R}_1^H\mathbf{R}_2^H$$

The operation $\mathbf{R}_2\mathbf{R}_1$ results in a new *spinor* with the following values:

$$\alpha_f = \alpha_1\alpha_2 - \beta_1\beta_2^*$$
$$\beta_f = \alpha_1\beta_2 + \beta_1\alpha_2^*$$

In practical terms, suppose a rotation of ϕ radians is required around the axis $\mathbf{n} = (n_x, n_y, n_z)$ following the left-hand rule. The values of α and β for the resulting *spinor* are:

$$\alpha = \cos(\phi/2) + jn_z \sin(\phi/2) \tag{A.10}$$

$$\beta = j(n_x + jn_y)\sin(\phi/2) \tag{A.11}$$

The expressions in (A.8) and (A.9) can be rewritten in matrix form as:

$$\begin{bmatrix} M_{xy+} \\ M_{z+} \end{bmatrix} = \begin{bmatrix} (\alpha^*)^2 & -\beta^2 & 2\alpha^*\beta \\ -\alpha^*\beta^* & -\alpha\beta & |\alpha|^2 - |\beta|^2 \end{bmatrix} \begin{bmatrix} M_{xy} \\ M_{xy}^* \\ M_z \end{bmatrix}$$

[4]In this thesis M_{xy} is used with two different meanings depending of the context. First M_{xy} may denote the modulus of the transversal component vector $\mathbf{M}_{xy}$. Alternatively, M_{xy} may be the complex number that expresses the component in the $\hat{x}$ direction as the real part and the component in the $\hat{y}$ direction as imaginary part. Context will indicate which of the two representations applies.

or by means of a 3×3 matrix:

$$\begin{bmatrix} M_{xy_+} \\ M^*_{xy_+} \\ M_{z_+} \end{bmatrix} = \begin{bmatrix} (\alpha^*)^2 & -\beta^2 & 2\alpha^*\beta \\ -(\beta^*)^2 & \alpha^2 & 2\alpha\beta^* \\ -\alpha^*\beta^* & -\alpha\beta & |\alpha|^2 - |\beta|^2 \end{bmatrix} \begin{bmatrix} M_{xy} \\ M^*_{xy} \\ M_z \end{bmatrix} \tag{A.12}$$

Additionally, when the expressions in (A.8) and (A.9) are split into their real and imaginary components, a general expression for rotation around an arbitrarily oriented axis is obtained:

$$\begin{bmatrix} M_{x+} \\ M_{y_+} \\ M_{z_+} \end{bmatrix} = \begin{bmatrix} (\alpha_R^2 - \alpha_I^2) - (\beta_R^2 - \beta_I^2) & 2(\alpha_R\alpha_I - \beta_R\beta_I) & 2(\alpha_R\beta_R + \alpha_I\beta_I) \\ -2(\alpha_R\alpha_I + \beta_R\beta_I) & (\alpha_R^2 - \alpha_I^2) + (\beta_R^2 - \beta_I^2) & 2(\alpha_R\beta_I - \alpha_I\beta_R) \\ -2(\alpha_R\beta_R - \alpha_I\beta_I) & -2(\alpha_R\beta_I + \alpha_I\beta_R) & |\alpha|^2 - |\beta|^2 \end{bmatrix} \begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix} \tag{A.13}$$

A.4. Rotating reference frame

Suppose that a reference frame consisting of the vectors $\mathbf{i}'$, $\mathbf{j}'$, and $\mathbf{k}'$, initially coincident with the canonical reference frame $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$, rotates around the $\mathbf{k}$ axis following the left-hand rule (or equivalently, with a rotation vector $\boldsymbol{\omega} = \omega(-\mathbf{k})$ following the right-hand rule, with the thumb pointing in the direction of $-\mathbf{k}$). Thus, the vectors of this rotating frame can be expressed with respect to the canonical reference frame as follows:

$$\mathbf{i}' = \cos(\omega t)\mathbf{i} - \sin(\omega t)\mathbf{j} \tag{A.14}$$
$$\mathbf{j}' = \sin(\omega t)\mathbf{i} + \cos(\omega t)\mathbf{j} \tag{A.15}$$
$$\mathbf{k}' = \mathbf{k} \tag{A.16}$$

Note that these vectors are functions of time; it is then straightforward to verify that

$$\frac{d\mathbf{i}'}{dt} = \boldsymbol{\omega} \times \mathbf{i}' \tag{A.17}$$
$$\frac{d\mathbf{j}'}{dt} = \boldsymbol{\omega} \times \mathbf{j}' \tag{A.18}$$
$$\frac{d\mathbf{k}'}{dt} = \boldsymbol{\omega} \times \mathbf{k}' \tag{A.19}$$
$$\tag{A.20}$$

Suppose a vector $\boldsymbol{M}$, with components in the canonical reference frame (M_x, M_y, M_z). The same vector will have components (M'_x, M'_y, M'_z) in the rotating reference frame, and can be denoted as $\boldsymbol{M}_{rot}$. $\boldsymbol{M}$ and $\boldsymbol{M}_{rot}$ are equal component by component when expressed in the same basis, that is

$$M'_x\mathbf{i}'(\mathbf{i},\mathbf{j},\mathbf{k}) + M'_y\mathbf{j}'(\mathbf{i},\mathbf{j},\mathbf{k}) + M'_z\mathbf{k}'(\mathbf{i},\mathbf{j},\mathbf{k}) = M_x\mathbf{i} + M_y\mathbf{j} + M_z\mathbf{k} \tag{A.21}$$

Then, by simply expressing the vectors of the rotating frame as indicated by equations (A.14)–(A.16), it can be written:

$$\begin{bmatrix} M'_x \\ M'_y \\ M'_z \end{bmatrix} = \begin{bmatrix} \cos(\omega t) & -\sin(\omega t) & 0 \\ \sin(\omega t) & \cos(\omega t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix} \tag{A.22}$$

Note that this matrix is equal to $\mathbf{R}_z(-\omega t)$(A.3). Now, suppose a vector lies on the transverse plane (x, y); in this case, a two-dimensional representation would suffice. Let this vector be denoted as $\mathbf{B}_1$ in the original system and $\mathbf{B}_{1,rot}$ in the rotating system. The translation of the previous matrix equality to the 2D case is straightforward:

$$\begin{bmatrix} B'_{1,x} \\ B'_{1,y} \end{bmatrix} = \begin{bmatrix} \cos(\omega t) & -\sin(\omega t) \\ \sin(\omega t) & \cos(\omega t) \end{bmatrix} \begin{bmatrix} B_{1,x} \\ B_{1,y} \end{bmatrix} \tag{A.23}$$

As is known, rotations in the 2D case can alternatively be expressed as phase shift operations in the complex plane. Let $M_{xy} = M_x + jM_y$ and $M_{x'y'} = M_{x'} + jM_{y'}$. The vector M_{xy} will have a magnitude $|M_{xy}|$ and a phase θ. With respect to the rotating system, the corresponding vector will acquire an additional phase of ωt. Therefore:

$$M_{x'y'} = M_{xy}e^{j\omega t} \tag{A.24}$$

It is easy to verify that this result directly arises from applying the matrix operation in expression (A.22).

On the other hand, let us define:

$$\frac{d\boldsymbol{M}}{dt} = \frac{dM_x}{dt}\mathbf{i} + \frac{dM_y}{dt}\mathbf{j} + \frac{dM_z}{dt}\mathbf{k} \tag{A.25}$$

$$\frac{\partial \boldsymbol{M}_{rot}}{\partial t} = \frac{\partial M'_x}{\partial t}\mathbf{i}' + \frac{\partial M'_y}{\partial t}\mathbf{j}' + \frac{\partial M'_z}{\partial t}\mathbf{k}' \tag{A.26}$$

From these definitions, along with the equality (A.21) and the derivatives expressed in equations (A.17)–(A.19), it can be written:

$$\begin{aligned}
\frac{d\boldsymbol{M}}{dt} &= \frac{dM_{x'}}{dt}\mathbf{i}'(\mathbf{i},\mathbf{j},\mathbf{k}) + \frac{dM_{y'}}{dt}\mathbf{j}'(\mathbf{i},\mathbf{j},\mathbf{k}) + \frac{dM_{z'}}{dt}\mathbf{k}'(\mathbf{i},\mathbf{j},\mathbf{k}) + \\
&= M_{x'}\frac{d\mathbf{i}'(\mathbf{i},\mathbf{j},\mathbf{k})}{dt} + M_{y'}\frac{d\mathbf{j}'(\mathbf{i},\mathbf{j},\mathbf{k})}{dt} + M_{z'}\frac{d\mathbf{j}'(\mathbf{i},\mathbf{j},\mathbf{k})}{dt} = \\
&= \frac{\partial \boldsymbol{M}_{rot}}{\partial t} + M_{x'}\boldsymbol{\omega}\times\mathbf{i}' + M_{y'}\boldsymbol{\omega}\times\mathbf{j}' + M_{z'}\boldsymbol{\omega}\times\mathbf{k} \\
&= \frac{\partial \boldsymbol{M}_{rot}}{\partial t} + \boldsymbol{\omega}\times\left(M_{x'}\mathbf{i}' + M_{y'}\mathbf{j}' + M_{z'}\mathbf{k}\right) \\
&= \frac{\partial \boldsymbol{M}_{rot}}{\partial t} + \boldsymbol{\omega}\times\boldsymbol{M}_{rot}
\end{aligned} \tag{A.27}$$

Note, therefore, that, in general:

$$\frac{d\boldsymbol{M}}{dt} = \frac{d\boldsymbol{M}_{rot}}{dt} \neq \frac{\partial \boldsymbol{M}_{rot}}{\partial t}$$

Appendix B. Background on tagging and phase constrast

B.1. Solution of the Bloch equations in the presence of a gradient

We depart from Eq. (2.11). In this case, let $\mathbf{B} = [\mathbf{G}(t) \cdot \mathbf{r}(t)]\,\hat{z}$. We can express

$$\begin{aligned}
\frac{d\mathbf{M}(t)}{dt} &= \gamma \mathbf{M}(t) \times \underbrace{[\mathbf{G}(t) \cdot \mathbf{r}(t)]}_{\Lambda(t)}\,\hat{z} \\
&= \gamma \left(M_x\hat{x} + M_y\hat{y} + M_z\hat{z}\right) \times \Lambda(t)\hat{z} \\
&= \gamma\Lambda(t)\left[M_x(-\hat{y}) + M_y\hat{x}\right]
\end{aligned}$$

This expression indicates (a) that there is no time variation in M_z and (b) that we can express this equation using complex arithmetic, i.e., $M_{xy} = M_x + jM_y$, as follows

$$\begin{aligned}
\frac{dM_{xy}(t)}{dt} &= -\gamma\Lambda(t)j(M_x + jM_y) \\
&= -j\gamma\Lambda(t)M_{xy}
\end{aligned}$$

Hence

$$\begin{aligned}
\frac{dM_{xy}}{M_{xy}} &= -j\gamma\Lambda(t)dt \\
\int_{t_0}^{t} \frac{dM_{xy}}{M_{xy}} &= -j\gamma \int_{t_0}^{t} \Lambda(\tau)d\tau \\
\log M_{xy}(t) - \log M_{xy}(t_0) &= \int_{t_0}^{t} -j\gamma\Lambda(\tau)d\tau \\
M_{xy}(t) &= M_{xy}(t_0)e^{-j\gamma \int_{t_0}^{t} \mathbf{G}(\tau)\cdot\mathbf{r}(\tau)d\tau} \qquad \text{(B.1)}
\end{aligned}$$

B.2. Tagging: pulses and gradients

Recall Figure 2.23 about the SPAMM sequence. For simplicity, we will assume that both excitation pulses are such that $\theta_1 = \theta_2 = \pi/2$ and both are applied on the x axis. The first pulse makes the magnetization vector rotate about this axis, so the complete longitudinal component M_z^0 becomes a transversal component located on the y axis. The gradient then is responsible for a phase dispersion in which the phase term will be position dependent (recall Eq. (B.1)). The second pulse makes the components on the $x-y$ plane rotate onto the $x-z$ plane, with unaltered components on the x axis and the previous components on the y axis will be now components on the z axis. Then, the x components will relax due to the T_2 effect, and this effect will be more favored by the spoiler gradient indicated in Figure 2.23. With respect to the z components, they will be function of the of the spatial position on the y of the spin in the time instant right before the pulse. Hence, if at this moment a conventional imaging sequence is applied, the longitudinal component of each spin will be spatially modulated, which will give rise to an intensity pattern in the resulting image that will vary according to this modulation.

For a quantitative explanation let $M_z^{\theta_i}(0_+)$ and $M_{xy}^{\theta_i}(0_+)$ be the longitudinal and transversal components, respectively, right after the pulse with flip angle θ_i. Then

$$M_{xy}^{\theta_1}(0_+) = jM_z^0$$

When gradient $\mathbf{G}$ is applied the effect is

$$\begin{aligned} M_{xy}(t) &= M_{xy}^{\theta_1}(0_+)e^{-j\gamma\int_0^t \mathbf{G}(\tau)\cdot\mathbf{r}d\tau} = M_{xy}^{\theta_1}(0_+)e^{-j\phi(\mathbf{r},t)} \\ &= jM_z^0\left(\cos(\phi(\mathbf{r},t)) - j\sin(\phi(\mathbf{r},t))\right). \end{aligned}$$

After the second pulse, in the case that the elapsed time between both pulses is t_{tag} then the longitudinal component will be

$$M_z^{\theta_2}(0+) = M_z^0\cos(\phi(\mathbf{r},t_{tag})) \tag{B.2}$$

which will rise to a longitudinal component with a sinusoidal spatial pattern, the specific shape of which will depend of the waveform of $\mathbf{G}$.

B.3. Phase Contrast: pulses and gradients

Recall Eq. (B.1) and let's assume that our spin of interest follows a trajectory given by $\mathbf{r}(t)$, starting at $\mathbf{r}(t_0)$. At this time instant the transversal component associated to this spin is $M_{xy}(t_0)$ and then we apply a gradient $\mathbf{G}(t)$. Before applying this equation let's write:

$$\mathbf{r}(t) = \sum_{n=0}^{\infty}\left.\frac{d^n\mathbf{r}(t)}{dt^n}\right|_{t=t_0}\frac{1}{n!}(t-t_0)^n$$

and now we rewrite the phase term in Eq. (B.1) as

$$\begin{aligned} \phi &= \gamma\int_{t_0}^{t}\mathbf{G}(\tau)\cdot\mathbf{r}(\tau)d\tau = \gamma\sum_{n=0}^{\infty}\int_{t_0}^{t}\mathbf{G}(\tau)\cdot\left.\frac{d^n\mathbf{r}(t)}{dt^n}\right|_{t=t_0}\frac{1}{n!}(\tau-t_0)^n d\tau \\ &= \gamma\sum_{n=0}^{\infty}\frac{1}{n!}\int_{t_0}^{t}\left[G_x(\tau)\left.\frac{d^n r_x(t)}{dt^n}\right|_{t=t_0} + G_y(\tau)\left.\frac{d^n r_y(t)}{dt^n}\right|_{t=t_0} + G_z(\tau)\left.\frac{d^n r_z(t)}{dt^n}\right|_{t=t_0}\right](\tau-t_0)^n d\tau \\ &= \sum_{n=0}^{\infty}\left[\phi_{nx}+\phi_{ny}+\phi_{nz}\right] \end{aligned}$$

with

$$\phi_{nu} = \frac{\gamma}{n!}\int_{t_0}^{t}G_u(\tau)\left.\frac{d^n r_u(t)}{dt^n}\right|_{t=t_0}(\tau-t_0)^n d\tau$$

with u any of the spatial coordinates (x, y, z). As can be observed the phase accumulation is additive in the components of the gradients.

Assume spins undergo a trajectory with constant velocity. If this is the case, then $\left.\frac{d^n r_u(t)}{dt^n}\right|_{t=t_0} = 0, \forall n \geq 2$. Let now $A(t)$ be a function that is non-null only within the interval $t_0 \leq t \leq t_0 + \Delta$ and let

$G_u(t) = A(t) - A(t-T)$, with $T \geq \Delta$. If this is the case, the following holds:

$$\begin{aligned}
\phi_{0u} &= \gamma r_u(t_0) \left[\int_{t_0}^{t_0+\Delta} A(\tau) d\tau - \int_{t_0+T}^{t_0+T+\Delta} A(\tau - T) d\tau \right] \overset{\alpha=\tau-T}{=} \\
&= \gamma r_u(t_0) \left[\int_{t_0}^{t_0+\Delta} A(\tau) d\tau - \int_{t_0}^{t_0+\Delta} A(\alpha) d\alpha \right] = 0 \\
\phi_{1u} &= \gamma v_u(t_0) \left[\int_{t_0}^{t_0+\Delta} A(\tau)(\tau - t_0) d\tau - \int_{t_0+T}^{t_0+T+\Delta} A(\tau - T)(\tau - t_0) d\tau \right] \overset{\alpha=\tau-T}{=} \\
&= \gamma v_u(t_0) \left[\int_{t_0}^{t_0+\Delta} A(\tau)(\tau - t_0) d\tau - \int_{t_0}^{t_0+\Delta} A(\alpha)(\alpha - t_0 + T) d\alpha \right] = 0 \\
&= -\gamma v_u(t_0) T \int_{t_0}^{t_0+\Delta} A(\alpha) d\alpha
\end{aligned}$$

As can be observed, with the selection of the gradient we have made, ϕ_{0u}, i.e., the term that depends on the spin position at $t = t_0$ becomes null while ϕ_{1u} has a value that is proportional to $v_u(t_0) = \left.\frac{dr_u(t)}{dt}\right|_{t=t_0}$ the velocity of the spin —in the direction of the gradient— at time instant $t = t_0$. This choice of gradient is referred to as *bipolar gradient.* Notice that the phase is independent of the shape of $A(t)$ as it only depends on its area, and it also depends on the separation T between $A(t)$ and $A(t-T)$.

Finally, if the gradient is appplied with the polarity indicated in Fig. 2.26A, and assuming that the RF pulse gives rise to an initial phase ϕ_0, we will end up with an expression similar to Eq. (2.34):

$$\phi_{1u} = 2\pi\bar{\gamma} m_1 v_u(t_0) + \phi_0$$

where

$$m_1 = T \int_{t_0}^{t_0+\Delta} |A(\alpha)| d\alpha$$

Consequently, if a gradient is applied only in one spatial direction we obtain a phase encoding in the transversal component. If at this point a second pulse converts the transversal component back into a longitudinal component, in a similar fashion as we did for the tagging sequence, we can make use of a regular sequence to get a complex image with velocity information in its phase.

Appendix C. Flow data acquisition and processing

The development of this part of the work was carried out by contributors from the Department of Energetic Engineering and Fluid Mechanics[5] at the University of Valladolid.

Arbitrary motions, including flow, need to be characterized in KomaMRI as spin trajectories. As anatomic geometries and flow models become moderately complex, it becomes unfeasible for users to manually define these trajectories, as they may result from complex and irregular flow patterns which are difficult to model. Thus, it is necessary to define a methodology for acquiring trajectories derived from realistic and arbitrarily complex flow fields. This methodology would require the use of external software specialized in generating and managing flow-related data, and should be based on two steps:

1. **Fluid field acquisition**. This involves obtaining accurate representations of the fluid fields, which can be generated through CFD, or may also come from experimental data or third-party repositories. The accuracy and resolution of these fluid fields are crucial, since they directly impact the quality of the spin trajectories derived in the next step.

2. **Particle seeding and trajectory tracking**. This consists of the introduction of a set of virtual particles within the fluid field. These particles will move according to the Eulerian velocity value at each point in space and time. Trajectories are resolved as stated in Equation 2.43, typically by means of numerical integration methods, where the particle positions are iteratively updated based on the velocity field at each time step. Note that the particle seeding strategy —random, uniform, etc.— also determines the results obtained [46].

Figure C.1 illustrates the methodology followed for this specific case, highlighting the technologies utilized at each step.

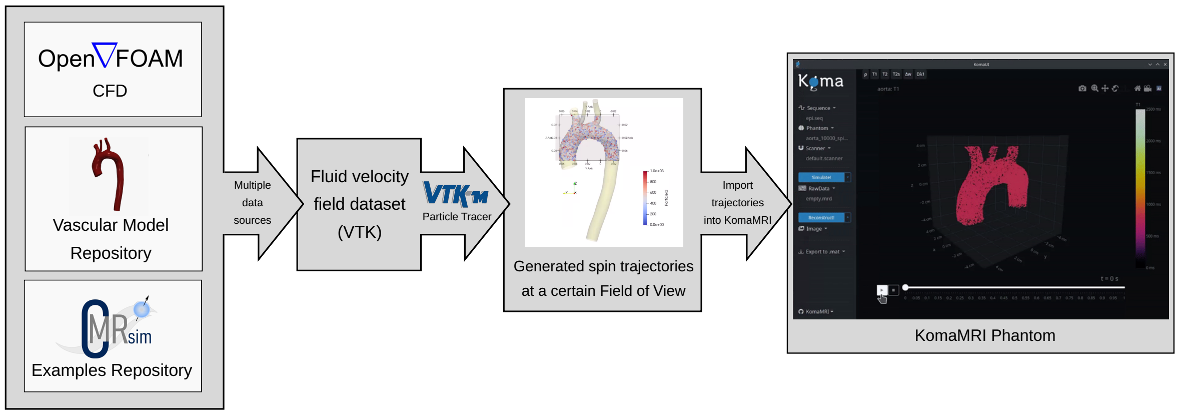

OpenFOAM
CFD
Vascular Model Repository
CMRsim
Examples Repository
Multiple data sources
Fluid velocity field dataset (VTK)
VTK-m
Particle Tracer
Generated spin trajectories at a certain Field of View
Import trajectories into KomaMRI
KomaMRI Phantom

Figure C.1: Diagram of the methodology implemented to derive spin trajectories from arbitrarily complex flow data and import them into the KomaMRI simulator.

Regarding the first step, several data sources were used to obtain the geometries and corresponding fields. First, the Vascular Model Repository (VMR) [91], a public[6] library of cardiovascular datasets, was accessed to obtain realistic aorta models. Additionally, the specialized software OpenFOAM [92] was employed to apply an in-house CFD method, generating fields for other simple geometries as well as more complex ones, such as an atrium with an appendage, which is still under development. Furthermore, although the models obtained from the VMR often include fluid fields alongside the geometry, these fields may lack the spatial or temporal resolution required for accurate particle tracking. For this reason, OpenFOAM is also used to recompute CFD simulations for these geometries, ensuring more precise velocity data. Lastly, and particularly tailored to the experiments in this work, some

[5]Group of prof. José Benito Sierra-Pallares, located at Escuela de Ingenierías Industriales, Universidad de Valladolid.

[6]Vascular Model Repository, available in https://www.vascularmodel.com/.

datasets available in the CMRsim repository[7] were also utilized, with the stenosis U-bend tube dataset being particularly relevant. The velocity field datasets from the mentioned sources are stored in VTK files, using either legacy (`.vtk`), image (`.vti`) or unstructured (`.vtu`) formats.

As for the second step, which involves obtaining the trajectories from the flow fields, the specialized library VTK-m [65] is employed. This library is specifically optimized to solve particle trajectories over arbitrary fluid fields, ensuring both precision and efficiency. Moreover, recent contributions [93] have further improved the performance of these operations on multi-core architectures, emphasizing the device-agnostic nature of this framework. VTK-m, and specifically the developed scripts[8], allow working with both structured meshes –that is, images–, and unstructured meshes of any type. This script developed in VTK-m not only computes the trajectories but also detects when a particle exits the volume defined by the field of view. When this occurs, two operations are performed:

- The particle is reseeded at a random position over the predefined entry surface.
- A reset flag is triggered and stored. This allows for the generation of the aforementioned `spin_reset` matrix.

Finally, the particle trajectories and reset flags are exported to an HDF5 file and subsequently imported into the KomaMRI simulator, thereby defining its phantom.

[7]CMRsim example resources: `https://gitlab.ethz.ch/ibt-cmr/mri_simulation/cmrsim/-/tree/master/notebooks/example_resources`.

[8]Available at: `https://github.com/jsierra-pallares/spinAdvection`.

Appendix D. Phantom file format

Phantom file directory tree (HDF5)

```
/                                                 HDF5 elements:
├── Version::String                                 Group
├── Name::String                                    Dataset
├── Ns::Int                                         Dimensions
├── Dims::Int                                       Attributes
├── position/ [m]
│   ├── x [N_spins x 1]
│   ├── y [N_spins x 1]
│   └── z [N_spins x 1]
├── contrast/
│   ├── rho    [N_spins x 1]
│   ├── T1     [N_spins x 1] [s]
│   ├── T2     [N_spins x 1] [s]
│   └── Deltaw [N_spins x 1] [rad/s]
└── motion/
    ├── motion_<id>/
    │   ├── action/::Action
    │   ├── time/  ::TimeCurve
    │   └── spins/ ::SpinSpan
    │    ...
    └── motion_<id>/
        ├── action/::Action
        ├── time/  ::TimeCurve
        └── spins/ ::SpinSpan
```

TimeCurve

```
...
└── time/
    ├── type::String = "TimeCurve"
    ├── periodic::String = <"true"|"false">
    ├── t [1 x N_times] [s]
    └── t_unit [1 x N_times] [s]
```

Action types

```
...
└── action/
    ├── type::String = "Translate"
    ├── dx::Number [m]
    ├── dy::Number [m]
    └── dz::Number [m]
```

```
...
└── action/
    ├── type::String = "Rotate"
    ├── pitch::Number [º]
    ├── roll::Number  [º]
    └── yaw::Number   [º]
```

```
...
└── action/
    ├── type::String = "HeartBeat"
    ├── circumferential_strain::Number
    ├── radial_strain::Number
    └── longitudinal_strain::Number
```

```
...
└── action/
    ├── type::String = "Path"
    ├── dx [N_spins x N_discrete_times] [m]
    ├── dy [N_spins x N_discrete_times] [m]
    └── dz [N_spins x N_discrete_times] [m]
```

```
...
└── action/
    ├── type::String = "FlowPath"
    ├── dx [N_spins x N_discrete_times] [m]
    ├── dy [N_spins x N_discrete_times] [m]
    ├── dz [N_spins x N_discrete_times] [m]
    └── spin_reset [N_spins x N_discrete_times]
```

SpinSpan types

```
...
└── spins/
    ├── type::String = "SpinRange"
    └── range::String
```

```
...
└── spins/
    └── type::String = "AllSpins"
```

Appendix E. Amdahl's Law

Amdahl's Law is a formula used to determine the maximum performance improvement of a system when only a part of it is improved. It was formulated by Gene Amdahl in 1967 [94].

Let speedup be the original execution time divided by an enhanced execution time [94]. The Amdahl's Law states that, when a fraction f of a computation is enhanced by a speedup factor S, the overall speedup can be expressed as follows [95]:

$$\text{speedup}_{\text{enhanced}}(f, S) = \frac{1}{(1-f) + \frac{f}{S}} \tag{E.1}$$

which implies that the performance improvement increases as the enhanced component is used for a larger fraction of the total time. Consequently, the improvement is limited by that fraction.

When the enhancement of f is due to the parallelization of the process across multiple devices (whether CPU cores or GPUs), the speedup factor is directly determined by the number of devices used, n [95]:

$$\text{speedup}_{\text{parallel}}(f, n) = \frac{1}{(1-f) + \frac{f}{n}} \tag{E.2}$$

With this assumption, we can say that:

- A fraction f of the program's execution time is perfectly parallelizable, which means that it can be executed simultaneously across multiple devices with no additional scheduling overhead.
- The remaining fraction, $(1-f)$, is completely sequential, which means that it cannot be parallelized and must be executed on a single device.

This means that the total speedup factor of a program is limited by its sequential, non-parallelizable part. Additionally, if the parallelizable fraction of the program is small —which means that most of the execution time is sequential—, fewer devices will be needed to achieve the maximum possible speedup. This is because the benefit of adding more devices quickly diminishes, as the majority of the work still needs to be done sequentially.

Bibliography

[1] P. Villacorta-Aylagas, C. Alberola-López, and M. Rodríguez-Cayetano, *Construcción de un editor y simulador de secuencias de imagen de resonancia magnética con Qt*. Bachelor Thesis, Universidad de Valladolid, 2022.

[2] R. H. Caverly, “MRI Fundamentals: RF Aspects of Magnetic Resonance Imaging (MRI)”, *IEEE Microwave Magazine*, vol. 16, no. 6, pp. 20–33, 2015.

[3] 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.

[4] C. Castillo-Passi, R. Coronado, G. Varela-Mattatall, C. Alberola-López, R. Botnar, and P. Irarrazaval, “KomaMRI.jl: An open-source framework for general MRI simulations with GPU acceleration”, *Magnetic Resonance in Medicine*, vol. 90, no. 1, pp. 329–342, 2023.

[5] 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.

[6] F. Liu, R. Kijowski, and W. F. Block, “MRiLab: Performing fast 3D parallel MRI numerical simulation on a simple PC”, in *2013 ISMRM & ISMRT Annual Meeting & Exhibition*, vol. 2072, Univerity of Wisconsin, 2013.

[7] F. Liu, J. V. Velikina, W. F. Block, R. Kijowski, and A. A. Samsonov, “Fast realistic MRI Simulations based on Generalized Multi-Pool Exchange Tissue Model”, *IEEE Transactions on Medical Imaging*, vol. 36, no. 2, pp. 527–537, 2017.

[8] J. Weine, C. McGrath, P. Dirix, S. Buoso, and S. Kozerke, “CMRsim–A python package for cardiovascular MR simulations incorporating complex motion and flow”, *Magnetic Resonance in Medicine*, vol. 91, no. 6, pp. 2621–2637, 2024.

[9] 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, 2014.

[10] C. G. Xanthis and A. H. Aletras, “coreMRI: A high-performance, publicly available MR simulation platform on the cloud”, *PLOS ONE*, vol. 14, no. 5, pp. 1–26, May 2019.

[11] C. G. Xanthis, I. E. Venetis, A. Chalkias, and A. H. Aletras, “High performance MRI simulations of motion on multi-GPU systems”, *Journal of Cardiovascular Magnetic Resonance*, vol. 16, no. 1, p. 48, 2014.

[12] 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.

[13] K. J. Layton, S. Kroboth, F. Jia, S. Littin, H. Yu, J. Leupold, J.-F. Nielsen, T. Stöcker, and M. Zaitsev, “Pulseq: A rapid and hardware-independent pulse sequence prototyping framework”, *Magnetic Resonance in Medicine*, vol. 77, no. 4, pp. 1544–1552, 2017.

[14] S. J. Inati, J. D. Naegele, N. R. Zwart, V. Roopchansingh, M. J. Lizak, D. C. Hansen, C.-Y. Liu, D. Atkinson, P. Kellman, S. Kozerke, H. Xue, A. E. Campbell-Washburn, T. S. Sørensen, and M. S. Hansen, “ISMRM Raw data format: A proposed standard for MRI raw datasets”, *Magnetic Resonance in Medicine*, vol. 77, no. 1, pp. 411–421, 2017.

[15] 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.

[16] Z. P. Liang and P. C. Lauterbur, *Principles of magnetic resonance imaging. A Signal Processing Perspective.* SPIE Optical Engineering Press Bellingham, 2000.

[17] E. Moya-Sáez, C. Alberola-López, and F. Simmross-Wattenberg, *Síntesis de imagen de resonancia magnética mediante GPU.* Bachelor’s Thesis, Universidad de Valladolid, 2017.

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

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

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

[21] M. Jouda, J. G. Korvink, and J. Anders, “Innovative Concepts for the Electronic Interface of Massively Parallel MRI Phased Imaging Arrays”, Ph.D. dissertation, 2016.

[22] 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.

[23] 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.

[24] M. Bernstein, K. King, and X. Zhou, *Handbook of MRI pulse sequences.* Elsevier, 2004.

[25] D. Weishaupt, V. Köchli, and B. Marincek, *How Does MRI Work?: An Introduction to the Physics and Function of Magnetic Resonance Imaging.* Jan. 2006.

[26] A. D. Elster. “Questions And Answers in MRI”. [Online], Available in: https://mriquestions.com (Last accessed: Oct. 24, 2024).

[27] G. B. Chavhan, P. S. Babyn, B. G. Jankharia, H.-L. M. Cheng, and M. M. Shroff, “Steady-State MR Imaging Sequences: Physics, Classification, and Clinical Applications”, *RadioGraphics*, vol. 28, no. 4, pp. 1147–1160, 2008.

[28] J. Lafuente and L. Hernández, “Técnica de la imagen por resonancia magnética”, in *Resonancia Magnética del Sistema Músculo-Esquelético*, Sociedad Española de Radiología Musculoesquelética, 2016, pp. 9–28.

[29] E. Martín-González, C. Alberola-López, and P. Casaseca-de-la-Higuera, “Parallelization and Deep Learning Techniques for the Registration and Reconstruction of Dynamic Cardiac Magnetic Resonance Imaging”, Ph.D. dissertation, 2023.

[30] L. Guo, J. Derbyshire, and D. Herzka, “Pseudo-projection-driven, self-gated cardiac cine imaging using cartesian golden step phase encoding”, *Magnetic resonance in medicine*, vol. 76, Oct. 2015.

[31] M. Ládrová, R. Martinek, J. Nedoma, P. Hanzlíková, M. Nelson, R. Vilimkova Kahankova, J. Brablík, and J. Kolarik, “Monitoring and Synchronization of Cardiac and Respiratory Traces in Magnetic Resonance Imaging: A Review”, *IEEE Reviews in Biomedical Engineering*, pp. 200–221, 2022.

[32] M. L. Shehata, S. Cheng, N. F. Osman, D. A. Bluemke, and J. A. Lima, “Myocardial tissue tagging with cardiovascular magnetic resonance,” *Journal of Cardiovascular Magnetic Resonance*, vol. 11, no. 1, p. 55, 2009.

[33] D. T. Wymer, K. P. Patel, W. F. Burke, and V. K. Bhatia, “Phase-Contrast MRI: Physics, Techniques, and Clinical Applications”, *RadioGraphics*, vol. 40, no. 1, pp. 122–140, 2020.

[34] K. S. Nayak, J.-F. Nielsen, M. A. Bernstein, M. Markl, P. D. Gatehouse, R. M. Botnar, D. Saloner, C. Lorenz, H. Wen, B. S. Hu, F. H. Epstein, J. N. Oshinski, and S. V. Raman, “Cardiovascular magnetic resonance phase contrast imaging,” *Journal of Cardiovascular Magnetic Resonance*, vol. 17, no. 1, p. 71, 2015.

[35] N. Wake, C. Ianniello, R. Brown, and C. M. Collins, “Chapter 14 - 3D Printed Imaging Phantoms”, in *3D Printing for the Radiologist*, N. Wake, Ed., Elsevier, 2022, pp. 175–189.

[36] K. E. Keenan, M. Ainslie, A. J. Barker, M. A. Boss, K. M. Cecil, C. Charles, T. L. Chenevert, L. Clarke, J. L. Evelhoch, P. Finn, D. Gembris, J. L. Gunter, D. L. Hill, C. R. Jack Jr., E. F. Jackson, G. Liu, S. E. Russek, S. D. Sharma, M. Steckner, K. F. Stupic, J. D. Trzasko, C. Yuan, and J. Zheng, "Quantitative magnetic resonance imaging phantoms: A review and the need for a system phantom", *Magnetic Resonance in Medicine*, vol. 79, no. 1, pp. 48–61, 2018.

[37] T. Puiseux, A. Sewonu, R. Moreno, S. Mendez, and F. Nicoud, "Numerical simulation of time-resolved 3D phase-contrast magnetic resonance imaging", *PLOS ONE*, vol. 16, no. 3, pp. 1–32, Mar. 2021.

[38] C. Graf, A. Rund, C. S. Aigner, and R. Stollberger, "Accuracy and performance analysis for Bloch and Bloch-McConnell simulation methods", *Journal of Magnetic Resonance*, vol. 329, p. 107 011, 2021.

[39] A. Barrero-Ripoll and M. Pérez-Saborid-Sánchez-Pastor, *Fundamentos y Aplicaciones de Mecánica de Fluidos*. McGraw-Hill, 2005.

[40] L. Jou and D. Saloner, "A numerical study of magnetic resonance images of pulsatile flow in a two dimensional carotid bifurcation: A numerical study of MR images", *Medical Engineering & Physics*, vol. 20, no. 9, pp. 643–652, 1998.

[41] S. Lorthois, J. Stroud-Rossman, S. Berger, L.-D. Jou, and D. Saloner, "Numerical Simulation of Magnetic Resonance Angiographies of an Anatomically Realistic Stenotic Carotid Bifurcation", *Annals of Biomedical Engineering*, vol. 33, pp. 270–283, 2005.

[42] S. Petersson, P. Dyverfeldt, R. Gårdhagen, M. Karlsson, and T. Ebbers, "Simulation of phase contrast MRI of turbulent flow", *Magnetic Resonance in Medicine*, vol. 64, no. 4, pp. 1039–1046, 2010.

[43] I. Marshall, "Computational simulations and experimental studies of 3D phase-contrast imaging of fluid flow in carotid bifurcation geometries", *Journal of Magnetic Resonance Imaging*, vol. 31, no. 4, pp. 928–934, 2010.

[44] A. Fortin, S. Salmon, J. Baruthio, M. Delbany, and E. Durand, "Flow MRI simulation in complex 3D geometries: Application to the cerebral venous network", *Magnetic Resonance in Medicine*, vol. 80, no. 4, pp. 1655–1665, 2018.

[45] S. D. Cohen, A. C. Hindmarsh, and P. F. Dubois, "CVODE, A Stiff/Nonstiff ODE Solver in C", *Computers in physics*, vol. 10, no. 2, pp. 138–143, 1996.

[46] A. Fortin, E. Durand, and S. Salmon, "Extension of an MRI Simulator Software for Phase Contrast Angiography Experiments", in *Biomedical Simulation - 6th International Symposium, ISBMS 2014*, Strasbourg, France: Springer, Oct. 2014, pp. 150–154.

[47] H. M. Hanson, B. Eiben, J. R. McClelland, M. van Herk, and B. C. Rowland, "Technical Note: Four-dimensional deformable digital phantom for MRI sequence development", *Medical Physics*, vol. 48, no. 9, pp. 5406–5413, 2021.

[48] W. P. Segars, G. Sturgeon, S. Mendonca, J. Grimes, and B. M. W. Tsui, "4D XCAT phantom for multimodality imaging research", *Medical Physics*, vol. 37, no. 9, pp. 4902–4915, 2010.

[49] J. Weine, C. McGrath, and S. Kozerke, "CMRSeq - A Python package for intuitive sequence design", in *2023 ISMRM & ISMRT Annual Meeting & Exhibition*, Toronto, Canada: ISMRM, 2023.

[50] D. G. F. Huilier, "An overview of the lagrangian dispersion modeling of heavy particles in homogeneous isotropic turbulence and considerations on related les simulations", *Fluids*, vol. 6, no. 4, 2021.

[51] 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.

[52] M. Kittisopikul, T. E. Holy, T. Aschan, and Contributors. "Interpolations.jl". [Software], Available in: https://github.com/JuliaMath/Interpolations.jl (Last accessed: Nov. 18, 2024).

[53] V. Churavy and Contributors. "KernelAbstractions.jl". [Software], Available in: https://github.com/JuliaGPU/KernelAbstractions.jl (Last accessed: Nov. 18, 2024).

[54] J. Computing and Contributors. “Functors.jl”. [Software], Available in: https://github.com/FluxML/Functors.jl (Last accessed: Nov. 18, 2024).

[55] S. Lyon and Contributors. “Adapt.jl”. [Software], Available in: https://github.com/JuliaGPU/Adapt.jl (Last accessed: Nov. 18, 2024).

[56] T. E. Holy, S. Kornblith, and Contributors. “HDF5.jl”. [Software], Available in: https://github.com/JuliaIO/HDF5.jl (Last accessed: Nov. 18, 2024).

[57] L. A. Wasser. “Hierarchical Data Formats - What is HDF5?” [Online], Available in: https://www.neonscience.org/resources/learning-hub/tutorials/about-hdf5 (Last accessed: Nov. 26, 2024).

[58] S. Lyon and Contributors. “PlotlyJS.jl”. [Software], Available in: https://github.com/JuliaPlots/PlotlyJS.jl (Last accessed: Nov. 18, 2024).

[59] N. Ortega and Contributors. “Oxygen.jl”. [Software], Available in: https://github.com/OxygenFramework/Oxygen.jl (Last accessed: Nov. 12, 2024).

[60] S. Ramírez. “FastAPI”. [Software], Available in: https://fastapi.tiangolo.com (Last accessed: Nov. 13, 2024).

[61] The Qt Company. “Qt, cross-platform software development for embedded & desktop”. [Software], Available in: https://www.qt.io/ (Last accessed: Nov. 12, 2024).

[62] The Qt Company. “Qt documentation”. [Online], Available in: https://doc.qt.io (Last accessed: Nov. 12, 2024).

[63] Kitware. “About CMake”. [Online], Available in: https://cmake.org/overview/ (Last accessed: Nov. 12, 2024).

[64] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit*, 4th ed. Kitware, 2006.

[65] K. Moreland, C. Sewell, W. Usher, L.-t. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, M. Larsen, C.-M. Chen, R. Maynard, and B. Geveci, “VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures”, *IEEE Computer Graphics and Applications*, vol. 36, no. 3, pp. 48–58, 2016.

[66] Mozilla Developer Network (MDN) Contributors. “Web technology for developers”. [Online], Available in: https://developer.mozilla.org/en-US/docs/Web (Last accessed: Nov. 12, 2024).

[67] R. Dahl, Node.js Developers, and Joyent Inc. “Node.js”. [Software], Available in: https://nodejs.org (Last accessed: Nov. 12, 2024).

[68] Kitware Inc. “VTK.js”. [Software], Available in: https://kitware.github.io/vtk-js/ (Last accessed: Nov. 12, 2024).

[69] A. Rossberg. “WebAssembly Specifications”. [Online], Available in: https://webassembly.github.io/spec/ (Last accessed: Nov. 12, 2024).

[70] Emscripten Contributors. “Emscripten documentation”. [Online], Available in: https://emscripten.org/docs/ (Last accessed: Nov. 12, 2024).

[71] Red Hat, Inc. “What is a REST API?” [Online], Available in: https://www.redhat.com/en/topics/api/what-is-a-rest-api (Last accessed: Feb. 18, 2025).

[72] Graham Wideman. “Orientation and Voxel-Order Terminology: RAS, LAS, LPI, RPI, XYZ and All That”. [Online], Available in: http://www.grahamwideman.com/gw/brain/orientation/orientterms.htm (Last accessed: Nov. 20, 2024).

[73] X. Zhang, R. Vinu Alexander, J. Yuan, and Y. Ding, “Computational Analysis of Cardiac Contractile Function”, *Current Cardiology Reports*, vol. 24, pp. 1983–1994, 2022.

[74] P. Villacorta-Aylagas, I. Ferández-Arias, C. Castillo-Passi, P. Irarrázaval-Mena, F. Simmross-Wattenberg, M. Rodríguez-Cayetano, and C. Alberola-López, “A Cross-Platform Editor and Simulator of Magnetic Resonance Imaging Sequences: Design and Implementation”, in *CASEIB 2022*, Valladolid, Spain: Sociedad Española de Ingeniería Biomédica (SEIB), 2022.

[75] OpenAPI Initiative. "OpenAPI Specification". [Online], Available in: https://swagger.io/specification/ (Last accessed: Feb. 18, 2025).

[76] P. Villacorta-Aylagas, C. Castillo-Passi, R. M. Menchón-Lara, P. Irarrázaval, J. B. Sierra-Pallares, and C. Alberola-López, "A KomaMRI Phantom Extension for Integrated Dynamic Imaging", in *2024 ISMRM & ISMRT Annual Meeting & Exhibition*, Singapore: ISMRM, May 2024.

[77] P. Villacorta-Aylagas, C. Castillo-Passi, R. M. Menchón-Lara, J. Anatol-Hernández, P. Irarrázaval, J. B. Sierra-Pallares, and C. Alberola-López, "New Methods for Defining Dynamic Phantoms in KomaMRI", in *ISMRM Iberian Chapter Annual Meeting 2024*, Porto, Portugal: ISMRM, Jul. 2024.

[78] P. Villacorta-Aylagas, C. Castillo-Passi, R. M. Menchón-Lara, J. Anatol-Hernández, P. Irarrázaval, J. B. Sierra-Pallares, and C. Alberola-López, "Flexible Motion Models for GPU-accelerated MRI Simulations with Complex Motion and Flow using KomaMRI", in *ESMRMB Annual Meeting 2024*, Barcelona, Spain: ESMRMB, Oct. 2024.

[79] F. Godenschweger, U. Kägebein, D. Stucht, U. Yarach, A. Sciarra, R. Yakupov, F. Lüsebrink, P. Schulze, and O. Speck, "Motion correction in MRI of the brain", *Physics in Medicine & Biology*, vol. 61, no. 5, R32, Feb. 2016.

[80] S. R. Tecelão, J. J. Zwanenburg, J. P. Kuijer, and J. T. Marcus, "Extended harmonic phase tracking of myocardial motion: Improved coverage of myocardium and its effect on strain results", *Journal of Magnetic Resonance Imaging*, vol. 23, no. 5, pp. 682–690, 2006.

[81] C. Yuan, G. T. Gullberg, and D. L. Parker, "The solution of Bloch equations for flowing spins during a selective pulse using a finite difference method", *Medical Physics*, vol. 14, no. 6, pp. 914–921, 1987.

[82] A. Hazra, G. Lube, and H.-G. Raumer, "Numerical simulation of Bloch equations for dynamic magnetic resonance imaging", *Applied Numerical Mathematics*, vol. 123, pp. 241–255, 2018.

[83] C. Rackauckas and Q. Nie, "DifferentialEquations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia", *Journal of Open Research Software*, vol. 5, no. 1, 2017.

[84] W. H. Kruskal and W. A. Wallis, "Use of Ranks in One-Criterion Variance Analysis," *Journal of the American Statistical Association*, vol. 47, no. 260, pp. 583–621, 1952.

[85] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other", *The annals of mathematical statistics*, pp. 50–60, 1947.

[86] S.-L. Peng, P. Su, F.-N. Wang, Y. Cao, R. Zhang, H. Lu, and P. Liu, "Optimization of phase-contrast MRI for the quantification of whole-brain cerebral blood flow", *Journal of Magnetic Resonance Imaging*, vol. 42, no. 4, pp. 1126–1133, 2015.

[87] P. Villacorta-Aylagas, I. Fernández-Arias, C. Castillo-Passi, P. Irarrázaval, F. Simmross-Wattenberg, M. Rodríguez-Cayetano, and C. Alberola-López, "A Web Version of KomaMRI. Sequence Editor and Remote Execution", in *ISMRM Iberian Chapter Annual Meeting 2023*, Valladolid, Spain: ISMRM, Jul. 2023.

[88] J. Anatol-Hernández, J. B. Sierra-Pallares, P. Villacorta-Aylagas, C. Catillo-Passi, R. M. Menchón-Lara, P. Irarrázaval, and C. Alberola-López, "Synthetic spin trajectory generation for advanced flow phantom in MRI simulation", in *ISMRM Iberian Chapter Annual Meeting 2024*, Porto, Portugal: ISMRM, Jul. 2024.

[89] E. Weisstein from Wolfram Research. "Rodrigues' Rotation Formula". [Online], Available in: https://mathworld.wolfram.com/RodriguesRotationFormula.html (Last accessed: Dec. 2, 2024).

[90] P. Le Roux, "Introduction to the Shinnar-Le Roux algorithm", Tech. Rep., 1995.

[91] N. Wilson, A. Ortiz, and A. Johnson, "The Vascular Model Repository: A Public Resource of Medical Imaging Data and Blood Flow Simulation Results", *Journal of medical devices*, vol. 7, pp. 0 409 231–409 231, Dec. 2013.

[92] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby, "A tensorial approach to computational continuum mechanics using object-oriented techniques", *Computer in Physics*, vol. 12, no. 6, pp. 620–631, Nov. 1998.

[93] K. Moreland, R. Maynard, D. Pugmire, A. Yenpure, A. Vacanti, M. Larsen, and H. Childs, "Minimizing development costs for efficient many-core visualization using MCD3", *Parallel Computing*, vol. 108, p. 102 834, 2021.

[94] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring), Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485.

[95] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era", *Computer*, vol. 41, no. 7, pp. 33–38, 2008.