An educational program on magnetic resonance for students
Hoekstra, Alex

Citation
Hoekstra, A. (2023). *An educational program on magnetic resonance for students*.

Version:	Not Applicable (or Unknown)
License:	License to inclusion and publication of a Bachelor or Master Thesis, 2023
Downloaded from:	https://hdl.handle.net/1887/3674221

Note: To cite this publication please use the final published version (if applicable).

An educational program on magnetic resonance for students

THESIS

submitted in partial fulfillment of the
requirements for the degree of

BACHELOR OF SCIENCE
in
PHYSICS

Author :	Alex Hoekstra
Student ID :	s2033283
Supervisor :	Dr. M.I. Huber
2nd corrector :	Dr.ir. P.S.W.M. Logman

Leiden, The Netherlands, December 15, 2023

An educational program on magnetic resonance for students

Alex Hoekstra

Huygens-Kamerlingh Onnes Laboratory, Leiden University
P.O. Box 9500, 2300 RA Leiden, The Netherlands

December 15, 2023

Abstract

In this project, the simulation software MRI2D was developed. It is a program that allows students to simulate the motion of spins during the magnetic resonance imaging experiment. Already available tools did not model multiple spins rotating to image tissues. Besides filling this niche, other design requirements were taken from existing educational tools. All requirements have been met. MRI2D is open-source software and can be downloaded for free. It comes with a manual for downloading and installing and three example exercises directly for classroom use. MRI2D is ready to now be implemented in education. Further development suggestions are given.

Contents

1 Introduction **1**

2 Background **3**
2.1 Equations for simulating MRI with compass motion 3
2.1.1 Physics for a compass: harmonic oscillator 3
2.1.2 Relevant MRI mechanics and equations 5
2.2 Physical demonstrations 7
2.2.1 MRI scanner 7
2.2.2 Existing physical magnetic resonance demonstrations 8
2.3 MRI computer simulation approach 9
2.3.1 Choice of programming language and graphics library 9
2.3.2 Existing MRI simulation software 9
2.4 Education theory 10
2.5 Summary of design requirements 11

3 Methods - Structure and design of MRI2D **13**
3.1 Developing a program 13
3.1.1 Interpreted language or compiled language 13
3.1.2 Code structure 14
3.1.3 Developing and organising code 15
3.2 MRI implementation 17
3.2.1 Simulation physics (S.1, S.2, S.3, S.4, S.5) 17
3.2.2 Visualizing resonance (S.3, S.4) 20
3.2.3 Integration of model and graphics into one program 24
3.2.4 Interpretation of units in MRI2D 25
3.3 Implementation of requirements regarding ease of use 26
3.4 Exercise development 28

4 Results **31**
4.1 Fulfilment of simulation requirements 32
4.1.1 Resonance without tissue (S.1, S.2, S.3, S.4) 32
4.1.2 Resonance with tissue (S.5) 33
4.2 Fulfilment of educational requirements 34
4.2.1 Safe and robust (E.1) 34
4.2.2 Available (E.2) 35
4.2.3 Affordable (E.3) 35
4.2.4 Easily distributed (E.4) 35
4.2.5 Flexible in use (E.5) 35
4.3 Possible demonstrations using MRI2D 36
4.3.1 Exercise 1: Demonstration of magnetic resonance 36
4.3.2 Exercise 2: Determine resonance frequency and magnetic field correlation 36
4.3.3 Example answer to exercise 2 38
4.3.4 Exercise 3: Performing an MRI scan with MRI2D 39
4.3.5 Teacher only: Defining tissues 40
4.3.6 Exercises tested 40

5 Discussion **43**
5.1 Simplification of spin precession frequency for modelling 43
5.1.1 Dimensions of precession in MRI2D 44
5.1.2 Complications with modelling precession frequency 44
5.2 Possible code environments 45
5.2.1 Other approaches to Object-Oriented Programming in Python 45
5.2.2 Possible structure improvements when using other applications for Python 46
5.2.3 Choice of download format 46
5.3 Education research 47
5.3.1 Suggestions for testing MRI2D education effectiveness 47
5.3.2 Possible exercises to be developed 47

6 Conclusion **51**

A User guide **57**
A.1 Installing the program 57
A.2 Running the program 58
A.3 Graphical user interface 58
A.3.1 Runtime screen 58
A.3.2 Analysis screen 59

A.4 Magnetic Resonance Exercises 61
A.4.1 Exercise 1: Demonstration of magnetic resonance 61
A.4.2 Exercise 2: Determine resonance frequency and magnetic field correlation 61
A.4.3 Exercise 3: Performing an MRI scan with MRI2D 62

B Video demonstration details 63
B.1 Measurement 1: Resonance frequency at B_0 magnitude 0.5 T 63
B.2 Measurement 2: Resonance frequency at B_0 magnitude 1.0 T 64

C Python code 65
C.1 Initialisation: Help programs 65
C.2 Simulation 66
C.2.1 Main program 66
C.2.2 Programs containing essential classes and functions 70

Chapter 1

Introduction

Magnetic Resonance Imaging (MRI) is an important imaging technique, as it is used for diagnosing diseases in the brain, heart, muscles and more organs because of its ability to image soft tissues with great detail, in any direction and without ionizing radiation. However, MRI scans take longer than other medical imaging techniques e.g. ultrasound [1]. The longer duration can lead to patient discomfort, especially because MRI scanners are cramped spaces and make loud noises during a scan.

Most people do not know how MRI works, some are even afraid of electromagnetic fields in general [2] [3]. Education can help alleviate these fears, but MRI education has a problem. Aside from the general public, there are non-physics students that need to understand MRI for their field, and they already find the underlying physics of Nuclear Magnetic Resonance (NMR) difficult, especially spin-magnetic field interactions [4]. Therefore we will develop a tool that can help explain these physics.

There is a natural need to simplify the situation to explain an aspect of a complicated topic well. When isolated, spin-magnetic field interactions can be fully explained, and still be helpful in understanding the whole subject later. Compasses are very useful in teaching students magnetic resonance as they are the simplest representation of the spin of an electron or nucleus [5] [6].

Another demonstration with compasses used it as a stepping stone to software further explaining the quantum nature of MRI [7]. They found that this was possibly too big a step, and more support is needed. Especially measuring the frequency better was suggested to give students a better idea of how to create the resonance condition. Software can help in this regard, as all physical constants can be scaled to make the frequency easy to see, and the exact motion can automatically be recorded on the

computer.

That need for clarity is why we developed MRI2D, a program that emulates MRI quantum mechanics with a classical physical model to bring the advantages of software to education with the compass analogy.

In this thesis, we will explain which aspects of MRI quantum mechanics we simulate, the design requirements based on the gaps left by available educational demonstrations and programs, which tools and techniques were used to develop MRI2D, and how to use our program in education.

This thesis is structured as follows: in chapter 2 the choice of approach is discussed by exploring the physics and the relevant existing resources, leading to design requirements. In chapter 3 the design requirements are specified and quantified further by discussing their implementation, which means section 3.3 and section 3.4 are the only sections suitable for readers who do not have experience in programming. In chapter 4 the design requirements are tested and the behaviour of the program is shown. In chapter 5 choices in development are discussed and compared to alternative options, accompanied by suggestions for future research and development. In chapter 6 a brief summary of the results of this thesis is given and contextualised.

Chapter 2

Background

In this chapter we will be looking at established literature covering resonance physics, MRI demonstrations, MRI simulation software, and education theory to determine design requirements for MRI2D.

2.1 Equations for simulating MRI with compass motion

In this section, we will connect the physics of a compass to the quantum mechanics of MRI. First we explore how forces create different kinds of oscillators and what their resonance frequencies are. Then the importance of the Larmor frequency for MRI is explained using the relevant quantum mechanics.

2.1.1 Physics for a compass: harmonic oscillator

The compasses are placed in two magnetic fields, which we call B_0 and B_1 to relate it to the MRI principle, despite some differences in the setup. One strong static field (B_0) is present, which aligns the compasses with this field, as it is much stronger than the earth magnetic field. These fields differ by three orders of magnitude, as they are roughly 10^{-3} T vs 10^{-6} T. This B_0 alignment is caused by the magnetic torque $\vec{T}$ experienced by the compass with a magnetic moment $\vec{m}$, pointing from south to north of the compass, when written as vector, in a field B_0:

$$\vec{T} = \vec{m} \times \vec{B_0} \tag{2.1}$$

Then a second magnetic field (B_1) is used to cause an excitation. The result is that these moments can be summed to give the total torque the compass experiences:

$$\vec{T} \;=\; \vec{m} \times \vec{B_0} \;+\; \vec{m} \times \vec{B_1} \tag{2.2}$$

The trick of this setup consists of the fact that the magnetic fields are orthogonal to each other. When writing out the cross products (assuming B_0 is along the x-axis and B_1 along the y-axis, we can calculate that the resulting torque component in the z-axis now is:

$$T_z \;=\; mB_0 \sin(-\theta) \;+\; mB_1 \sin\left(\frac{\pi}{2} - \theta\right) \tag{2.3}$$

When ignoring friction, as compasses are built in a way to minimise the friction of the rotation, the equation of motion then becomes:

$$J\ddot{\theta} \;=\; mB_0 \sin(-\theta) \;+\; mB_1 \sin\left(\frac{\pi}{2} - \theta\right) \tag{2.4}$$

To estimate the resonance frequency for small excitation angles θ, we can use the following approximations:

$$\sin(-\theta) \approx -\theta \tag{2.5a}$$

$$\sin\left(\frac{\pi}{2} - \theta\right) = \cos(\theta) \approx 1 \tag{2.5b}$$

Substituting these approximations in equation 2.2 gives:

$$\begin{aligned} J\,\ddot{\theta} &= -mB_0\theta \;+\; mB_1(t) \\ J\,\ddot{\theta} \;+\; mB_0\theta &= mB_1(t) \\ \ddot{\theta} \;+\; \frac{mB_0}{J}\theta &= \frac{mB_1(t)}{J} \end{aligned} \tag{2.6}$$

The left side can be solved to arrive at the resonance frequency of this harmonic oscillator:

$$\omega^2 = \frac{mB_0}{J} \iff \omega = \sqrt{\frac{mB_0}{J}} \iff 2\pi = \sqrt{\frac{mB_0}{J}} \tag{2.7}$$

Thus the expression for the resonance frequency in Hz becomes:

$$f = \frac{1}{2\pi}\sqrt{\frac{mB_0}{J}} \tag{2.8}$$

However, this is for a zero-friction situation. In this zero-friction situation, frequencies other than the resonance frequency are not damped out and would continue to be present in the oscillations of the compasses making it harder to distinguish the resonance frequency from the other frequencies. Therefore, introducing a little friction adds to the realism and facilitates finding the resonance frequency.

Introducing friction also affects the resonance frequency:

$$\begin{aligned} J\,\ddot{\theta} &= -mB_0\theta \;-\; k\dot{\theta} \;+\; mB_1(t) \\ J\,\ddot{\theta} \;+\; k\dot{\theta} \;+\; mB_0\theta &= mB_1(t) \\ \ddot{\theta} \;+\; \frac{k}{J}\dot{\theta} \;+\; \frac{mB_0}{J}\theta &= \frac{mB_1(t)}{J} \\ \Rightarrow \omega &= \sqrt{1 \;-\; \frac{k^2}{4mB_0}} \cdot \sqrt{\frac{mB_0}{J}} \\ \Rightarrow \omega &= \sqrt{1 \;-\; \frac{k^2}{4mB_0}} \cdot \omega_{\text{no friction}} \end{aligned} \tag{2.9}$$

This shows that when the damping constant k, used to simulate friction, is sufficiently small in comparison with mB_0/J, the effect on the resonance frequency is minimal. On top of that, a compass demonstrator also only shows the principle of magnetic resonance and not the actual values as in an MRI experiment, so this is not considered to be a problem while it increases the realism of the simulator.

Thus compasses are able to demonstrate the principle of magnetic resonance, despite the compasses oscillating in the same plane as the magnetic fields. Insofar they differ from the three-dimensional precession of spins. Its 2D character also makes it suitable for a digital demonstrator on a 2D screen, where the students can set and interact with the values of strength of the magnetic fields and the frequency of the excitation signal $B_1(t)$ interactively to verify this principle and see its effects and sensitivity to the settings. Added benefits are the configurable friction and also that other physical quantities and constants can be changed easily.

2.1.2 Relevant MRI mechanics and equations

For MRI, the only particle we have to look at is the proton. As a hydrogen nucleus, it mostly is present in water but also in lipids. A proton has an intrinsic angular momentum $\vec{P}$, which gives rise to a magnetic moment $\vec{\mu}$

due to the charge of the proton. The value of this magnetic moment is decided by the intrinsic angular momentum, also known as spin, multiplied by the gyromagnetic ratio γ.

$$\vec{\mu} = \gamma \vec{P} \tag{2.10}$$

When a nucleus is placed in a magnetic field, its magnetic moment will align at an angle of 54.7° with B_0 and precess around B_0 as shown in Figure 2.1. The spin can also point in the opposite direction, but that is the higher energy state, so it will be less populated in equilibrium.

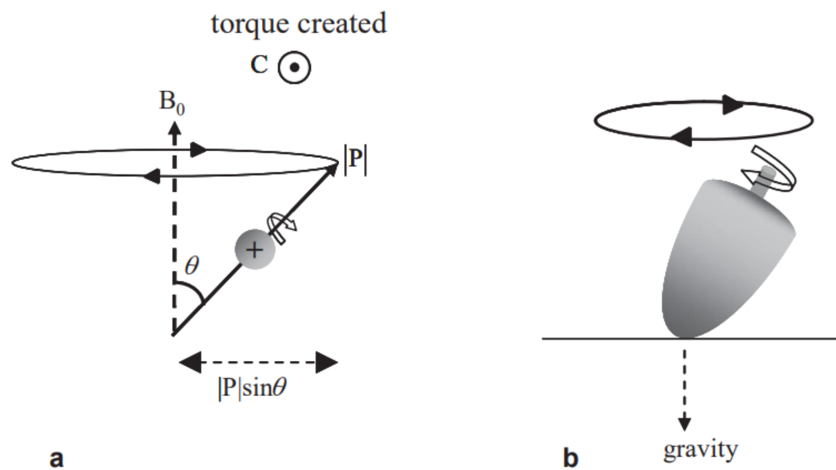


Figure 2.1: *Precession: quantum (a) and classical (b). Torque is created when a rotating object is not aligned with a vertical force, causing precession. Page 209 [1]*

The rate at which this precession occurs is called the Larmor frequency. Via the cross product of $\vec{\mu}$ and $\vec{B_0}$, the Larmor frequency is found:

$$\omega = 2\pi f = \gamma B_0 \tag{2.11}$$

While all the protons in a sample or patient are indeed precessing around B_0 as described (with a small number in the opposite z-direction), they will not be in the same phase. This means the net magnetization of all the protons in equilibrium is simply a positive z-component.

The forces leading to this equilibrium being the lowest energy state depend on the surroundings of each particular proton. This means that when they are brought out of equilibrium by an electromagnetic field B_1 perpendicular to B_0, the time it takes them to return to equilibrium reveals

information about the surroundings of each proton. In clinical MRI this aspect is used to differentiate between healthy and diseased tissues [1].

Perpendicular magnetic fields, and tissue differences leading to different relaxation times in MRI brings us to our first two design requirements for MRI2D.

Our simulated spins should be exposed to *two perpendicular adjustable magnetic fields* B_0 *and* B_1 (S.3) whose excitations *cause different motion depending on the tissue* (S.5) each spin is a part of. We number each design requirement for easy reference in following chapters. The S denotes design requirements that are related to the simulation of physics, which make up one of two types of design requirements we specify, the other being requirements relevant to ease of use in education.

2.2 Physical demonstrations

2.2.1 MRI scanner

Real MRI scanners may seem the best choice for MRI education, but the request for MRI education alternatives wouldn't be as large if it were that simple. There are indeed many advantages to working with the device itself, since every part of the experience should be valuable as training.Hence the downsides listed here will be of a logistical nature, not about the simulation physics.

One of the concerns with full-scale MRI scanners is safety [8]. Intricate safety procedures are harder to follow with younger or larger groups of students. Failure to follow these procedures could result in injuries [9].

This led to our first educational design requirement (E.1): *our design should be safe and robust*. Any accidental misuse should not have serious consequences.

The second hurdle for demonstrations with a real MRI scanner is the varying degree of availability of these devices. Efforts to reduce waiting lists for patients show the limited availability of MRI scanners in time [10]. Besides the aspect of time there is an aspect of travel distance as well.

The design intended in this thesis therefore must be *readily available, preferentially at the educational institution itself* (E.2).

The final problem with MRI scanners is the price. With prices ranging from 30,000 to 700,000 euros, purchase is rarely feasible for use only in a purely educational setting [11]. Smaller systems unfit for imaging human bodies can be used for education to reduce costs. For example, an NMR

spectrometer has been used for imaging small samples, letting students get acquainted with signal processing [12]. A spectrometer for Electron Paramagnetic Resonance (EPR) has also been used by students to study the influence of coils on a magnetic resonance image [13]. Such a device could already be present in an institute for its intended research use, but otherwise it would likely still be too costly to acquire solely for MRI education.

Existing magnetic resonance hardware is too costly for educational institutes, *our design should use more affordable options* (E.3).

2.2.2 Existing physical magnetic resonance demonstrations

Educational setups for magnetic resonance do not have the high requirements that professional instruments do. One option would be to build an EPR spectrometer instead of purchasing one [14]. This line of thinking could be taken further, by simplifying the device to its constituent parts, a functional setup can be constructed with more standard lab equipment [15]. Reducing the amount of specialised tools solves monetary concerns, but both of these described methods require considerable time to prepare before the setup can be used by students. Preferably, each student should be able to work on their own unit.

As such, it should be *easy to distribute multiple copies of our design* (E.4).

That distribution requirement pushes us away from spectrometers, so we will take a look at simple physical models of quantum mechanics. A variety of options for simulating the effects of magnetic fields on a single spin have been developed over the years. Rotation of a magnet in a plane has allowed studying the resonance frequency in different magnetic field strengths against the torque of a glass fibre suspension [16]. Later three-dimensional demonstrations of single spins were created, using airflow to suspend a ball with almost no friction that would inhibit easy rotation [17] [18].

The niche that seems unfulfilled is *a setting that allows for multiple nuclei to be observed at the same time* (S.2). Gradients in magnetic fields could then be used to show the difference in behaviour that leads to the signal in MRI more clearly, without requiring parameter changes. A grid of nuclei could also be used to represent the pixels of a very low resolution image.

2.3 MRI computer simulation approach

2.3.1 Choice of programming language and graphics library

Software is an easy first option to fulfil our requirements. An important choice in developing software is choosing the programming language. Which of the languages, such as C, C#, C++, Java, Python would be the best option is not explored here, as the author was already proficient in Python with little experience in the other popular choices and the Leiden Institute of Physics uses Python in education.

Other considerations when picking a programming language are discussed in section 3.1.

Of course, visualization tools are also needed. Building a program for displaying graphics from the ground up is unnecessary in Python, as Pygame offers modules allowing the Simple DirectMedia Layer (SDL) library to be accessed smoothly [19]. This is not a novel choice, since Pygame has been a popular choice for physics simulations for some time now [20] [21] [22] [23] [24].

2.3.2 Existing MRI simulation software

A wide range of MRI simulation software is already available. This means requirements should be chosen such that our program is filling a gap, to ensure it adds to the existing software.

Our exploration of physical demonstrations in subsection 2.2.2 ended with setups using a single ball with a magnet embedded. A software equivalent of that kind of experiment has already been developed [25]. This software leads us to the same requirement as the physical demonstration did: our program should feature multiple nuclei in a grid to offer something new. That kind of simulator was found to complement traditional education [26]. This affirms our choice for the software approach in an educational environment.

Other software has visualised what happens with the data gathered in MRI scans as fully as possible [27]. Especially training software that helps in teaching the interpretation of images is abundant [28] [29] [30]. While images do represent reality, putting too much focus on images and the layout of tissues skips important steps in the complicated concepts

behind MRI. Software that helps explain how received frequencies lead to an image via Fourier transform already exists [31].

The existing simulation software thus reduces our niche to exploring the quantum mechanics generating these frequencies. To accomplish that, our model will have to *describe the situation in the time and space domain* (S.1), before Fourier transforms to and from the frequency domain.

Finally, there is one simulation in the literature that seems to conform to most of the requirements listed [32]. The PhET project offers many different simulations in many fields of physics, but for us the relevant simulation is named Simplified MRI. It offers a grid of nuclei in adjustable magnetic fields. And, to demonstrate how MRI interacts with different kinds of tissue, it simulates tumours as more densely packed nuclei without changing any of the resonance characteristics. As described in subsection 2.1.2, we will aim to show that nuclei in different tissues resonate at different frequencies.

Our software will differentiate itself from the simulations by fulfilling all previous requirements in addition to *showing visible oscillations of nuclei* (S.4). We can simulate the spin motion instead of having a quantum mechanical like Simplified MRI where motion is omitted in favour of energy levels. Our approach should allow students to get a grasp of the magnetic resonance phenomenon as required to understand MRI, without requiring abstract quantum mechanical knowledge.

2.4 Education theory

In the interest of the educational value of the software that will be developed, it would be wrong not to take research in physics education into account.

The last mentioned existing software in subsection 2.3.2 has already given some brief insight and general suggestions for the use of an MRI simulation in education. Most importantly, it is indicated that there must be a balance between letting students discover by themselves and giving them guidance [32].

The importance of freedom in choices from a psychological perspective is well documented. Self-determination theory explains the value of autonomy for well-being [33].

What we can learn from this is that giving students freedom is paramount to the usefulness of a simulator. Instruction can always be made more ex-

plicit by detailed exercises or teachers. Software programmed with a restrictive lesson plan can never be used freely, while flexible software can be accompanied by any level of guidance.

To ensure software can be used in education with the optimal balance of freedom and guidance, the software itself will have to *allow users to make a lot of choices* (E.5). These choices must also be presented to students without the need for much explanation; if they are not aware of options they can use, they will not profit from the possibilities those options offer.

2.5 Summary of design requirements

Table 2.1: *All described design requirements categorised in the order of development and testing (not the order in which they were described above). The rightmost column lists numbers for referencing.*

Category	Requirement	#
Simulation	Time domain	S.1
Simulation	Multiple nuclei	S.2
Simulation	Adjustable magnetic fields	S.3
Simulation	Visible resonance	S.4
Simulation	Tissue differences	S.5
Educational	Safe and robust	E.1
Educational	Available for education	E.2
Educational	Affordable	E.3
Educational	Easily distributed	E.4
Educational	Flexible in use	E.5

All requirements are described in more detail in chapter 3.

Chapter 3

Methods - Structure and design of MRI2D

The design requirements (see section 2.5, Table 2.1) that arose from the analysis and overview of existing approaches in chapter 2, are specified further when necessary as their implementation is discussed.

In section 3.1 context is provided for choice of software and strategy used for developing the program, useful to readers looking to develop programs themselves. The most technical text is in section 3.2, where the code implementing each requirement is discussed for experienced programmers who want to understand the whole program. In section 3.3 the educational requirements are quantified, making this section suitable for researchers and teachers. For readers planning to develop their own exercises with MRI2D, section 3.4 describes how the example exercises (section A.4) were developed in detail.

3.1 Developing a program

3.1.1 Interpreted language or compiled language

Python is an interpreted language, unlike C, C#, C++ and Java. That means an interpreter simply reads the program in the order it was written and executes the program in the same step. This is different from a compiled language, which uses a compiler that first translates the code into another language before it can be executed. Compiled languages can execute faster, but the compiling takes a lot of time. Compilers tend to be the choice for commercial products that go to non-programmers, as the code

will not be edited, and it will be run many times. Interpreted languages are thus often preferred by scientists who write scripts for analyzing their data, since it is closer to a one-time use program, or it is edited before being used again. An edited program would have to be compiled again if created with a compiled language [34].

To summarise the difference: interpreted languages are faster with writing or editing time included, while compiled languages could be faster if run by a large amount of clients unfamiliar with programming.

3.1.2 Code structure

Interpreted languages lack the kind of code check that compiling has. Simply writing a long list of instructions for the computer without structure leads to code that is very difficult for people to understand. The first step to make a program more readable, is to put a subroutine that is used often inside a function. This way, instead of repeating the code each time, the function can be called. Not only does this reduce the size of code, but by giving the subroutine a comprehensive name people will more easily understand the actions the program performs. Functions take in variables called arguments, and from this they can produce an output.

If functions need to take in and edit the same groups of variables, it gets tiresome to write and hard to read when every output variable needs to become the input (arguments) for other functions. Added difficulty in writing and reading can lead to more bugs and slower debugging for the programmer themselves, and make it even harder for other people to work with the program. Therefore we place the functions in a structure with the variables they often output and need as input. This categorises variables in a group, and show more clearly what the difference is in arguments needed for different functions (because the large group stays the same and need not be mentioned explicitly, any variables from outside the group will stand out as arguments).

Object-Oriented Programming (OOP) can be used in Python to give a program this described structure. A structured program is easier to develop, debug and maintain, because OOP gives you the ability to easily test smaller parts of the program separately.

As alluded to, OOP can categorise code even further than functions by using Classes. Classes contain attributes and methods. Attributes are variables inside the class, methods are functions inside the class. The different names exist because methods can access attributes without each attribute needing to be listed as an argument. The way to program OOP is to make

clear choices on how to split your code into Classes. There is not necessarily one way to do this correctly, as long as it leads to good performance and readability. More on possible OOP structures in subsection 3.1.3.

On top of providing graphics support at a low level, Pygame offers the Sprite module as an option for OOP. Sprite Classes assume you will create an instance of the Class for each object on the screen. The benefit of this approach is access to Sprite's support of collision detection and rendering groups. Both of those benefits are only relevant to programs where the elements on screen can move in such a way that they can overlap. As stated before, our program will have nuclei locked in a grid, able to rotate but not translate, therefore gaining no benefit from what the Sprite module offers. If another structure has advantages for our program, it should be used instead of the Sprite Classes. The chosen OOP structure without the Sprite module will be discussed now along with the implementation of all requirements.

3.1.3 Developing and organising code

Object-Oriented Programming (OOP) was used to develop a structured program in Python. However, the structure is different from the Pygame Sprite classes mentioned in the previous section. This project does not have many different types of elements on the screen, rather many elements of the same type: nuclei are the only type necessary. Also, the same equations are used for all nuclei, only with different numbers. The data structure that Python offers for vectorised calculation is NumPy arrays. Since the Pygame examples outline a different use case, the Sprite module is not compatible with NumPy arrays in this way. Not taking advantage of this vectorised calculation that Python offers would mean a slower program, therefore another structure was used, shown in Figure 3.1 where `graphics.py` uses Pygame modules and `model.py` takes advantage of NumPy arrays. The precise content of the python files is described in detail in section 3.2.

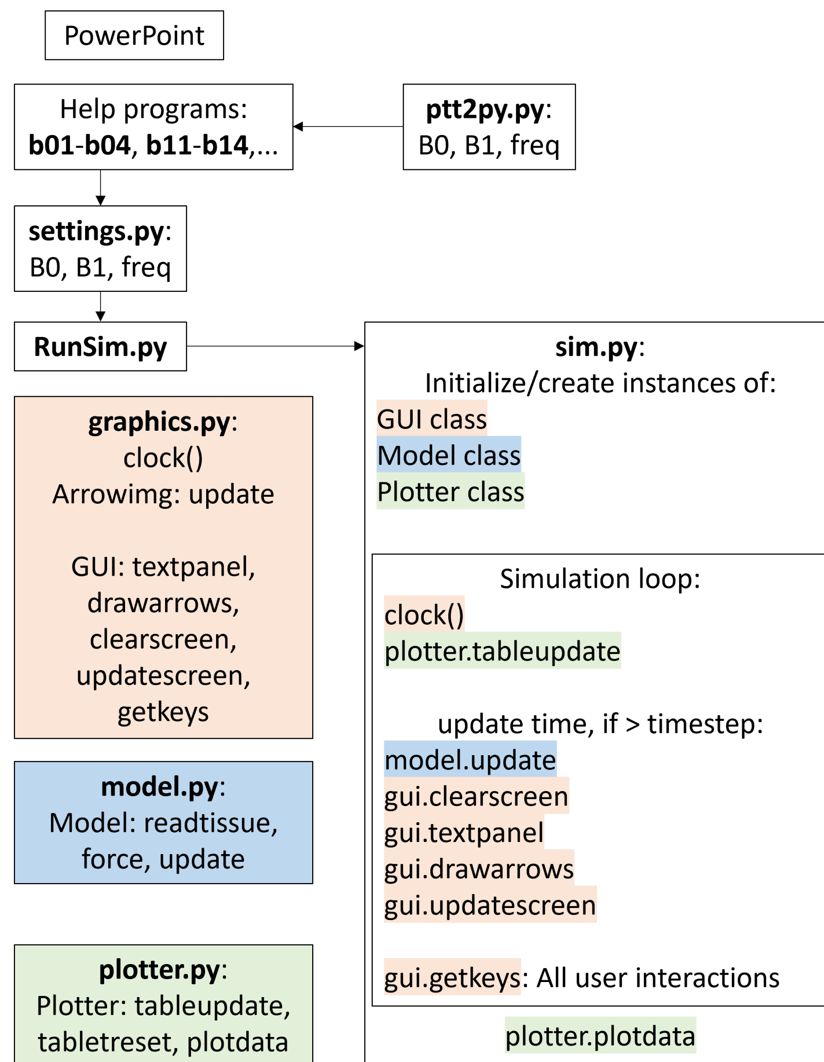


Figure 3.1: *File structure of the code, following object-oriented programming. Pygame is used most in graphics.py and NumPy is used most in model.py. The simulation requirements are implemented as methods in model.py and graphics.py, which are called in sim.py.*

3.2 MRI implementation

The way simulation requirements were implemented is explained in this section. To do that, the code is described in detail, particularly the *Model* class in `model.py` and the *GUI* class in `graphics.py`.

One instance of the *Model* class handles all physics calculations, satisfying all simulation design requirements (S.1, S.2, S.3, S.4 and S.5).

The Graphical User Interface (GUI) in MRI2D is the only instance of the *GUI* class, part of which helps with the implementation of two simulation requirements: S.3 *Adjustable magnetic fields* by providing buttons and the current values, and S.4 *Visible resonance* by displaying spins at the right angle at every moment as calculated by the *Model* class.

3.2.1 Simulation physics (S.1, S.2, S.3, S.4, S.5)

The simulation can quickly be summarised as a grid of compasses (arrows) representing nuclei, influenced by two perpendicular magnetic fields B_0 and B_1. B_1 is the driving magnetic field, which means its strength increases and decreases according to the B_1 frequency controlled by the user.

The *Model* class in `model.py` holds the code representing physical reality. This means the five simulation requirements are implemented here, except for some visual aspects of requirements S.3 and S.4.

The *Model* class is the only content of this file. It contains four methods (`Constructor`, `readtissue`, `force` and `update`) to calculate the necessary physics.

Before diving into the explanation of these methods, let us clarify what the necessary physics mean in terms of the simulation requirements.

The *Model* class is built around the `force` method, which calculates the torques compasses experience due to the magnetic field at that moment; constantly changing as a result of the driving magnetic field B_1. The torque returned by the `force` method leads to an acceleration in the `update` method, which is integrated to an angular velocity, in turn integrated to an angle change for each timestep. This chain integration to evolve angles through time step by step is intended to satisfy the requirement for time domain simulation (S.1).

The other simulation requirements are also implemented in the code delivering NumPy arrays to the `force` method for vectorised calculation described at the beginning of this section. The `Constructor`*, always the

*Represented in Python by *__init__*. The underscores indicate it is a hidden method,

first method of the class, initialises magnetic fields with perpendicular gradients to match their orientation. This should let each compass experience a unique magnetic field (S.2).

As mentioned before, the implementation of magnetic fields that are adjustable during simulation (S.3) also involves code in `graphics.py`, but for `model.py` it entails the `force` method accepting magnetic field arrays as an argument, instead of always using the same arrays whose initialisation was just mentioned.

Visible resonance (S.4) also depends `graphics.py`, but our work with the equations in section 2.1 showed that compasses experiencing torque as implemented in the force method should lead to resonance.

For the final simulation requirement, the `Constructor` calls on `readtissue` to implement the *tissuemask* array chosen by the user. It alters the torque arrows experience in the `update` method accordingly. Because the `update` method is run every frame, this effect of tissue differences will be present throughout the simulation run giving nuclei in different tissues different resonance frequencies (S.5).

Table 3.1: *Parameters for the constructor of the Model class grouped according to their purpose, which is described in the second column.*

Parameters	Description
b0set, b1set, f1set	Starting values for the three configurable quantities
m,n	Dimensions of the arrow grid
xmax, ymax	The largest coordinates available to the grid in terms of pixels
tissuefile	Which optional phantom (*tissuemask*) is used
gui	The instance of the GUI class, concerning graphical operations

model.py

With that in mind, let us look at the methods in the *Model* class:

- `Constructor`(b0set, b1set, f1set, m, n, xmax, ymax, tissuefile, gui):
  As described in Table 3.1, the first three arguments are the initial values of the magnetic fields; B_0 magnitude, B_1 magnitude and

not for use outside of the class. It initialises attributes: variables inside the class. It is called upon when an instance of the class is created.

B_1 frequency. Next follow the dimensions of the grid of arrows, the position of the grid on the screen, a possible tissue to simulate and an instance of the next class to be discussed, the *GUI* class. With all of those parameters an instance of the *Model* class can be created: a Python object that governs the arrows, fields and tissue using the methods discussed next. For more realistic behaviour, not every compass arrow is exactly identical: some variation ("noise") is added to simulate minor variations in magnetic moment or inertia. Angles and angular velocities are initiated at 180 degrees (down) and zero speed. Using the screen size and the dimension of the arrow array (m,n), each arrow is placed at their screen coordinates to form a grid. The actual images of the rotated arrows are stored in the *imglist* with bitmaps for computational speed, this process is part of the *GUI* code discussed later.

- `readtissue`(tissuefile):
  If a *tissuefile* is selected, this reads one of the .tis files from the data folder to create the *tissuemask*.These files each contain an array of factors, simulating the effect of different kinds of tissues on magnetic behaviour. A tissue file is disguised to hide them from students by using the .tis extension, where it is in fact a simple comma separated value file (.csv) which can be edited in any text editor. 1 is the normal value, expectations for different tissues are between 0 and 1.

- `force`(bx, by, theta):
  This function takes in 3 arrays B_0, B_1 and theta. All three arrays contain unique values for each arrow. Using these, it returns the torque on each:

```
# construct magnetic vector in polar coordinates
forcerad = np.arctan2(by, bx)
forcemag = np.sqrt(bx ** 2 + by ** 2)

# convert to radians for sine
thetarad = theta / 360 * 2 * np.pi
anglediff = thetarad - forcerad
# for sine, so outside of 0 to pi domain is fine
```

```
    force = forcemag * np.sin(anglediff)
    return force
```

- `update`(tsim, dt):
  This function contains the numerical integration for the time step *tsim* to $tsim + dt$. First it gets the forces on the arrows, to calculate the accelerations. With the accelerations, the angular velocities are updated. Then the arrows are rotated. Finally, the bitmaps are also updated by calling the `update` function of the *arrowlist* for each arrow.

  dt is the time a frame stays on screen. Due to time control in sim.py, *dt* can be as small as the computer allows or if the computer is very slow, MRI2D sets the simulated timestep to *maxdt*, a tenth of a second, in which case simulated time becomes different from actual time. Setting a maximum limit is necessary because real physics doesn't have timesteps: a slow computer with noticeable timesteps would not be an accurate simulator. Often operating systems have a minimum of a hundredth of a second [35]. Therefore MRI2D has a framerate in between 100 Hz and 10 Hz in simulated time.

  Also note that while angular velocities refers to an array, the velocity per arrow is a scalar whose sign indicates direction. The dimensionality of the velocities array is there only because of the grid.

3.2.2 Visualizing resonance (S.3, S.4)

The visual aspects of the program are handled in graphics.py, in the *Arrowimg* class and the *GUI* class. *Arrowimg* handles the compass motion, while *GUI* connects user input to the physics simulation and displays the result of the physics calculation by using *Arrowimg*.

Setting up, clearing and drawing the screen requires a lot of code. `graphics.py` is the file with the most lines. The size does not necessarily make it the most important, since most of this code is standard Pygame

syntax and screen coordinates for the images.

The two requirements whose implementation depends on graphics.py also are magnetic fields being adjustable during simulation (S.3), and resonance in motion being visible (S.4).

The `getkeys` method of *GUI* changes the values of magnetic fields if the user clicks on buttons on screen, or uses the keyboard shortcuts.

Rendering the resonating motion of compasses is a task for the entire program, but for the most relevant code, the *Arrowimg* class is called in `model.py` as discussed earlier. *GUI*'s `Constructor` creates *imglist*, a list of 360 rotated arrow images with which the *Arrowimg* object is initialised. The images are rotated before the program fully starts to save computation time when the program is running. The *GUI*'s *drawarrows* method contains code to draw ("blit") every arrow in the list on the screen.

graphics.py

The purpose of graphics.py is to show the runtime screen. Most of the code is inside *GUI*, but graphics.py also contains:

- *clock* function:
  Pygame measures in milliseconds, this function converts to seconds. A very simple and short procedure, but otherwise this conversion would have to be done for each physics equation. A function named *clock* and time in seconds will both make the program more readable.

- *Arrowimg* class:
  Most of the arrow graphics. This class handles the image and the *Rect* from Pygame. Needs the *imglist* so it can quickly update an arrow to the correct angle; loading a premade image is fast, unlike creating a rotated image from the original.
  - `update` method: takes in any float theta and loads the nearest integer angle, as *imglist* contains 360 images.

The *GUI* class is defined which can set up, clear, update and draw the screen. The constructor of *GUI* initialises the following members:

- The Pygame graphics library

- The screen (set-up)
- 360 pre-rotated images of which the closest to the calculated angle is chosen
- Bitmaps of display panel with indicators and buttons for magnetic field strength
- Switches (toggle buttons) for turning the magnetic fields on/off
- Positions of display panel and button
- Font objects

The functions/methods of this class are:

- `textpanel`(self,b0mag,b1mag,b1freq,b0on,b1on):
  Displays the text panel, the keyboard controls on the left, the B_0/B_1 on/off buttons and the RESET button on the right.
- `drawarrows`(self,arrowlist,m,n):
  Goes through the nested list (to represent the two dimensions) with the *Arrowimg* objects and displays each one at the correct angle and position with the nested list slicing. It puts the images on the screen and blits the image and *Rect*.
- `clearscreen`(self):
  Pygame doesn't get rid of the previous frame by itself, so this function is needed to clear the screen. Done here by simply filling the screen with black, so every screen frame starts off as a clean black rectangle.
- `updatescreen`(self):
  The screen is drawn in a buffer. This function moves the pointer of the visual screen to this buffer (flips the screen) when the frame is finished. The speed of this action avoids a flickering screen during the updating of the screen frame.
- `getkeys`(self):
  Regulates every functionality both with the mouse as well as with the optional key control. It returns one or more commands (as keys can be pressed simultaneously) of the following:

- 'RIGHT', 'LEFT' = Increase, decrease B_0 field
- 'UP', 'DOWN' = Increase, decrease B_1 field
- 'PLUS','MINUS' = Increase, decrease frequency of B_1 field
- 'V' = Set speeds to zero
- 'B' = Show for debugging actual precise float values
- 'ESC' = Quit program
- 'RESET' = Reset speed to zero and angle to non-zero
- 'B0' = Switch BO field on or off
- 'B1' = Switch B1 field on or off

Note that the last three do not correspond to keys, they are just custom names for functionalities only accessible via the clickable buttons on screen.

plotter.py

An additional module to help show resonance is plotter.py. This module generates plots to provide an overview of the simulation when it is finished.

The *Plotter* class maintains tables with a specific plotter (minimum) time step to avoid occupying too much memory when the time step is small.

It contains three functions/methods next to the constructor:

- `Constructor`(tsims, dtplot):
  Stores the time, plotter time step and creates the empty tables.
- tableupdate(tsim, b0mag, b1mag, theta, b1freq):
  Appends the current values of simulated time and model state to the tables if enough time has passed (more than *dtplot*, default 0.1 second).
- `tabletreset`(tsim):
  Reset events are drawn as red lines. Whenever the reset button was pressed, this function was called to store the timestamp in a separate list.

- `plotdata()`:
  Shows the final results in three rows of formatted subplots (number of subplots can be adjusted if necessary). The plots help in finding resonance, and can even show the different resonance frequencies for different arrows in case a *tissuemask* was used.

3.2.3 Integration of model and graphics into one program

As seen in Figure 3.1, `sim.py` contains the main function. `RunSim.py` calls this main function with the contents of `settings.py` as the arguments. Calling the main function with arguments in a Python console is the manual way of controlling the initial settings. It can also be called without arguments, using the default values in the definition of main. This program controls the simulation. It maintains the time, contains the simulation loop with the updating of the model and screen, as well as getting the user keyboard or mouse inputs, all by using the previously explained modules as shown in Figure 3.1.

sim.py

Before the loop the following objects are initialised:

- Parameters of the graphical user interface (GUI)
- The GUI itself
- Model using the *Model* class from model.py
- The clock
- The plotter is prepared

During the loop the following actions take place every frame:

- Get system time, and with a time step (actual or limited) update the simulation time
- Update plotter tables here so the initial values are also in the plot

- Update simulated time *tsim*
- When there is a time step detected, to show change, update:
  - Model
  - Screen
- The *getkeys* method from *GUI* is used to get controls. Mouse actions are also translated into keys/commands, with names. Next, depending on which keys/commands have been found, adjustments are made to e.g. the magnetic fields B_0 and B_1. The reset command resets the speed to zero and the angle to a non-zero value, which is useful when looking for the resonance frequency.
- The *ESC* command & key, sets the flag *running* to False which means the loop will stop repeating.

After the loop is terminated, two actions take place:

- The GUI screen is closed by deleting the (only) instance of the *GUI* class.
- Plotter screen is started, showing the time histories of the simulation, which helps in finding the right parameters for resonance.

3.2.4 Interpretation of units in MRI2D

Requirement S.4 in Table 2.1 states that resonance has to be visible. This means that the resonance frequency should be well below 25 Hz we choose a minimum of around 1 Hz and a maximum of around 10 Hz; a slower oscillation is easier to see, but a higher frequency leads to a resonance steady state faster. This unit for frequency is a given because time in the simulation is equivalent to real time.

MRI2D needs the values for the magnetic field magnitudes to be in the thousands for the resonance frequency to be around 1 Hz. This power of ten raises some questions about the unit for the magnetic field, as MRI scanners have B_0 fields in the order of single digits expressed in Tesla.

The magnetic moment of the nuclear spin does not have any meaning in this simulation, because a two-dimensional harmonic oscillator (equiv-

alent to a compass) is used. Strictly following the compass analogy, a reasonable mass for compasses could be estimated, and added to the force method. There is no guarantee this would lead to a satisfying scale of numbers for MRI education, and was not tried due to time constraints and tester availability as all preset software would have to be changed and experiments retested.

Therefore the magnetic field units in MRI2D are 10^{-4} T to match real B_0 values in MRI scanners.

3.3 Implementation of requirements regarding ease of use

The educational requirements found in chapter 2 are almost automatically fulfilled by virtue of it being free software. However, further clarification or quantization of these requirements is necessary before we can test if they are fulfilled.

Safe and robust (E.1)

MRI2D is safe because it is Open Source and therefore all code can be viewed by users. Safety seems easily demonstrated at first, as a computer program is unlikely to bring physical harm to people. Software however has a risk of malware, and it can be difficult to know which programs to trust. If a user has security concerns, they can read the commented code of MRI2D to find out exactly what the program does and does not execute.

Robustness comes at the cost of less flexibility. For that reason, different starting methods are provided for the teacher to choose how robust the program needs to be for their students, or how much freedom their students can have without making critical mistakes. With the most robust option, which is to execute RunSim.py to start the program with default settings, MRI2D will always start and thus is robust.

Available for education (E.2)

MRI2D needs to be able to be run on computers already owned by educational institutes or students to truly be available nearby and without waiting list. Computers and operating systems differ per institute but MRI2D being programmed in Python means it should run on Windows, Linux and macOS without any extra effort.

Affordable (E.3)

MRI2D is free of charge. Institutions have different budgets, but there is no need to research a reasonable price for them as all similar software discussed in Background is free. Only physical setups or software for highly specialised professionals (pertaining to scan image interpretation) had a cost. MRI2D is neither of those, so it needs to be free like other software at the same educational level.

Easily distributed (E.4)

An installation guide needs to be created for ease of distribution. Sending the program to multiple computers is trivial, the question is whether students can use the program. The installation guide provided has to allow a majority of students to set up the software by themselves. The guide will be tested by individual users instead of a classroom, but with minimal supervision the situations should not be too different.

Flexible in use (E.5)

Three quantities pertaining to the magnetic field allow adjustments during runtime as the simulation requires (S.2); magnitude for constant magnetic field B_0, magnitude or amplitude of driving magnetic field B_1 and frequency of the driving magnetic field B_1.

Other quantities have to be set before starting the program, preferably as an argument when called or as a number in the text-like file *settings.py*. Otherwise changing a number in the code itself is required.

Tissue shapes need to be able to be made without programming, as they decide how many "imaging" exercises can be done. At least 3 preset tissue files need to be provided with MRI2D so the teacher can use exercises without spending time on preparing the program itself.

This specification of E.5 should allow for a wide variety of possible exercises.

3.4 Exercise development

The exercises were developed in accordance with the findings of section 2.4, concerning Self-Determination Theory (SDT) and the value of agency in exercises for students [33].

Three exercises of different levels are created to show the scope of MRI2D and to give an example of an MRI2D lesson structure.

A teacher can first perform a qualitative demonstration, before students carry out quantitative measurements. Without a tissue, every arrow represents one nuclear spin and each nuclear spin oscillates in the same way if they experience identical magnetic fields, making these two exercises useful for general NMR education, not just MRI.

Lastly, tissue differences are studied to lead into the subject of MRI. A phantom, a mask to simulate tissue, changes the magnetic forces spins experience depending on their location. When simulating differences in the resonance properties of different locations in the phantom, the principle of imaging based on resonance can be explored. The qualitative aspect of this exercise can also be done by students, as finding the shape of the tissue should be an easy task after having completed the previously recommended measurements. An optional qualitative measurement on tissue will also be provided.

Table 3.2: *Attributes of MRI2D example exercises. Exercise numbers indicate order of complexity, starting from simplest. The first exercises are done with roughly identical spins, 3a and 3b feature a tissue-mimicking phantom. Qualitative explorations are fit for demonstrations, while quantitative exercises are best suited to students carrying out the measurements themselves. Descriptions are very brief to keep the table readable. 3b is optional, it is mentioned mostly for students who finish the mandatory exercises early.*

#	Phantom	Activity	Description
1	No mask	Demonstration	Notice resonance
2	No mask	Measurements	Find correlation resonance frequency and B_0
3a	Tissue	Demonstration	Imaging, find shape of tissue
3b	Tissue	Measurements	Find resonance frequencies of different tissues

This way the exercises are ordered from simplest to most complex. Quick demonstrations can easily be used for qualitative observations if time is the largest constraint. Otherwise, if there is more time a more scientific quantitative exercise can be performed where the students gather data points and record them to gain understanding of the model.

When students have found the ideal values for resonance without tissue, they can accurately image tissues as the spins which do not resonate at the same values. With the experience of the quantitative exercise the students should be able to find the resonance frequency of the deviating spins in tissue. The last part, finding the resonance frequency of the tissue, is not required. The students can know the shape of the tissue by the non-resonating spins that will stand out already. This optional task makes it a good final exercise with its flexibility in duration per student to reduce the amount of students finishing early and possibly distracting others.

The people testing the exercises were the author, two experts and a layperson individually; not in a classroom setting. The latest version of Python in which MRI2D was tested is 3.9.13.

Chapter 4

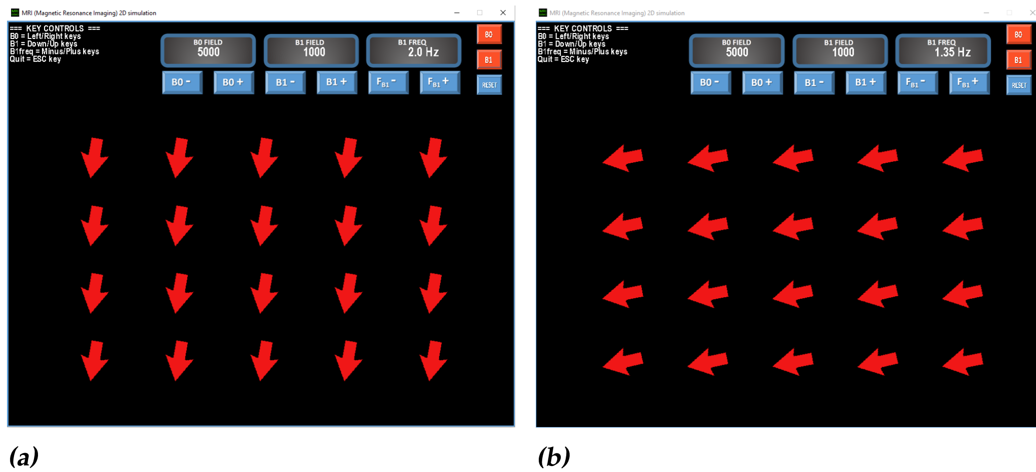
Results

Here we describe the program MRI2D created for this thesis, the test results and the ways in which it can be used in education. It was tested successfully by different users. The reader can download MRI2D for free [36].

Table 4.1: *All described design requirements categorised in the order of development and testing. The rightmost column lists numbers for referencing. Copied from section 2.5.*

Category	Requirement	#
Simulation	Time domain	S.1
Simulation	Multiple nuclei	S.2
Simulation	Adjustable magnetic fields	S.3
Simulation	Visible resonance	S.4
Simulation	Tissue differences	S.5
Educational	Safe and robust	E.1
Educational	Available for education	E.2
Educational	Affordable	E.3
Educational	Easily distributed	E.4
Educational	Flexible in use	E.5

Test results for each of the requirements found in chapter 2, listed in Table 4.1, will be given. The simulation requirements are explicitly shown in video format (see Appendix B), but screenshots showing similar situations are included in this text. The educational requirements are more difficult to show in such a manner, and their tests are therefore described. Example exercises are also provided to show MRI2D can be used in education.



(a) ***(b)***

Figure 4.1: *Screenshots at maximum amplitude: (a) for non-resonant spins and (b) at the resonance frequency for the same magnetic field magnitudes. 1.35 Hz is the resonance frequency for these magnetic fields, and 2.0 Hz is far from it. Shown by the larger amplitude at resonance (b). Not pictured: when only the static magnetic field B_0 is enabled, the spins point straight down.*

4.1 Fulfilment of simulation requirements

4.1.1 Resonance without tissue (S.1, S.2, S.3, S.4)

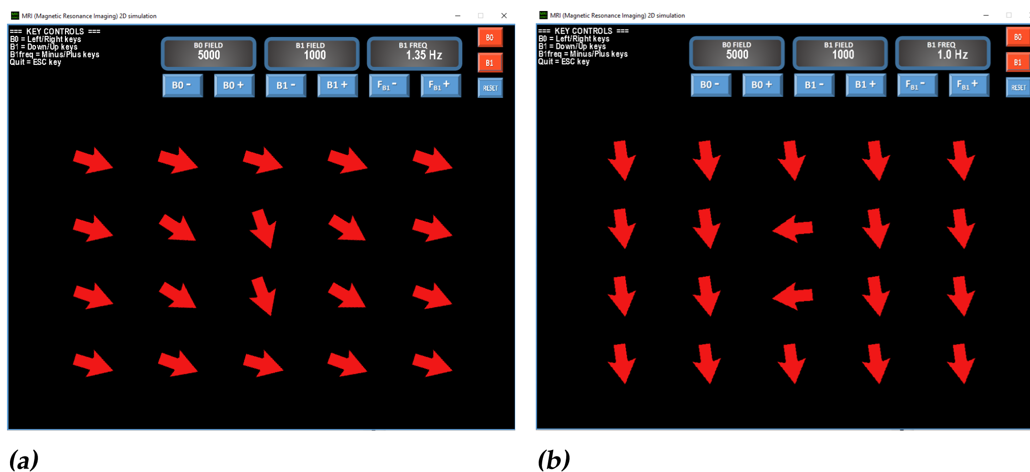
As shown in Figure 4.1 and the video in Appendix B, the magnetic fields B_0 and B_1 lead to torque on the nuclear spins. The resulting motion conforms to the expected driven harmonic oscillator with damping discussed in section 2.1. The *modelled forces leading to realistic motion in the time domain* means the first requirement (S.1) is satisfied.

Though the video in Appendix B shows it better than Figure 4.1, spins in different lattice locations experience slightly different magnetic fields due to gradients. Each nuclear spin also has small random noise in their damping factor, maximally 1.5% more or less than the set damping factor. *MRI2D simulates multiple nuclear spins each experiencing unique forces due to their lattice location* (S.2).

Besides the keyboard controls shown in the top-left of the MRI2D window (Figure 4.1), buttons of a Graphical User Interface (GUI) are featured in the remaining space along the top. Digital dials show the current values for B_0 magnitude, B_1 magnitude and B_1 frequency. Below each physical quantity dial there are two buttons to *decrease or increase the value while the simulation is running* (S.3). Buttons to turn the magnetic fields on or off are

located at the top-right, along with a RESET button which sets the speed of all spins to 0, and their displacement angle to an arbitrary 40 degrees to see if they resonate to higher amplitudes, or get damped to lower amplitudes.

Comparing both images in Figure 4.1, the nuclear spins in Figure 4.1a have a smaller displacement angle at their maximum amplitude than the spins in Figure 4.1b. As Figure 4.3 further on in this chapter shows, the expected resonance frequency for these magnetic field values is close to 1.35 Hz (the driving frequency in the second image). So what Figure 4.1 shows, is that *nuclear spins display a higher amplitude at their resonance frequency* (S.4).



(a) ***(b)***

Figure 4.2: *Screenshots of MRI2D modelling tissue differences. The middle two arrows are a different tissue, with a rotational inertia factor of 0.60 as opposed to 1.00. On either side of them are four arrows that only slightly differ from the normal spins, as their factor is 0.90. The entire outside border is still the same as in Figure 4.1. (a) The different central tissue does not resonate with the other spins. (b) At the resonance frequency of the central tissue we see the inverted situation in which the central tissue is resonating, while now the other spins are not.*

4.1.2 Resonance with tissue (S.5)

When a *tissuemask* is applied, nuclear spins experience the magnetic fields differently depending on the factor specified for the tissue at their location. As Figure 4.2 shows, *these tissue differences cause the specified nuclear spins to have a different resonance frequency* (S.5).

4.2 Fulfilment of educational requirements

4.2.1 Safe and robust (E.1)

Since MRI2D is a program, not a piece of equipment, any possible safety issues would derive from malware like computer viruses. The source code of MRI2D is viewable in Appendix C, online before downloading and can be inspected locally after downloading and before running the program. This open nature makes the code transparent and avoids the risk of hidden malware that could be contained in an executable program, a file type where the code cannot easily be viewed.

By robustness we mean that the code does not crash and the program starts and runs without glitches after finishing installment. Robustness can come at the cost of freedom or user control, which is why MRI2D offers four different levels of starting methods. The most robust starting method still offers the flexibility required in chapter 3, as that pertains to control over three quantities during runtime. Other levels of starting methods can offer greater control at the risk of user errors. The novice user should start MRI2D at level 1.

Level 1: Start with default values
This level is set by running *sim.py* or *RunSim.py* in Python.
• No additional options beyond runtime control over B_0, B_1 and frequency.
• No possible user error, because there is no custom input at this level.

Level 2: Selecting different presets
For this level a teacher needs to prepare scripts to edit *settings.py*, to give the students the choice with which *damping* and *tissue file* to start, in addition to starting values for B_0, B_1 and frequency. The students run these scripts, possibly via Powerpoint buttons. In summary:
• Limited presets provided for tissue files, damping, B_0, B_1 and frequency.
• No possible user error.

Level 3: Edit starting parameters in a script
Students can write any starting values in code in two different ways, either by manually editing *settings.py* or calling *sim.py* main function with arguments from a Python console, best done when students are familiar with an Integrated Development Environment (IDE). In summary:
• Any values available for tissue files, damping, B_0, B_1 and frequency.
• Only syntax errors possible. The program checks for syntax errors before starting to prevent faulty simulations.

Level 4: Edit the program itself to add options
This can be done by editing *sim.py*. Changing any numbers should not cause problems. Code in the other scripts could be edited, but they govern physics, graphics and the plots. Changes in files other than *sim.py* lead to a different simulation, instead of the same simulation with different physical constants, and are therefore discouraged. In summary:
• Full control over any values in MRI2D, including spin lattice dimensions, which require a matching tissue file. For example, the provided tissue files assume a 5 x 4 lattice.
• Any kind of Python error becomes possible, and such errors can be hard to detect.

4.2.2 Available (E.2)

MRI2D has been tested on different Windows computers and across versions, on Windows 10 and 11. No tests have taken place on other platforms, but all of the used Python modules are compatible with Linux and macOS.

4.2.3 Affordable (E.3)

The MRI2D software is free, like most software used for educational purposes as described in subsection 2.3.2. Educational institutions already in possession of computers can use MRI2D without incurring any additional costs.

4.2.4 Easily distributed (E.4)

We found that new users were able to install MRI2D with minimal guidance when using the user guide, found in Appendix A. The guide also contains comments on any peculiarities encountered during testing, but of course we strove to prevent them as much as possible.

4.2.5 Flexible in use (E.5)

The magnitude of both B_0 and B_1, in addition to the frequency of B_1 can all be edited freely during runtime. The degree of flexibility required is therefore met exactly, and as described in the definition of requirement E.1

in subsection 4.2.1, more quantities can be changed before starting MRI2D. To show that MRI2D really is flexible, we will discuss a series of exercises in the following section to prove this requirement has been met.

4.3 Possible demonstrations using MRI2D

A number of example exercises is discussed to illustrate how MRI2D could be used in education.

4.3.1 Exercise 1: Demonstration of magnetic resonance

The fastest and most simple exercise is a qualitative demonstration of magnetic resonance. Students can get an intuitive understanding of the simulation by observing how the B_0 magnitude, the B_1 magnitude and the driving frequency can lead to resonance. The students can do this themselves, but it is faster when a teacher demonstrates this exercise.

Any starting method will suffice here: simply running the program with default settings is enough, but buttons with hyperlinks (Ctrl+K) in Powerpoint can be used to quickly show different preset situations. An example interactive slideshow has been provided, which runs a set of programs that change the *settings.py* and similarly a button with a hyperlink to start the simulation.

The teacher can show what resonance is using the example settings and they can explain the buttons to students.

4.3.2 Exercise 2: Determine resonance frequency and magnetic field correlation

The student can determine the relation between B_0 and the resonance frequency without any tissues present. As explained in section 2.1, the resonance frequency can be approximated by linearising the equation to a harmonic oscillator.

$$f_{reso} = C \cdot {B_0}^p \tag{4.1}$$

But in reality, for larger angles θ the sine and cosine approximations might not hold, in which case students will disprove this hypothesis. First exploring this relation experimentally will make the theory easier to understand and remember. An example lesson is described in the following paragraph.

The assignment for the lesson would be to determine the resonance frequency per B_0 magnitude for a series of B_0 magnitudes. It is good experimental practice to record all quantities for reproducibility, even though B_1 magnitude does not show up in the simplified theoretical solutions. See Table 4.2 for an example table provided to students.

Table 4.2: *Example table to fill with measurements for exercise 2. For regression analysis, the B_0 magnitude will be the independent variable. The B_1 frequency at which the largest amplitude is recorded is the resonance frequency, and will be the dependent variable. The B_1 magnitude is kept constant, but should be recorded at least once for repeatability. Its value here is chosen to be as high as possible without making measurements at B_0 magnitude $2000 \cdot 10^{-4}$ T impossible, because determining resonance frequency becomes hard when B_1 magnitude is larger than half of the B_0 magnitude.*

B_0 magnitude (10^{-4} T)	B_1 magnitude (10^{-4} T)	B_1 frequency (Hz)
2000	1000	0.80
5000	1000	....
10000	1000	....
20000	1000	....
50000	1000	....
100000	1000	....

In order to find the functional relationship of B_0 and resonance frequency, a curve fitting step will also be necessary.

Any software can be used for fitting, we suggest two of the many options: Python module or WolframAlpha. Python with `scikit-learn` makes sense, if teaching Python was also a goal. Otherwise WolframAlpha is the easiest, since it is available online to everyone without the need to install any software. Simply query "power fit" with the collected points (B_0, frequency).

If fitting is too complicated for the students, the teacher could collect the students' data points into one collection, and use that collection when fitting the data. In that case one could encourage the students to measure at different B_0 magnitudes. If different groups of students have measured at different B_0 magnitudes, then their measurements are recognisable in the final graph, which could enhance the students' feeling of contribution.

4.3.3 Example answer to exercise 2

Table 4.3: *Example answers of exercise 2. Frequency was measured in steps of 0.05 Hz initially.*

B_0 magnitude (10^{-4} T)	B_1 magnitude (10^{-4} T)	B_1 frequency (Hz)
2000	1000	0.80
5000	1000	1.35
10000	1000	2.10
20000	1000	2.90
10000	1000	4.60
12000	1000	6.50

Now that we have the measurements in Table 4.3, we can put them into our software of choice for curve fitting. Equation 4.2 gives the expected power relation, so we try a power fit.

Figure 4.3 shows that the program behaviour is as expected. The regression fit gave the following parameters when rounded off to significant decimals:

$$f_{reso} = 2.0 \cdot {B_0}^{0.5} \tag{4.2}$$

The coefficient 2.0 relates to the magnetic moment, but in the MRI2D simulation model, this is not a parameter that has a quantitative meaning. Educationally valuable is the power found in the fit: 0.5. This value is as expected because the resonance frequency scales with the square root of the B_0 magnitude as shown in Equation 2.8. Linear and parabola fits are not shown as they do not match the data points.

Important to note with fitting is the number of free parameters versus the number of data points. Standard second degree polynomial fitting will have three parameters, making it easy to minimise residual errors. This error reduction is artificial in the case of few data points. Fortunately, even when an inverted parabola appears to be the best fit, it can be easily shown that the shape implies unexpected behaviour at high B_0 magnitudes. Beyond the maximum, the resonance frequency would reduce as the B_0 magnitude increases, and even go negative if it followed a parabola in relation to B_0, which is nonsensical.

Therefore the better model fits to compare the data to would be linear and power, with the latter being the expected relation as a result of the harmonic oscillator physics. Both only have two fitting parameters, making

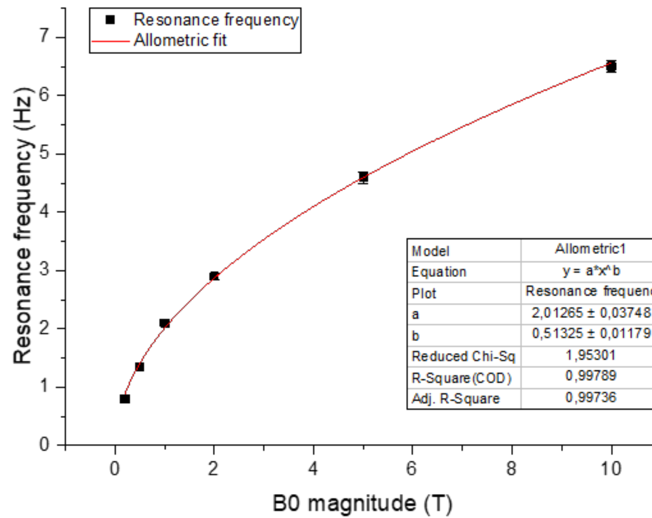


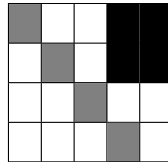
Figure 4.3: *Example answers to exercise 2. Hypothesis was $b = 0,5$. Origin 2016 was used to obtain this curve fit and plot from the data in Table 4.3. Important to note that the choice of decimal point for B_0 magnitude can multiply or divide parameter a by 10 to the power b: for our expected power that becomes a factor of $\sqrt{10} \approx 3$.*

their fit equally justified from a statistical standpoint for the same number of data points, unlike the second degree polynomial. The consequences of a linear relation would be less problematic than a second degree polynomial: the only issue arises around B_0 magnitudes near zero, but practically, it would be hard to see the impact of B_0 in that range without lowering the magnitude B_1 as well.

4.3.4 Exercise 3: Performing an MRI scan with MRI2D

In this exercise, students will be creating a resonance image. Students are provided one or more tissue masks, either the default set or custom ones made by the teacher. By applying what they learned from exercise 2, students are expected to bring into resonance the arrows that are unchanged by the tissue mask. By looking at the arrows not in resonance, they can find the tissue shape.

Table 4.4: Example of student answer for exercise 3a. Cells are coloured according to the resonance frequencies of the nuclear spins at the respective positions.



For exercise 3b, students can find the resonance frequencies of the different tissues. Use a similar method to exercise 2, but make sure to pay attention only to the spins of one tissue.

4.3.5 Teacher only: Defining tissues

As a teacher you can make your own tissue files (see below for its use). They are located in the `MRI2D/data/` subfolder. You can edit these files manually or use the tissue tool to aid in editing them. The files are simple CSV (comma separated values) files which can be edited with Notepad, Excel or any text editor:

Table 4.5: The contents of an example tissue file, where the tissue is a diagonal line from the top-right corner to the centre of the bottom row.

1.00	1.00	1.00	1.00	0.80
1.00	1.00	1.00	0.80	1.00
1.00	1.00	0.80	1.00	1.00
1.00	0.80	1.00	1.00	1.00

The force an arrow experiences is multiplied by the factor at the corresponding position (meaning a factor of one keeps the behaviour the same, as without a tissue file).

The file extension is set to `.tis` to hide the format for the students. Tissues can be detected by watching which arrows behave differently. These observations can be confirmed in the analysis screen. If there is a problem with the tissue files, for example when they do no match the lattice dimensions, running `resettings.py` will restore default values to solve the problem.

4.3.6 Exercises tested

The exercises were successfully completed by four testers: the author, two experts and a layperson. Exercise 2 takes 30 minutes when performed in a

competent and precise manner: understanding MRI2D controls and measuring in steps of 0.1 Hz around the estimated resonance frequency.

Appendix A contains instructions for downloading and installing MRI2D and its prerequisite software. The exercises are described in section A.4 in a format that can be given to students directly.

Chapter 5

Discussion

In this thesis, a program was developed, MRI2D, that simulates magnetic resonance with compass analogues to help in MRI education.

It was tested individually by users of different levels of expertise with minimal instruction and performed to the satisfaction of all users. The program closes the gap that the existing educational tools to teach MRI leave as described in subsection 2.3.2. It does not require instruments, and does not use images from MRI scans, but instead looks at the motion of nuclear spins in the time domain.

5.1 Simplification of spin precession frequency for modelling

Our model does not fully correspond to the physics of reality. It simulates a simpler situation by modelling the three-dimensional precession as an oscillation in two dimensions. MRI2D explains the most important aspects of MRI. Logically, other physical phenomena could have been chosen here, like how the three-dimensional rotation of a spin leads to it getting flipped. Many of those choices are valid, as MRI is a vast enough subject that even with the extensive MRI educational software out there, not everything is covered at levels well below expert. The two aspects of simplification that we will discuss here are the consequences of the choice of dimension and frequency.

5.1.1 Dimensions of precession in MRI2D

The real situation involves three dimensions, as spins precess around the direction of the static magnetic field B_0. This three-dimensional precession of spins can only be seen well for a maximum of around four spins, and that is not enough to show imaging. So instead of three-dimensional motion, we have made a two-dimensional model to make differences in resonance frequency visible for a five by four grid of nuclear spins on a screen. Reducing by one dimension was necessary to fulfil these requirements for multiple spins (S.2) and visibility (S.4). Simulating the precession in this way also has influence on the absolute numbers of resonance frequency and magnitudes of the two magnetic fields B_0 and B_1. Frequencies of around 1 Hz and units of 10^{-4} T were chosen to end up with magnetic fields from 0.1 T to 10 T as described in subsection 3.2.4.

An alternative approach is using colour to represent frequency, effectively adding a dimension. A large grid or lattice of spins should not make it much harder to see the colours, unlike with a three-dimensional motion. This way the model retains the physics of three dimensions, without having to visually show motions in three dimensions on a two-dimensional screen. The colour approach was not the first choice here as it is not as clearly visible as two dimensional movement, so it would go against what we wanted (requirement S.4). It was not explored in practice because of time constraints, but it could be an interesting addition for the future.

5.1.2 Complications with modelling precession frequency

Our approach to frequency was to handle spins as if they were compasses in a magnetic field. This was a simple way to avoid modelling quantum mechanical behaviour and is accessible for students who did not (yet) have quantum mechanics. Also, compasses had been used in MRI education before, using physical compasses in conjunction with magnets and coils in physical demonstrations described in subsection 2.2.2. Obviously, in MRI2D many more situations and quantities could be explored.

However, this choice also brought some complications with it. The resonance frequency of a harmonic oscillator scales with the square root of the static magnetic field. That power relation is unlike the Larmor frequency of the precession in reality, which scales linearly with the static magnetic field B_0. There are two possible solutions to this problem: projecting the magnetization vector $\vec{\mu}$ in a plane (a) or use a model containing unrealistic physics (b) described below.

a) Projection is possibly the simplest fix. By keeping physics true to the three-dimensional situation, the relation is conserved. And to make it easily visible on a screen, the motion is simply projected to two dimensions. Ignoring the third dimension can make the oscillation look unnatural, but it should not make it any harder to distinguish frequency.

b) The other option of uncoupling the displacement angle of the compass from frequency is more complicated, and would have to use a lot of tricks unique to software models. What we mean by this is that the oscillating motion of the displacement angle itself does not have to follow the frequency of the driving magnetic field but can instead always be oscillating at the resonance frequency of that particular spin. Resonance would still be found by writing down the physical quantities at which maximum amplitude occurred, which would be when the driving frequency matches the resonance frequency of spins. Whether such fake physics is harder or easier to program is hard to guess, but it would be a lot easier to image tissues in such a model than in MRI2D.

5.2 Possible code environments

We have chosen for a standard project structure for a Python program consisting of multiple `.py` files and folders for images. This structure was chosen for safety from malware, compatibility with many operating systems, costs and flexibility (E.1, E.3 and E.5 from Table 2.1). Although using an exectuable file would avoid user errors better (E.1 and E.4), the other requirements would not be satisfied nearly as well. Here we will mention some considerations for alternative Object-Oriented Programming (OOP) structures, code environments or distribution packages.

5.2.1 Other approaches to Object-Oriented Programming in Python

MRI2D uses Classes and functions as explained in subsection 3.1.3 regarding Object-Oriented Programming (OOP). The most notable aspect there was not following the Sprite Class examples used in Pygame documentation [19]. This was done because Sprites would make the code run more slowly. Sprite Classes are not the only other option for organising code, and if NumPy array mathematics are kept, other OOP structures should

not be noticeably slower or faster than MRI2D currently is. Therefore the choices should be decided by ease of developing, bug fixing and updating or editing. When done by one developer these choices are highly personal, so we cannot offer any further guidance for this other than the expectation that a different code structure may be better if design requirements change.

5.2.2 Possible structure improvements when using other applications for Python

Advanced Python users can consider editing MRI2D to better use interface applications like Jupyter Notebook. The current MRI2D structure is suitable for any Python application, but that means that it could be improved if a specialised application is the only one used. Jupyter Notebook can divide one program into multiple cells, which can be selectively run while still sharing variables like a normal program. This is mentioned here, as `sim.py` has a very clear split between initialisation and the simulation loop, which could be put in different cells. If Pygame code is changed to work well with the Jupyter Notebook environment, this could mean loading the images only happens once per lesson, instead of each time the program is started. Right now there seems to be no need for this as the loading time is less than 10 seconds even in non-ideal conditions, but if changes happen that make the loading time a large enough obstacle, Jupyter Notebook should be the first option to explore.

5.2.3 Choice of download format

Other than safety, an important reason for MRI2D to be distributed as the raw code files is that Python can be installed on many platforms (Windows, Linux, macOS and more). However, it does make installing MRI2D a multiple step process. An executable file would be simpler in his aspect, but it could have safety issues (malware) and would be platform specific. One way around this problem would be a teacher making such an executable file for the platform used by the school or university computers. The downsides of executable files like malware concerns are alleviated when a teacher created the file. The final option would be a web application. This is best for larger or paid projects, as web applications need maintenance and updates to stay safe and working, as internet browsers change. They also need more development time to be compatible with multiple platforms. This additional development time and contin-

ued maintenance for web applications is why this format was not a good choice for MRI2D. Pygame on the other hand is not expected to update in a way that breaks the program. If MRI2D becomes incompatible with a newer version of Python, an older version of Python will have to be used until MRI2D is updated.

5.3 Education research

MRI2D exercises have been tested by individuals, but not in a classroom setting. MRI2D was designed to complement other educational software as described in subsection 2.3.2: to teach aspects of MRI physics that have not yet been simulated, but also combining good implementation ideas of multiple tools. It is expected that MRI2D is beneficial to education, as existing software covering different subjects in MRI was found to be a helpful addition to traditional education [26]. Nevertheless, this area is not explored in this thesis, as organising proper classroom tests was outside the scope of a thesis describing the development of the software. The simulation software is ready to be tested in educational settings.

5.3.1 Suggestions for testing MRI2D education effectiveness

To find out if MRI2D adds to education, and at which level it is best used, tests will have to be conducted. Ideally in situations most closely resembling classroom circumstances, most importantly that means: one teacher for 30 high school students, or a lecturer with teaching assistants for a large group of bachelor students. Whether MRI2D is best as the students' first contact with MRI principles, or as a follow-up to an introductory lesson can also be studied.

Methodologies for this kind of experiment have already been published, an example could be the already mentioned research by Fernández et al. [26]. We do not provide a test form for checking students' information retention. Since the specific learning goals of courses can vary, a test form should be made to fit the courses at the institution conducting the education experiment.

5.3.2 Possible exercises to be developed

The provided example exercises were merely created to give an idea of what a lesson with MRI2D could look like. As mentioned, we designed

the program with a lot of flexibility in mind, so more exercises than have been discussed are certainly possible. Here we will discuss ideas for two other exercises, one more for exploring general resonance principles with trigonometric functions, and the other more focused on MRI simulation.

The first idea that could be turned into an exercise has to do with the method of measuring the resonance frequency. The example exercises asked students to try different driving frequencies, and see which produced the highest steady amplitude. However, information about the resonance frequency can still be gathered when the driving frequency is mismatched. With multiple oscillators in a system, a beat wave will emerge. By knowing the driving frequency and measuring the frequency of the beat wave, the resonance frequency can be calculated. How to measure the beat wave frequency with enough precision has not been figured out yet. Therefore, this exercise idea is mentioned here as an inspiration to future developers.

The other exercise idea is about the influence of B_1 magnitude on resonance frequency. B_1 does not appear in the resonance frequency obtained from the near-zero approximation of the harmonic oscillator in section 2.1. However, that is an approximation, and we have found that the B_1 magnitude can have an effect on the resonance frequency in MRI2D. The exact correlation has not been explored, as it is unsure if the non-approximated mathematics hold up in a discrete environment: in software, time happens in steps instead of continuously. Additionally, if B_0 is too small compared to B_1, it is hard to find the resonance frequency, as there will be a range of driving frequencies at which the maximum amplitude of the spins will exceed 180 degrees, and seemingly chaotic behaviour occurs. The edges of this range can be identified, but where in this range the resonance frequency lies depends on the correlation that the exercise is looking for. Maybe this is not a big problem, as the relation between the B_0 magnitude and the resonance frequency is known and can easily be found by performing exercise 2. The effect of the damping factor can also be explored in this context, but is best done separately as performing an experiment with three changing quantities can become hard to follow for students and is in general not good scientific practice.

Finding the relation between resonance frequency and B_0 magnitude was thought to be sufficiently interesting and complicated for an MRI2D exercise at this stage, but if the program was changed or other software was developed with the choices mentioned at the end of subsection 5.1.2, resonance frequency measurements would be a lot easier. That could make the provided example exercises less interesting, but it would be better for exploring the effect of the B_1 magnitude.

The examples described here are still the same format as exercise 2 in subsection 4.3.2, i.e. measuring the resonance frequency. MRI2D could be used for other concepts, but because it was made for this kind of exercise, other software (see subsection 2.3.2) might complement other exercises better.

MRI2D is suitable for exercises showing magnetic resonance in an imaging context by showing the motion of spins. This can give students a better understanding of the mechanics behind signals in MRI.

Chapter 6

Conclusion

A real-time two-dimensional simulation of the basic quantum mechanics behind MRI was developed, with adjustable magnetic fields and the capability to model tissues.

Made in Python using the Pygame module for rendering graphics, the program shows magnetic resonance in the time domain utilizing the angular movement of arrows. These are equivalent to compasses as used in demonstrations of MRI, and thus analogous to spins in the quantum world. This representation of spin motion can give a deeper understanding even to students unfamiliar with energy levels in quantum mechanics.

The simulation is free to download as open-source software [36]. It is safe, robust and available to anyone with a device compatible with Python. Due to the nature of software, providing an entire class of students with their own digital environment is as easy as providing it to only one individual.

Three example exercises have been provided for use in education and to illustrate a lesson template that can be adjusted for different audiences by focusing on the basic quantum mechanics behind MRI. Already available software focuses on the data analysis side of MRI, so MRI2D is made to complement them and cover part of the gap in MRI education.

Bibliography

[1] N. B. Smith and A. Webb, *Introduction to Medical Imaging: Physics, Engineering and Clinical Applications*, Cambridge Texts in Biomedical Engineering, Cambridge University Press, 2010.

[2] S. Weinberger, *Mind Games: a community of people who believe the government is beaming voices into their minds*, The Washington Post (January 16, 2007).

[3] A. Boyle, *Reality check on Russia's 'zombie ray gun' program*, NBC News: Cosmic Log (April 7, 2012).

[4] S. Faletič, M. Michelini, and D. Buongiorno, *Nuclear Magnetic Resonance as an Applicative Topic in the Physics Curriculum for Students of Life Sciences*, pages 754–755, 2010.

[5] E. Cookson, D. Nelson, M. Anderson, D. L. McKinney, and I. Barsukov, *Exploring magnetic resonance with a compass*, The Physics Teacher **57**, 633 (2018).

[6] S. Murphy, D. L. Jones, J. Gross, and D. Zollman, *Apparatus for investigating resonance with application to magnetic resonance imaging*, American Journal of Physics **83**, 942 (2015).

[7] D. McBride, S. Murphy, and D. Zollman, *Student Understanding of the Correlation between Handsâon Activities and Computer Visualizations of NMR/MRI*, AIP Conference Proceedings **1289**, 225 (2010).

[8] D. Grainger, *Safety Guidelines for Magnetic Resonance Imaging Equipment in Clinical Use*, Technical report, Medicines and Healthcare products Regulatory Agency, London, United Kingdom, 2021.

[9] J. G. Delfino, D. M. Krainak, S. A. Flesher, and D. L. Miller, *MRI-related FDA adverse event reports: A 10-yr review*, Medical Physics **46**, 5562 (2019).

[10] H. Bhullar, B. County, S. Barnard, A. Anderson, and M. E. Seddon, *Reducing the MRI outpatient waiting list through a capacity and demand time series improvement programme*, **134**, 1537 (2021).

[11] LBN Medical, *MRI Machine Price*, `https://lbnmedical.com/how-much-does-an-mri-machine-cost/`, 2023, [Accessed: 2023-12-12].

[12] D. E. Vincent, T. Wang, T. A. Magyar, P. I. Jacob, R. Buist, and M. Martin, *Birdcage volume coils and magnetic resonance imaging: A simple experiment for students*, Journal of Biological Engineering **11** (2017).

[13] A. I. Smirnov, R. L. Belford, and I. Morse, Reef (Philip D., *Magnetic resonance imaging in a hands-on student experiment using an EPR spectrometer*, Concepts in Magnetic Resonance **11**, 277 (1999).

[14] Z. Jang, *Design and construction of low-cost EPR spectrometers for education*, New Physics: Sae Mulli **68**, 225 (2018).

[15] R. A. Butera and D. H. Waldeck, *An EPR Experiment for the Undergraduate Physical Chemistry Laboratory*, Journal of Chemical Education **77**, 1489 (2000).

[16] J. J. Hill, *A Magnetic Resonance Demonstration Model*, American Journal of Physics **31**, 446 (1963).

[17] O. Ennemoser and W. Ambach, *Magnetic resonance imaging in medical education: A demonstration experiment for students*, European Journal of Physics **12**, 52 (2000).

[18] CERN Teachers Lab, *Electron Spin Resonance model experiment*, `https://indico.cern.ch/event/36368/contributions/1777448/attachments/723409/992912/electron-spin-resonance-qrg.pdf`, 2008, [Accessed: 2023-12-12].

[19] P. Shinners, *About Pygame*, `https://www.pygame.org/wiki/about`, [Accessed: 2023-12-12].

[20] N. Elkunchwar, V. Iyer, M. Anderson, K. Balasubramanian, J. Noe, Y. Talwekar, and S. Fuller, *Bio-inspired source seeking and obstacle avoidance on a palm-sized drone*, in *2022 International Conference on Unmanned*

Aircraft Systems, ICUAS, pages 282–289, Institute of Electrical and Electronics Engineers Inc., 2022.

[21] M. S. Tanveer, S. M. Kabir, and A. S. Shihavuddin, *Determination of initial projectile velocity in the presence of static fields using deep actor critic method*, in *2020 11th International Conference on Electrical and Computer Engineering*, pages 455–458, Institute of Electrical and Electronics Engineers Inc., 2020.

[22] D. H. Perico, T. P. Homem, A. C. Almeida, I. J. Silva, C. O. Vilão, V. N. Ferreira, and R. A. Bianchi, *A Robot Simulator Based on the Cross Architecture for the Development of Cognitive Robotics*, pages 317–322, Institute of Electrical and Electronics Engineers Inc., 2016.

[23] A. Joshi and M. Kshirsagar, *Mathematical Modeling for Planetary Motion using Python's PyGame Module, Matplotlib and Linux Shell Scripting*, International Journal of Innovative Science and Research Technology **7** (2022).

[24] C. Wang, X. Zhang, Z. Jia-wei, Z. Ding, and A. Lanxuan, *Navigation behavioural decision-making of MASS based on deep reinforcement learning and artificial potential field*, Journal of Physics: Conference Series **1357**, 012026 (2019).

[25] L. G. Hanson, *A graphical simulator for teaching basic and advanced MR imaging techniques*, Radiographics : a review publication of the Radiological Society of North America, Inc **27** (2007).

[26] D. Treceño-Fernández, J. Calabia-Del-campo, F. Matute-Teresa, M. L. Bote-Lorenzo, E. Gómez-Sánchez, R. de Luis-García, and C. Alberola-López, *Magnetic Resonance Simulation in Education: Quantitative Evaluation of an Actual Classroom Experience*, Sensors (Basel, Switzerland) **21** (2021).

[27] J. Johansson and M. Båth, *Simulated MRI-Scanning: Visualising signal sampling and image reconstruction of a human brain*, Master's thesis, University of Gothenburg, Gothenburg, Sweden, 2017.

[28] F. Liu, J. V. Velikina, W. F. Block, R. Kijowski, and A. A. Samsonov, *Fast Realistic MRI Simulations Based on Generalized Multi-Pool Exchange Tissue Model*, IEEE Transactions on Medical Imaging **36**, 527 (2017).

[29] Psychology Software Tools Inc., *IACI MRI Simulation Software*, https://pstnet.com/products/iaci-mri-simulation-software/, 2007, [Accessed: 2023-12-12].

[30] S. M. Smith, M. Jenkinson, M. W. Woolrich, C. F. Beckmann, T. E. Behrens, H. Johansen-Berg, P. R. Bannister, M. D. Luca, I. Drobnjak, D. E. Flitney, R. K. Niazy, J. Saunders, J. Vickers, Y. Zhang, N. D. Stefano, J. M. Brady, and P. M. Matthews, *Advances in functional and structural MR image analysis and implementation as FSL*, NeuroImage **23**, S208 (2004).

[31] D. Moratal, A. Vallés-Luch, L. Martí-Bonmati, and M. E. Brummers, *k-Space tutorial: an MRI educational tool for a better understanding of k-space*, Biomedical Imaging and Intervention Journal **4** (2008).

[32] S. B. McKagan, K. K. Perkins, M. Dubson, C. Malley, S. Reid, R. LeMaster, and C. E. Wieman, *Developing and researching PhET simulations for teaching quantum mechanics*, American Journal of Physics **76**, 406 (2008).

[33] E. L. Deci and R. M. Ryan, *The "What" and "Why" of Goal Pursuits: Human Needs and the Self-Determination of Behavior*, Psychological Inquiry **11**, 227 (2000).

[34] Q. Larson, *Interpreted vs Compiled Programming Languages: What's the Difference?*, https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/, 2020, [Accessed: 2023-12-12].

[35] P. Shinners, *pygame.time - pygame documentation*, https://www.pygame.org/docs/ref/time.html, 2003, [Accessed: 2023-12-12].

[36] A. Hoekstra, *MRI2D repository*, https://github.com/MRI2D/MRI2D, 2023, [Accessed: 2023-12-12].

[37] A. Hoekstra, *MRI2D Video Demonstration*, https://vimeo.com/893059133, 2023, [Accessed: 2023-12-12].

Appendix A

User guide

A.1 Installing the program

The program can be downloaded as a zip file and extracted to any folder. To make the program run, it needs an installed version of Python, with some modules, as a runtime environment. To install the program therefore follow the steps below:

1. Download MRI2D from the `github` page by clicking on the green Code button, and selecting Download zip: `https://github.com/MRI2D/MRI2D`

2. Extract the MRIDemonstrator.zip to a folder.

3. Download the Python Windows installer or the Python installer for your platform from the Downloads page on the official Python site: `https://www.python.org/downloads/`

4. Very important: When running the installer in the first screen select the Python folder to be added to the PATH.

5. Then open a Run as Administrator Command prompt (Windows, in Apple/Linux: a console), and Enter the following command:

```
pip install numpy matplotlib pygame
```

6. This should now run a set of installation scripts. If not, the Python folder was likely not added to PATH during installation.

A.2 Running the program

The program can be run by double clicking on: *RunSim.py*. Start-up settings and tissues files can be specified in *settings.py*. This file is a configuration file, so leave the format intact while editing it. If something goes wrong with this file, run *resettings.py* to restore the default settings.

A.3 Graphical user interface

The program has two main interfaces: the Runtime screen and the Analysis screen.

A.3.1 Runtime screen

The runtime screen looks as follows:

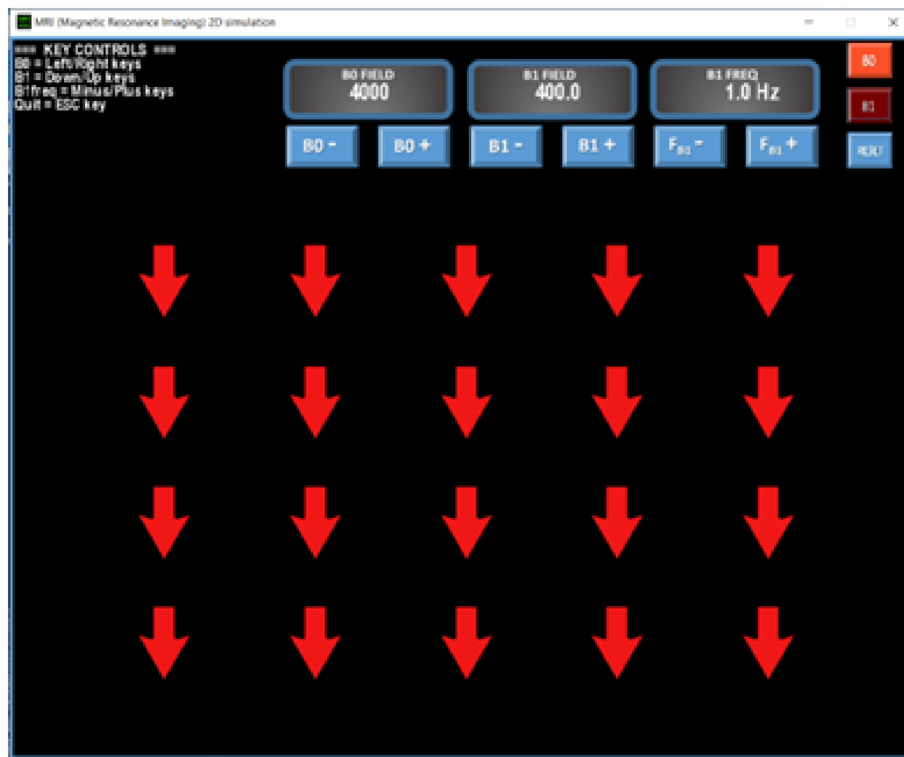


Figure A.1: *Screen when program starts*

From top to bottom, the screen contains the elements listed below:

- Shortcut information on keys
- Three displays to show the current values of:
  - B_0 field strength
  - B_1 field strength
  - B_1 frequency
- Each of the displays has two buttons, to be selected and pressed with the mouse, to increase (+) or decrease (-) the value logarithmically
- On the right side three more buttons are shown, which act as switches:
  - B_0 field on/off
  - B_1 field on/off
  - Reset/swing button: resets arrow speeds and swings them to 45 degrees to test resonance
- Field with 4 rows of 5 simulated magnetic arrows which can resonate using the right settings (e.g. 4000/400/1.229).

The simulation can be ended using the close window button or the ESCape key. Quitting the simulation will prompt the analysis screen.

A.3.2 Analysis screen

When the program is ended it will show three time-history plots (Figure A.2) to help in analysing, saving and understand the simulation results.

The analysis screen is a *matplotlib* plot screen, with the following controls on the top row:

- **Home/Back/Forward**: to go back and forward when selecting, panning and zooming through plots. Home restores the default start value for these display settings.
- **Pan** button (with arrows)
- **Zoom** button: select an area to zoom in.
- **Layout settings**: spacing, etc.
- **Axis settings**: ticks, titles, etc.

- **Save image** button: saves the plot as an image.

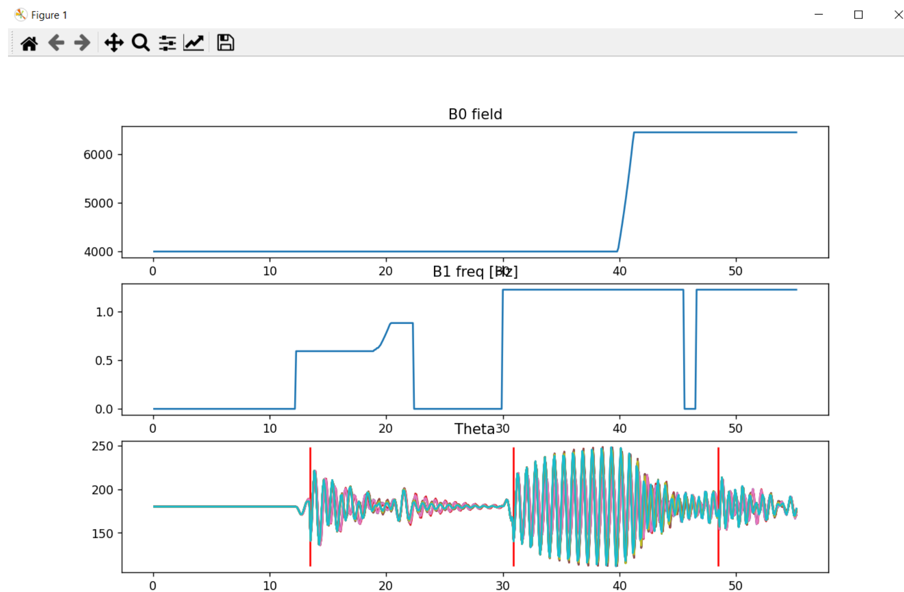


Figure A.2: *Analysis screen, shown after each simulation*

In the example plot it can be seen which combination of B_0 (field strength) and B_1 frequency lead to resonance. The vertical red lines indicate the selection of the Reset/Swing button. It can be seen that for most combinations the oscillation is quickly damped, but not for the previously mentioned 4000-400-1.229 settings for B_0, B_1 and B_1 frequency at t=31 seconds. When B_1 is turned off, the B_1 frequency graph drops to zero. The presence of different tissues can also be seen in the angle *theta*-plot, with the red oscillation not reaching the same amplitude as the blue oscillation. The shape of the tissue cannot be found here as the locations of the arrows are not indicated.

Tissue can be detected in the runtime screen by the different behaviour of the magnetic arrows.

A.4 Magnetic Resonance Exercises

A.4.1 Exercise 1: Demonstration of magnetic resonance

Run the program by selecting RunSim.py. Experiment with the controls and settings to find a qualitative relation between B_0 and the resonance frequency. If B_0 magnitude is increased, should B_1 frequency be decreased or increased to give the nuclear spins a larger amplitude?

A.4.2 Exercise 2: Determine resonance frequency and magnetic field correlation

Let us hypothesise the formula for the resonance frequency is given by:

$$f_{reso} = C \cdot {B_0}^p \tag{A.1}$$

The assignment is now to determine the resonance frequency per B_0 magnitude for a series of B_0 magnitudes. Write down the resonance frequency for each B_0 magnitude in a table like Table A.1

Table A.1: *Example table to fill with measurements for exercise 2. The B_1 frequency at which the largest amplitude is recorded is the resonance frequency, and will be the dependent variable. The B_1 magnitude is kept constant, but should be recorded at least once for repeatability. Its value here is carefully chosen to be as high as possible without making measurements at B_0 magnitude $2000 \cdot 10^{-4}$ T impossible.*

B_0 magnitude (10^{-4} T)	B_1 magnitude (10^{-4} T)	B_1 frequency (Hz)
2000	1000	0.80
5000	1000	....
10000	1000	....
20000	1000	....
50000	1000	....
100000	1000	....

When you have collected enough data points, you can use a program to fit a curve to these points. Use B_0 magnitude as X and resonance frequency as Y. Try a linear fit and a power (also called allometric) fit. Check if the hypothesised Equation A.1 was correct, and which value you have found for p. Does it match up with harmonic oscillator theory?

A.4.3 Exercise 3: Performing an MRI scan with MRI2D

Now that you have measured the normal situation with magnetic resonance, you are ready to image shapes of different tissues. Enable a tissue mask as instructed by your teacher.

First use data from Exercise 2 to bring the unchanged nuclear spins to resonance. Look carefully which spins do not behave like the others. The smaller the amplitude, the darker you can colour the pixel in the lattice location of that spin. For example, if the four top-right spins barely move, while a diagonal of top-left to bottom-middle oscillates a little, but not as much as the other spins, your answer should be Table A.2.

Table A.2: *Example of student answer for exercise 3. Cells are coloured according to the resonance frequencies of the nuclear spins at the respective positions.*

To be sure you have the right shades, try lower driving frequencies. Find the resonance frequencies of the different tissues using a similar method as you did for exercise 2.

Appendix B

Video demonstration details

This appendix provides supplementary information for section 4.1. The video can be viewed online [37].

The video shows two resonance frequency measurements, each at a different B_0 magnitude. The spins will show the highest amplitude when the frequency of the driving B_1 field matches the resonance frequency of the spins.

B.1 Measurement 1: Resonance frequency at B_0 magnitude 0.5 T

0:00 - 0:10 Configuration. The default settings are changed. B_0 magnitude is set to $5000 \cdot 10^{-4}$ T and the B_1 frequency is set to 1.25 Hz in `settings.py`. The program is started with these settings by executing `RunSim.py`.
0:10 - 2:25 Experiment. When the arrows appear to be in a steady state, the `RESET` button is pressed. This step is repeated thrice before the frequency is increased to ensure accurate measurements of the steady state amplitude at each B_1 frequency. The frequency is increased in steps of around 0.05 Hz.
2:25 - 4:00 Analysis. The *Theta*-plot is analysed to find the highest amplitude in steady state. The plot coordinates of the cursor are displayed at the bottom of the window to measure the amplitude. The resonance frequency for a B_0 magnitude of $5000 \cdot 10^{-4}$ T is around 1.35 Hz.

B.2 Measurement 2: Resonance frequency at B_0 magnitude 1.0 T

4:00 - 4:15 Configuration. B_0 magnitude is set to $10000 \cdot 10^{-4}$ T and the B_1 frequency is set to 2.0 Hz in `settings.py`. The initial B_1 frequency was increased because a higher resonance frequency is expected for this B_0 magnitude.
4:15 - 5:45 Experiment. As per section B.1, the frequency is increased in steps of 0.05 Hz and the `RESET` button is used thrice per B_1 frequency to ensure accurate measurements. With a higher B_0 magnitude than in section B.1 and the same B_1 magnitude, the amplitudes of the spins are lower. The approximation used to find Equation 2.8 assumed displacement angles around 0, so it gets more relevant the larger B_0 magnitude is compared to B_1 magnitude. The resonance frequency is easier to see in this measurement because of this larger ratio between the B_0 and B_1 magnitudes.
5:45 - 6:37 Analysis. Once again the plots are analysed to find the B_1 frequency corresponding to the highest amplitude shown by the spins in a steady state. The spins reached steady states faster than in section B.1 because of the higher resonance frequency induced by the higher B_0 magnitude. The resonance frequency for B_0 magnitude of $10000 \cdot 10^{-4}$ T is around 2.0 Hz.

Appendix C

Python code

C.1 Initialisation: Help programs

```
import os

def setvar(name, value):
    '''
    Takes in a 2-character long variable name (string),
    then writes the given value (integer or float)
    to the right spot in settings.txt
    '''
    dir_path = os.path. dirname(os.path.realpath(__file__))
    os.chdir(dir_path)

    sets = open("settings.py", "r")
    lines = sets.readlines()
    sets.close()

    news = open("settings.py", "w")
    for line in lines:
        line = line.rstrip()
        if line[:2] == name:
            newline = name+" = "+str(value)
        else:
            newline = line
        news.write(newline+"\n")
    news.close()
```

scripts/ppt2py.py

```
# Initial value of B0
b0 = 5000

```

```
# Initial value of B1
b1 = 1000

# Initial value of B1 frequency
f1 = 1.0

# Tissuefile
tf = 'tissue2'

# Friction damping (0 - 1.0, normal = 0.6)
kd = 0.6
```

scripts/settings.py

```
from sim import main
from settings import b0,b1,f1,kd,tf
import os

# Protect against PowerPoint starting in wrong working directory
dir_path = os.path. dirname(os.path.realpath(__file__))
os.chdir(dir_path)


# List of B0 magnitude, B1 magnitude, B1 frequency, Damper and
    Tissue file
settings = [b0,b1,f1,kd,tf]

# Call main with these settings
main(*settings)
```

scripts/RunSim.py

C.2 Simulation

C.2.1 Main program

```
"""
MRI 2D by Alex Hoekstra
Python 3.9.13
"""
import time
import numpy as np
from graphics import GUI, clock
from model import Model
from plotter import Plotter


```

```
12 def main(b0set = 4000, b1set = 1000, f1set = 1, kdamper = 0.6,
   tissuefile=""):
13     '''Main function containing simulation loop'''
14 
15     # Dimensions of compass arrow grid (m,n)
16     m, n = 5, 4
17 
18     # Initialize GUI window with a caption, xmax,ymax
19     xmax,ymax = 1000,800
20 
21     # Create model and gui
22     #model = Model(xmax,ymax,m,n)
23     gui = GUI("MRI (Magnetic Resonance Imaging) 2D simulation",
    xmax,ymax)
24 
25     # Not starting from 0 as then the unaffected arrows are
    quite boring
26     model = Model(b0set, b1set, f1set, kdamper, m,n,xmax,ymax,
    tissuefile,gui)
27 
28     # factor for speed of control by keys and mouse
29     adjustfactor =  np.sqrt(2) # Doubling in 2 seconds. factor
    per second, >1 for logical behaviour
30 
31     # Set up timer for loop
32     print("Starting simulation")
33     tstart = clock()
34     t0 = tstart
35     tsim = 0      # simulated time for harmonic oscillation of B1
36     maxdt = 0.1 # time step max so slow PCs won't have big time
    steps
37     running = True
38 
39     # Create a plotter
40     dtplot = 0.1 #delta time for tables with plot data
41     plotter = Plotter(tsim,dtplot)
42 
43     # Main simulation loop
44     while running:
45         # Time control in loop
46         t = clock()
47         dt = min(t-t0,maxdt) # set maximum limit to dt
48         t0 = t
49 
50         # Plot data to be added
51         plotter.tableupdate(tsim, model.b0on*model.b0mag,
52                             model.theta, model.b1on*model.b1freq
    )
53 
```

```
54         # Simulated time, also protected for time steps larger than maxdt
55         tsim = tsim + dt
56 
57         # If a real timestep has been made, we calculate and draw
58         if dt>0:
59 
60             # Update compass arrows model
61             model.update(tsim,dt)
62 
63             # Update GUI
64             gui.clearscreen()
65             gui.textpanel(model.b0mag, model.b1mag, model.b1freq, model.b0on, model.b1on)
66             gui.drawarrows(model.arrowlist,m,n)
67             gui.updatescreen()
68 
69         # Key inputs
70         # B_0 magnitude with right/left
71         keyspressed = gui.getkeys()
72 
73         if 'RIGHT' in keyspressed:
74             model.b0mag *= adjustfactor**dt
75         if 'LEFT' in keyspressed:
76             model.b0mag /= adjustfactor**dt
77 
78         # B_1 magnitude with up/down
79         if 'UP' in keyspressed:
80             model.b1mag *= adjustfactor**dt
81         if 'DOWN' in keyspressed:
82             model.b1mag /= adjustfactor**dt
83 
84         # Frequency (B_1) with +/-
85         if 'PLUS' in keyspressed:
86             model.b1freq *= adjustfactor**(dt/1.4)
87         if 'MINUS' in keyspressed:
88             model.b1freq /= adjustfactor**(dt/1.4)
89 
90         # B_0 on/off switch
91         if "B0" in keyspressed:
92             model.b0on = not model.b0on
93 
94         # B_1 on/off switch
95         if "B1" in keyspressed:
96             model.b1on = not model.b1on
97 
98         # Angular velocity reset
99         if "V" in keyspressed:
```

```
100                     model.v = model.v * 0#
101
102         # Total reset: angular velocity & position
103         if "RESET" in keyspressed:
104             model.v = model.v*0
105             model.theta = 140+0*model.theta
106             plotter.tabletreset(tsim) # Keep track of reset
    times for plots
107
108         # Magnetic fields details
109         # Debug messages
110         if "B" in keyspressed and t%.2>.18:
111             #time requirement so the message isn't printed too
    often
112             print("____________________\n",
113                 "B0 magnitude | ", round(b0mag,5), "\n",
114                 "B1 magnitude | ", round(b1mag,5), "\n",
115                 "B1 frequency | ", round(b1freq,5),
116                 "\n____________________")
117
118         # Quit with Esc
119         if "ESC" in keyspressed:
120             running = False
121
122         # Runtime limit
123         if t>300:
124             running = False
125
126     # Exit when loop is ended
127     # close screen
128     print("Simulation ran",t-tstart,"seconds")
129     del gui
130
131
132     # Plot store tables with data
133     plotter.plotdata()
134     print("Ready.")
135
136 # "If this program is run, run main."
137 # Checking this makes it easier to keep this .py file in folders
        for importing
138 if __name__ == "__main__":
139         main()
```

scripts/sim.py

C.2.2 Programs containing essential classes and functions

Model

```
import numpy as np
from graphics import Arrowimg
import os

class Model():
    '''
    This class contains all physics.
    It sets start values for physical quantities and contains calculations
    that are called upon every frame. It needs the time and timesteps as inputs,
    they are not regulated within this class. (See sim.py, the main program.)
    '''
    def __init__(self, b0set, b1set, f1set, kdamper, \
                 m, n, xmax, ymax, tissuefile, gui):
        # Array size
        self.m, self.n = m,n

        # Adjustable values
        self.b0mag, self.b1mag, self.b1freq = b0set, b1set, f1set
        self.kdamper = kdamper

        # Pixel distance between compass arrows
        xdist = xmax / (self.m + 1)
        ydist = ymax / (self.n + 2)

        # + 1 because the centers are half distance from borders on each side
        # Arrays with the x and y pixel positions of the compass arrows
        xposarr = np.arange(xdist, xmax, xdist)
        yposarr = np.arange(2.0 * ydist, ymax, ydist)

        # Setting up the arrows
        # Saving them in a 2D list so using the physics arrays will be easier
        self.arrowlist = []
        for i in range(m):
            self.arrowlist.append([])
            for j in range(n):
                self.arrowlist[i].append(Arrowimg(xposarr[i], yposarr[j], gui.imglist))

```

```
38        # Simulating tissue in the grid
39        self.readtissue(tissuefile)
40
41        # Starting the magnetic field arrays
42        # Optional x gradient and y gradient to build magnetic
    fields on
43        self.b0gradient, self.b1gradient = np.zeros([m, n]), np.
    zeros([m, n])
44        for i in range(m):
45            self.b0gradient[i, :] = np.linspace(.1, 1, n)
46        for j in range(n):
47            self.b1gradient[:, j] = np.linspace(.1, 1, m)
48
49        # For realism, not every compass arrow is exactly
    identical, so add noise
50        # Use gradient factor for noise
51        self.noise = 0.03
52        self.b0gradient = 1 + self.noise * (np.random.rand(m, n)
     - 0.5)
53        self.b1gradient = 1 + 0 * self.b1gradient
54
55        # Start with B0 and B1 on
56        self.b0on = True
57        self.b1on = True
58
59        # Set up physics variables
60        # Angles and angular velocities
61        theta0 = 180.
62        self.theta, self.v = np.ones([m, n]) * theta0, np.zeros
    ([m, n])
63
64    def readtissue(self,tissuefile):
65        '''
66        Imports the file containing the properties of compasses
67        by their coordinates, to simulate different tissues, and
     "image" them.
68        '''
69        # Read tissue file or set to one
70        if tissuefile == "":
71            self.tissuemask = np.ones([self.m, self.n])
72        else:
73            # Check for .csv or .tis extension, if not add ".tis
    "
74            filename = os.path.join("data",tissuefile)
75            if not (".tis" in filename) and \
76               not (".csv" in filename):
77                filename += ".tis"
78
79            # Read lines from tissuefile
```

```
80            f = open(filename,"r")
81            lines= f.readlines()
82            f.close()
83            print("Reading "+filename)
84
85            # Read tissue data from lines from tissue CSV file
86            # Use list for appending
87            tissuelist = []
88            for line in lines:
89                if line.strip()[0]=="#":
90                    continue # Comment line skip to next
91                else:
92                    row = []
93                    columns = line.split(",")
94                    for col in columns:
95                        row.append(float(col))
96
97                # Add row of data
98                tissuelist.append(row)
99
100           # Convert list to numpy array
101           # Row,column  = x,y so transpose data from file
102           self.tissuemask = np.array(tissuelist).T
103
104
105       #if tissuesimulation:
106       #    tissuemask[int(m / 3):2 * int(m / 3), int(n / 3):2
    * int(n / 3)] = .1
107       #    # This picks one or a few center arrows to
    experience a lot less force
108       #    print("Simulated tissue:", tissuemask)
109
110
111   def force(self, bx, by, theta):
112       # Force function
113       '''
114       Takes in 3 arrays B0, B1 and the angles of the arrows.
115       Returns the torque on each.
116       '''
117       # arctan2 is the inverse tangent that can automatically
    deal
118       # with any quadrant
119       forcerad = np.arctan2(by, bx)
120       forcemag = np.sqrt(bx ** 2 + by ** 2)
121
122       # theta was in degrees, we work with radians for numpy
    sine
123       thetarad = theta / 360 * 2 * np.pi
124       # angle between the spin (thetarad) and the force
```

```
125         anglediff = thetarad - forcerad
126         # doesn't have to be smallest angle, or any specific
    domain
127         # since it will go into a sine function
128 
129         # possible to add extra factors here
130         force = forcemag * np.sin(anglediff)
131         return force
132 
133     def update(self, tsim, dt):
134         '''
135         Time integration from acceleration, to velocity, to
    theta.
136         '''
137         # Model update function
138         if dt > 0.:
139             # Update magnetic field arrays, separate for x and y
     (B0 and B1)
140             self.b0 = self.b0mag * self.b0gradient
141             self.b1 = self.b1mag * np.cos(2 * np.pi * self.
    b1freq * tsim) * self.b1gradient
142             a = self.tissuemask * self.force(self.b0on * self.b0
    , self.b1on * self.b1, self.theta)\
143                     - self.kdamper * self.v
144 
145             # can add a factor here if necessary to simulate MoI
146 
147             self.v = self.v + a * dt
148             self.theta = self.theta + self.v * dt
149 
150             # Now all array calculations are done
151             # We have to switch to loops/lists for graphics
152 
153             # Set all arrows from arrowlist to their new angle
154             for i in range(self.m):
155                 for j in range(self.n):
156                     # Update the arrow for this iteration
157                     # Slicing looks different because arrowlist
    a nested list
158                     self.arrowlist[i][j].update(self.theta[i, j
    ])
```

scripts/model.py

Graphics

```
1 import os
2 import pygame as pg
```

```
from fastfont import Fastfont

# Some colours in RGB values
black = (0,0,0)
white= (255,255,255)
blue = (91,155,213) #RGB value from display panel image

pg.init()

# Clock function
def clock():
    '''
    Uses pygame ticks to get time units.
    Because physics use time, not ticks.
    '''
    time = pg.time.get_ticks()*0.001
    return time

#Class
class Arrowimg():
    '''
    Most of the arrow graphics. This class handles the image and
    the "Rect".
    Needs the imglist so it can quickly update to correct angle.
    '''
    def __init__(self,posx,posy,imglist):
        self.imglist = imglist
        fname = os.path.join('bitmaps','arrow.png')
        img_source = pg.image.load(fname).convert_alpha()
        self.img = img_source.copy()
        rectarrow = pg.Rect(posx,posy,10,10)
        img_rect = img_source.get_rect(center=rectarrow.center)
        self.rect = self.img.get_rect(center=img_rect.center)

    def update(self, theta):
        '''
        Takes in any float theta and
        Loads the nearest integer angle,
        as imglist contains 360 images.
        '''
        angle = int(theta+.5)%360
        self.img = self.imglist[angle]
        self.rect = self.img.get_rect(center=self.rect.center)


class GUI():
    '''
    Graphical User Interface class,
    set up, clear update and draw screen.
```

```
    '''
    def __init__(self,caption,xmax,ymax):

        # Initialize pygame
        pg.init()

        # Set up screen
        self.xmax = xmax
        self.ymax = ymax
        self.screen = pg.display.set_mode((xmax,ymax))
        pg.display.set_caption(caption)
        pg.display.set_icon(pg.image.load("icon.gif"))

        # Load rotated images into imagelist for all angle 0-359
 deg
        self.imglist = []
        print("Loading rotated arrow images",end="")

        # Load original arrow image, for if rotation is
necessary
        fname = os.path.join('bitmaps','arrow.png')
        img_source = pg.image.load(fname).convert_alpha()


        # Loading all rotated images, if not exist, rotate and
save
        for i in range(360):

            # Show progress bar with points
            if i%30==0:
                print(".",end="")

            # Load from (or save in) sprites folder
            spritename = os.path.join('sprites', 'arrow' + str(i
) + '.png')

            if os.path.exists(spritename):
                img = pg.image.load(spritename).convert_alpha()
            else:
                img = pg.transform.rotate(img_source, i).
convert_alpha()
                pg.image.save(img, spritename)

            # Add to rotated arrow img to imglist
            self.imglist.append(img)

        print("")

        # Display panel bitmap
```

```
95         self.panelimg = pg.image.load(os.path.join('bitmaps','displays-h130.png'))
96         self.panelrect = self.panelimg.get_rect()
97 
98         # Light button images, same size, so we can use same rect object)
99         self.imgb0on  = pg.transform.scale(pg.image.load(os.path.join('bitmaps','B0on.png')),(50,40))
100        self.imgb0off = pg.transform.scale(pg.image.load(os.path.join('bitmaps','B0off.png')),(50,40))
101        self.imgb1on  = pg.transform.scale(pg.image.load(os.path.join('bitmaps','B1on.png')),(50,40))
102        self.imgb1off = pg.transform.scale(pg.image.load(os.path.join('bitmaps','B1off.png')),(50,40))
103        self.imgreset = pg.transform.scale(pg.image.load(os.path.join('bitmaps', 'RESET.png')), (50, 40))
104        self.imgbrect = self.imgb0on.get_rect()
105
106        # Button positions
107        self.B0xy = self.xmax-50,26
108        self.B1xy = self.xmax-50,76
109        self.RESETxy = self.xmax-50,126
110
111        # Calculate where panel starts
112        self.panelx0  = int(self.xmax/2)-int(self.panelrect.width/2)+100
113        self.panely0  = 20
114
115        # Create font objects
116        self.font = Fastfont(self.screen,'Arial',17,white,True,False)
117        self.dispfont = Fastfont(self.screen,'Arial',25,white,True,0,0) # 0,0 = center this font in x and y
118        #       (pygame screen, font,size,   colour RGB,bold,italic)
119
120
121
122    def textpanel(self,b0mag,b1mag,b1freq,b0on,b1on):
123        '''
124        Displays the text panel listing the key controls on the left,
125        the B0/B1 on/off buttons and the RESET button on the right.
126        (NOT the increase/decrease buttons for B0/B1 magnitude and B1 freq)
127        '''
128
```

```
129        # Blit background image for display panel, center of
   screen
130        self.panelrect.centerx = int(self.xmax / 2) + 100
131        self.panelrect.y = self.panely0
132        self.screen.blit(self.panelimg, self.panelrect)
133
134        # Text
135
136        # Render text at x,y-position
137        xtxt = 6
138        y = 5
139        dy = 15
140
141        self.font.printat(self.screen,xtxt,y,"=== KEY CONTROLS
    ===")
142        y += dy
143        self.font.printat(self.screen,xtxt,y,"B0 = Left/Right
   keys")
144        y += dy
145        self.font.printat(self.screen,xtxt,y,"B1 = Down/Up keys
   ")
146        y += dy
147        self.font.printat(self.screen,xtxt,y,"B1freq = Minus/
   Plus keys")
148        y += dy
149        self.font.printat(self.screen,xtxt,y,"Quit = ESC key")
150        y += dy
151        #self.font.printat(screen,xtxt,y,"Reset vel: v")
152
153        # Positon of values in displays
154
155        # Use display panel x-coordinate to set text x-
   coordinate
156        yb0 = self.panely0 + 26
157        xb0 = self.panelx0 + 99
158        xb1  = self.panelx0 + 307
159        xb1f = self.panelx0 + 521
160        self.dispfont.printat(self.screen, xb0, yb0, str(round(
   b0mag, 3)))
161        self.dispfont.printat(self.screen, xb1, yb0, str(round(
   b1mag, 3)))
162        self.dispfont.printat(self.screen, xb1f, yb0, str(round(
   b1freq, 3)) + " Hz")
163
164        # Button for B0 on/off
165        self.imgbrect.center = self.B0xy
166        if b0on:
167            self.screen.blit(self.imgb0on,self.imgbrect)
168        else:
```

```
169             self.screen.blit(self.imgb0off,self.imgbrect)
170 
171         # Button for B1 on/off
172         self.imgbrect.center = self.B1xy
173         if b1on:
174             self.screen.blit(self.imgb1on,self.imgbrect)
175         else:
176             self.screen.blit(self.imgb1off,self.imgbrect)
177 
178         # RESET button
179         self.imgbrect.center = self.RESETxy
180         self.screen.blit(self.imgreset, self.imgbrect)
181 
182     def drawarrows(self,arrowlist,m,n):
183         '''
184         Goes through the nested list (to represent the two
    dimensions) with
185         Arrowimg objects and displays each one at the correct
    position and angle.
186         '''
187         for i in range(m):
188             for j in range(n):
189                 # Slicing looks weird because it's a nested list
190                 # To put the image on the screen, we blit the
    image and Rect
191                 self.screen.blit(arrowlist[i][j].img, arrowlist[
    i][j].rect)
192 
193     def clearscreen(self):
194         '''
195         Pygame doesn't get rid of the previous frame by itself,
    so a
196         function is needed to clear the screen unless. Done here
     by simply
197         drawing a black rectangle over everything.
198         '''
199         self.screen.fill(black)
200         rect = self.screen.get_rect()
201 
202 
203         # Draw a border
204         dx = 4
205         pg.draw.rect(self.screen,blue,rect,dx)
206 
207 
208     def updatescreen(self):
209         '''
210         Display all the prepared elements of a frame.
211         '''
```

```
212         pg.display.flip()
213
214     def getkeys(self):
215         '''
216         Regulates every functionality that has an optional key
    control.
217         Buttons for increasing or decreasing B0/B1 magnitude and
     B1 frequency
218         are also part of this, since they require the same
    limits on how much
219         they should change between frames when held down.
220         '''
221         # Use keynames to report which keys have been pressed
222         # This dictionary translates key code in key names
223         keynames = { pg.K_RIGHT: 'RIGHT',
224                      pg.K_LEFT: 'LEFT',
225                      pg.K_UP: 'UP',
226                      pg.K_DOWN: 'DOWN',
227                      pg.K_EQUALS: 'PLUS',
228                      pg.K_MINUS: 'MINUS',
229                      pg.K_v:  'V',
230                      pg.K_b:  'B',
231                      pg.K_ESCAPE: 'ESC',
232                      pg.K_HOME:  'RESET',
233                      }
234
235         # Keys list to return
236         activekeys = []
237
238         # Check events
239         for event in pg.event.get():
240             # Quit event (closing window) is same ESCAPE key
241             if event.type==pg.QUIT:
242                 activekeys.append(keynames[pg.K_ESCAPE])
243
244             # Mouse clicked on buttons
245             elif event.type == pg.MOUSEBUTTONUP:
246                 mousex,mousey = event.pos
247                 butnr = event.button
248                 # Check on buttons B0 and B1, size 50,40 with x,
    y-distance to center of button
249                 if abs(mousex-self.B0xy[0])<25 and abs(mousey-
    self.B0xy[1])<20:
250                     activekeys.append("B0")
251                 elif abs(mousex-self.B1xy[0])<25 and abs(mousey-
    self.B1xy[1])<20:
252                     activekeys.append("B1")
253                 elif abs(mousex - self.RESETxy[0]) < 25 and abs(
    mousey - self.RESETxy[1]) < 20:
```

```
                    activekeys.append("RESET")

        # Check mouse buttons status and position for buttons
        # Translate to corresponding keyname
        buttons = ["LEFT","RIGHT","DOWN","UP","MINUS","PLUS"]
        leftmousebutton = pg.mouse.get_pressed()[0]
        if leftmousebutton:
            # Mouse positions in panel coordinates
            xmouse = pg.mouse.get_pos()[0] - self.panelx0
            ymouse = pg.mouse.get_pos()[1] - self.panely0

            # Positions of 6 buttons: size
            ybuttons = 74
            dybuttons = 50
            xbuttons = list(range(5,5+5*106+1,106)) # start coordinates of + and - buttons
            dxbuttons = 80


            if ybuttons < ymouse < ybuttons+dybuttons:
                for i in range(6):
                    if xbuttons[i]<= xmouse <=xbuttons[i]+dxbuttons:
                        activekeys.append(buttons[i])

        # Check keys
        pg.event.pump()
        keyboard = pg.key.get_pressed()
        for code in keynames.keys():
            if keyboard[code]:
                activekeys.append(keynames[code])

        return activekeys


    def __del__(self):
        pg.display.quit()
        pg.quit()
```

scripts/graphics.py

Plotter

```
import numpy as np
import matplotlib.pyplot as plt

# Show plot at end of program to help in finding reso frequency

```

```
6  class Plotter():
7      '''
8      Handles everything necessary to create graphs after the simulation is done,
9      to allow proper analysis.
10     Tracks and later displays B0 magnitude, B1 freq, and theta through time.
11     B1 magnitude isn't tracked, since its only requirement for finding the
12     resonance frequency is being strong enough, not a specific value.
13     '''
14     def __init__(self,tsim,dtplot):
15         # Store starting time for plotting timer
16         self.dtplot = dtplot
17         self.tplot = tsim
18
19         # Tabular data for plotting and timing
20         self.thetatab = []
21         self.b0tab = []
22         self.f1tab = []
23         self.timetab = []
24         self.devtab = []
25         self.treset = []
26
27     def tableupdate(self,tsim,b0mag,theta,b1freq):
28         '''
29         Adds a new data point in each plot if the time since last update is
30         longer than dtplot, the time steps of the plots. Saving data on every
31         tick would be a waste of storage.
32         '''
33         # Check whether it is time to store an update values to tables for plotting
34         if tsim - self.tplot > self.dtplot:
35             # Plotting timer
36             self.tplot = tsim
37
38             # Plot data
39             self.timetab.append(tsim)
40             self.b0tab.append(b0mag)
41
42             # reshape theta into one long 1D array for plotting
43             m,n = theta.shape
44             allthetas = theta.reshape(m * n)
45
46             self.thetatab.append(np.mod(allthetas,360))
47             self.devtab.append(np.std(allthetas))
```

```
48              self.f1tab.append(b1freq)
49
50      def tabletreset(self,tsim):
51          '''Tracks RESET button usage'''
52          # Save the times of resetting arrows for red lines in
    the plot
53          self.treset.append(tsim)
54
55      def plotdata(self):
56          '''
57          Draws the saved data (B0 magnitude, B1 frequency and
    theta) in 3 plots.
58          '''
59
60          # Three or four rows with a plot, increase this number
    to add a plot
61          nrows = 3
62
63          # Plot data
64          plt.subplot(nrows*100+11)
65          plt.title("B0 field")
66          plt.plot(self.timetab, self.b0tab)
67
68          plt.subplot(nrows*100+12)
69          plt.title("B1 freq [Hz]")
70          plt.plot(self.timetab, self.f1tab)
71
72          plt.subplot(nrows*100+13)
73          plt.title("Theta")
74          plt.vlines(self.treset, np.min(self.thetatab), np.max(
    self.thetatab), "r")
75          plt.plot(self.timetab, self.thetatab)
76
77  #         plt.subplot(nrows*100+14)
78  #         plt.title("Std Dev Theta")
79  #         plt.plot(self.timetab, self.devtab)
80
81          plt.show()
```

scripts/plotter.py