**INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA**
**Physics Department**


# Implementation of Compressed Sensing Methods on a Tabletop MRI System

Francisco Gomes da Silva
50888

Bachelor's Final Project Report
Bachelor's Degree in Applied Physics Engineering

Advisorship:
Dra.Rita Nunes, ISR-IST
Dr.Pedro Patrício, ISEL

July 2025

# Table of Contents

# 1
# Introduction

Magnetic Resonance Imaging (MRI) is a powerful non-invasive technique widely used, primarily in the medical field, to obtain high quality images inside of the human body. Its high soft tissue contrast, absence of ionizing radiation, versatility, and other benefits to other medical imaging techniques have made it indispensable in medical diagnosis. Despite its already widespread use, scientific interest in this technique continues to grow, driven by the constant need to improve image quality, reduce acquisition times, and extract more quantitative information from MRI data, resulting in new advanced acquisition and reconstruction techniques.

This final report describes the work carried out in this field of study during the internship at Evolutionary Systems and Biomedical Engineering Lab, a research laboratory of the Instituto de Sistemas e Robótica at the Instituto Superior Técnico. The curricular internship, as part of the Bachelor's Degree in Applied Physics Engineering at the Instituto Superior de Engenharia de Lisboa, took place between February 17 and July 12, 2025.

The main objective was to implement and test compressed sensing methodology on an open-source tabletop MRI scanner, the Ilumr, developed by Resonance Innovations.

This report is organized into five main chapters. Chapter 1 provides an introduction and contextualization of the project. Chapter 2 outlines the theoretical background necessary to understand the principles of MRI, as well as the advanced acquisition method explored in this project. It also provides an overview of the working principles and technical specifications of the Ilumr tabletop MRI scanner. Chapter 3 describes the experimental procedures, including sequence analysis, the development and implementation of the necessary code and the system setup. Chapter 4 presents and discusses the results obtained from the implementation process. Finally, Chapter 5 summarizes the conclusions of the work and suggests potential directions for future developments.

# 2
# Theoretical Model

## 2.1 Basic Principles of MRI

MRI is a non-invasive imaging technique used to obtain high quality images. This technique, based on the principles of nuclear magnetic resonance (NMR), exploits the magnetic properties of atomic nuclei, particularly hydrogen nuclei (protons), which are abundant in human body and biological tissues in general due to their high water content.

### 2.1.1 NMR and Larmor Frequency

In the presence of a strong magnetic field ($\vec{B_0}$) the magnetic moments of the hydrogen nuclei align parallel or anti-parallel with $\vec{B_0}$ (see appendix A.1). The spins also precess around the direction of $\vec{B_0}$ at a given frequency named the Larmor Frequency ($w_0$) that is given by:

$$(1) w_0 = \gamma \cdot B_0$$

, where $w_0$ is the Larmor Frequency (rad/s), $\gamma$ the gyromagnetic ratio (42.58 Mhz/T for hydrogen) and $B_0$ the magnetic field strength (T).

The Larmor Frequency is the frequency in which a particle with a net spin can absorb a photon. This is the basis of magnetic resonance because only at this frequency nuclear spins resonate and absorb energy from an external RF field [1].

### 2.1.2 Radiofrequency Excitation and Relaxation Process

In order to perturb the system from equilibrium, a RF pulse, at the Larmor Frequency, is applied. This pulse tips the net magnetization vector, that is align with the main magnetic field ($\vec{B_0}$) and with Z axis into the transverse plane, XY-plane (see appendix A.2). The amount of tilt depends on the duration and magnitude of the RF pulse.

When the RF pulse is turned off, a relaxation process begins and the protons return to their equilibrium state, emitting signal that can be detected (see appendix A.3).

There are two relaxations processes happening at the same time:

- **$T1$ Relaxation (Longitudinal Relaxation):**

  $T1$ describes how quickly the protons return to the equilibrium and realign with the main magnetic field ($\vec{B_0}$). This time depends on the type of tissue the excited hydrogen protons are part of. Fat typically has a short $T1$ while fluids have longer relaxation times. This process occurs due to spin-lattice interactions, in other words $T1$ determine how quickly the tissue recovers its ability to generate signal after excitation. The time constant $T1$ describes the exponential recovery of longitudinal magnetization along the $\vec{B_0}$ which, following the application of a 90º RF pulse is given by:

  $$(2) M_z(t) = M_0(1 - e^{-t/T_1})$$

  , where $M_0$ is the maximum longitudinal magnetization.

- **$T2$ Relaxation (Transverse Relaxation):**

  $T2$ describes how quickly the protons lose the phase coherence in the transverse plane, as the result of spin to spin interactions, that they had gained with the RF pulse. It results in the decay of the net magnetization vector and of the signal detected by the receiver coils. Additionally, magnetic field inhomogeneities cause further dephasing, accelerating the decay. The effective decay of the transverse magnetization observed is in practice described by $T2^*$, but with specialized pulse sequences, such as spin-echo, we can refocus spins in order to compensate for $B_0$ field inhomogeneities and only detect the relevant intrinsic $T2$ signal. $T2$ is always shorter than $T1$ and also varies across different tissues. The time constant describing the decay of transverse magnetization due to dephasing of spin vectors in the XY-plane is:

  $$(3) M_{xy}(t) = M_0 \cdot e^{-t/T_2}$$

  ,where $M_0$ is the initial transverse magnetization (immediately after the RF pulse).

The signal detected by the receiver coil arises from the transverse magnetization, so it reflects the transverse magnetization relaxation. $T1$ indirectly influences the signal by controlling the recovery of longitudinal magnetization, which determines the magnetization available for excitation by subsequent pulses. The time-domain signal generated immediately after excitation, known as the Free Induction Decay (FID), is a decaying sinusoidal waveform induced in the receiver coil by the precessing transverse magnetization (see appendices A.4, A.5). Since the intensity of the signal detected depends on the type of tissue that the protons are part of, there will be areas with higher signal than others. This different relaxation times allow us to have contrast in the MRI images [4] [9].

### 2.1.3 Gradients and Spatial Encoding

In order to localize the origin of the signal detected, magnetic field gradients are applied in different directions. These gradients cause the magnetic field and consequently the Larmor Frequency to vary linearly with spatial position, enabling spatial encoding of the initial signal.

During the RF pulse, a gradient is applied along the Z-axis to select a specific slice (slice selection process). In this way only spins in a narrow frequency range, corresponding to the specific location along the gradient, are excited. Once a slice is selected, two spatial encoding mechanisms are used within that slice:

- A frequency encoding gradient (readout gradient) is applied along the X-axis during the signal acquisition. This causes spins at different X positions to precess at different frequencies.
- A phase encoding gradient is applied along the Y-axis before readout, causing position dependent phase differences that persist into the signal.

In three-dimensional imaging, an additional phase-encoding gradient is applied along the Z-axis, allowing volume acquisition. These gradients define the field of view and resolution of the final image, as the strength and duration of the gradients determine the range and granularity of spatial encoding.

### 2.1.4 K-Space and Acquisition Time

The now spatially encoded signals are sampled and stored in a matrix known as k-space (see figure 2.1), which represents the spatial frequency content of the image. Each point in k-space corresponds to a specific combination of phase and frequency encoding. This matrix can be two or three-dimensional depending on the type of final image desired [3].

The center of k-space contains low spatial frequency information, which determines the image contrast and overall structure. The outer regions correspond to higher spatial frequencies, which contribute to more fine details and edges. Filling the k-space requires multiple repetitions of the pulse sequence, each one with different phase-encoding gradient values.

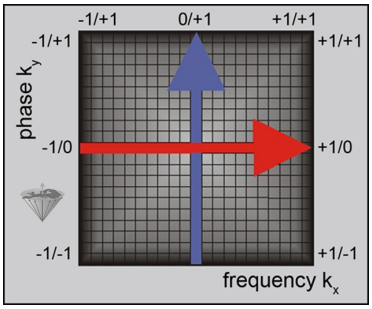


Figure 2.1: The two dimensional k-space where encoded signals are sampled and stored in.

Source: magnetic-resonance.org [3]

During the acquisition, data is collected line by line in k-space. Each $TR$ (Repetition Time) is used to acquire one line, for 2D, or one line within the plane, for 3D, of the k-space (see appendix A.6). Therefore, the majority of total scan time in MRI is used filling the k-space, especially in high-resolution and 3D acquisitions, where the number of phase encoding steps (lines needed) is significantly higher.

The total acquisition time ($TA$) can be calculated as follows:

$$(4) TA_{2D} = TR \cdot N_{PE} \cdot N_{avg}$$

$$(5) TA_{3D} = TR \cdot N_{PEy} \cdot N_{PEz} \cdot N_{avg}$$

, where $TR$ is the repetition time, $N_{PE}$ the number of phase encoding steps, $N_{PEy}/N_{PEz}$ the phase encodes in 3D and $N_{avg}$ the number of averages (repetitions of the acquisition to improve signal-to-noise ratio).

### 2.1.5 Fourier Transform and Final Image

When k-space is fully sampled, the data is transformed from the frequency domain into the spatial domain using the inverse of the Fourier Transform (IFT). Specifically, a 2D or 3D Inverse Fast Fourier Transform (IFFT) is applied, depending on the dimensionality of the acquisition. This mathematical transformation converts the phase and frequency-encoded signal into spatially localized pixel intensities. The result is an image where each pixel corresponds to a specific location in the selected slice or volume, and the intensity reflects the magnitude of the detected signal at that specific location.

This process, from the RF excitation and relaxation, through spatial encoding and signal acquisition, to k-space sampling and image reconstruction using the Inverse Fourier Transform, forms the complete theoretical basis of image formation in MRI.

### 2.1.6 Pulse Sequences in MRI

MRI pulse sequences define when and how RF pulses and gradients are applied. These determine how the signal is generated, encoded, and acquired. Different sequences can emphasize in the final signal, a particular relaxation time, resulting in different image contrasts (T1-weighted, T2-weighted or a mixture of the two). Some sequences can even minimize scan time and reduce sensitivity to artifacts. Each one of these represents a different trade-off between image quality, acquisition speed, contrast weighting, and hardware limitations. The choice of the sequence is a critical part of the MRI protocol, as it directly impacts the final image produced. Examples of commonly used sequences are Spin Echo, Gradient Echo and Rapid Acquisition with Relaxation Enhancement (RARE). The last one will be discuss in more detail as it was the one used in this project.

### RARE Sequence

The RARE sequence, also known as Fast Spin Echo (FSE), is a sequence used to accelerate a T2-weighted image acquisition. It is based in a Spin Echo sequence that reduces acquisition time by collecting multiple lines of the k-space after a single excitation pulse.

After an initial 90° degree excitation pulse, a train of 180° degree refocusing pulses (Echo-train) is applied. Each one of these echoes is phase-encoded differently, allowing the acquisition of several k-space lines during one repetition time. This echo train leads to a much faster scan compared to the conventional Spin Echo sequence, which collects a single line of the k-space per repetition time due to applying a single echo per excitation. The sequence diagram is shown in figure 2.2.

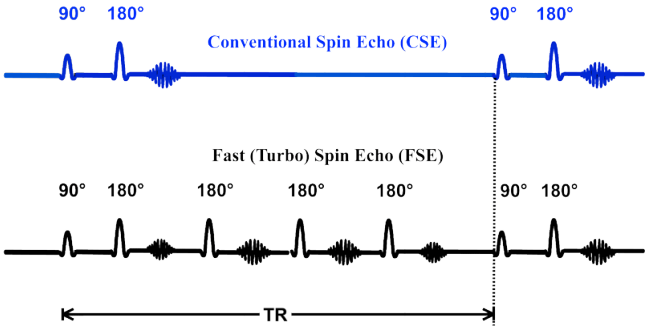


Figure 2.2: RARE/FSE sequence diagram
Source: mriquestions.com

## 2.2 Compressed Sensing

As discussed in the context of MRI, acquiring some types of images often results in slow data acquisition. Since image quality depends on sampling enough of k-space to meet the Nyquist criterion and since each line in k-space requires a separate excitation and readout cycle, acquisition times can become prohibitively long, especially for high-resolution or 3D imaging. Compressed Sensing (CS) offers a powerful framework to overcome this limitation by enabling accurate image reconstruction from a incomplete sampled k-space data, reducing the scan time needed to form an image (see figure 2.3 and appendix A.7).

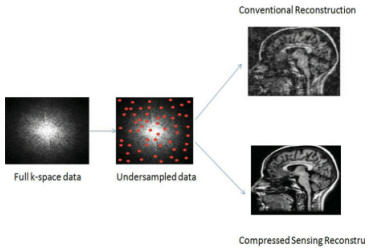


Figure 2.3: From left to right: fully sampled k-space, undersampled k-space, and two reconstructed images — the top image shows the normal Cartesian reconstruction, while the bottom one shows the CS reconstruction from the undersampled k-space.
Source: biomedikal.in [5]

CS relies on three key principles:

- **Sparcity**

  A signal is sparse if it can be represented with few non zero coeficients. This compression method discardes low-energy coefficients and retaines only the significant ones. MR images are not strictly sparse in the pixel domain, but they can be compressed in domains such as wavelets where most of the signal energy is concentrated in a few coefficients.

- **Incoherence**

  Incoherence describes the relationship between the sampling basis (Fourier in MRI) and the sparsity basis (wavelets). The more incoherent the two bases are, the more noise-like the aliasing effects become, which makes it easier to separate the true signal from the artifacts during the reconstruction process. In MRI, incoherence is achieved using a random undersampling mask of k-space, especially with variable density patterns that sample more densely near the center of k-space and sparsely at the edges.

- **Nonlinear Reconstruction**

  Because in CS less lines are sampled compared to the full acquisition, traditional linear reconstruction (inverse FFT) is not enough. Instead CS uses nonlinear reconstruction algorithms that exploit sparsity. The normal approach is to solve an optimization problem that enforces data consistency and promotes sparsity:

$$(6)\ \min_{m} \|\Psi m\|_1 \quad \text{subject to} \quad \|F_S m - y\|_2 < \varepsilon$$

  Where $m$ is the image to be reconstructed, $\Psi$ the sparsifying transform (wavelets), $F_S$ the undersampled Fourier operator, $y$ is the measured k-space data from the MRI scanner and $\varepsilon$ (the threshold parameter is roughly the expected noise level) controls the fidelity of the reconstruction to the measured data.

  Putting the two expressions in words, among all solutions that are consistent with the acquired data, we want to find a solution that is compressible by the transform $\Psi$. This problem can be solved using iterative algorithms such as the Iterative Soft Thresholding [2] [5].

In practice, the reconstruction will involve:

1. Applying the sparsifying transform (wavelet);
2. Thresholding small coefficients;
3. Enforcing consistency with acquired k-space data;
4. Iterating until convergence.

## 2.3 Ilumr Tabletop MRI System Overview

The Ilumr [1] is a compact tabletop MRI scanner that was developed by Resonance Innovations. This scanner is ideal for educational, research, and development purposes, as it provides full access to the MRI acquisition chain while having a simplified and accessible hardware platform. The system operates with a static magnetic field of 0.33 Tesla, permanent magnets to be able to have a homogeneous field, three orthogonal gradient coils for spatial encoding, and a transmitter/receiver radiofrequency coil [7] (see appendix A.8).

The system has a base set of some MRI sequences, all written in Python and fully modifiable. The system also supports the addition of new sequences or data to its base set. This open-source approach makes the Ilumr ideal to develop and test new sequences, adapt existing ones to support advanced methods or just to learn and teach the MRI basics.

All interactions with the scanner are web-based and performed through JupyterLab using the matipo framework, which provides tools for sequence configuration, parameter management, acquisition triggering, and data visualization.

[1] Official Resonint website with information about the Ilumr: https://www.resonint.com/ilumr

# 3
# Experimental Work

## 3.1 Overview of the Workflow

The main objective of the experimental work was to implement, validate and analyze the technique, ultimately in a three-dimensional acquisition context, using the Ilumr tabletop MRI system. This involved modifying the existing acquisition sequence selected, controlling the scan workflow, acquiring undersampled data, reconstructing images using CS algorithms and finally analyze and interpret the results.

To achieve this, a complete experimental pipeline was developed and tested. The work was divided into four main sections:

1. Modification of the existing RARE pulse sequence to support 3D k-space undersampling;
2. Modification of the existing Jupyter notebook to control the new 3D CS RARE sequence;
3. Modification of the existing Jupyter notebook, used to generate the 2D undersampling mask and image reconstruction, to support the 3D CS;
4. Data acquisition and image reconstruction.

## 3.2 Modification of the RARE Sequence

### 3.2.1 Original RARE Sequence

In the original RARE Sequence [6], the code is designed to iterate over all indices. The code receives the echo train length (`p.n_ETL`), the number of phase encoding steps in the two directions (`p.n_phase_1, p.n_phase_2`) and with the function `etl_sequence_format` returns the number of runs (`n_runs`) and the number of echos (`n_echos`) to make the full acquisition loop needed to obtain the k-space data (see lines 88-101 in appendix B.1).

After this, the code calculates the value of the gradient steps (`g_phase_1_step`, `g_phase_2_step`) in order to use them in the following loop:

Listing 3.1: Gradient steps calculation in RARE.py.

```
    if p.n_phase_1>1:
        g_phase_1_step = -p.g_phase_1/(p.n_phase_1//2)
    else:
        g_phase_1_step = 0
    log.debug(f'phase 1 step: {str(g_phase_1_step)}')
    if p.n_phase_2>1:
         g_phase_2_step = -p.g_phase_2/(p.n_phase_2//2)
    else:
        g_phase_2_step = 0
    log.debug(f'phase 2 step: {str(g_phase_2_step)}')

    n_runs, n_echos = etl.sequence_format(p.n_ETL, p.n_phase_1,
       p.n_phase_2)

    log.debug(f"n_runs: {n_runs}, n_echos: {n_echos}")
```

To make these calculations, the code divides the total gradient range (`g_phase_1`, `g_phase_2`) for half the number of phase-encoding steps and applies a negative sign to ensure the correct direction. This means the phase-encoding gradient will vary symmetrically from -g to +g across the k-space, covering both negative and positive sides of k-space, ensuring uniform coverage and proper image reconstruction.

After this, and before arriving at the main loop, the code calculates critical times such as the echo time, pre-reading gradient times, and time delays for acquisitions and inversions. Then the gradient duty cycle is calculated to ensure that the time the gradients are on does not exceed 30% of the total repetition time. This is made to ensure that the coils do not overheat during acquisition. Finally, before the main loop, pregenerated fixed gradient ramps, for execution optimization, are calculated (see lines 167-182 in appendix B.1).

The main loop runs for each scan (`n_scan`, complete repetition of the acquisition to improve the achieved signal-to-noise ratio) and inside each scan for each run (`n_runs`), previously calculated, these parts:

1. Inversion pulse (optional);
2. 90º degree radiofrequency pulse;
3. Loop for each echo;
4. Spoiler pulse and final waiting.

Below are the main loop parts relevant for this report, specially the ones inside the echo loop that need to be changed in order to do an undersampled acquisition. These ones are involved directly in the way gradient steps are applied to the total gradient being used.

Listing 3.2: Relevant sections of the main acquisition loop in RARE.py.

```
    for i_scan in range(p.n_scans):
        p_rf_acc = p_rf_acc % 360 # prevent growing too large and
            causing floating point rounding errors
        p_90 = p_acq = phase_cycle_90[i_scan % n_phase_cycle]
        p_180 = phase_cycle_180[i_scan % n_phase_cycle]
        for i_run in range(n_runs):
            (...) #inversion pulse and 90 degree rf pulse
            for i_echo in range(n_echos):
                # calculate interlaced phase index
                i_phase = i_echo * n_runs + i_run
                if i_phase < p.n_phase_1*p.n_phase_2:
                    # calculate gradient
                    i_phase_1 = i_phase % p.n_phase_1
                    i_phase_2 = i_phase // p.n_phase_1
                    g_phase = p.g_phase_1 +
                        i_phase_1*g_phase_1_step + p.g_phase_2 +
                        i_phase_2*g_phase_2_step
                else:
                    # special case when p.n_phase_1*p.n_phase_2
                        does not divide evenly into p.n_ETL
                    # and there are extra echos
                    g_phase = g_ZERO
            (...) #refocusing pulse
            (...) #data aquistion
        (...) #spoiler pulse
```

This loop iterates over each echo inside the echo train. For each echo, it calculates the variable `i_phase` that determines the current position in the phase-encoding grid, ensuring that all lines in k-space are covered efficiently. Then it calculates the corresponding phase-encoding indices (`i_phase_1, i_phase_2` ) using an interleaved indexing scheme. This allows the sequence to transform a linear sequential index into 2D coordinates. The total gradient (`g_phase`) to apply for each run and for each echo is then calculated by adding the multiplication of the gradient step in the first direction with its corresponding phase-encoding index to the multiplication of the gradient step in the second direction with its corresponding phase-encoding index.

### 3.2.2 RARE Modified Version for 3D CS Compatibility

To enable the 3D undersampling capability of the RARE sequence, several modifications were introduced in the sequence code, based on the work done by Prof. Rita Nunes about the two-dimensional undersampling capability of the same sequence.

Firstly, new variables were defined:

- `set_phase_12`: When set to True, the original program uses the new parts added, enabling the 3D undersampling capability of the sequence;
- `g_phase_12_list`: List of tuples, where each tuple contains a pair of indices (`i_phase_1`, `i_phase_2`). List that defines the only phase-encoding lines, from all the the 3D k-space lines, that will be acquired. It represents the undersampling mask indices and enables arbitrary and sparse sampling patterns, which are essential for CS;
- `n_phase_1_target, n_phase_2_target`: These variables define the original matrix size of the k-space. The use of target variables is important to maintain the spatial encoding and reconstruction grid, even when only a sub-set of all k-space data is being acquired.

Then, the effective changes on the original code were made. The calculation of phase encoding gradient steps was modified to use the new target values `n_phase_1_target`, `n_phase_2_target` (see lines 123-132 in appendix B.2) The main acquisition loop was adapted. When `set_phase_12_samp` is False, the code uses the standard sequential calculation for phase-encoding indices that was in the original sequence code. When `set_phase_12_samp` is True, the code iterates over the custom list `g_phase_12_list`, directly selecting which (`i_phase_1, i_phase_2`) pairs need to be acquired. This allows for arbitrary and sparse k-space coverage, required for compressed sensing.

Listing 3.3: Relevant sections of the main acquisition loop in RARE-3D-CS-ONLY.py.

```
1    for i_echo in range(n_echos):
2                # calculate interlaced phase index
3                i_phase = i_echo * n_runs + i_run
4                # print(p.g_phase_2_lines)
5                if i_phase < p.n_phase_1*n_phase_2:
6                    # calculate gradient
7                    if not p.set_phase_12_samp:
8                        i_phase_1 = i_phase % p.n_phase_1
9                        i_phase_2 = i_phase // p.n_phase_1
10                       g_phase = p.g_phase_1 +
                             i_phase_1*g_phase_1_step + p.g_phase_2
                             + i_phase_2*g_phase_2_step
11                   else:
12                       for y in range (len(p.g_phase_12_list)):
13                           i_phase_1, i_phase_2 =
                                 p.g_phase_12_lines[y]
14                           g_phase = p.g_phase_1 +
                                 i_phase_1*g_phase_1_step +
                                 p.g_phase_2 +
                                 i_phase_2*g_phase_2_step
```

Additionally, the number of runs (`n_runs`) and the echo train length (`n_echos`) were set to 1 when CS is active in order to simplify the acquisition and reduce computational complexity, similar to the 2D CS implementation from Prof. Rita Nunes.

## 3.3 Jupyter Notebook for Sequence Control

As with all sequences in Ilumr, it was preferable to create a Jupyter notebook in order to configure, run, and process data from the newly modified 3D CS RARE sequence. The notebook `3D_RARE_CS.ipynb` serves as an interface that enables full control over the scan parameters and acquisition behavior. To do so, I based it on the existing Jupyter notebook interfaces used to run the original RARE sequence and the 2D CS RARE sequence. The primary goal of the notebook was to simplify the configuration and execution of the undersampled 3D scans by allowing the user to define the sampling mask (list of phase encoding indices) and the key sequence parameters.

The notebook is structured into modular blocks for clarity and reproducibility. It begins by importing necessary packages (see 1º Block in appendix B.3). The notebook loads the custom sequence (`RARE_3D_CS_ONLY.py`) and sets the required global parameters, such as shimming, RF pulses, and gradient definitions. The sampling mask is loaded to a variable (`pe_lines`) from a numpy file (`g_phase_12_list.npy`) that contains the pre-generated 3D compressed sensing phase encoding indices that were randomly selected and saved into the file in the Jupyter notebook program explained within the next section 3.4 (see lines 1-27, 2º Block in appendix B.3).

Then, some scan parameters are set to run in the sequence. These are the readout gradient, phase encoding gradients (in both directions), slice selection gradient, sample rate, number of samples (that is oversampled by a factor of 4), and some time durations. After, if the 3D CS mode is enabled (`using_3D_CS = True`) the notebook sets in the sequence additional parameters:

- The variable `set_phase_12_samp = True` in order to enable the 3D CS mode in the sequence file;
- The dimensions of the original k-sapce `n_phase_1_target` and `n_phase_2_target`. By default we are undersampling 64x64 data);
- The undersampling dimensions, `n_phase_1` and `n_phase_2`. This must match the percentage of undersampling selected in the program from the next chapter 3.4;
- The custom list of phase indices `g_phase_12_list=pe_lines`.

With all the setup done, the execution of the sequence (`RARE_3D_CS_ONLY.py`) is launched, and the execution time is printed after the scan (see lines 29-90, 2º Block in appendix B.3).

After the acquisition, the raw k-space data (`seq.data`) is deinterlaced to reorder it and displayed to visualize the raw signal. Then, the reordered k-space data (`kdata`) is transformed into the image domain using the inverse Fourier Transform, and three undersampling images and the corresponding k-space, from a representative slice of the three main anatomical planes (axial, sagittal, and coronal), are shown (see 2º Block, lines 92-101, 3º and 4º Block in appendix B.3).

Finally, for 3D CS, the file with the list of indices for the undersampled acquisition (`g_phase_12_list.npy`) is used to reshape the data into a consistent k-space volume with unsampled points filled with zeros. The resulting k-space array (`kdata_sq`) is saved as a numpy file (`kdata_3D_CS.npy`) for the further CS reconstruction mentioned in the next section of this chapter 3.4 (see 5º Block in appendix B.3).

## 3.4 Jupyter Notebook for 3D Compressed Sensing Methodology

To enable image reconstruction from undersampled 3D k-space data, a dedicated Jupyter notebook was developed. This notebook is based on a 2D Compressed Sensing implementation originally provided by Prof. Rita Nunes, and it was adapted and extended to support 3D reconstruction using a wavelet-based sparse representation. The process has four main stages: generation of the random undersampling mask, loading and preparing the acquired data, CS reconstruction, and visualization of the reconstructed images.

The notebook begins defining the acceleration factor variable (`acc_factor`), which controls the sampling percentage in k-space. For example, if we define `acc_factor = 8` it means that the generated mask will represent $1/8$ of the original k-space. The variable corresponding to the size of the image (`im_size`) is also defined (by default, it's set to 64 corresponding to 64x64 k-space). Then, using a pre-made function and the previous variables, a probability density function (`pdf`) and its corresponding random undersampling mask (`random_mask`) are generated. The non-zero positions of the mask are converted to a list of index tuples to use during acquisition (`g_phase_12_list`) and finally, the list of index tuples, the mask, and the PDF are saved into numpy files. The last two are plotted to confirm the sampling pattern (see 2º and 3º Blocks in appendix B.4).

In the second part, the numpy file that contains the undersampled k-space data acquired with the Ilumr scanner (`kdata_3D_CS.npy`) is loaded, along with the numpy files with the sampling mask (`random_mask_3D_CS.npy`) and the associated probability density function (`pdf_3D_CS.npy`). These are expanded into 3D volumes with shape (64, 64, 256) to match the acquired k-space data dimensions (see 4º Block and 5º block, lines 1-3, in appendix B.4).

After loading the files, reconstruction pipeline starts. The reconstruction begins with preprocessing steps that prepare the acquired k-space data for the iterative CS. This step corrects the variable-density nature of the undersampling pattern, ensuring that the resulting dataset is unbiased, and a first soft-thresholding step is made before the iterative loop. Then, the pipeline follows the standard Iterative Soft Thresholding approach in the Wavelet domain. Therefore, the reconstruction process alternates between enforcing sparsity in the wavelet domain and data consistency with the acquired k-space samples.

First, a fixed regularization variable $\lambda$ (`lamb`) is defined to control the soft-thresholding strength later in the sparse domain. A Daubechies 4 wavelet transform operator (`W`) is also defined. The sampling mask is divided by the PDF in order to obtain the unbiased k-space data (`Mu`). An inverse 3D Fourier Transform is then applied to convert the unbiased k-space data (`Mu`) to the image domain (`imu`), data is normalized and a final 3D Fourier Transform is applied to convert the now normalized image (`imu`) back to k-space (`Mu`). The initial image estimate is projected into the sparse domain by multiplying the wavelet transform operator (`W`) with the image (`imu`). Then the soft-thresholding is applied in the wavelet domain and finally the data is converted back from the wavelet domain to the image domain (see 5º Block in appendix B.4).

After the first part, the iterative soft-threshold loop begins in order to do CS reconstruction to the aliased image data `imu` (`Mu` in k-space). The number of iterations for the loop (`niter`) and the stopping criteria (`tol`) are defined. If the error falls below a set tolerance (`tol`) the loop stops early. At each iteration, the image data (`imu` = `im_est`) is projected into the wavelet domain. Then soft-thresholding is applied to promote sparsity, and the result (`Wim_thres`) is transformed back to the image domain (listing 3.4).

Listing 3.4: Sections of the reconstruction loop in 3D-CS-Methodology.py.

```
1 W = sp.linop.Wavelet(imu.shape, axes=None, wave_name='db4',
      level=3)
2   (...)
3         imprev = im_est
4     Wim = W*imprev
5     im_thres = W.H*Wim_thres
6     M_est = functions.fftc_3D(im_thres)
```

A Fourier Transform is then applied to convert the image (`im_thres`) to k-space, and data consistency between the k-space and the acquired k-space is enforced. Instead of full substitution, a blending factor $\alpha$ is used to softly guide the estimate toward the acquired data. Finally, the updated image estimate (`im_est`) is obtained by converting the consistent k-space data (`M_est`) to image data, and the change from the previous iteration is computed to monitor convergence. After the loop, a normalization is applied to the new reconstructed image data (`im_est`) in order to improve image contrast (see 7º Block in appendix B.4).

In the final part of the notebook, visualization routines are included to inspect the final reconstruction quality. As in the previous program (see section 3.3), due to the size of the 3D dataset and limited computational resources, the full volume rendering was not performed. Instead, the code extracts and displays 2D representative slices from the three main anatomical planes: axial, sagittal, and coronal (the specific slices within the anatomic planes can be selected using the variables `x_show`, `y_show` and `z_show`). For each plane, it shows the initial aliased image, the final reconstructed image, and the corresponding error map. Additionally, a convergence plot of the reconstruction error over iterations is generated to monitor the algorithm's stability and the stopping behavior (see 8º, 9º, 10º and 11º Blocks in appendix B.4).

## 3.5 Acquisition and Reconstruction

The full acquisition and reconstruction workflow involved several sequential steps, connecting the developed programs into a complete pipeline for each reconstruction.

First, with the reconstruction notebook `3D_CS_Methodology.ipynb`, the undersampling mask is generated with the desired acceleration factor and image size (typically 64×64). The resulting file, containing the sampling indices (`g_phase_12_list.npy`), is manually placed in the directory used by the notebook `3D_RARE_CS.ipynb`, responsible for the acquisition sequence control. Next, the acquisition parameters are set in the notebook, and the sample we want to acquire is positioned properly inside the Ilumr scanner. The sequence `3D_RARE_CS_ONLY.py` is then executed via the notebook, and the acquired undersampled k-space data is automatically saved in the numpy format.

The output file containing the acquired undersampled data (`3D_CS_Methodology.ipynb`) is manually transferred to the directory of the reconstruction notebook (`kdata_3D_CS.npy`). After setting the appropriate reconstruction parameters, including the regularization weight, number of iterations, and others, the notebook is executed to perform the compressed sensing reconstruction. The final output, given by the reconstruction notebook, is the result of the experiment conducted: a set of reconstructed images visualized across axial, sagittal, and coronal planes, along with error maps and convergence plots used to evaluate reconstruction quality.

This is the practical execution used in order to acquire each experiment result that will be presented in the chapter 4.

# 4
# Results

In order to test the performance of the developed programs, to verify the CS theoretical model, and to accomplish the initial objective of reconstructing an undersampled k-space data set, two different samples were used: a simple phantom containing water with copper sulfate, used as a contrast agent, and a more complex biological sample, a piece of broccoli. Each sample was acquired using a fully sampled dataset as a reference (64x64x256), followed by one or more undersampled acquisitions, reconstructed using the CS reconstruction methodology implemented.

## 4.1 Sample 1 - Water with Copper Sulfate

This first sample provides a homogeneous contrast and the simplest anatomical structure possible, ideal for validating the acquisition and the reconstruction workflow.

**Fully Sampled Acquisition**

The image was first acquired with full k-space sampling using a resolution of 64 × 64 × 256, resulting in a scan time of 153.889 seconds. This dataset served as the ground truth to evaluate the performance of the compressed sensing reconstruction. The images acquired are shown in figure 4.1.

Figure 4.1: From left to right: coronal, axial and sagittal fully sampled images from the sample 1 acquisition.

**Undersampled Acquisition (1/8 = 12.5%)**

A second acquisition was performed using an acceleration factor of 8, 1/8 (12.5%) of the original k-space, corresponding to a 8 × 64 × 256 resolution (see figure 4.2). To match the desired undersampling, `n_phase_1` was changed to 8 and `n_phase_2`, `n_phase_1_target` and `n_phase_2_target` remain 64. The scan time for this acquisition was significantly reduced to 34.492 seconds.

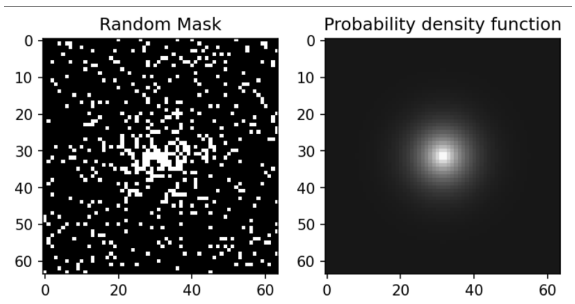


Figure 4.2: Random undersampling mask and probability density function used.

The resulting data was reconstructed using the compressed sensing pipeline with the following parameters: $\lambda = 0.01$, $\alpha = 1$, $N_{iterations} = 10$ and $Tol = 10^{-12}$. The results are shown in figures 4.3, 4.4 and 4.5. The bottom-left image, within the figures, represents the difference between the initial and the reconstructed images.

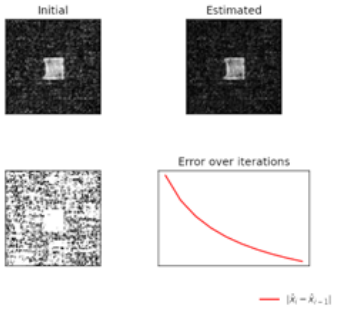


Figure 4.3: Reconstruction routine analysis plot (coronal plane).

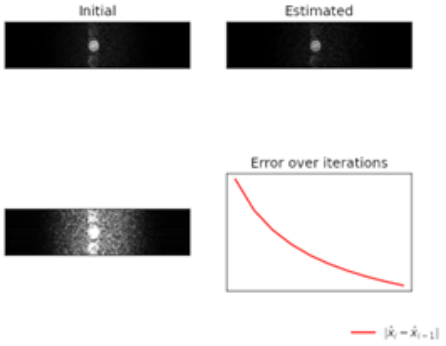


Figure 4.4: Reconstruction routine analysis plot (axial plane).

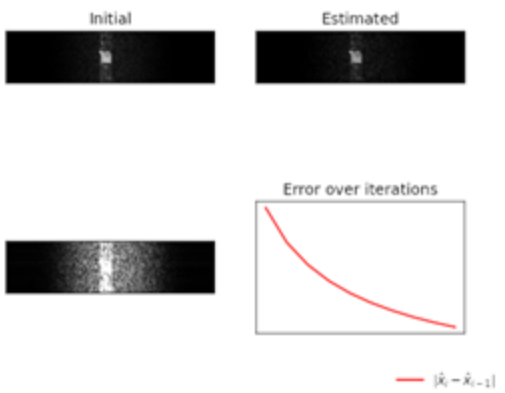


Figure 4.5: Reconstruction routine analysis plot (sagittal plane).

The images from the fully sampled image dataset and from the undersampled and reconstructed image dataset were then compared (see figures 4.6, 4.7 and 4.8).

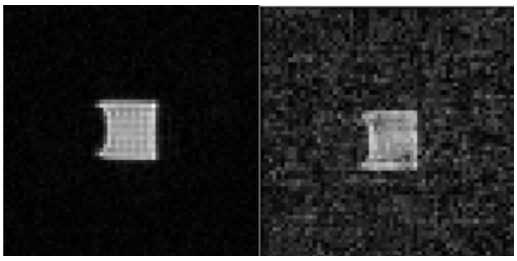

Figure 4.6: The original (left) and reconstructed (right) images from the coronal plane.

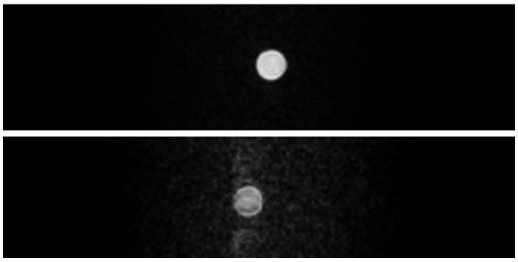

Figure 4.7: The original (top) and reconstructed images (bottom) from the axial plane.

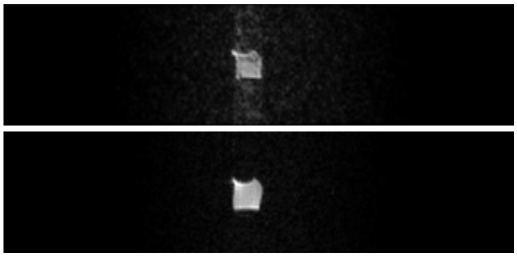

Figure 4.8: The original (top) and reconstructed (bottom) images from the sagittal plane.

Given the simplicity of the sample, the reconstruction preserved, as expected, the overall structural fidelity and contrast well, with few visible artifacts even with a high acceleration factor, few iterations in the reconstruction loop, and a low soft-thresholding parameter ($\lambda$).

## 4.2 Sample 2 - Broccoli

The second sample used in this study was a piece of broccoli. It was selected for its heterogeneous texture and complex structure, ideal for evaluating the limits of the CS reconstruction methodology used, but also for its practical compatibility with the Ilumr scanner, as it fits inside the glass tubes designed for the system.

### Fully Sampled Acquisition

As with the previous sample, a fully sampled acquisition with a resolution of 64 × 64 × 256 was first acquired, resulting in a total scan time of 153.726 seconds. The fully sampled acquisition provided a reference for evaluating the performance of the CS reconstructions performed. The images acquired are shown in figure 4.9.

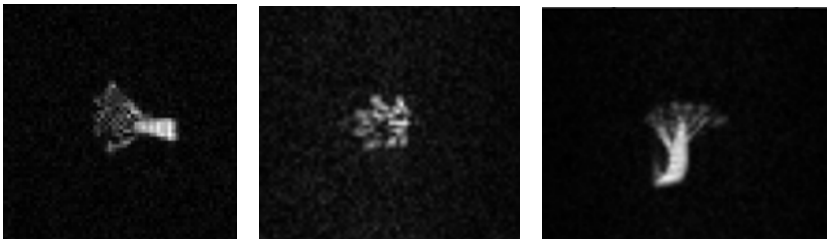

Figure 4.9: From left to right: coronal, axial and sagittal fully sampled images from the sample 2 acquisition.

### Undersampled Acquisition (1/2 = 50%)

A second acquisition was carried out using an acceleration factor of 2, 1/2 (50%) of the original k-space, corresponding to a 32 × 64 × 256 resolution (see figure 4.10). To match the desired undersampling, `n_phase_1` was changed to 32 and `n_phase_2`, `n_phase_1_target` and `n_phase_2_target` remain 64. The acquisition time for this dataset acquisition was reduced to 130.818 seconds.

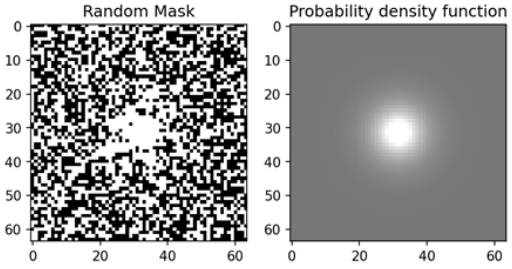


Figure 4.10: Random undersampling mask and probability density function used.

The resulting data was reconstructed using the compressed sensing pipeline with the following parameters: $\lambda = 0.2$, $\alpha = 0.2$, $N_{iterations} = 30$ and $Tol = 10^{-12}$. The results are shown in figures 4.11, 4.12 and 4.13.

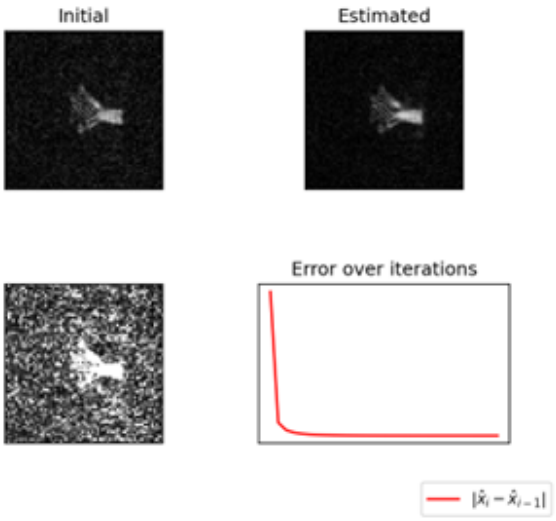


Figure 4.11: Reconstruction routine analysis plot (coronal plane).

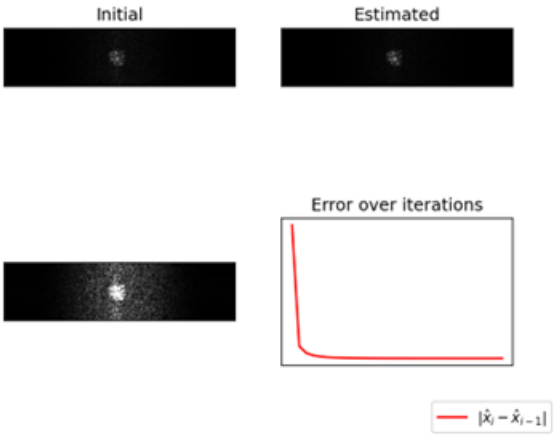


Figure 4.12: Reconstruction routine analysis plot (axial plane).

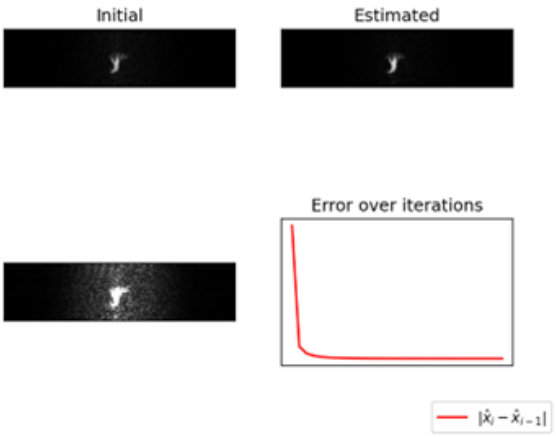


Figure 4.13: Reconstruction routine analysis plot (sagittal plane).

The images from the fully sampled image dataset and from the undersampled and reconstructed image dataset were then compared (see figures 4.14, 4.15 and 4.16).

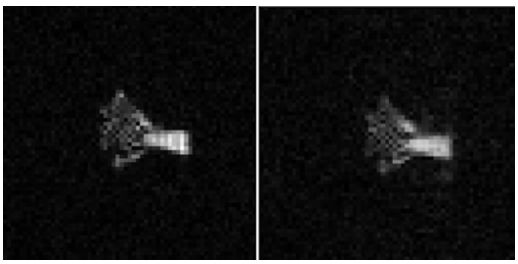

Figure 4.14: The original (left) and reconstructed (right) images from the coronal plane.

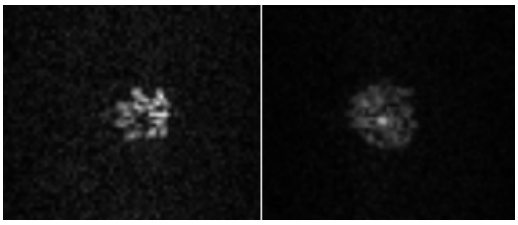

Figure 4.15: The original (left) and reconstructed (right) images from the axial plan).

Figure 4.16: The original (left) and reconstructed (right) images from the sagittal plane.

The reconstructed images demonstrated a very good approximation to the fully sampled references. The structural details, such as edges, textures, and internal patterns, were successfully recovered. Minor smoothing was observed in more fine-detail regions, but overall no major aliasing artifacts appeared. This confirmed that the moderated undersampling and the CS methodology used were viable for this more complex sample. The chosen $\alpha$ and $\lambda$ values allowed a balanced trade-off between enforcing sparsity and retaining fidelity to the original data.

**Undersampled Acquisition (1/4 = 25%)**

A third acquisition was conducted increasing the acceleration factor to 4, 1/4 (20%) of the original k-space, corresponding to a 16 × 64 × 256 resolution (see figure 4.17). To match the desired undersampling, `n_phase_1` was changed to 16 and `n_phase_2`, `n_phase_1_target` and `n_phase_2_target` remained 64. This reduced the scan time significantly to 66.068 seconds, but at the cost of a more aggressive reduction in the acquired data.

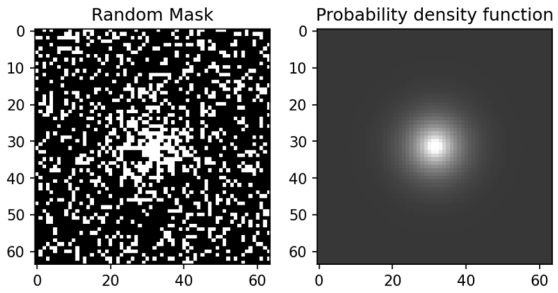


Figure 4.17: Random undersampling mask and probability density function used.

The resulting data was again reconstructed using the compressed sensing pipeline with the following parameters: $\lambda = 0.5$, $\alpha = 1$, $N_{iterations} = 30$ and $Tol = 10^{-12}$. The results are shown in figures 4.18, 4.19 and 4.20.

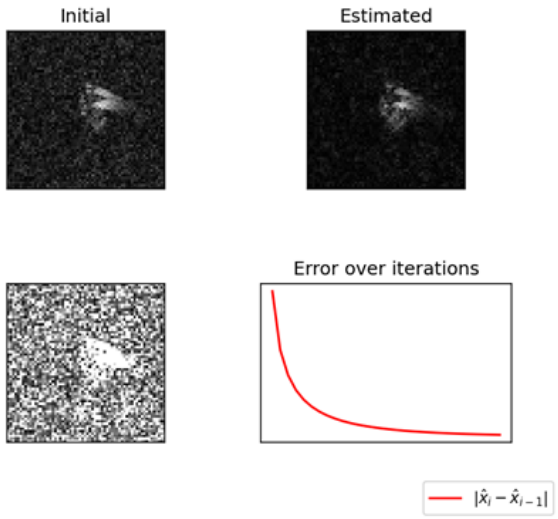


Figure 4.18: Reconstruction routine analysis plot (coronal plane).

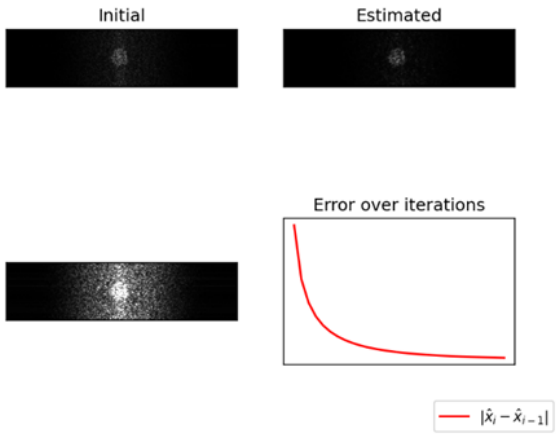


Figure 4.19: Reconstruction routine analysis plot (axial plane).

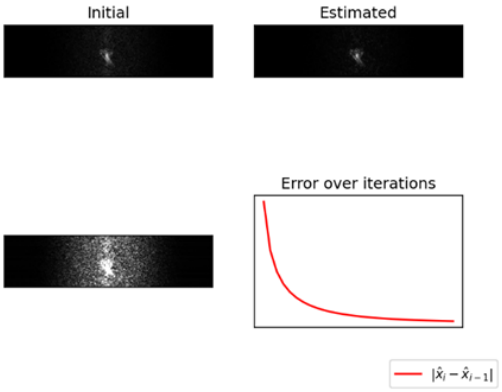


Figure 4.20: Reconstruction routine analysis plot (sagittal plane).

The images from the fully sampled image dataset and from the undersampled and reconstructed image dataset were then compared (see figures 4.21, 4.22 and 4.23).

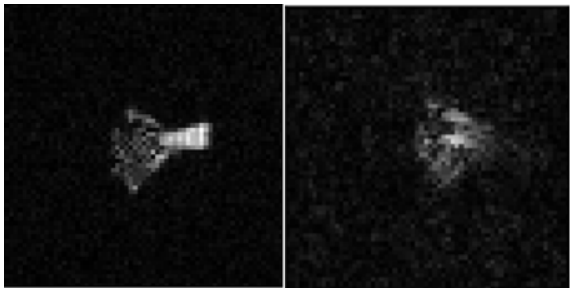

Figure 4.21: The original (left) and reconstructed (right) images from the coronal plane.

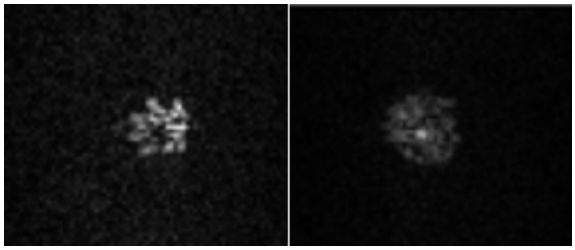

Figure 4.22: The original (left) and reconstructed (right) images from the axial plane.

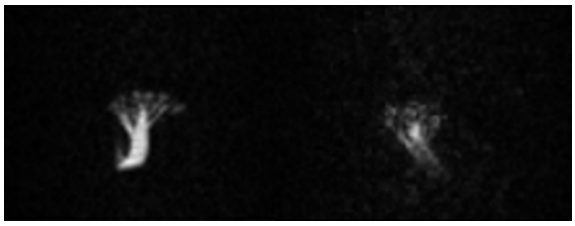

Figure 4.23: The original (left) and reconstructed (right) images from the sagittal plane.

Increasing the acceleration factor made the image reconstruction process noticeably more difficult. While the overall shape and main structures of the broccoli were preserved, some textures and certain structural components suffered damage. The increase of the $\lambda$ value reflected a strategy to compensate for the reduced amount of acquired information. Despite these limitations, the CS algorithm still managed to reconstruct interpretable images.

## 4.3 Final Observations

The experiments confirm that, as expected, the use of the CS methodology leads to substantial reductions in acquisition time. The most significant reduction was observed in the water phantom using an acceleration factor of 8, where the acquisition time went from 153.889 seconds to just 34.492 seconds, corresponding to a 77.6% decrease. In the case of the broccoli sample, the 1/2 undersampling acquisition required 130.818 seconds, representing a 14.9% reduction, while the 1/4 undersampling acquisition took 66.068 seconds, achieving a 57.0% time saving.

The experiments also confirm that the reconstruction quality can vary notably depending on the complexity of the sample being acquired, the undersampling factor chosen, and the reconstruction parameters used. The findings showed that the last two need to be chosen accordingly to the sample being acquired and to the final result needed. For simpler structures, such as the water phantom, a reduced number of iterations (10) and a relatively low regularization parameter (0.01) $\lambda$ can be enough to produce a high quality image reconstruction with few artifacts in the final images, even if the undersampling factor chosen is considerably high (1/8).

However, when dealing with more complex samples, such as the broccoli, a higher regularization parameter $\lambda$ (0.2) and more iterations (30) are needed in order to achieve a good reconstruction. It was also clear that for more complex samples or for acquisitions where fine details are important, the undersampling factor chosen cannot be as high as desired, as it can lead to the loss of some structural details and to the appearance of reconstruction artifacts, such as in the broccoli 1/4 undersampling acquisition.

Overall, from undersampled k-space data acquisitions, it was possible to reconstruct high quality images, reducing the scan time while maintaining image fidelity, confirming both the practical benefits and theoretical expectations of the CS methodology in MRI. In addition, it was possible to understand the influence of the reconstruction parameters on the final image produced. The experimental outcomes closely align with the theoretical model that was presented.

# 5
# Final Remarks

## 5.1 Conclusion

To conclude, the work conducted in order to create, implement and validate a 3D CS methodology for MRI, on the Ilumr tabletop scanner, was a success as it demonstrated the feasibility of accelerating data acquisition without compromising the final image quality. By making modifications in the RARE sequence and by developing a pipeline for the control of the new acquisition sequence and for the data reconstruction, a new framework was integrated and properly tested.

Throughout the internship, it was possible to deepen knowledge in the fields of NMR and MRI and also to gain a solid understanding of both the theoretical and practical aspects of compressed sensing in a real experimental setup. The analysis of the results obtained confirmed the expectations: with simpler samples, or when detail is not important, aggressive undersampling can still lead to useful images. But, with more complex samples, a lower acceleration factor and careful tuning of the reconstruction parameters and the number of iterations are needed in order to achieve a good balance between noise suppression and detail preservation.

In addition to achieving the initial objectives, this project also provided additional contributions to the continuing development of the Ilumr platform by incorporating a new 3D CS sequence and a complete workflow for running, reconstructing, and evaluating it. This work not only serves as a prototype but also as a useful resource for future researchers and students working with CS or other advanced acquisition methods.

## 5.2 Future Perspectives

Although the outcomes of the work carried out were a success, diverse opportunities remain to further extend this work.

The next important step would be the implementation of full 3D volume visualization. Due to computational limitations in this project, only orthogonal slices (axial, coronal and sagittal) were visualized. A 3D volume rendering would provide a more complete view of the reconstruction quality and structural integrity across the entire dataset. Another future approach would be to combine the acquisition and reconstruction phases into a single unified program and to improve the algorithm's robustness by performing more acquisitions with different biological samples.

Finally, implementing and testing non-Cartesian acquisition schemes, such as radial sampling, into the methodology developed would be a huge step forward. The current CS acquisition sequence was constructed on the existing Cartesian structure supplied by the RARE sequence, which was preserved and expanded. Moving towards radial trajectories has significant potential, since integrating these two advanced strategies can eventually lead to the enhancement of the existing CS method.

Together, these opportunities would expand the capabilities and efficiency of the compressed sensing approach implemented in this project, and contribute to the research in this field of study.

# Bibliography


[1] J. P. Hornak, *The Basics of MRI* [Online]. Available: https://www.cis.rit.edu/htbooks/mri/inside.htm. [Accessed: Jun. 20, 2025].

[2] M. Lustig, D. L. Donoho, J. M. Santos, and J. M. Pauly, "Compressed sensing MRI," *IEEE Signal Processing Magazine*, vol. 25, no. 2, pp. 72–82, Mar. 2008.

[3] P. A. Rinck and The Round Table Foundation (TRTF), "Chapter 7 — Page 2: MR Imaging and k-Space," in *Magnetic Resonance: A Peer-Reviewed, Critical Introduction**. [Online]. Available: https://www.magnetic-resonance.org/ch/07-02.html. [Accessed: Jun. 26, 2025].

[4] L. Leroi, *Quantitative MRI: towards fast and reliable $T_1$, $T_2$ and proton density mapping at ultra-high field*, M.Sc. thesis, Nov. 2018. [Online]. Available: https://www.researchgate.net/publication/329697798_Quantitative_MRI_towards_fast_and_reliable_T_T_and_proton_density_mapping_at_ultra-high_field. [Accessed: Jun. 26, 2025].

[5] K. Tripathi, "Compressed sensing for rapid MRI: Need for speed," *Biomedikal.in*, Oct. 11, 2017. [Online]. Available: https://biomedikal.in/2017/10/compressed-sensing-for-rapid-mri-need-for-speed/. [Accessed: Jun. 27, 2025].

[6] Resonint Limited, "ilumr-courseware: Courseware for the ilumr educational MRI system," GitHub repository, 2024. [Online]. Available: https://github.com/Resonint/ilumr-courseware. [Accessed: Jun. 29, 2025].

[7] Resonint Limited, *ilumr User Guide*, [Online]. Available: https://www.resonint.com/ilumr-user-guide. [Accessed: Jul. 2, 2025].

[8] S. Currie, N. Hoggard, I. J. Craven, M. Hadjivassiliou, and I. D. Wilkinson, "Understanding MRI: basic MR physics for physicians," *Postgrad. Med. J.*, vol.89, no.1050, pp.209–223, Apr. 2013, doi:10.1136/postgradmedj-2012-131342.

[9] S. Abdulla, "T1 and T2 signal," *Radiology Cafe*, Oct. 10, 2021. [Online]. Available: https://www.radiologycafe.com/frcr-physics-notes/mr-imaging/t1-and-t2-signal/. [Accessed: Jul. 7, 2025].

# A
# Supplementary Figures

## A.1 Alignment of Protons

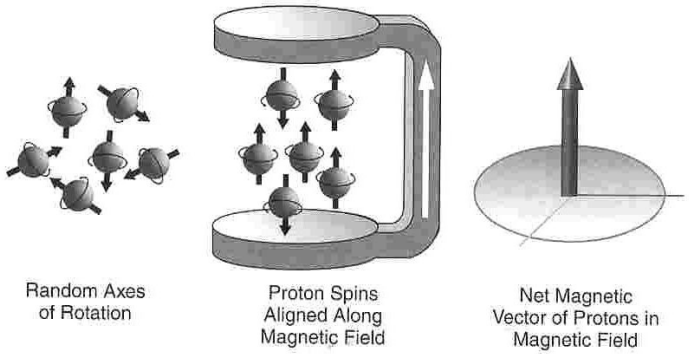


Figure A.1: Alignment of the protons and the net magnetic vector.
Source: sas.upenn.edu

## A.2 Radiofrequency Excitation

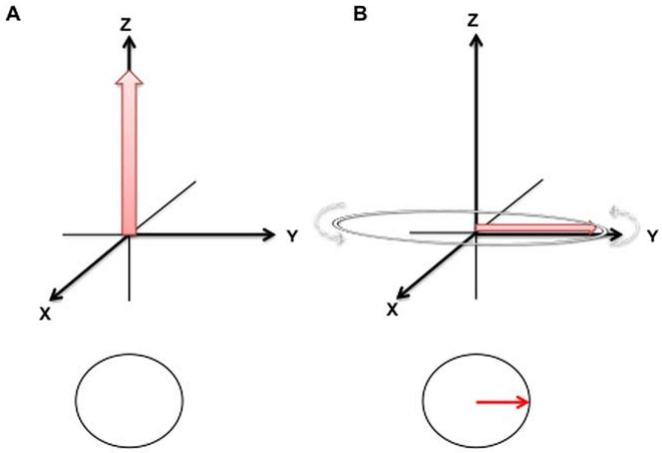


Figure A.2: Excitation after the radiofrequency pulse.
Source: Understanding MRI: basic MR physics for physicians [8]

## A.3 Relaxation Process

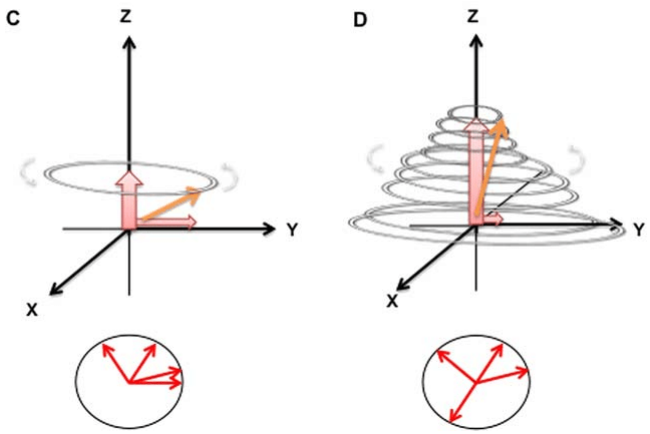


Figure A.3: Recovery after the excitation.
Source: Understanding MRI: basic MR physics for physicians [8]

## A.4 Free Induction Decay

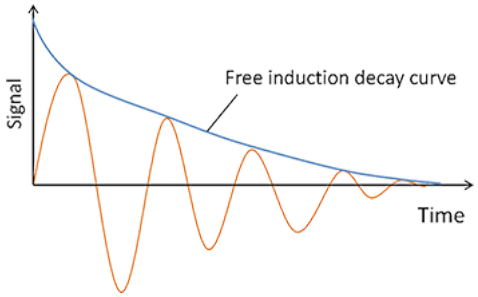


Figure A.4: Free induction decay signal curve.
Source: radiologycafe.com [9]

## A.5 $T2^*$ or free induction decay

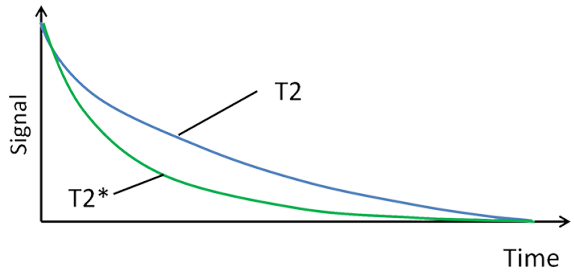


Figure A.5: $T2^*$ decay curve and free induction decay curve.
Source: radiologycafe.com [9]

## A.6 3D K-Space

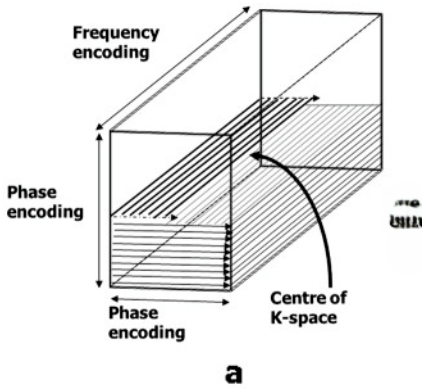


Figure A.6: Three dimensional k-space.
Source: biomedikal.in [5]

## A.7 2D and 3D Undersampled K-Space Mask

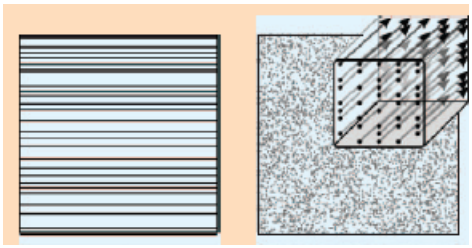

Figure A.7: Two and three dimensional undersampled k-space masks.
Source: IEEE Signal Processing Magazine [2].

## A.8 Ilumr Hardware

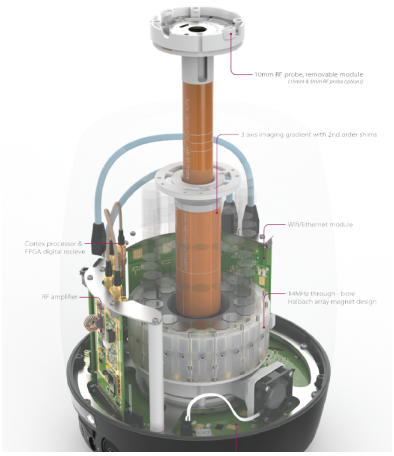

Figure A.8: The hardware of the Ilumr scanner.
Source: Ilumr User Manual [7].

# B
# Existing and Developed Code

## B.1 RARE.py

```
from matipo import sequence as seq
from matipo import ParDef
from matipo import datalayout
from collections import namedtuple
import numpy as np
from functools import partial
from matipo.util.pulseshape import calc_soft_pulse
from matipo.util import etl

# TODO: move to library
def float_array(v):
    a = np.array(v, dtype=float)
    a.setflags(write=False)
    return a

import logging
logging.basicConfig()
log = logging.getLogger(__name__)
log.setLevel(logging.DEBUG)

g_ZERO = float_array((0,0,0))

def soft_pulse(freq, phase, amp, width, bandwidth):
    N, dt, pts = calc_soft_pulse(width, bandwidth)
    pts *= amp
    softpulse = seq.pulse_start(freq, phase, pts[0]) + seq.wait(dt)
    for amp in pts[1:]:
        softpulse += seq.pulse_update(freq, phase, amp) +
           seq.wait(dt)
    softpulse += seq.pulse_end()
    return softpulse

# TODO: move to library
def gen_grad_ramp(g_start, g_end, t, n):
```

```
    g_step = (g_end - g_start)/n
    t_step = t/n
    ret = b''
    for i in range(n):
        ret += seq.gradient(*(g_start+(i+1)*g_step))
        ret += seq.wait(t_step)
    return ret

PARDEF = [
    ParDef('n_scans', int, 2, min=1),
    ParDef('f', float, 1e6),
    ParDef('a_90', float, 0),
    ParDef('t_90', float, 32e-6),
    ParDef('f_slice_offset', float, 0),
    ParDef('bw_90', float, 0),
    ParDef('a_180', float, 0),
    ParDef('t_180', float, 32e-6),
    ParDef('t_inv', float, 1e-3),
    ParDef('t_dw', float, 1e-6),
    ParDef('n_samples', int, 320),
    ParDef('t_read', float, 320e-6),
    ParDef('t_phase', float, 160e-6),
    ParDef('n_phase_1', int, 32),
    ParDef('n_phase_2', int, 32),
    ParDef('n_ETL', int, 0),
    ParDef('t_end', float, 1),
    ParDef('g_slice', float_array, (0, 0, 0), min=(-1, -1, -1),
        max=(1, 1, 1)),
    ParDef('g_read', float_array, (0, 0, 0), min=(-1, -1, -1),
        max=(1, 1, 1)),
    ParDef('g_phase_1', float_array, (0, 0, 0), min=(-1, -1, -1),
        max=(1, 1, 1)),
    ParDef('g_phase_2', float_array, (0, 0, 0), min=(-1, -1, -1),
        max=(1, 1, 1)),
    ParDef('t_spoil', float, 1e-3),
    ParDef('g_spoil', float_array, (0, 0, 0), min=(-1, -1, -1),
        max=(1, 1, 1)),
    ParDef('t_grad_ramp', float, 100e-6),
    ParDef('n_grad_ramp', int, 10),
    ParDef('shim_x', float, 0, min=-1, max=1),
    ParDef('shim_y', float, 0, min=-1, max=1),
    ParDef('shim_z', float, 0, min=-1, max=1),
    ParDef('shim_z2', float, 0, min=-1, max=1),
    ParDef('shim_zx', float, 0, min=-1, max=1),
    ParDef('shim_zy', float, 0, min=-1, max=1),
    ParDef('shim_xy', float, 0, min=-1, max=1),
    ParDef('shim_x2y2', float, 0, min=-1, max=1),
    ParDef('enable_IR', bool, False),
    ParDef('enable_dummy_run', bool, False)
]
```

```
79
80  # TODO: move to library
81  ParameterSet = namedtuple('ParameterSet', [pd.name for pd in
        PARDEF])
82
83  def get_options(p: ParameterSet):
84      return seq.Options(
85          amp_enabled=True,
86          rx_gain=7)
87
88  def get_datalayout(p: ParameterSet):
89      n_runs, n_echos = etl.sequence_format(p.n_ETL, p.n_phase_1,
            p.n_phase_2)
90      # note: if the total number of phase steps does not divide
            evenly into n_ETL
91      # there will be additional data in the last run with no phase
            encoding
92      # which may be ignored or exploited
93      return datalayout.Scans(
94          p.n_scans,
95          datalayout.Repetitions(
96              n_runs,
97              datalayout.Repetitions(
98                  n_echos,
99                  datalayout.Acquisition(
100                     n_samples=p.n_samples,
101                     t_dw=p.t_dw))))
102
103 def main(p: ParameterSet):
104
105     if p.n_phase_1>1:
106         g_phase_1_step = -p.g_phase_1/(p.n_phase_1//2)
107     else:
108         g_phase_1_step = 0
109     log.debug(f'phase 1 step: {str(g_phase_1_step)}')
110     if p.n_phase_2>1:
111          g_phase_2_step = -p.g_phase_2/(p.n_phase_2//2)
112     else:
113         g_phase_2_step = 0
114     log.debug(f'phase 2 step: {str(g_phase_2_step)}')
115
116     n_runs, n_echos = etl.sequence_format(p.n_ETL, p.n_phase_1,
            p.n_phase_2)
117
118     log.debug(f"n_runs: {n_runs}, n_echos: {n_echos}")
119
120     t_echo = p.t_180 + 4*p.t_grad_ramp + 2*p.t_phase + p.t_read
121
122     log.debug(f"echo time: {t_echo}")
123
```

```
    t_phase_read = p.t_read/2 - p.t_grad_ramp/2
    log.debug(f"t_phase_read: {t_phase_read*1e6} us")
    t_phase_read_180 = (t_echo - p.t_90 - p.t_180)/2 -
       t_phase_read - 3*p.t_grad_ramp
    log.debug(f"preread gradient to 180 time:
       {t_phase_read_180*1e6} us")
    if t_phase_read_180 <= 0:
        raise Exception('phase gradient time too short')

    t_acq_delay = (p.t_read - p.n_samples*p.t_dw)/2
    if t_acq_delay < 0:
        raise Exception('t_read too short!')


    # time from end of inversion pulse to start of excitation pulse
    t_inv_90 = p.t_inv - (p.t_90+p.t_180)/2
    if t_inv_90 <= 0:
        raise Exception('t_inv too short!')

    n_phase_cycle = 4
    phase_cycle_90 = [0, 180, 0, 180]
    phase_cycle_180 = [90, 90, 270, 270]

    p_rf_acc = 0 # accumation of RF phase by running at a
       different frequency
    p_90_inc = 0 # phase accumulated by a 90 pulse (to be
       calculated)
    p_180_inc = 0 # phase accumulated by a 180 pulse (to be
       calculated)

    enable_softpulse = p.bw_90 > 0
    if enable_softpulse:
        t_inv_90 -= p.t_grad_ramp # keep t_inv setting the time
           between centres of pulses

        # calculate the the gradient amplitude required to match
           half the slice gradient area during the read prephasing
           gradient time
        g_unslice =
           -p.g_slice*0.5*(p.t_90+p.t_grad_ramp)/(t_phase_read+p.t_grad_ramp)
        log.debug(f'g_unslice: {str(g_unslice)}')

        log.debug('using softpulse')
        # to offset the slice, the RF pulses must have a frequency
           offset,
        # which introduces a phase change in the transmitter
           relative to the receiver,
        # p_90_inc and p_180_inc are the amount of phase
           accumulated during a 90 and 180 pulse
        p_90_inc = 360*p.f_slice_offset*p.t_90
```

```
162         p_180_inc = 360*p.f_slice_offset*p.t_180
163         log.debug(f'p_90_inc: {p_90_inc}, p_180_inc: {p_180_inc}')
164     else:
165         g_unslice = g_ZERO
166 
167     # gradient duty cycle check TODO: implement duty cycle checks
        in driver
168     t_cpmg = p.t_90/2 + t_echo/2 + p.n_phase_2*t_echo - p.t_180/2
        + p.t_spoil
169     if p.enable_IR:
170         t_cpmg += p.t_180/2 + p.t_inv - p.t_90/2
171     t_rep = t_cpmg + p.t_end
172     grad_total_area = (
173         np.abs(p.g_slice)*(p.t_90+p.t_grad_ramp)
174         + np.abs(p.g_read+g_unslice)*(t_phase_read+p.t_grad_ramp)
175         +
            p.n_phase_2*2*(np.abs(p.g_phase_1)+np.abs(p.g_phase_2))*p.t_phase
176         + p.n_phase_2*np.abs(p.g_slice)*p.t_180
177         + np.abs(p.g_spoil)*p.t_spoil
178     )
179     grad_duty_cycle = grad_total_area/t_rep
180     log.debug(f'gradient duty cycle: {str(grad_duty_cycle)}')
181     if np.any(grad_duty_cycle > 0.3):
182         raise Exception('Gradient duty cycle too high!')
183 
184     # pregenerate fixed ramps (optimisation)
185     g_ramp_zero_to_slice = gen_grad_ramp(g_ZERO, p.g_slice,
        p.t_grad_ramp, p.n_grad_ramp)
186     g_ramp_slice_to_zero = gen_grad_ramp(p.g_slice, g_ZERO,
        p.t_grad_ramp, p.n_grad_ramp)
187     g_ramp_zero_to_read_unslice = gen_grad_ramp(g_ZERO,
        p.g_read+g_unslice, p.t_grad_ramp, p.n_grad_ramp)
188     g_ramp_read_unslice_to_zero =
        gen_grad_ramp(p.g_read+g_unslice, g_ZERO, p.t_grad_ramp,
        p.n_grad_ramp)
189     g_ramp_zero_to_read = gen_grad_ramp(g_ZERO, p.g_read,
        p.t_grad_ramp, p.n_grad_ramp)
190     g_ramp_read_to_zero = gen_grad_ramp(p.g_read, g_ZERO,
        p.t_grad_ramp, p.n_grad_ramp)
191 
192     yield seq.shim(p.shim_x, p.shim_y, p.shim_z, p.shim_z2,
        p.shim_zx, p.shim_zy, p.shim_xy, p.shim_x2y2)
193     yield seq.wait(0.01)
194 
195     if p.enable_dummy_run:
196         i_run = 0
197         p_90 = phase_cycle_90[-1]
198         p_180 = phase_cycle_180[-1]
199 
200         if p.enable_IR:
```

```
            # inversion pulse
            yield seq.pulse_start(p.f, p_rf_acc, p.a_180)
            yield seq.wait(p.t_180)
            yield seq.pulse_end()
            yield seq.wait(t_inv_90)

        if enable_softpulse:
            p_rf_acc += p_90_inc/2
            yield (
                g_ramp_zero_to_slice
                + soft_pulse(p.f+p.f_slice_offset,
                    (p_90-p_rf_acc)%360, p.a_90, p.t_90, p.bw_90)
                + seq.wait(100e-9)
                + seq.pulse_update(p.f, 0, 0) # reset transmitter
                    back to reference frame frequency
                + g_ramp_slice_to_zero
            )
            p_rf_acc += p_90_inc/2 + 360*p.f_slice_offset*(100e-9)
        else:
            yield (
                seq.pulse_start(p.f, p_90, p.a_90)
                + seq.wait(p.t_90)
                + seq.pulse_end()
                + seq.wait(p.t_grad_ramp)
            )

        yield (
            g_ramp_zero_to_read_unslice
            + seq.wait(t_phase_read)
            + g_ramp_read_unslice_to_zero
            + seq.wait(t_phase_read_180)
        )

        for i_echo in range(n_echos):
            g_phase = g_ZERO

            if enable_softpulse:
                p_rf_acc += p_180_inc/2
                yield (
                    # refocusing pulse
                    g_ramp_zero_to_slice
                    + seq.pulse_start(p.f+p.f_slice_offset,
                        (p_180-p_rf_acc)%360, p.a_180)
                    + seq.wait(p.t_180)
                    + seq.pulse_update(p.f, 0, 0) # reset
                        transmitter back to reference frame
                        frequency
                    + g_ramp_slice_to_zero
                    + seq.pulse_end()
                )
```

```
                p_rf_acc += p_180_inc/2
            else:
                yield (
                    # refocusing pulse
                    seq.pulse_start(p.f, p_180, p.a_180)
                    + seq.wait(p.t_180)
                    + seq.pulse_end()
                )

            yield (
                # phase blip
                seq.gradient(*g_phase)
                + seq.wait(p.t_phase)
                + seq.gradient(*g_ZERO)
                + seq.wait(p.t_grad_ramp)

                # readout
                + g_ramp_zero_to_read
                + seq.wait(t_acq_delay)
                # + seq.acquire(p.f, p_acq, p.t_dw, p.n_samples)
                + seq.wait(p.t_read-t_acq_delay)
                + g_ramp_read_to_zero

                # unphase blip
                + seq.gradient(*(-g_phase))
                + seq.wait(p.t_phase)
                + seq.gradient(*g_ZERO)
                + seq.wait(p.t_grad_ramp)
            )

        # spoiler pulse
        yield seq.gradient(*p.g_spoil)
        yield seq.wait(p.t_spoil)
        yield seq.gradient(*g_ZERO)
        yield seq.wait(2*p.t_grad_ramp)

        yield seq.wait(p.t_end)

    for i_scan in range(p.n_scans):
        p_rf_acc = p_rf_acc % 360 # prevent growing too large and
            causing floating point rounding errors
        p_90 = p_acq = phase_cycle_90[i_scan % n_phase_cycle]
        p_180 = phase_cycle_180[i_scan % n_phase_cycle]
        for i_run in range(n_runs):
            if p.enable_IR:
                # inversion pulse
                yield seq.pulse_start(p.f, p_rf_acc, p.a_180)
                yield seq.wait(p.t_180)
                yield seq.pulse_end()
                yield seq.wait(t_inv_90)
```

```
295
296             if enable_softpulse:
297                 p_rf_acc += p_90_inc/2
298                 yield (
299                     g_ramp_zero_to_slice
300                     + soft_pulse(p.f+p.f_slice_offset,
                            (p_90-p_rf_acc)%360, p.a_90, p.t_90,
                            p.bw_90)
301                     + seq.wait(100e-9)
302                     + seq.pulse_update(p.f, 0, 0) # reset
                            transmitter back to reference frame
                            frequency
303                     + g_ramp_slice_to_zero
304                 )
305                 p_rf_acc += p_90_inc/2 +
                        360*p.f_slice_offset*(100e-9)
306             else:
307                 yield (
308                     seq.pulse_start(p.f, p_90, p.a_90)
309                     + seq.wait(p.t_90)
310                     + seq.pulse_end()
311                     + seq.wait(p.t_grad_ramp)
312                 )
313
314             yield (
315                 g_ramp_zero_to_read_unslice
316                 + seq.wait(t_phase_read)
317                 + g_ramp_read_unslice_to_zero
318                 + seq.wait(t_phase_read_180)
319             )
320
321             for i_echo in range(n_echos):
322                 # calculate interlaced phase index
323                 i_phase = i_echo * n_runs + i_run
324                 if i_phase < p.n_phase_1*p.n_phase_2:
325                     # calculate gradient
326                     i_phase_1 = i_phase % p.n_phase_1
327                     i_phase_2 = i_phase // p.n_phase_1
328                     g_phase = p.g_phase_1 +
                            i_phase_1*g_phase_1_step + p.g_phase_2 +
                            i_phase_2*g_phase_2_step
329                 else:
330                     # special case when p.n_phase_1*p.n_phase_2
                            does not divide evenly into p.n_ETL
331                     # and there are extra echos
332                     g_phase = g_ZERO
333
334                 if enable_softpulse:
335                     p_rf_acc += p_180_inc/2
336                     yield (
```

```
                        # refocusing pulse
                        g_ramp_zero_to_slice
                        + seq.pulse_start(p.f+p.f_slice_offset,
                            (p_180-p_rf_acc)%360, p.a_180)
                        + seq.wait(p.t_180)
                        + seq.pulse_update(p.f, 0, 0) # reset
                            transmitter back to reference frame
                            frequency
                        + g_ramp_slice_to_zero
                        + seq.pulse_end()
                    )
                    p_rf_acc += p_180_inc/2
                else:
                    yield (
                        # refocusing pulse
                        seq.pulse_start(p.f, p_180, p.a_180)
                        + seq.wait(p.t_180)
                        + seq.pulse_end()
                    )

                yield (
                    # phase blip
                    seq.gradient(*g_phase)
                    + seq.wait(p.t_phase)
                    + seq.gradient(*g_ZERO)
                    + seq.wait(p.t_grad_ramp)

                    # readout
                    + g_ramp_zero_to_read
                    + seq.wait(t_acq_delay)
                    + seq.acquire(p.f, p_acq, p.t_dw, p.n_samples)
                    + seq.wait(p.t_read-t_acq_delay)
                    + g_ramp_read_to_zero

                    # unphase blip
                    + seq.gradient(*(-g_phase))
                    + seq.wait(p.t_phase)
                    + seq.gradient(*g_ZERO)
                    + seq.wait(p.t_grad_ramp)
                )

            # spoiler pulse
            yield seq.gradient(*p.g_spoil)
            yield seq.wait(p.t_spoil)
            yield seq.gradient(*g_ZERO)
            yield seq.wait(2*p.t_grad_ramp)

            yield seq.wait(p.t_end)
```

## B.2 RARE_3D_CS_ONLY.py

```
    from matipo import sequence as seq
from matipo import ParDef
from matipo import datalayout
from collections import namedtuple
import numpy as np
from functools import partial
from matipo.util.pulseshape import calc_soft_pulse
from matipo.util import etl

# TODO: move to library
def float_array(v):
    a = np.array(v, dtype=float)
    a.setflags(write=False)
    return a

import logging
logging.basicConfig()
log = logging.getLogger(__name__)
log.setLevel(logging.DEBUG)

g_ZERO = float_array((0,0,0))

def soft_pulse(freq, phase, amp, width, bandwidth):
    N, dt, pts = calc_soft_pulse(width, bandwidth)
    pts *= amp
    softpulse = seq.pulse_start(freq, phase, pts[0]) + seq.wait(dt)
    for amp in pts[1:]:
        softpulse += seq.pulse_update(freq, phase, amp) +
           seq.wait(dt)
    softpulse += seq.pulse_end()
    return softpulse

# TODO: move to library
def gen_grad_ramp(g_start, g_end, t, n):
    g_step = (g_end - g_start)/n
    t_step = t/n
    ret = b''
    for i in range(n):
        ret += seq.gradient(*(g_start+(i+1)*g_step))
        ret += seq.wait(t_step)
    return ret

PARDEF = [
    ParDef('n_scans', int, 2, min=1),
    ParDef('f', float, 1e6),
    ParDef('a_90', float, 0),
    ParDef('t_90', float, 32e-6),
    ParDef('f_slice_offset', float, 0),
```

```
48      ParDef('bw_90', float, 0),
49      ParDef('a_180', float, 0),
50      ParDef('t_180', float, 32e-6),
51      ParDef('t_inv', float, 1e-3),
52      ParDef('t_dw', float, 1e-6),
53      ParDef('n_samples', int, 320),
54      ParDef('t_read', float, 320e-6),
55      ParDef('t_phase', float, 160e-6),
56      ParDef('n_phase_1', int, 32),
57      ParDef('n_phase_2', int, 32),
58      ParDef('n_phase_2_target', int, 32),
59      ParDef('n_phase_1_target', int, 32),
60      ParDef('n_ETL', int, 0),
61      ParDef('t_end', float, 1),
62      ParDef('g_slice', float_array, (0, 0, 0), min=(-1, -1, -1),
            max=(1, 1, 1)),
63      ParDef('g_read', float_array, (0, 0, 0), min=(-1, -1, -1),
            max=(1, 1, 1)),
64      ParDef('g_phase_1', float_array, (0, 0, 0), min=(-1, -1, -1),
            max=(1, 1, 1)),
65      ParDef('g_phase_2', float_array, (0, 0, 0), min=(-1, -1, -1),
            max=(1, 1, 1)),
66      ParDef('t_spoil', float, 1e-3),
67      ParDef('g_spoil', float_array, (0, 0, 0), min=(-1, -1, -1),
            max=(1, 1, 1)),
68      ParDef('t_grad_ramp', float, 100e-6),
69      ParDef('n_grad_ramp', int, 10),
70      ParDef('shim_x', float, 0, min=-1, max=1),
71      ParDef('shim_y', float, 0, min=-1, max=1),
72      ParDef('shim_z', float, 0, min=-1, max=1),
73      ParDef('shim_z2', float, 0, min=-1, max=1),
74      ParDef('shim_zx', float, 0, min=-1, max=1),
75      ParDef('shim_zy', float, 0, min=-1, max=1),
76      ParDef('shim_xy', float, 0, min=-1, max=1),
77      ParDef('shim_x2y2', float, 0, min=-1, max=1),
78      ParDef('enable_IR', bool, False),
79      ParDef('enable_dummy_run', bool, False),
80      ParDef('set_phase_2_samp', bool, False),
81      ParDef('set_phase_12_samp', bool, False),
82      # ParDef('g_phase_1_lines',list, [20,5,2,-2,-5,-20])
83      ParDef('g_phase_12_list',list, [(0, 0), (1, 0), (2, 0), (5,
            0), (11, 0)] )
84  ]
85
86  # TODO: move to library
87  ParameterSet = namedtuple('ParameterSet', [pd.name for pd in
        PARDEF])
88
89
90  def get_options(p: ParameterSet):
```

```
    return seq.Options(
        amp_enabled=True,
        rx_gain=7)


def get_datalayout(p: ParameterSet):
    n_runs, n_echos = etl.sequence_format(p.n_ETL, p.n_phase_1,
       p.n_phase_2)
    # note: if the total number of phase steps does not divide
       evenly into n_ETL
    # there will be additional data in the last run with no phase
       encoding
    # which may be ignored or exploited
    return datalayout.Scans(
        p.n_scans,
        datalayout.Repetitions(
            n_runs,
            datalayout.Repetitions(
                n_echos,
                datalayout.Acquisition(
                    n_samples=p.n_samples,
                    t_dw=p.t_dw))))

def main(p: ParameterSet):
    if p.set_phase_12_samp:
        # p.n_ETL=1
        n_phase_1=32 #32
        n_phase_2=64
        n_phase_1_target=p.n_phase_1_target
        n_phase_2_target=p.n_phase_2_target
    else:
        n_phase_2=p.n_phase_2
        n_phase_2_target=p.n_phase_2
    print(f'n_phase_2: {n_phase_2}')
    print(f'n_ETL:{p.n_ETL}')
    if p.n_phase_1>1:
        g_phase_1_step = -p.g_phase_1/(n_phase_1_target//2)
    else:
        g_phase_1_step = 0
    log.debug(f'phase 1 step: {str(g_phase_1_step)}')
    if n_phase_2>1:
         g_phase_2_step = -p.g_phase_2/(n_phase_2_target//2)
    else:
        g_phase_2_step = 0
    log.debug(f'phase 2 step: {str(g_phase_2_step)}')

    n_runs, n_echos = etl.sequence_format(p.n_ETL, n_phase_1,
       n_phase_2)
    print(f"n_runs: {n_runs}, n_echos: {n_echos}")
    log.debug(f"n_runs: {n_runs}, n_echos: {n_echos}")
```

```
137
138     t_echo = p.t_180 + 4*p.t_grad_ramp + 2*p.t_phase + p.t_read
139
140     log.debug(f"echo time: {t_echo}")
141
142     t_phase_read = p.t_read/2 - p.t_grad_ramp/2
143     log.debug(f"t_phase_read: {t_phase_read*1e6} us")
144     t_phase_read_180 = (t_echo - p.t_90 - p.t_180)/2 -
            t_phase_read - 3*p.t_grad_ramp
145     log.debug(f"preread gradient to 180 time:
            {t_phase_read_180*1e6} us")
146     if t_phase_read_180 <= 0:
147         raise Exception('phase gradient time too short')
148
149     t_acq_delay = (p.t_read - p.n_samples*p.t_dw)/2
150     if t_acq_delay < 0:
151         raise Exception('t_read too short!')
152
153
154     # time from end of inversion pulse to start of excitation pulse
155     t_inv_90 = p.t_inv - (p.t_90+p.t_180)/2
156     if t_inv_90 <= 0:
157         raise Exception('t_inv too short!')
158
159     n_phase_cycle = 4
160     phase_cycle_90 = [0, 180, 0, 180]
161     phase_cycle_180 = [90, 90, 270, 270]
162
163     p_rf_acc = 0 # accumation of RF phase by running at a
            different frequency
164     p_90_inc = 0 # phase accumulated by a 90 pulse (to be
            calculated)
165     p_180_inc = 0 # phase accumulated by a 180 pulse (to be
            calculated)
166
167     enable_softpulse = p.bw_90 > 0
168     if enable_softpulse:
169         t_inv_90 -= p.t_grad_ramp # keep t_inv setting the time
                between centres of pulses
170
171         # calculate the the gradient amplitude required to match
                half the slice gradient area during the read prephasing
                gradient time
172         g_unslice =
                -p.g_slice*0.5*(p.t_90+p.t_grad_ramp)/(t_phase_read+p.t_grad_ramp)
173         log.debug(f'g_unslice: {str(g_unslice)}')
174
175         log.debug('using softpulse')
176         # to offset the slice, the RF pulses must have a frequency
                offset,
```

```
177         # which introduces a phase change in the transmitter
                relative to the receiver,
178         # p_90_inc and p_180_inc are the amount of phase
                accumulated during a 90 and 180 pulse
179         p_90_inc = 360*p.f_slice_offset*p.t_90
180         p_180_inc = 360*p.f_slice_offset*p.t_180
181         log.debug(f'p_90_inc: {p_90_inc}, p_180_inc: {p_180_inc}')
182     else:
183         g_unslice = g_ZERO
184 
185     # gradient duty cycle check TODO: implement duty cycle checks
            in driver
186     t_cpmg = p.t_90/2 + t_echo/2 + n_phase_2*t_echo - p.t_180/2 +
            p.t_spoil
187     if p.enable_IR:
188         t_cpmg += p.t_180/2 + p.t_inv - p.t_90/2
189     t_rep = t_cpmg + p.t_end
190     grad_total_area = (
191         np.abs(p.g_slice)*(p.t_90+p.t_grad_ramp)
192         + np.abs(p.g_read+g_unslice)*(t_phase_read+p.t_grad_ramp)
193         +
                n_phase_2*2*(np.abs(p.g_phase_1)+np.abs(p.g_phase_2))*p.t_phase
194         + n_phase_2*np.abs(p.g_slice)*p.t_180
195         + np.abs(p.g_spoil)*p.t_spoil
196     )
197     grad_duty_cycle = grad_total_area/t_rep
198     log.debug(f'gradient duty cycle: {str(grad_duty_cycle)}')
199     if np.any(grad_duty_cycle > 0.3):
200         raise Exception('Gradient duty cycle too high!')
201 
202     # pregenerate fixed ramps (optimisation)
203     g_ramp_zero_to_slice = gen_grad_ramp(g_ZERO, p.g_slice,
            p.t_grad_ramp, p.n_grad_ramp)
204     g_ramp_slice_to_zero = gen_grad_ramp(p.g_slice, g_ZERO,
            p.t_grad_ramp, p.n_grad_ramp)
205     g_ramp_zero_to_read_unslice = gen_grad_ramp(g_ZERO,
            p.g_read+g_unslice, p.t_grad_ramp, p.n_grad_ramp)
206     g_ramp_read_unslice_to_zero =
            gen_grad_ramp(p.g_read+g_unslice, g_ZERO, p.t_grad_ramp,
            p.n_grad_ramp)
207     g_ramp_zero_to_read = gen_grad_ramp(g_ZERO, p.g_read,
            p.t_grad_ramp, p.n_grad_ramp)
208     g_ramp_read_to_zero = gen_grad_ramp(p.g_read, g_ZERO,
            p.t_grad_ramp, p.n_grad_ramp)
209 
210     yield seq.shim(p.shim_x, p.shim_y, p.shim_z, p.shim_z2,
            p.shim_zx, p.shim_zy, p.shim_xy, p.shim_x2y2)
211     yield seq.wait(0.01)
212 
213     if p.enable_dummy_run:
```

```
        i_run = 0
        p_90 = phase_cycle_90[-1]
        p_180 = phase_cycle_180[-1]

        if p.enable_IR:
            # inversion pulse
            yield seq.pulse_start(p.f, p_rf_acc, p.a_180)
            yield seq.wait(p.t_180)
            yield seq.pulse_end()
            yield seq.wait(t_inv_90)

        if enable_softpulse:
            p_rf_acc += p_90_inc/2
            yield (
                g_ramp_zero_to_slice
                + soft_pulse(p.f+p.f_slice_offset,
                    (p_90-p_rf_acc)%360, p.a_90, p.t_90, p.bw_90)
                + seq.wait(100e-9)
                + seq.pulse_update(p.f, 0, 0) # reset transmitter
                    back to reference frame frequency
                + g_ramp_slice_to_zero
            )
            p_rf_acc += p_90_inc/2 + 360*p.f_slice_offset*(100e-9)
        else:
            yield (
                seq.pulse_start(p.f, p_90, p.a_90)
                + seq.wait(p.t_90)
                + seq.pulse_end()
                + seq.wait(p.t_grad_ramp)
            )

        yield (
            g_ramp_zero_to_read_unslice
            + seq.wait(t_phase_read)
            + g_ramp_read_unslice_to_zero
            + seq.wait(t_phase_read_180)
        )

        for i_echo in range(n_echos):
            g_phase = g_ZERO

            if enable_softpulse:
                p_rf_acc += p_180_inc/2
                yield (
                    # refocusing pulse
                    g_ramp_zero_to_slice
                    + seq.pulse_start(p.f+p.f_slice_offset,
                        (p_180-p_rf_acc)%360, p.a_180)
                    + seq.wait(p.t_180)
```

```
                    + seq.pulse_update(p.f, 0, 0) # reset
                        transmitter back to reference frame
                        frequency
                    + g_ramp_slice_to_zero
                    + seq.pulse_end()
                )
                p_rf_acc += p_180_inc/2
            else:
                yield (
                    # refocusing pulse
                    seq.pulse_start(p.f, p_180, p.a_180)
                    + seq.wait(p.t_180)
                    + seq.pulse_end()
                )

            yield (
                # phase blip
                seq.gradient(*g_phase)
                + seq.wait(p.t_phase)
                + seq.gradient(*g_ZERO)
                + seq.wait(p.t_grad_ramp)

                # readout
                + g_ramp_zero_to_read
                + seq.wait(t_acq_delay)
                # + seq.acquire(p.f, p_acq, p.t_dw, p.n_samples)
                + seq.wait(p.t_read-t_acq_delay)
                + g_ramp_read_to_zero

                # unphase blip
                + seq.gradient(*(-g_phase))
                + seq.wait(p.t_phase)
                + seq.gradient(*g_ZERO)
                + seq.wait(p.t_grad_ramp)
            )

        # spoiler pulse
        yield seq.gradient(*p.g_spoil)
        yield seq.wait(p.t_spoil)
        yield seq.gradient(*g_ZERO)
        yield seq.wait(2*p.t_grad_ramp)

        yield seq.wait(p.t_end)

    for i_scan in range(p.n_scans):
        p_rf_acc = p_rf_acc % 360 # prevent growing too large and
            causing floating point rounding errors
        p_90 = p_acq = phase_cycle_90[i_scan % n_phase_cycle]
        p_180 = phase_cycle_180[i_scan % n_phase_cycle]
        for i_run in range(n_runs):
```

```
307             if p.enable_IR:
308                 # inversion pulse
309                 yield seq.pulse_start(p.f, p_rf_acc, p.a_180)
310                 yield seq.wait(p.t_180)
311                 yield seq.pulse_end()
312                 yield seq.wait(t_inv_90)
313 
314             if enable_softpulse:
315                 p_rf_acc += p_90_inc/2
316                 yield (
317                     g_ramp_zero_to_slice
318                     + soft_pulse(p.f+p.f_slice_offset,
                        (p_90-p_rf_acc)%360, p.a_90, p.t_90,
                        p.bw_90)
319                     + seq.wait(100e-9)
320                     + seq.pulse_update(p.f, 0, 0) # reset
                        transmitter back to reference frame
                        frequency
321                     + g_ramp_slice_to_zero
322                 )
323                 p_rf_acc += p_90_inc/2 +
                    360*p.f_slice_offset*(100e-9)
324             else:
325                 yield (
326                     seq.pulse_start(p.f, p_90, p.a_90)
327                     + seq.wait(p.t_90)
328                     + seq.pulse_end()
329                     + seq.wait(p.t_grad_ramp)
330                 )
331 
332             yield (
333                 g_ramp_zero_to_read_unslice
334                 + seq.wait(t_phase_read)
335                 + g_ramp_read_unslice_to_zero
336                 + seq.wait(t_phase_read_180)
337             )
338 
339             for i_echo in range(n_echos):
340                 # calculate interlaced phase index
341                 i_phase = i_echo * n_runs + i_run
342                 # print(p.g_phase_2_lines)
343                 if i_phase < n_phase_1*n_phase_2:
344                     # calculate gradient
345                     if not p.set_phase_12_samp:
346                         i_phase_1 = i_phase % p.n_phase_1
347                         i_phase_2 = i_phase // p.n_phase_1
348                         g_phase = p.g_phase_1 +
                            i_phase_1*g_phase_1_step + p.g_phase_2
                            + i_phase_2*g_phase_2_step
349                     else:
```

```
                        i_phase_1, i_phase_2 =
                            p.g_phase_12_list[i_phase]
                        g_phase = p.g_phase_1 +
                            i_phase_1*g_phase_1_step + p.g_phase_2
                            + i_phase_2*g_phase_2_step
                    # print(i_phase_2)
                    print(f'i_phase: {i_phase}, i_phase_2:
                        {i_phase_2}')


                    print(f'g_phase: {g_phase}, g_phase_2:
                        {p.g_phase_2}, g_phase_1: {p.g_phase_1},
                        g_phase_2_step: {g_phase_2_step}')
                else:
                    # special case when p.n_phase_1*n_phase_2 does
                        not divide evenly into p.n_ETL
                    # and there are extra echos
                    g_phase = g_ZERO

                if enable_softpulse:
                    p_rf_acc += p_180_inc/2
                    yield (
                        # refocusing pulse
                        g_ramp_zero_to_slice
                        + seq.pulse_start(p.f+p.f_slice_offset,
                            (p_180-p_rf_acc)%360, p.a_180)
                        + seq.wait(p.t_180)
                        + seq.pulse_update(p.f, 0, 0) # reset
                            transmitter back to reference frame
                            frequency
                        + g_ramp_slice_to_zero
                        + seq.pulse_end()
                    )
                    p_rf_acc += p_180_inc/2
                else:
                    yield (
                        # refocusing pulse
                        seq.pulse_start(p.f, p_180, p.a_180)
                        + seq.wait(p.t_180)
                        + seq.pulse_end()
                    )

                yield (
                    # phase blip
                    seq.gradient(*g_phase)
                    + seq.wait(p.t_phase)
                    + seq.gradient(*g_ZERO)
                    + seq.wait(p.t_grad_ramp)

                    # readout
```

```
                    + g_ramp_zero_to_read
                    + seq.wait(t_acq_delay)
                    + seq.acquire(p.f, p_acq, p.t_dw, p.n_samples)
                    + seq.wait(p.t_read-t_acq_delay)
                    + g_ramp_read_to_zero

                    # unphase blip
                    + seq.gradient(*(-g_phase))
                    + seq.wait(p.t_phase)
                    + seq.gradient(*g_ZERO)
                    + seq.wait(p.t_grad_ramp)
                )

            # spoiler pulse
            yield seq.gradient(*p.g_spoil)
            yield seq.wait(p.t_spoil)
            yield seq.gradient(*g_ZERO)
            yield seq.wait(2*p.t_grad_ramp)

            yield seq.wait(p.t_end)
```

## B.3 3D_RARE_CS.ipynb

### 1º block

```
    from matipo import SEQUENCE_DIR, GLOBALS_DIR
from matipo.sequence import Sequence
from matipo.util.autophase import autophase
from matipo.util.decimation import decimate
from matipo.util.fft import fft_reconstruction
from matipo.util.etl import deinterlace
import numpy as np
import yaml
import matplotlib.pyplot as plt
import time
from scipy.spatial.transform import Rotation
# set matplotlib figure size, default is quite small
plt.rcParams['figure.figsize'] = [10, 5]
plt.rcParams['figure.dpi'] = 100
# progress_handler for Sequence.run() that simply prints the
    progress
def print_progress(p, l):
    print(p, '/', l)
```

**2º block**

```
    # load pulse sequence
using_3D_CS = True

seq = Sequence('RARE_3D_CS_ONLY.py')

seq.loadpar(GLOBALS_DIR+'shims.yaml')
seq.loadpar(GLOBALS_DIR+'frequency.yaml')
seq.loadpar(GLOBALS_DIR+'hardpulse_90.yaml')
seq.loadpar(GLOBALS_DIR+'hardpulse_180.yaml')

#seq.loadpar(GLOBALS_DIR+'softpulse_90_8.0mm.yaml')

#with open(GLOBALS_DIR+'gradient_calibration.yaml', 'r') as f:
 #           gradient_calibration =
    float(yaml.load(f)['gradient_calibration'])

g_slice = seq.par.g_slice*np.array([0, 0, 1.0]) # use g_slice
   loaded from softpulse_90 file
#slice_offset = 3 #mm

DECIMATION = 4

rot = Rotation.from_rotvec([0, np.pi/4, 0]) # rotate around y axis

pe_lines = np.load('g_phase_12_list.npy')
#np.load('g_phase_12_list.npy')
#[(i, j) for i in range(64) for j in range(64)]
#[(0,0),(1,0),(0,1),(1,1)]
#np.load('maks_lines_acc_8.npy')

seq.setpar(
    f_slice_offset=3e3,
    #(slice_offset*1e-3)*(1/gradient_calibration)*g_slice[2],
    #3e3,

    g_slice=g_slice,

    n_ETL=0,

    # for 2D, just use phase_2 and set n_phase_1 to 1 with no
       gradient
    #n_phase_1=1,
    #g_phase_1=(0,0,0),

    g_phase_2= ([0, 1.0, 0]),
    g_phase_1= ([0, 0, 1.0]),
    # oversample by a factor of 4 for flat filtering, 64*4 = 256
       samples
```

```
    t_dw=8e-6,
    n_samples=64*DECIMATION,
    g_read=([1.0, 0, 0]),

    # t_inv=1.2,
    # enable_IR=False,

    t_end=.8
    # enable_dummy_run=True
)

if using_3D_CS:
    seq.setpar(
        # 64 phase steps
        set_phase_12_samp=True,
        n_phase_1=32,   #32
        n_phase_2=64,
        n_phase_1_target=64,
        n_phase_2_target=64,
        g_phase_12_list=pe_lines
    )
else:
    seq.setpar(
        # 64 phase steps
        set_phase_12_samp=False,
        n_phase_2=64
    )

# set calculated read gradient pulse duration
seq.setpar(t_read=seq.par.t_dw*seq.par.n_samples)
# set phase  pulse duration so that the area is half the read
   pulse area
# seq.setpar(t_phase=seq.par.t_read/2)
seq.setpar(t_phase=np.abs(np.linalg.norm(seq.par.g_read)/
np.linalg.norm(seq.par.g_phase_2))*seq.par.t_read/2)

# print out the parameter set for reference
print(seq.par)

start = time.time()

# run sequence, progress_handler is optional
await seq.run(progress_handler=print_progress)
end_time = time.time()

delta_time = end_time - start
print(f'acquisition time: {delta_time:.3f}')

# deinterlace and decimate data
if seq.par.set_phase_12_samp:
```

```
    kdata = deinterlace(seq.data, seq.par.n_ETL,
        seq.par.n_phase_1, seq.par.n_phase_2, seq.par.n_samples)

    print(np.shape(seq.data))
    print(np.shape(kdata))
else:
    kdata = decimate(
        deinterlace(seq.data, seq.par.n_ETL, 1, seq.par.n_phase_2,
            seq.par.n_samples),
        DECIMATION, axis=1)
```

**3º block**

```
    plt.plot(kdata.flatten().real)
plt.plot(kdata.flatten().imag)
plt.show()
```

**4º block**

```
    # use fourier reconstruction to get the 2D projection image
image = fft_reconstruction(kdata)#, gaussian_blur=1,
    upscale_factor=2)
print(np.shape(image))

plt.figure(figsize=(10, 12))
# plot kspace data
plt.subplot(3, 2, 1)
plt.imshow(np.log(np.abs(kdata[:,:,115])))
plt.title('K-Space Data XY')

# plot image data
plt.subplot(3, 2, 2)
plt.imshow(np.abs(image[:,:,115]), cmap='gray')
plt.title('Image XY')

plt.subplot(3, 2, 3)
plt.imshow(np.log(np.abs(kdata[:,30,:])))
plt.title('K-Space Data XZ')

plt.subplot(3, 2, 4)
plt.imshow(np.abs(image[:,30,:]), cmap='gray')
plt.title('Image XZ')

plt.subplot(3, 2, 5)
plt.imshow(np.log(np.abs(kdata[30,:,:])))
plt.title('K-Space Data YZ')

plt.subplot(3, 2, 6)
```

```
plt.imshow(np.abs(image[30,:,:]), cmap='gray')
plt.title('Image YZ')

plt.tight_layout()
plt.show()
```

**5º block**

```
    mask_lines=np.load('g_phase_12_list.npy')
zero_image =
   np.zeros((seq.par.n_phase_2_target,seq.par.n_phase_2_target))
for (x,y) in mask_lines:
    zero_image[int(y), int(x)] = 1 #Para arrays numpy, o acesso e
       [y, x]

plt.imshow(zero_image, cmap='gray')
plt.title('Random Mask')
plt.show()

if using_3D_CS:
    mask_lines=seq.par.g_phase_12_list
    kdata_sq = np.zeros([seq.par.n_phase_2_target,
       seq.par.n_phase_2_target,256], dtype='complex')
    k_idx = 110  #slice

    for i, (x, y) in enumerate(mask_lines):
        x, y = int(x), int(y)
        linha = i // 64   #Para arrays numpy, o acesso e [y, x]
           [coluna,linha]
        coluna = i % 64
        kdata_sq[y, x] = kdata[linha, coluna,:]
    print(np.shape(kdata_sq))

    # Visualizacao
    plt.imshow(np.log(np.abs(kdata_sq[:,:,k_idx] + 1e-10)))
    plt.title(f'K-space XY Data with zeros_slice {k_idx}')
    plt.show()

    np.save('kdata_3D_CS.npy',kdata_sq)
```

## B.4 3D_CS_Methodology.ipynb

### 1º block

```
    import numpy as np
from matplotlib import pyplot as plt
from utils import functions
from utils import simulation
import sigpy as sp
```

### 2º block

```
    # Select acceleration factor
acc_factor = 8
im_size=64

## ====================== Generate Random Undersampling Mask
    ========================================

# Generate Probability Density Function and corresponding
    Under-sampling mask
random_mask, pdf = simulation.generate_Sampling_Mask(iter_=10,
    tol=1, imsize=im_size, p=8, pctg=1/acc_factor)#, radius=0.5)


## ====================== Index Fiding of the Generate Random
    Undersampling Mask ========================================

indices = np.where(random_mask)
g_phase_12_list = list(zip(indices[1], indices[0]))

# Convert random_mask into 0s and 1s
mask_01 = random_mask


#print(mask_01)
print(g_phase_12_list)
print(len(g_phase_12_list))

#save the results
np.save('g_phase_12_list.npy', g_phase_12_list)
np.save('random_mask_3D_CS.npy', mask_01)
np.save('pdf_3D_CS.npy', pdf)
```

### 3º block

```
    fig, ax = plt.subplots(1, 2, dpi=150)
ax[0].imshow(random_mask, cmap="gray", vmin=0, vmax=1)
ax[0].set_title('Random Mask')
ax[1].imshow(pdf, cmap="gray", vmin=0, vmax=1)
ax[1].set_title('Probability density function')
```

### 4º block

```
    #Loading k-space acquired on ilumr
kspace=np.load('kdata_3D_CS.npy')
print(kspace.shape)
#Loading of sampled lines in two directions
mask_lines=np.load('g_phase_12_list.npy')

#Loading of pdf
pdf=np.load('pdf_3D_CS.npy')

#loading of mask
mask=np.load('random_mask_3D_CS.npy')
```

### 5º block

```
    # Expandir mask e pdf para shape (64, 64, 256)
mask = np.repeat(mask[:, :, np.newaxis], 256, axis=2)
pdf = np.repeat(pdf[:, :, np.newaxis], 256, axis=2)

# Choose value for lambda
lamb = 0.01

# Sparse representation: apply Wavelet transform
W= sp.linop.Wavelet(kspace.shape, axes=None, wave_name='db4',
    level=3)

# Apply the subsampling mask
Mu = np.divide(np.multiply(kspace,mask),pdf)
print("Shape of Mu:", Mu.shape)
print("Shape of kspace:", kspace.shape)

# Convert to image domain
imu = functions.ifftc_3D(Mu)

# Standardize image to interval [0,1]
imu=(imu-np.min(np.abs(imu)))/np.max(np.abs(imu))
print("Shape of imu:", imu.shape)

#Convert back to k-scpace, to store standardized data
```

```
Mu=functions.fftc_3D(imu)
print("Shape of Mu after fftc_3D:", Mu.shape)

# Convert to sparse domain
Wimu = W*imu
print("Shape of Wimu:", Wimu.shape)

# Apply soft-thresholding in the sparse (in this case - Wavelet) domain
Wimu_thresh = functions.SoftThresh(Wimu, lamb)
print("Shape of Wimu_thresh:", Wimu_thresh.shape)

# Convert to image domain
imu_thres = W.H*Wimu_thresh
print("Shape of imu_thres:", imu_thres.shape)
```

### 6º block

```
    # Plot images:
# 1) aliased image from undersampling
# 2) the result of thresholding c) in the sparse domain
z_show = 124  # escolha a fatia que quiser (0 a 255)

fig, ax = plt.subplots(2, 2, dpi=200, constrained_layout=True)
ax[0, 0].imshow(np.abs(Wimu[:, :, z_show]), cmap="gray", vmin=0, vmax=1)
ax[0, 0].set_title('Under-Sampled')
ax[0, 1].imshow(np.abs(Wimu_thresh[:, :, z_show]), cmap="gray", vmin=0, vmax=1)
ax[0, 1].set_title('Under-Sampled\nThresholded')
ax[1, 0].imshow(np.abs(imu[:, :, z_show]), cmap="gray", vmin=0, vmax=1)
ax[1, 1].imshow(np.abs(imu_thres[:, :, z_show]), cmap="gray", vmin=0, vmax=1)
for (m, n), subplot in np.ndenumerate(ax):
    subplot.set_xticks([])
    subplot.set_yticks([])
```

### 7º block

```
    # Parameters
niter = 10  # number of iterations
tol = 1e-12  # stopping criteria
## ========================= CS reconstruction
# We are going to reconstruct the aliased image imu (Mu in k-space) plotted in
# the first column above.

```

```
# Initialize wavelet operator
W = sp.linop.Wavelet(imu.shape, axes=None, wave_name='db4',
    level=3)

# Initialize vectors for saving errors across iterations
er = np.zeros((niter, 1))
er_ = np.zeros((niter, 1))

im_est = imu

for ii in np.arange(niter):
    imprev = im_est
    # Compute wavelet transform
    Wim = W*imprev

    # apply soft-thresholding in the sparse (in this case -
        Wavelet) domain
    Wim_thres = functions.SoftThresh(Wim, lamb)

    #return to the image domain
    im_thres = W.H*Wim_thres

    #compute the Fourier Transform
    M_est = functions.fftc_3D(im_thres)

    # enforce data consistency
    #M_est[mask>0] = Mu[mask>0]
    #M_est[np.abs(kspace)>0] = Mu[np.abs(kspace)>0]
    alpha = 1
    M_est[mask > 0] = alpha * Mu[mask > 0] + (1 - alpha) *
        M_est[mask > 0]   #if alpha = 1 M_est[mask>0] = Mu[mask>0]

    #update image estimate
    im_est = functions.ifftc_3D(M_est)

    er[ii]= np.linalg.norm(im_est-imprev)
    if er[ii] < tol:
        break

#normalizar para melhorar o contraste
im_est = (np.abs(im_est) - np.min(np.abs(im_est))) /
    np.ptp(np.abs(im_est))
```

### 8º Block

```
    # final solution
z_show = 122 # escolha a fatia que quiser (0 a 255)
fig, ax = plt.subplots(2, 2, dpi=150)
plt.subplots_adjust(hspace=0.6)
ax[0, 0].imshow(np.abs(imu[:,:,z_show]), cmap="gray", vmin=0,
    vmax=1)
ax[0, 0].set_title('Initial')
ax[0, 1].imshow(np.abs(im_est[:,:,z_show]), cmap="gray", vmin=0,
    vmax=1)
ax[0, 1].set_title('Estimated')
ax[1, 0].imshow(np.abs(imu[:,:,z_show]-im_est[:,:,z_show])*5,
    cmap="gray", vmin=0, vmax=1)
ax[1, 1].set_title('Final Error')
for m, subplot in np.ndenumerate(ax):
    subplot.set_xticks([])
    subplot.set_yticks([])
ax[1, 1].plot(er,'red', label='$|\hat{x}_i - \hat{x}_{i-1}|$')
ax[1, 1].set_title('Error over iterations')
ax[1, 1].legend(bbox_to_anchor=(1.2, -0.2), ncols=2)
```

### 9º Block

```
    y_show = 34
fig, ax = plt.subplots(2, 2, dpi=150)
plt.subplots_adjust(hspace=0.6)
ax[0, 0].imshow(np.abs(imu[:,y_show,:]), cmap="gray", vmin=0,
    vmax=1)
ax[0, 0].set_title('Initial')
ax[0, 1].imshow(np.abs(im_est[:,y_show,:]), cmap="gray", vmin=0,
    vmax=1)
ax[0, 1].set_title('Estimated')
ax[1, 0].imshow(np.abs(imu[:,y_show,:]-im_est[:,y_show,:])*5,
    cmap="gray", vmin=0, vmax=1)
ax[1, 1].set_title('Final Error')
for m, subplot in np.ndenumerate(ax):
    subplot.set_xticks([])
    subplot.set_yticks([])
ax[1, 1].plot(er,'red', label='$|\hat{x}_i - \hat{x}_{i-1}|$')
ax[1, 1].set_title('Error over iterations')
ax[1, 1].legend(bbox_to_anchor=(1.2, -0.2), ncols=2)
```

**10º Block**

```
    x_show = 29
fig, ax = plt.subplots(2, 2, dpi=150)
plt.subplots_adjust(hspace=0.6)
ax[0, 0].imshow(np.abs(imu[x_show,:,:]), cmap="gray", vmin=0,
    vmax=1)
ax[0, 0].set_title('Initial')
ax[0, 1].imshow(np.abs(im_est[x_show,:,:]), cmap="gray", vmin=0,
    vmax=1)
ax[0, 1].set_title('Estimated')
ax[1, 0].imshow(np.abs(imu[x_show,:,:]-im_est[x_show,:,:])*5,
    cmap="gray", vmin=0, vmax=1)
ax[1, 1].set_title('Final Error')
for m, subplot in np.ndenumerate(ax):
    subplot.set_xticks([])
    subplot.set_yticks([])
ax[1, 1].plot(er,'red', label='$|\hat{x}_i - \hat{x}_{i-1}|$')
ax[1, 1].set_title('Error over iterations')
ax[1, 1].legend(bbox_to_anchor=(1.2, -0.2), ncols=2)
```

**11º Block**

```
    fig, ax = plt.subplots(1, 4, dpi=150)
ax[0].imshow(np.log(np.abs(kspace[:,:,z_show]+1e-16)))
ax[0].set_title('K-Space Data')
ax[1].imshow(mask[:,:,z_show], cmap="gray", vmin=0, vmax=1)
ax[1].set_title('Random Mask')
ax[2].imshow(pdf[:,:,z_show], cmap="gray", vmin=0, vmax=1)
ax[2].set_title('pdf')
ax[3].imshow(np.log(np.abs(Mu[:,:,z_show]+1e-16)))#, cmap="gray",
    vmin=0, vmax=1)
ax[3].set_title('np.log(np.abs(Mu+1e-16))')
plt.tight_layout()
```