The Thesis Committee for Somnath Rakshit
Certifies that this is the approved version of the following thesis:

A GPU-accelerated MRI Sequence Simulator for Differentiable Optimization and Learning

APPROVED BY

SUPERVISING COMMITTEE:

Jonathan I. Tamir, Supervisor

Ying Ding

A GPU-accelerated MRI Sequence Simulator for Differentiable Optimization and Learning

by

Somnath Rakshit

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Information Studies

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2021

I dedicate this work to my family who have always stood by me, through thick and thin.

Acknowledgments

There are lots of people whom I would like to thank. First and foremost, I would like to thank my thesis supervisor, Jon Tamir. I started this project without any background in MRI. However, Jon was extremely patient in explaining each and every critical concept to me. He also spent countless hours with me debugging various parts of the codebase, and introducing me to new ideas. Thanks are also due to Ke Wang from UC Berkeley, who first wrote the non-parallelized version of this simulator, much before I started working on this project. My thanks also go to Geetali Tyagi, who was my partner when we began working on this for a class project, and Sidharth Kumar, who helped me not only understand various concepts, but was also always ready to debug the codebase with me. I am also grateful to Amazon for providing research support through the Amazon AWS Machine Learning Research grant. I am sure that I would not be able to reach here without all of their help, and for this, I thank them sincerely.

A GPU-accelerated MRI Sequence Simulator for Differentiable Optimization and Learning

Somnath Rakshit, MSIS
The University of Texas at Austin, 2021

Supervisor: Jonathan I. Tamir

The Extended Phase Graph (EPG) Algorithm is a powerful tool for magnetic resonance imaging (MRI) sequence simulation and quantitative fitting, but simulators based on EPG are mostly written to run on CPU only and (with some exception) are poorly parallelized. A parallelized simulator compatible with other learning-based frameworks would be a useful tool to optimize scan parameters, estimate tissue parameter maps, and combine with other learning frameworks. In this thesis, I present our work on an open source, GPU-accelerated EPG simulator written in PyTorch. Since the simulator is fully differentiable by means of automatic differentiation, it can be used to take derivatives with respect to sequence parameters, e.g. flip angles, as well as tissue parameters, e.g. T_1 and T_2. Here, I describe the simulator package and demonstrate its use for a number of MRI tasks including tissue parameter estimation and sequence optimization.

Table of Contents

List of Figures

List of Listings

Chapter 1

Introduction

Magnetic Resonance Imaging (MRI) is a powerful and non-invasive medical imaging modality, but it is slow and prohibitively expensive. Thus, to innovate and bring down costs, researchers and engineers aim to develop new methods that tightly integrate the imaging physics with computational algorithms. For this purpose, there exist several MRI simulators which can mimic the scanner behavior and be used to design and evaluate new algorithms. However, these simulators typically suffer from two major problems: poor parallelization—making simulations time-consuming, and lack of auto-differentiation support—which makes it difficult to incorporate advances in machine learning and artificial intelligence. In our work, we develop a massively parallelized and auto-differentiable MRI simulator based on the Extended Phase Graph (EPG) Algorithm, which can utilize graphical processing units (GPUs) effectively, thereby leading to quick and accurate simulations.

Extended Phase Graphs were first described by Hennig in 1988 [1]. However, the term "Extended Phase Graphs" first appeared in a later paper by the same author in 1991 [2]. Since then, the EPG algorithm has become an invaluable tool for fast simulation of MRI sequences. This algorithm also forms the backbone of many sequence simulations [1, 3, 4, 5], and can also be used for quantitative parameter fitting [6, 7]. The EPG algorithm allows simulation of multiple nuclear magnetic resonance phenomena such as relaxation, dephasing, diffusion, motion and magnetization transfer. It works using a spatial Fourier transformation of the isochromat signals present in a voxel, which in turn are described by the Bloch equations [8].

Recently, multiple developers have shared EPG simulators online in independent repositories. However, most of them are written to run on CPU without large-scale parallelization [9]. More recently, parallelized implementations have been developed [10], though they are still limited to CPU and do not include auto-differentiation. Auto-Differentiation is a generalized way of taking a program that computes a value and constructing a procedure to compute the derivatives of that value in an automated manner. Since most advances in machine learning and artificial intelligence require computing derivatives of complicated functions, it is difficult to incorporate these newer methods into EPG simulators without using auto-differentiation although it is possible to explicitly compute these derivatives.

Graphical Processing Units have revolutionized the field of machine learning using large datasets as they have more processor chips than CPUs by orders of magnitudes and can thus, support large scale parallelization. They can produce remarkable performance gains for computationally intensive tasks involving data parallelization. It is not uncommon for GPU accelerated implementations to obtain solution to a problem in hours instead of weeks compared to CPU based implementations. This property of GPUs fits extremely well as a solution to the problem of parallelizing the EPG algorithm. Modern frameworks like PyTorch can take advantage of GPUs to perform complicated operations and compute the gradient of complicated functions using auto-differentiation. Thus, auto-differentiation and GPU acceleration would enable the use of EPG within other physics-based learning frameworks [11]. Hence, an open source, parallelized and PyTorch-based differentiable simulator could help develop and disseminate novel imaging techniques. This is similar to how packages like MIRT [12], BART [13], and SigPy [14] have made advanced parallel imaging and compressed sensing reconstruction methods widely accessible. To satisfy this need, we developed a GPU-accelerated EPG implementation as part of the `mri-sim-py.epg` Python package [15, 16]. Our parallelized simulator is compatible with other learning-based frameworks and thus can be used to directly optimize MRI scan parameters. Since the simulator is fully differentiable by means of automatic differentiation, it can be used to take derivatives with respect to sequence parameters like flip angles, as well as tissue parameters like T_1 and T_2. Hence, our open source, parallelized, and PyTorch-based differentiable simulator can help spearhead development and disseminate novel imaging techniques. Python was chosen as it can be demonstrated easily by means of Jupyter notebooks and interfaced with PyTorch, a popular machine learning and auto-differentiation framework. This simulator can be interfaced with different learning architectures such as physics-based learning, neural networks, etc. Several applications are shown such as tissue relaxation estimation, and optimizing the flip angles of a multi-echo spin-echo experiment using the Cramer Rao lower bound [17, 18, 19]. It is seen that our simulator is roughly 100,000 times faster than the previous CPU-only and poorly parallelized simulators. Here, we provide an overview of computational MRI methods that incorporate physics-based constraints, the software's goals, organization, demonstration of a few examples of its usage, and ideas on how to extend it further.

Chapter 2

Specific Approach

This chapter outlines the different approaches used to integrate the developed simulator[1] to various applications. We first provide an overview of computational MRI methods that incorporate physics-based constraints and then show how different techniques such as auto-differentiation, unrolled least squares solver, and a neural network can effectively use our simulator to estimate various parameters like T_1, T_2 and the echo train sequence.

2.1 MRI Physics

2.1.1 Spin Dynamics

It is possible to approximate an MRI system by a dynamical system based on the Bloch equations [8], as shown in Figure 2.1 [20]. The spins of each voxel are aggregated to create a net magnetization that at first points towards the scanner's main magnetic field, and then evolves based upon intrinsic biophysical tissue parameters, and user controls which involve the use

[1] S. Rakshit, K. Wang, and J. I. Tamir, "A GPU-accelerated Extended Phase Graph Algorithm for differentiable optimization and learning," in *Proceedings of the 2021 ISMRM & SMRT Annual Meeting & Exhibition, Online*, vol. 4819, 2021.
Here, the author was involved in parallelizing the non-parallelized simulator, developing and running experiments, and writing the abstract.

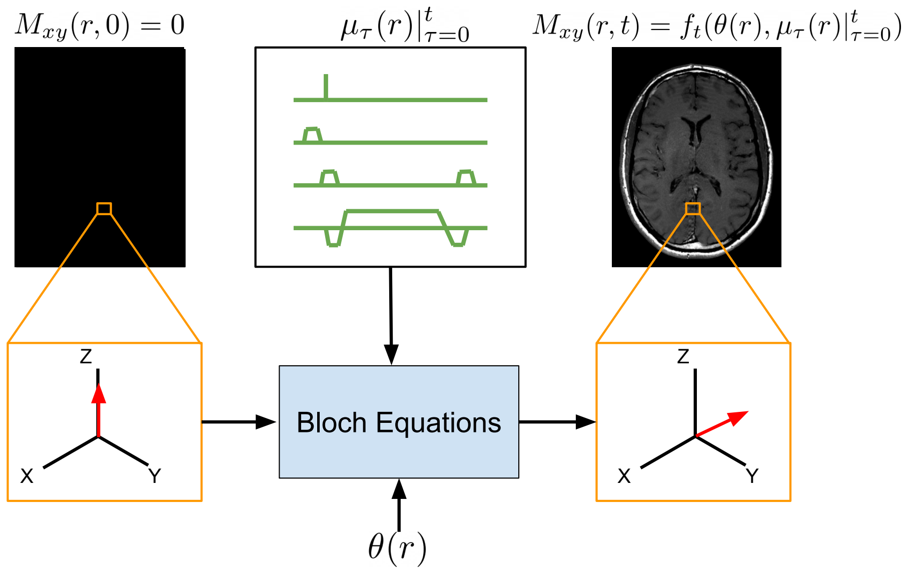

$M_{xy}(r,0) = 0$
$\mu_\tau(r)|_{\tau=0}^{t}$
$M_{xy}(r,t) = f_t(\theta(r), \mu_\tau(r)|_{\tau=0}^{t})$
Z
X
Y
Bloch Equations
Z
X
Y
$\theta(r)$

Figure 2.1: The MRI dynamical system as approximated by the Bloch equations. A net magnetization is created by the aggregation of spins at each voxel position. This initially points to the longitudinal direction and evolves based on user pulse sequence control inputs and intrinsic tissue parameters. The acquired image is the transverse component of the magnetization.

of RF pulses and magnetic field gradients. The direction of the magnetization vector changes from longitudinal field direction into the transverse plane due to the RF pulses. The degree of rotation of the magnetization vector, or flip angle, are determined by RF pulses depending on their strength and duration. Nearby receive coils sense the magnetization's transverse component through Faraday's Law of Induction.

There are many factors that influence tissue signal evolution, but here, we focus on relaxation parameters, the type of contrast mechanism most com-

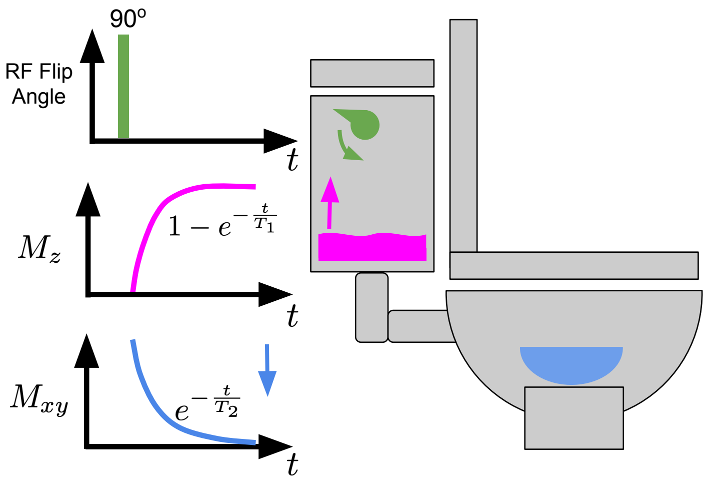


Figure 2.2: Through the toilet analogy, signal relaxation is visualized. A toilet flush represents RF excitation, whereas the water in the tank represents longitudinal magnetization, and water in the bowl representing transverse magnetization. When the toilet is flushed (excitation), water transfers from the tank into the bowl, generating a detectable signal. As water is drained out of the toilet bowl (T_2 relaxation), the water is refilled into the tank (T_1 recovery). After each flush, new water will be transferred from the tank to the bowl.

monly measured with MRI. MRI relies heavily on relaxation which determines the rate at which magnetization regains equilibrium, which is, in essence, resetting the system. Based on time constant T_1, longitudinal magnetization exponentially transforms to its initial state, and transverse magnetization becomes zero exponentially as a function of time constant T_2. A good analogy for relaxation is the toilet, where the water in the tank represents longitudinal magnetization, water in the bowl represents transverse magnetization, and the toilet flush represents RF excitation, as shown in Figure 2.2 [20]. On flushing the toilet, water transfers from the tank to the bowl, generating a detectable signal. As water is drained out of the toilet bowl (T_2 relaxation), the water is refilled into the tank (T_1 recovery). After each flush, new water will be transferred from the tank to the bowl. Ignoring other system effects, the magnetization evolution after a 90° RF pulse is given by:

$$M_z(t) = 1 - e^{-\frac{t}{T_1}} \tag{2.1}$$

$$M_{xy}(t) = e^{-\frac{t}{T_2}} \tag{2.2}$$

where $M_z(t)$ and $M_{xy}(t)$ are the longitudinal and transverse magnetization components respectively at time t after the excitation, and have initial conditions of $M_z(0) = 0$ and $M_{xy}(0) = 1$ immediately following the RF pulse. Even though there are many other tissue parameters and different other factors that control the magnetization, here, we will focus on Fast Spin Echo (FSE) imaging. Here, the magnetization is sampled at T equidistant intervals with distance TE. In FSE imaging, the magnetization is largely affected by the relaxation parameters, i.e., $\theta = (T_1, T_2)$, and RF refocusing flip angles. It is seen that spatially, the flip angles vary smoothly due to RF transmit field inhomogeneity effects that give rise to varying levels of RF power across the imaging volume. Different types of image contrasts can be created using various sequence timings and flip angles. Using different sequence parameters, four common types of image contrasts primarily due to PD, T_1, and T_2 are shown in Figure 2.3 [20].

Vector of magnetization points at the echo times can be modeled by solving the Bloch equations [8]. These are differential equations which, given

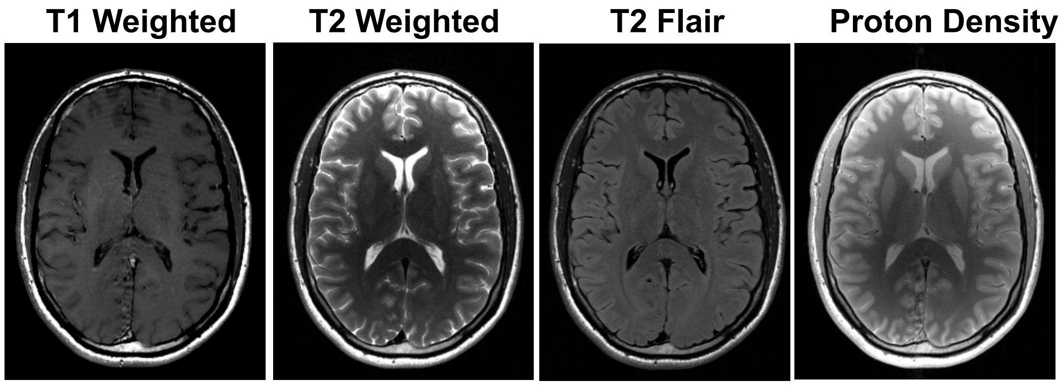
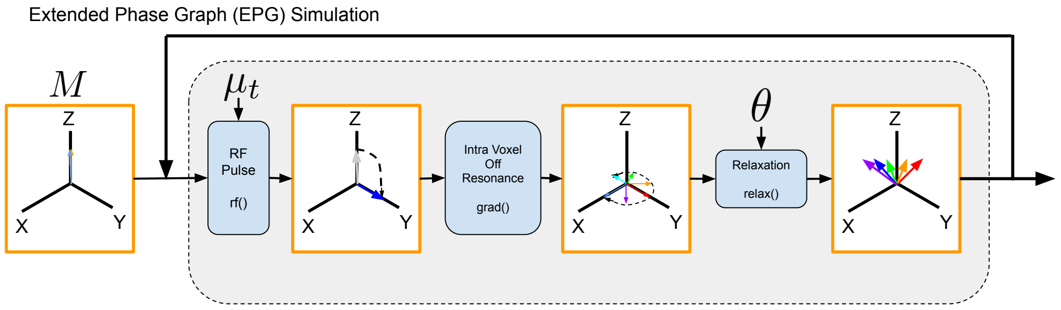


Figure 2.3: Various image contrasts based on T_1, T_2, and Proton Density generated by different RF flip angles and sequence timings for FSE-based scans.

pulse sequence inputs, makes it possible to calculate the magnetization evolution for individual spins in a spatial region. Additional contrast mechanisms such as diffusion and chemical exchange can be modeled though they are beyond our scope here.

2.1.2 Simulating Spins

The simulation of spin magnetization evolution can be a powerful tool for developing and evaluating physics constrained reconstruction methods. Bloch equations are linear differential equations and they can be efficiently computed through numerical solutions through approximation. Specifically, with discrete RF pulses, spin can be simulated using Bloch equations successively applying rotation, followed by relaxation to the magnetization vector. The simulation process is illustrated in Figure 2.4 [20]. Although the overall signal evolution is non-linear, it is differentiable with respect to the relaxation



Extended Phase Graph (EPG) Simulation
M
μ_t
RF Pulse
rf()
Intra Voxel Off Resonance
grad()
θ
Relaxation
relax()
Z
X
Y

Figure 2.4: Using discrete RF pulses, spin simulation can be performed using Bloch equations by successive applications of rotation followed by relaxation to the magnetization vector M. The extended phase graph (EPG), which simulates the signal evolution of a distribution of spins across a voxel, is another alternative to the Bloch equations.

and other parameters.

In real-world imaging scenarios, multiple resonant frequencies are frequently present within each voxel because of local magnetic field inhomogeneities. For each resonant frequency that appears within one voxel, the Bloch equations must be solved separately and then added together to obtain the resultant signal. Due to the need to simulate hundreds to thousands of isochromats in one voxel, this calculation can be computationally expensive.

The extended phase graph (EPG) Algorithm, which simulates the signal evolution of a distribution of spins across a voxel, is an alternative to the Bloch equations [3]. With EPG, resonant frequencies are uniformly discretized within one voxel. Using the Fourier series, EPG can keep track of signal evolution over multiple resonant frequencies utilizing the discretization. Using the EPG, spin simulation involves successive rotations followed by relaxations to the magnetization vector, similar to the Bloch equations. Signal evolution using EPG is also nonlinear, but differentiable.

2.2 Auto-differentiation

The evaluation of derivatives has been one of the core requirements of many machine learning methods whereas most of the traditional learning algorithms compute derivatives and Hessians of an objective function analytically [21]. Computing the derivatives of complex functions manually is error-prone and time consuming. Other alternatives like numerical differentiation can be easy to implement but they can be quite slow and inaccurate due to rounding errors [22]. Also, it does not scale well for high-dimensional gradients, making it a poor choice for models with a large number of parameters. Symbolic differentiation addresses these issues, but often results in complex expressions leading to the problem of "expression swell" [23].

Automatic differentiation (AD) performs a "non-standard interpretation of a given computer program by replacing the domain of the variables to incorporate derivative values and redefining the semantics of the operators to propagate derivatives per the chain rule of differential calculus" [24]. Although AD has been used in other fields widely, its application in machine learning was not popular until recently. Following the advent of deep learning [25, 26]

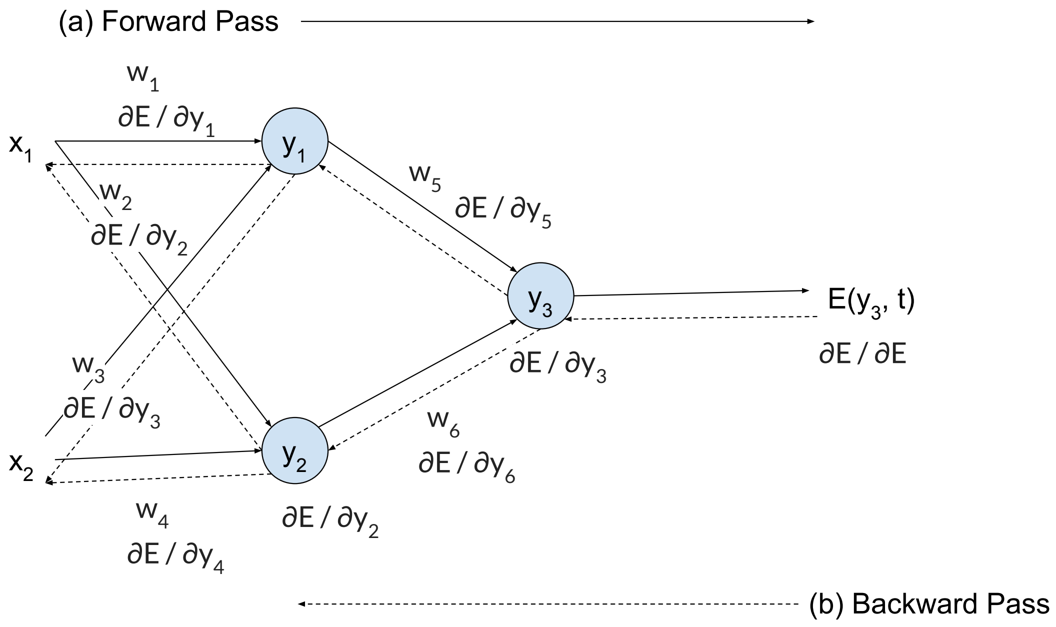


Figure 2.5: Overview of backpropagation. (a) By feeding forward the training input x_i, corresponding activations y_i are generated. Error E is computed as the difference between the actual output y_3 and the target output t. (b) By propagating the error adjoint backwards, we get the gradient with respect to the weights $\nabla_{w_i} E = \left(\frac{\partial E}{\partial w_1}, ..., \frac{\partial E}{\partial w_6}\right)$, the result of which is then utilized in a gradient-descent procedure. It is also possible to compute the gradient with respect to inputs $\nabla_{x_i} E$ within the same backward pass.

as state-of-the-art in many learning tasks, and current workflow trends that focus on reusing code and rapid prototyping in frameworks like Theano [27], Torch [28], Tensorflow [29] and Pytorch [30] have made general-purpose auto-differentiation methods widely accessible.

Presence of the term "automatic" in AD can lead to machine learning practitioners to put the label of AD in any method that does not involve the computation of derivatives manually. However here, we use AD to refer to those methods that compute derivatives by storing values during the execution of code to generate numerical derivative evaluations, and not derivative expressions. It gives an ideal asymptotic efficiency with a small constant factors of overhead for evaluating derivatives at machine precision. With AD, it is possible to apply differentiation without significant effort, allowing for branching, looping, and recursion, compared to the effort of composing closed-form expressions under symbolic differentiation. Due to its generality, AD has found broad applications in diverse fields such as engineering design optimization [31, 32], optimal control [33], and computational fluid dynamics [34, 35, 36].

Backpropagation is a specialized counterpart of AD used in machine learning for training neural networks, and has been independently re-invented by multiple researchers [37, 38]. As a result of the work of Rumelhart et al. [39], it has been one of the most studied and widely used training algorithms. Deep learning with backpropagation involves finding the minimum of an objective function from the weight space of deep neural networks. The required derivative can be calculated utilizing the chain rule to determine the partial derivative of the objective with respect to each weight via backward propagation of its sensitivity, as shown in Figure 2.5 [24].

2.3 Unrolled Least Squares Solver

Here, we show how we can use an unrolled least squares solver to estimate the optimal flip angles, and relaxation parameter T_2. Pytorch Lightning [40] was chosen for this purpose since it removes the need to write boilerplate code while making a wide variety of tasks easy to write. We first estimate the relaxation parameter T_2 using an unrolled least squares solver with gradient descent. We then use the estimated value of T_2 to optimize the flip angles with an additional gradient descent. In this way, we include a "double gradient descent," in which the gradient of the least squares solver with respect to the flip angles is automatically computed through AD [11]. However, this process has a number of challenges. For example, since the least-squares is non-convex, the T_2 vector has to be initialized so that it is within a small neighborhood of the true relaxation values. Otherwise, the training time will be prohibitive and potentially not converge. Also, to simulate real world conditions, it is essential to train on signals that contain noise. Finally, to incorporate the safety limits on RF heating, we added a constraint for Specific Absorption Rate (SAR). These details are discussed in detail in this chapter.

2.3.1 Creating the Dictionary

Before training, the T_2 vector has to be initialized in a way that it is close to its true value. This will help in quick convergence using gradient descent since less number of epochs are needed to reach the true value. Otherwise, if the T_2 vector is initialized to a constant value, some of the elements in the vector whose true value is close to the said constant will converge quickly whereas others may not converge in the same number of epochs. To solve this problem, we first estimate the T_2 values through a dictionary fitting procedure [41]. Specifically, we create a vector of 100 equispaced T_2 relaxation values between 10 and 250 milliseconds. Next, we use the given flip angles and this set of T_2 values to generate a set of 100 signals. This set of 100 signals is referred to as the dictionary. The code for this purpose is shown in the `on_batch_start()` function in Listing 2.1. Next, we initialize the predicted T_2 vector by calling the `get_myt2_init()` function where we pass the newly created dictionary and the true signal values as target. The distance vector is then created by adding the difference in the signal values and target values across all signals, and then taking its square. Finally, the indices which give the minimum distance are selected and these indices are used to select the T_2 values which lie closest to the true signal. These T_2 values are used to initialize the predicted T_2 vector.

```python
def get_myt2_init(self, dictionary, target):
    with torch.no_grad():
        distances = torch.sum((dictionary - target[..., None])
    ** 2, dim=1)
        min_index = torch.argmin(distances, dim=1)
        return self.pre_t2[min_index]

def on_batch_start(self, batch) -> None:
    with torch.no_grad():
        t1, t2, target = batch
        nums = torch.tensor(
            np.linspace(10, 250, self.config["pre_signals"]),
    dtype=torch.float32
        )
        self.pre_t2 = nums
```

Listing 2.1: Code to create the signal dictionary.

```
        self.pre_t2 = nums
        new_t1 = torch.cat(self.config["pre_signals"] * [t1]).
    type(torch.float32)
        new_t1 = t1.type(torch.float32)
        self.pre_signals = FSE_signal_TR(
            self.theta_hat,
            self.TE,
            self.config["TRs"],
            T1=new_t1.to(self.theta_hat.device),
            T2=nums.to(self.theta_hat.device)).T
```

Listing 2.1 (continued): Code to create the signal dictionary.

2.3.2 Applying a Specific Absorption Rate Constraint

To account for FDA limits on RF heating, we added a constraint for SAR. SAR or Specific Absorption Rate describes the potential for heating of the subject's body due to the application of the RF pulses needed for MRI scans. It is measured as the total power absorbed by unit mass of the body. Due to high exposure to these pulses, detrimental hot spots may occur in the tissue. Therefore, in most countries MRI systems are capped to a maximum SAR of 4 W/kg. The optimal flip angle schedule that minimizes T_2 fitting error may not be suitable for practical MRI scans since SAR is proportional to the sum of the squares of the flip angles. Hence, we added a constraint by optimizing the loss function while capping the sum of squares at a limit, say A. Assuming a flip angle sequence of T flip angles where each flip angle is equal to θ_k, we know that,

$$SAR \propto \sum_{k=1}^{T} (\theta_k)^2 \tag{2.3}$$

Considering a loss equal to L, estimated T_2 as $E(T_2)$, this is equivalent to the following optimization problem:

$$\begin{aligned} &\min_{\theta} L(\theta, T_2) \\ &\text{s.t. } SAR \leq A. \end{aligned} \tag{2.4}$$

Here, $\theta = (\theta_1, \ldots, \theta_T)$. Equation 2.4 is reflected in Listing 2.2. Note that we also limit the minimum and maximum flip angle value to $\theta_k \in [1, 179]$ so that the flip angles remain within a suitable range.

```python
def on_before_zero_grad(self, optimizer):
    self.theta_hat.data = torch.clamp(
        self.theta_hat.data, min=1 * np.pi / 180, max=179 * np.
   pi / 180
    )
    s = torch.pow(self.theta_hat, 2).sum(dim=1)
    A = self.config["A"]
    mask = s > A
    B = self.config["A"] / s
    B[~mask] = 1.0
    self.theta_hat.data[mask] = torch.transpose(
        self.theta_hat.data[mask].T * torch.sqrt(B[mask]), 0,
   1)
```

Listing 2.2: Code used to apply the Specific Absorption Rate (SAR) constraint.

2.3.3 Least Squares Solver

The least squares MRI signal decoder is where the T_2 value is estimated. First, the predicted T_2 vector (`myt2`) is initialized as shown in Section 2.3.1. Then, for a fixed number of iterations, MRI signals are generated using `myt2`. The loss is then calculated by taking the sum of the squared error. Using the gradients of this loss with respect to `myt2`, gradient descent is performed. The final value of `myt2` obtained after all iterations are complete is the estimated value of T_2 parameter. We also calculate the Cramer Rao Lower Bound

(CRLB) value using the given flip angles and the estimated value of T_2, which is then used as a performance metric. The gradients of the least squares solver and the CRLB are automatically calculated by our simulator using AD as described in Section 4.3. The code used to compute the CRLB value using the given flip angles and the estimated value of T_2 is shown in Listing 2.3, whereas the code present inside the MRI least squares solver that is used to estimate the value of T_2 is shown in Listing 2.4.

```python
import torch
from mri-sim-py.epg import epg_parallel


def FIM_T2_and_M0(angles_rad, TE, TRs, M0, T1, T2):
    def myfun(m0, t2):
        return m0[:, None, None] * epg_parallel.FSE_signal_TR(
    angles_rad, TE, TRs, T1, t2)

    batch_size = T2.shape[0]
    dZ = torch.autograd.functional.jacobian(myfun, (M0, T2),
    create_graph=True)
    dSdM0 = dZ[0]
    dSdT2 = dZ[1]
    F1 = torch.sum(dSdT2 ** 2, dim=1)
    F2 = torch.sum(dSdT2 * dSdM0, dim=1)
    F3 = torch.sum(dSdM0 ** 2, dim=1)
    FIM = torch.zeros([batch_size, 2, 2], device=T2.device)
    for i in range(batch_size):
        FIM[i, 0, 0] = F1[i, 0, i]
        FIM[i, 0, 1] = F2[i, 0, i]
        FIM[i, 1, 0] = F2[i, 0, i]
        FIM[i, 1, 1] = F3[i, 0, i]
    return FIM


def CRLB_T2(angles_rad, TE, TR, M0, T1, T2):
    return torch.inverse(FIM_T2_and_M0(angles_rad, TE, TR, M0,
    T1, T2))[:, 0, 0]
```

Listing 2.3: Code used to calculate the CRLB value using the given flip angles and the estimated value of T_2.

```
def least_sqaure_dec(s, theta_hat, TE, T, T1, is_test=False):
    if is_test:
        torch.train()
    myt2 = (
        self.get_myt2_init(self.pre_signals, s)
        .to(s.device)
        .requires_grad_()
    )
    for i in range(config["num_iters"]):
        f = FSE_signal_TR(theta_hat, TE, config["TRs"], T1, myt2).reshape(
            T1.shape[0], -1
        )
        l = (f - s).pow(2).sum()
        loss_grad = grad(outputs=l, inputs=myt2, create_graph=not is_test)
        myt2 = myt2 - self.step_size * loss_grad[0]
    M0 = torch.tensor([1.0], dtype=torch.float32, device=myt2.device)
    self.CRLB = CRLB_T2(theta_hat, TE, config["TRs"], M0, T1, myt2)
    if is_test:
        torch.eval()
    return myt2
```

Listing 2.4: Code present inside the MRI least squares signal decoder.

2.3.4 Adding Noise to the Signals

To emulate real-world scanning, we add noise to our true signals before they are used in estimation of flip angles and T_2 parameter in order to emulate a particular signal-to-noise ratio (SNR). To simulate an SNR of 35 dB, we create a random normal vector with mean = 0, and standard deviation = $\frac{\sqrt{10^{-35}}}{10}$. This noise is then added to the true value of the signals. The code used for this purpose is shown in Listing 2.5.

```
def forward(self, t1, t2, is_test=False):
    s = FSE_signal_TR(self.theta_hat, self.TE, self.config["TRs
"], T1=t1, T2=t2)
    s = s.view(-1, self.T)
    with torch.no_grad():
        std = 1.0 / np.sqrt(np.power(10, self.config["snr"] /
10))
        noise = (
            torch.distributions.normal.Normal(loc=0, scale=std)
            .sample(s.shape)
            .to(device=s.device)
        )
        s.add_(noise)
    x = self.dec(s, self.theta_hat, self.TE, self.T, t1,
is_test)
    return x
```

Listing 2.5: Code used to add noise to the signals before passing to the least sqaures decoder.

2.3.5 Loss Functions

After the estimated T_2 is returned by the least squares decoder, we then calculate the mean squared loss between the true T_2 and the estimated T_2. This loss is then used for backpropagation to update the value of flip angles. The mean squared loss (MSE) is represented by Equation 2.5.

$$MSE = \frac{1}{n}\sum_{k=1}^{n}\left(T_2^{(k)} - \hat{T_2}^{(k)}\right)^2 \tag{2.5}$$

Here, $T_2^{(k)}$ is the true value and $\hat{T_2}^{(k)}$ is the estimated value of the k^{th} sample in the batch, and n is the batch size. The code used for this purpose is shown in Listing 2.6.

```python
def custom_loss(self, pred, real, epsilon=1e-2):
    return (pred - real).pow(2).mean()
```

Listing 2.6: Code used to define the loss function.

2.3.6 Optimizing T_2 in Addition to Flip Angles

The `training_step()` function is called once for every batch in every epoch. In this function, the current batch passed through the MRI signal decoder, and then T_2 and $\hat{T_2}$ are used to compute the loss which is then used for backpropagation to update the flip angles. The code used for this purpose is shown in Listing 2.7.

```python
def training_step(self, train_batch, batch_idx, optimizer_idx=
    None):
    t1, t2, target = train_batch
    t1, t2 = t1.type(torch.float32), t2.type(torch.float32)
    self.t2_pred = self(t1, t2, is_test=False)
    self.t2_true = t2
    self.loss = self.custom_loss(self.t2_pred, self.t2_true)
    self.batched_loss = self.loss.mean()
    logs = {"train_loss": self.batched_loss}
    return {"loss": self.batched_loss, "log": logs}
```

Listing 2.7: Code present inside the outer training function.

2.4 Solving using a Neural Network

Here, we show how we can use a neural network to optimize the flip angles and estimate relaxation parameters T_1 and T_2. Neural networks are powerful approximation models which consist of neurons and weights. A neuron or edge typically has a weight that adjusts with learning. The magnitude of weight leads to a stronger or weaker connection signal to the corresponding signal. A neuron might have a threshold so that an aggregate signal crosses a certain threshold before a signal is sent. Neurons tend to be divided into layers that perform different transformations on their inputs. There are multiple layers in a model. Signals pass from the input layer to the output layer, possibly after traversing them multiple times. One such model is shown in Figure 2.6. This model consists of two components–the MRI simulator, and the MRI signal decoder. The simulator takes flip angles as inputs and generates MRI signal values as output through EPG, as described in Section 3.1. The MRI decoder consists of a feed-forward neural network and has multiple layers. The number of nodes in the final layer is equal to the number of parameters to be estimated. In this case, since we are estimating T_1 and T_2 parameters, we have two nodes in the final layer of the MRI signal decoder.

Since the simulator is completely differentiable, it can be combined with other commonly used differentiable layers in Pytorch to test different neural network architectures, and we can still use backpropagation to update the model parameters using AD. This allows to incorporate the latest innovations in machine learning and artificial intelligence. In this model, error E is calculated as the sum of the mean squared error between the true and predicted values of T_1 and T_2 respectively. By propagating this error backwards, we get the gradients of error with respect to the weights. These gradients can be used by various optimizers such as Adam [42], Gradient Descent [43], Adagrad [44], RMSProp [45], etc. Here, the weights include that of the MRI signal decoder,

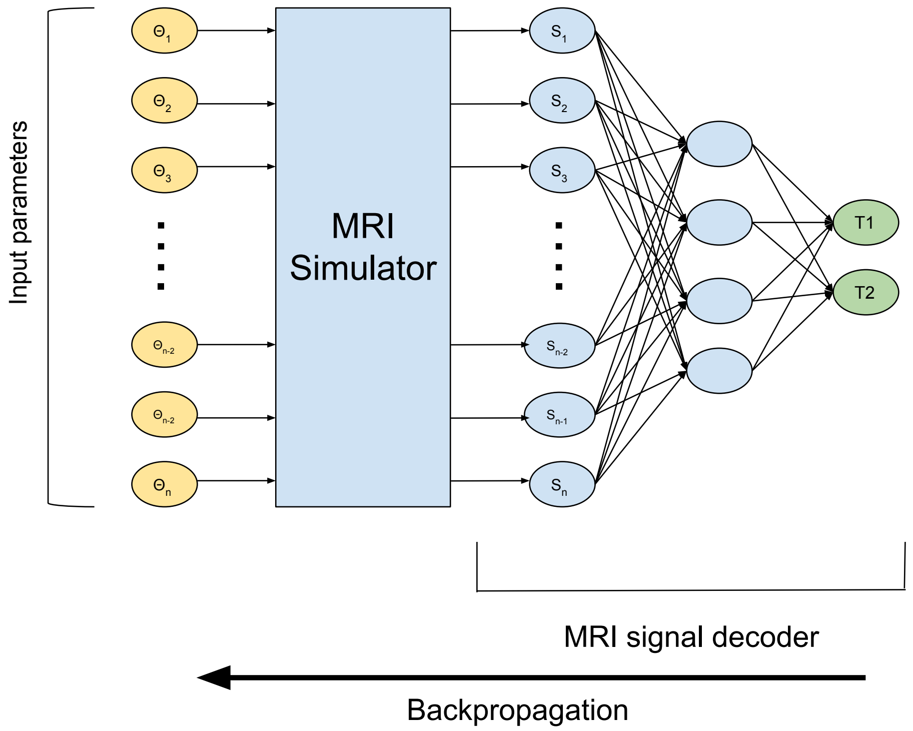


Figure 2.6: Overview of the neural network model that incorporates MRI physics using the EPG simulator to estimate T_1 and T_2. The flip angles are passed as input parameters. The EPG simulator then takes these flip angles to estimate signal values which are then used by the MRI signal decoder to estimate the values of T_1 and T_2.

as well as the flip angles, which are used as input parameters. Since the flip angles are also included in the model weights, they are also optimized while training the model yielding the optimized set of flip angles after completing the training.

Chapter 3

Toolbox

This chapter outlines the different components of our EPG simulator[2]. We discuss how we parallelized the simulator within Pytorch, as well as how we used the auto-differentiation feature to obtain gradients, which can then be used to estimate both tissue parameters such as T_2, as well as scan parameters such as RF flip angles.

3.1 Software Design

Python was chosen as it can be demonstrated easily by means of Jupyter notebooks and interfaced with PyTorch, a popular machine learning auto-differentiation framework. The EPG Algorithm has three components: RF excitation, gradient application, and tissue relaxation. Figure 3.1 shows how these three components of the EPG algorithm are organized.

[2]S. Rakshit, K. Wang, and J. I. Tamir, "A GPU-accelerated Extended Phase Graph Algorithm for differentiable optimization and learning," in *Proceedings of the 2021 ISMRM & SMRT Annual Meeting & Exhibition, Online*, vol. 4819, 2021.
Here, the author was involved in parallelizing the non-parallelized simulator, developing and running experiments, and writing the abstract.

```python
import numpy as np
import torch
from mri-sim-py.epg import epg_parallel

# Initializing hyperparameters
T = 32
TE = 9
TRs = np.array([860, 1830, 2800])
T1_vals = np.linspace(500, 1000, 100)   # 100 elements
T2_vals = np.linspace(20, 500, 100)     # 100 elements
device = torch.device('cuda'
                      if torch.cuda.is_available() else 'cpu')

# Ensure both T1_vals and T2_vals have the same
# number of elements
assert T1_vals.shape[0] == T2_vals.shape[0]

# Convert T1_vals and T2_vals to their corresponding tensors
T1 = torch.tensor(T1_vals, dtype=torch.float32, device=device)
T2 = torch.tensor(T2_vals, dtype=torch.float32, device=device)

# Convert angles from degrees to radians
angles_rad = torch.ones([1, T], device=device)*170./180.*np.pi

# Run EPG simulator
sig = epg_parallel(angles_rad, TE, TRs, T1, T2, B1=1.)
# Returns a tensor with size = torch.Size([100, 32, 1])
```

Listing 3.1: Code snippet demonstrating how to generate MRI signals from flip angles, and T_2 as input parameter.

A code snippet showing how the simulator can be used to generate FSE signals from flip angles, and T_1 and T_2 as input parameters is shown in Listing 3.1. Here, `T` is the length of flip angle train, `TE` is the echo time, `TRs` are the repetition times, and `T1_vals` and `T2_vals` are the list of T_1 and T_2 relaxation time values for which we want to find the MRI signal values using `angles_rad`, which is a tensor of size $1 \times T$ and contains the flip angles in radians. The major components present in the simulator are described below along with

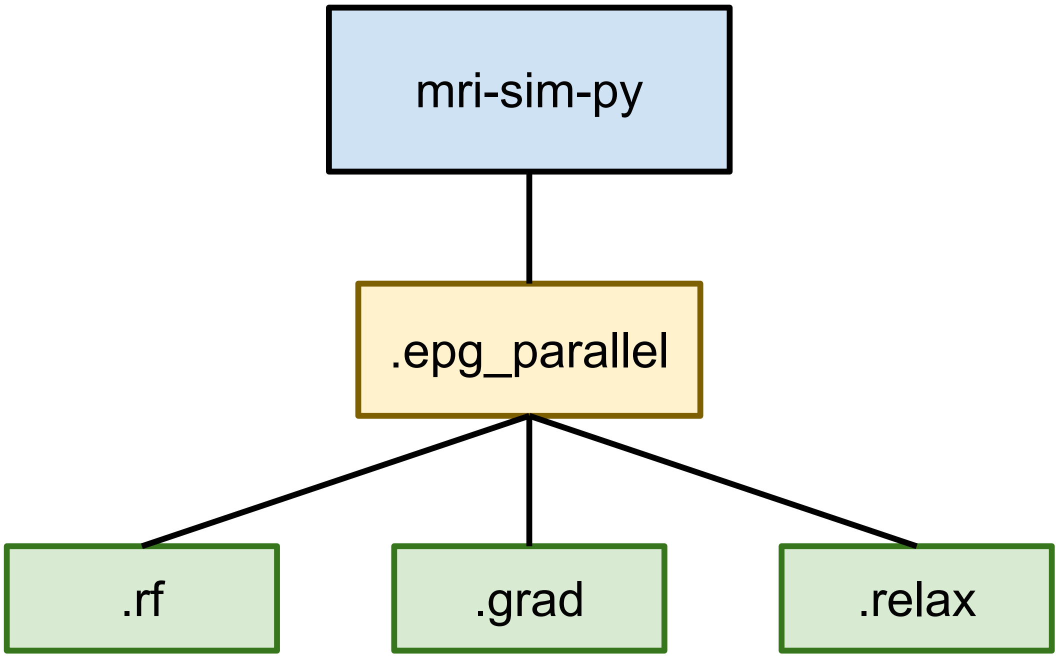


Figure 3.1: Components of the EPG simulator. Our simulator has three components: RF excitation, gradient application, and tissue relaxation.

their code. Note that other arbitrary sequences can also be simulated.

3.1.1 RF Excitation

During RF excitation, the EPG states are propagated through an RF rotation of α radians along the Y direction, i.e. phase of $\frac{\pi}{2}$. This function takes two parameters as input–`FpFmZ`, and `alpha`. Here, `FpFmZ` is an $N \times 3 \times (3T-1)$ shaped tensor of $F+$, $F-$ and Z EPG states [1] for all elements in the batch, whereas `alpha` is the RF pulse flip angle in radians. The updated `FpFmZ` tensor and `RR`, which is the RF rotation matrix of size 3×3 are returned from this function. The code for this function is present in Listing 3.2.

```python
def rf_ex(FpFmZ, alpha):
    "Same as rf2_ex, but only returns FpFmZ" ""
    return rf2_ex(FpFmZ, alpha)[0]


def rf2_ex(FpFmZ, alpha):
    """Propagate EPG states through an RF excitation of
    alpha (radians) along the y direction, i.e. phase of pi/2.
    in Pytorch
    INPUT:
        FpFmZ = N x 3 x (3T - 1) tensor of F+, F- and Z states
                for all elements in the batch.
        alpha = RF pulse flip angle in radians

    OUTPUT:
        FpFmZ = Updated FpFmZ state.
        RR = RF rotation matrix (3x3).

    """

    try:
        alpha = alpha[0]
    except:
        pass

    if torch.abs(alpha) > 2 * np.pi:
        warn("rf2_ex: Flip angle should be in radians! alpha=%f
   " % alpha)

    cosa2 = torch.cos(alpha / 2.0) ** 2
    sina2 = torch.sin(alpha / 2.0) ** 2

    cosa = torch.cos(alpha)
    sina = torch.sin(alpha)
    RR = torch.tensor(
        [
            [cosa2, -sina2, sina],
            [-sina2, cosa2, sina],
            [-0.5 * sina, -0.5 * sina, cosa],
        ],
        device=alpha.device,
    )
```
Listing 3.2: Code present inside `rf2_ex()` function.

```python
    FpFmZ = torch.matmul(RR, FpFmZ)

    return FpFmZ, RR
```
Listing 3.2 (continued): Code present inside `rf2_ex()` function.

3.1.2 Gradient Application

The `grad()` function is where gradient application takes place. This function propagates EPG states through a “unit” gradient pulse. The `grad()` function also assumes CPMG condition [3], i.e. all states are real-valued. It takes one parameter, i.e., `FpFmZ` as input and returns the updated `FpFmZ` state, where `FpFmZ` has been described already in Section 3.1.1. The code for this function is shown in Listing 3.3. Here, the $F+$ and $F-$ states are shifted and then the highest $F-$ state is set equal to 0 for the entire batch of elements. Finally, the first element of the matrix is set equal to the next diagonal element before returning the updated state of `FpFmZ`.

```python
def grad(FpFmZ):
    """Propagate EPG states through a "unit" gradient. Assumes
   CPMG condition,
    i.e. all states are real-valued.

    INPUT:
        FpFmZ = N x 3 x (3T - 1) tensor of F+, F- and Z states
                for all elements in the batch.

    OUTPUT:
        Updated FpFmZ state.

    """
    x = FpFmZ.clone()  # required to avoid in-place memory op
    FpFmZ[:, 0, 1:] = x[:, 0, :-1]  # shift Fp states
    FpFmZ[:, 1, :-1] = x[:, 1, 1:]  # shift Fm states
    FpFmZ[:, 1, -1] = 0  # Zero highest Fm state
```
Listing 3.3: Code present inside `grad()` function.

```
    FpFmZ[:, 0, 0] = FpFmZ[:, 1, 0]

    return FpFmZ
```

Listing 3.3 (continued): Code present inside `grad()` function.

3.1.3 Tissue Relaxation

The `relax()` function is used to propagate EPG states through a period of relaxation over an interval T. It takes three parameters as input. They are an $N \times 3 \times (3T - 1)$ shaped tensor of $F+$, $F-$ and Z states for all elements in the batch (`FpFmZ`), the decay matrix (`EE`), and the RF rotation matrix (`RR`). The updated $F+$, $F-$ and Z states of the batch of elements are returned as output. The code for this function is shown in Listing 3.4. This function first performs a matrix products of `EE` and `FpFmZ`. Then, the first element of the last row for each batch in `FpFmZ` is added with `RR`. Finally, the new `FpFmZ` tensor is returned.

```
def relax(FpFmZ, EE, RR):
    """Propagate EPG states through a period of relaxation over
    an interval T.
    torch
    INPUT:
        FpFmZ = N x 3 x (3T - 1) tensor of F+, F- and Z states
                for all elements in the batch.
        EE = decay matrix, 3x3 = diag([E2 E2 E1]);
        RR = RF rotation matrix (3x3).

    OUTPUT:
        FpFmZ = updated F+, F- and Z states.

    """
    FpFmZ = torch.matmul(EE, FpFmZ)  # Apply Relaxation
    FpFmZ[:, 2, 0] = FpFmZ[:, 2, 0] + RR  # Recovery
```

Listing 3.4: Code present inside `relax()` function.

```python

    return FpFmZ
```
Listing 3.4 (continued): Code present inside `relax()` function.

3.1.4 RF Rotation

The `rf()` function is used to perform a RF rotation. It propagates EPG states through an RF rotation of alpha (radians), which are present for the entire batch in matrices. `rf()` function assumes CPMG condition, i.e. magnetization lies on the real X axis [3]. It takes in two parameters as input: `matrices`, which is the output of the `get_precomputed_matrices()` function, and `FpFmZ`, which has been described earlier. Inside this function, a matrix multiplication of `matrices` and `FpFmZ` is performed and its result is returned as output. The code for this function is shown in Listing 3.5.

```python
def rf(matrices, FpFmZ):
    """Propagate EPG states through an RF rotation of
    alpha (radians), which are present for the entire batch
    in matrices. Assumes CPMG condition, i.e.
    magnetization lies on the real x axis.
    """
    return torch.matmul(matrices, FpFmZ)
```
Listing 3.5: Code present inside `get_precomputed_matrices()` function.

3.1.5 Fast Spin Echo - Time to Echo

The `FSE_TE()` function is used to propagate EPG states through a full FSE echo time (`TE`), consisting of relaxation, RF refocusing, and gradient dephasing. It is called once for every refocusing flip angle in the flip angle train. This function assumes CPMG condition, i.e. all states are real-valued. The

FSE_TE() function takes five inputs which are FpFmZ, i, EE, RR, and matrices. Here, i is the position of the current flip angle for which FSE_TE() is being called, EE is the decay matrix, RR is the current rotation matrix, and matrices is the pre-computed rotation matrix for the entire batch for the given flip angle train. As output, this function returns the updated FpFmZ, consisting of $F+$, $F-$ and Z states. Inside this function, relax(), grad(), rf(), grad(), and relax() functions are called to update FpFmz, as shown in the code present in Listing 3.6.

```python
def FSE_TE(FpFmZ, i, EE, RR, matrices):
    """Propagate EPG states through a full TE, i.e.
    relax -> grad -> rf -> grad -> relax.
    Assumes CPMG condition, i.e. all states are real-valued.

    INPUT:
        FpFmZ = N x 3 x (3T - 1) tensor of F+, F- and Z states
                for all elements in the batch.
        i = Current flip angle
        EE = decay matrix, 3x3 = diag([E2 E2 E1]);
        RR = RF rotation matrix (3x3).
        matrices = Pre-computed rotation matrix for the entire
   batch
                     for the given flip angle train.

    OUTPUT:
        FpFmZ = updated F+, F- and Z states.

    """
    FpFmZ = relax(FpFmZ, EE, RR)
    FpFmZ = grad(FpFmZ)
    FpFmZ = rf(matrices[:, :, :, i], FpFmZ)
    FpFmZ = grad(FpFmZ)
    FpFmZ = relax(FpFmZ, EE, RR)

    return FpFmZ
```

Listing 3.6: Code present inside FSE_TE() function.

3.1.6 Pre-computing Rotation Matrices

The `get_precomputed_matrices()` function is used to pre-compute the rotation matrices for the entire batch of data. It is called once per batch of data. This function requires three inputs: `batch_size`, `alphas`, and `T`. Here, `batch_size` is an integer representing the number of T_1 and T_2 values that are processed in parallel, `alphas` represent the flip angles, and `T` is the length of the flip angle train. The rotation matrix for the entire batch is then returned from this function. The code inside this function is shown in Listing 3.7.

```python
def get_precomputed_matrices(batch_size, alphas, T):
    """
    Returns the rotation matrix for the entire batch
    for the given flip angle train.
    """
    cosa2 = torch.cos(alphas / 2.0) ** 2
    sina2 = torch.sin(alphas / 2.0) ** 2
    cosa = torch.cos(alphas)
    sina = torch.sin(alphas)

    RR = torch.zeros(batch_size, 3, 3, T, device=alphas.device)

    RR[:, 0, 0, :] = cosa2
    RR[:, 0, 1, :] = sina2
    RR[:, 0, 2, :] = sina

    RR[:, 1, 0, :] = sina2
    RR[:, 1, 1, :] = cosa2
    RR[:, 1, 2, :] = -sina

    RR[:, 2, 0, :] = -0.5 * sina
    RR[:, 2, 1, :] = 0.5 * sina
    RR[:, 2, 2, :] = cosa
    return RR
```

Listing 3.7: Code present inside `get_precomputed_matrices()` function.

3.1.7 Creating Decay Matrix

The `relax_mat()` function is used to pre-compute the decay matrix. It is also called once per batch of data. This function takes three parameters as inputs: `T`, `T1`, and `T2`, where `T` is the length of the flip angle train, and `T1` and `T2` are relaxation time parameters. A diagonal matrix is produced for the decay of states due to relaxation alone. This is done for the entire batch in parallel using the input parameters as shown in the code present at Listing 3.8.

```python
def relax_mat(T, T1, T2):
    """
    Creates and returns the decay matrix
    """
    E2 = torch.exp(-T / T2)
    E1 = torch.exp(-T / T1)

    # Decay of states due to relaxation alone.
    mat = torch.stack([E2, E2, E1], dim=1)
    EE = torch.diag_embed(mat)
    return EE
```

Listing 3.8: Code present inside `relax_mat()` function.

3.1.8 Generating Signals Through Fast Spin Echo

The simulator is first called using the `FSE_signal_TR()` function. This calls `FSE_signal2_TR()`, which then calls `FSE_signal2_TRs_ex()` function. `FSE_signal2_TRs_ex()` then calls `FSE_signal2_ex()` function where fast spin echo CPMG sequence is simulated with a specific flip angle train. The functions allow us to support multiple interfaces: a simple interface that assume 90 degree excitation and only return the transverse magnetization, and a richer interface that simulates arbitrary excitation and returns both the longitudinal and transverse magnetization components. First, an excitation pulse equal to `angle_ex_rad` is applied in the Y direction. Then, the flip angle train is applied in the X direction. Here, `angle_ex_rad` $= \frac{\pi}{2}$.

```python
# Full FSE EPG function across T time points
def FSE_signal_TR(angles_rad, TE, TRs, T1, T2, B1=1.0):
    """Same as FSE_signal2_TR, but only returns Mxy"""
    return FSE_signal2_TR(angles_rad, TE, TRs, T1, T2, B1)


def FSE_signal2_TR(angles_rad, TE, TRs, T1, T2, B1=1.0):
    """Same as FSE_signal2, but includes finite TR"""
    pi = torch.tensor(np.pi, device=T2.device)
    return FSE_signal2_TRs_ex(pi / 2, angles_rad, TE, TRs, T1, T2, 1.0)

def FSE_signal2_TRs_ex(angle_ex_rad, angles_rad, TE, TRs, T1, T2, B1=1.0):
    """Same as FSE_signal2_ex, but includes finite TR"""
    T = angles_rad.shape[1]
    Mxy, Mz = FSE_signal2_ex(angle_ex_rad, angles_rad, TE, T1, T2, B1)  # n x T x 1
    URs = torch.tensor(TRs - T * TE, device=angles_rad.device)
    E1 = torch.exp(-URs[None, :] / T1[:, None])[:, None, :] # n x 3 -> n x 1 x 3
    sig = torch.flatten(Mxy * (1 - E1) / (1 - Mz[:, -1, :][:, None, :] * E1), start_dim=1)[:, :, None] # n x T x 3 -> n x 3T x 1
    return sig

def FSE_signal2_ex(angle_ex_rad, angles_rad, TE, T1, T2, B1=1.0):
    """Simulate Fast Spin-Echo CPMG sequence with specific flip angle train.
    Prior to the flip angle train, an excitation pulse of angle_ex_rad degrees
    is applied in the Y direction. The flip angle train is then applied in the X direction.
    INPUT:
    """
```

Listing 3.9: Code used to generate FSE signals.

```python
    """
        angles_rad = array of flip angles in radians equal to echo train length
        TE = echo time/spacing
        T1 = T1 value in seconds
        T2 = T2 value in seconds

    OUTPUT:
        Mxy = Transverse magnetization at each echo time
        Mz = Longitudinal magnetization at each echo time

    """
    batch_size = T2.shape[0]
    T = angles_rad.shape[1]
    Mxy = torch.zeros((batch_size, T, 1), requires_grad=False, device=T2.device)
    Mz = torch.zeros((batch_size, T, 1), requires_grad=False, device=T2.device)
    P = torch.zeros((batch_size, 3, 2 * T + 1), dtype=torch.float32, device=T2.device)
    P[:, 2, 0] = 1.0
    try:
        B1 = B1[0]
    except:
        pass

    # pre-scale by B1 homogeneity
    angle_ex_rad = B1 * angle_ex_rad
    angles_rad = B1 * angles_rad

    P = rf_ex(P, angle_ex_rad)  # initial tip
    EE = relax_mat(TE / 2.0, T1, T2)
    E1 = torch.exp(-TE / 2.0 / T1)
    RR = 1 - E1
    matrices = get_precomputed_matrices(batch_size, angles_rad, T)
    for i in range(T):
        P = FSE_TE(P, i, EE, RR, matrices)
        Mxy[:, i, 0] = P[:, 0, 0]
        Mz[:, i, 0] = P[:, 2, 0]
    return Mxy, Mz
```

Listing 3.9 (continued): Code used to generate FSE signals.

Chapter 4

Experiments and Results

This chapter outlines the different experiments conducted using our developed simulator and results obtained from the said experiments. We discuss how our simulator is used to generate signals in parallel, estimate T_2, and estimate the optimal flip angle sequence using the Cramer Rao Lower Bound (CRLB).

4.1 Using the Software

Our simulator can be used to generate signals using different T_1 and T_2 for a multi-echo spin-echo sequence. This can be done in parallel with negligible impact to runtime. Figure 4.1 shows 100 signals simulated using various values of T_1 and T_2 for a multi-echo spin-echo sequence with 60-degree refocusing flip angles.

To compare the time taken by the parallelized simulator and the naive simulator running on CPU and GPU, we plot the evaluation time for different batch sizes. This plot is shown in Figure 4.2. We see that the time taken by the naive simulators increases linearly until batch size ~ 1000, and then shows

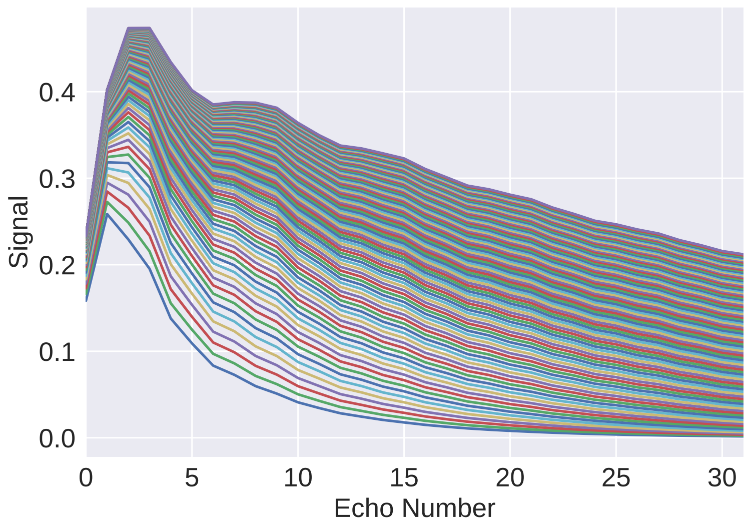


Figure 4.1: 100 signals simulated using various values of T_1 and T_2 for a multi-echo spin-echo sequence with 60-degree refocusing flip angles.

sub-linear increase till batch size = 30,000. Also, the parallelized simulator running on CPU increases linearly until batch size $\sim$ 1000 before showing a drop and then increasing sub-linearly. However, the parallelized simulator running on GPU shows negligible impact in evaluation time throughout the range of batch sizes taken here, i.e., until 30,000. In fact, we stopped at batch size = 30,000 since we filled all 32GB of the NVIDIA V100 Tensor Core GPU.

4.2 Estimating T_2

Here, we show an application to estimating T_2 relaxation. A volunteer was scanned under IRB approval and informed consent. A multi-echo spin-echo acquisition of the brain was acquired on a 3 Tesla Siemens Vida scanner. Scan

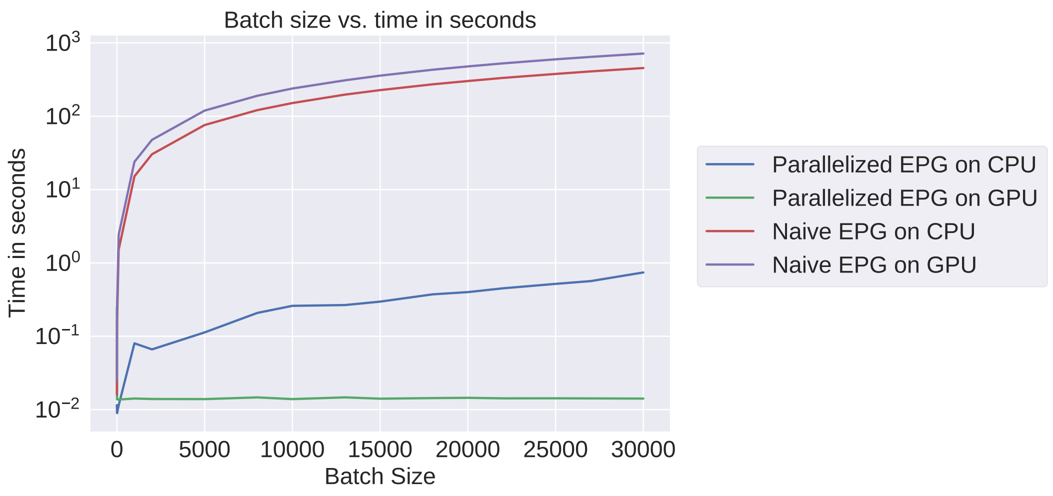


Figure 4.2: Comparison of the time taken by the naive and our parallelized EPG simulators running on CPU and GPU

parameters included: 32 echoes, TR=2800 ms, TE=9 ms, matrix size = 288×288, nominal refocusing flip angles of 105 degrees. Using the EPG simulator, a least-squares solver with gradient descent was implemented, taking advantage of automatic differentiation. Figures 4.3 (a) and 4.3 (b) show images of the resulting T2 map, and Proton Density map respectively, which was estimated in 543 seconds using batch size = 14,000, for a total number of 58 million signal evaluations. These results match with the prior literature [46].

4.3 Estimating Cramer Rao Lower Bound

The Cramer Rao Lower Bound is defined as the inverse of the Fisher Information, which provides a lower bound on the variance of any unbiased estimator [17, 18, 19]. It can be used as a statistical tool for estimating se-

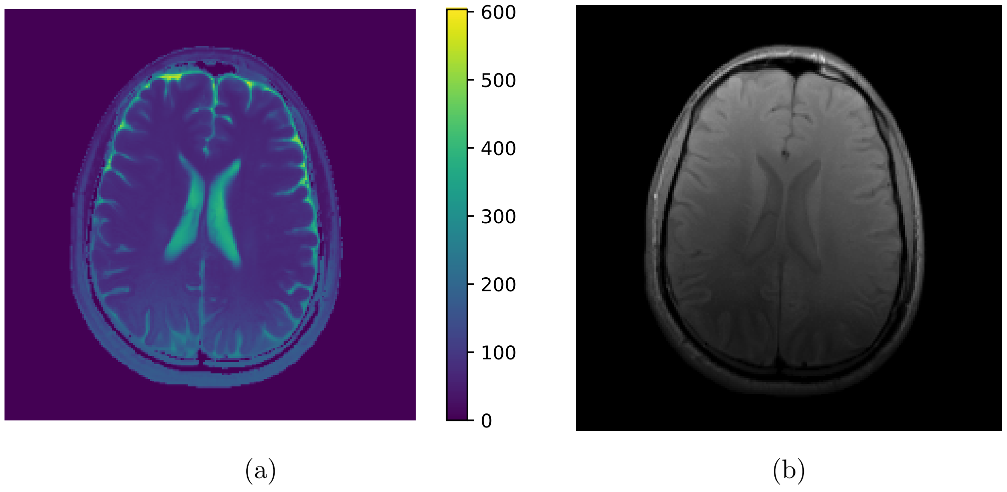
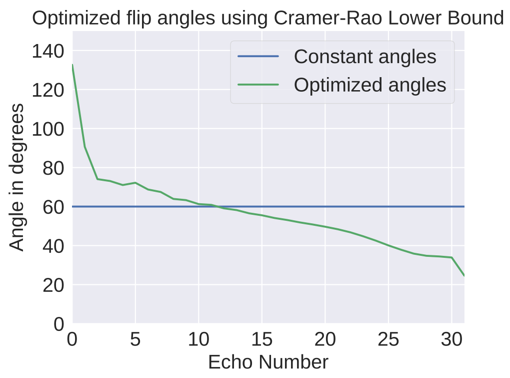
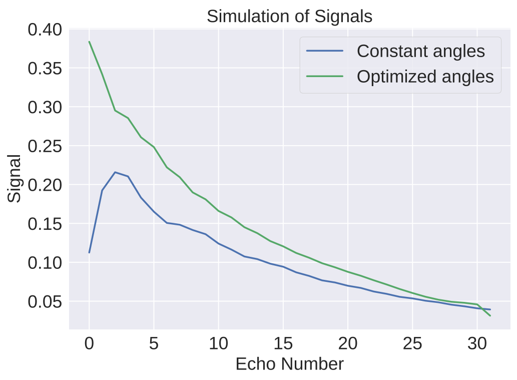


Figure 4.3: Applications of the EPG simulator to T_2 relaxation showing plots of the resulting (a) T_2 map in milliseconds, and (b) Proton Density map, in arbitrary units respectively.

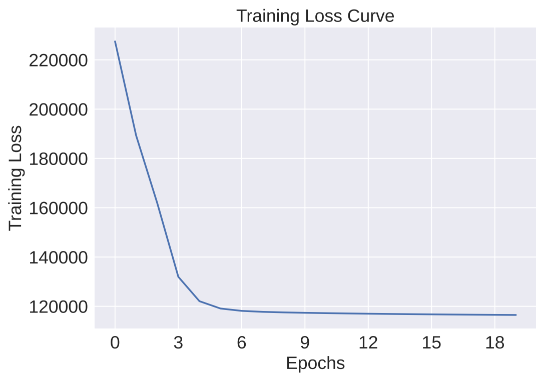
quence parameters to try to minimize the variance of the final estimate. By minimizing the variance with respect to the sequence parameters, we can find new sequence parameters that can give better estimates of tissue parameters. Here, we minimize the CRLB with respect to the sequence parameters subject to a SAR constraint as described in Equation 2.4 in Section 2.3.2. The CRLB is the loss represented by L in the above mentioned equation. Figure 4.4 (a) shows the constant flip angles equal to 60 degrees in blue, and the optimized flip angles in green. Note that they have the same SAR level. We also note that in our optimized sequence, the flip angles start large and then slowly decrease. Figure 4.4 (b) shows the signal simulation curves for the two sets of flip angles with $T_1 = 1000$ ms and $T_2 = 100$ ms. We note that the optimized flip angle produces higher signal value than the constant flip angles and



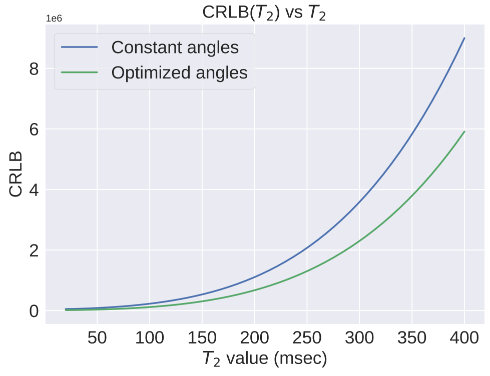
(a)



(b)



(c)



(d)

Figure 4.4: (a) The constant flip angle train equal to 60 degrees and the optimized flip angle train obtained using Cramer Rao Lower Bound. (b) The simulation of a particular signal using the constant and optimized flip angles with $T_1 = 1000$ ms and $T_2 = 100$ ms. (c) The loss curve generated while obtaining the optimized flip angle train. (d) Cramer Rao Lower Bound of constant flip angles vs. optimized flip angles over T_2 values ranging from 20 ms to 400 ms (lower is better).

therefore, produces higher SNR, which helps in estimating tissue parameters. Figure 4.4 (c) shows the loss curve during optimization. We note that the loss has converged to a local minimum after 20 epochs. Figure 4.4 (d) shows the CRLB of the different flip angle trains over T_2 values ranging from 20 to 400 ms. Note that the optimized flip angle train reduces the CRLB for all T2 values compared to the constant flip angle train, despite only being optimized for a single T_2 value = 100 ms.

Chapter 5

Conclusions and Future Direction

5.1 Summary

Physical modeling of signal dynamics and tissue parameters have been used since the advent of MRI. In early works, physical constraints were used to reduce systematic errors and derive quantitative maps that reduce image artifacts. Despite this, scan times are long and sophisticated reconstruction algorithms are lacking, which prevents clinical adoption of these techniques.

Over the last decade, computational imaging and compressed sensing have redefined what MRI is capable of. Natural image statistics can be leveraged to reduce scanning time appreciably. This purpose has been met with the development of many advanced reconstruction algorithms. It is now feasible to perform physics-constrained computational MRI in clinics with reasonable scan and reconstruction times using these numerical tools and combining sparsity-based modeling [47]. Incorporating physical dynamics resulting from both tissue-specific and scanner-specific features, reconstructions and acquisitions can be made to work robustly across an array of patient populations.

5.2 Future Directions

Although this thesis contains multiple innovations, further work can be done to build on this. Many topics discussed in this thesis provide exciting avenues for future research. Some of them are discussed here. This work focuses on applying auto-differentiation techniques to optimize MRI system parameters for more robust scanning and reconstruction. Our work brings together MRI physics with optimization and deep learning. Most available MRI simulators are extremely slow, and are not suitable for auto-differentiation. Our work reformulates the simulation operators and maintains massively parallel runtime on GPU. As a result, we are able to simulate 30,000 concurrent voxels in 0.01 seconds. This has the potential to open new opportunities in computational MRI with deep learning where fast large-scale optimization is crucial. To demonstrate the effectiveness of this method, we have applied our differentiable simulator to MRI reconstruction, pulse sequence design, and deep learning based regression.

To emulate more sophisticated simulation of MRI signals, more parameters can be added to this simulator such as diffusion, magnetization transfer, perfusion, flow, etc. More work can also be done in generating a graphical representation of the model components. This can help in easy visualization of the workflow and aid in better understanding of how different components of the model work together. We could also model other physical characteristics of the MRI sequence such as the envelope of the RF pulses. Spatial encoding [4] could also be included as an application, as we currently only demonstrate 1-D simulation here. This can ultimately be connected to full deep learning-based image reconstruction.

Bibliography

[1] J. Hennig, "Multiecho imaging sequences with low refocusing flip angles," *Journal of Magnetic Resonance*, vol. 78, no. 3, pp. 397–407, 1988.

[2] J. Hennig, "Echoes—how to generate, recognize, use or avoid them in MR-imaging sequences. Part I," *Concepts in Magnetic Resonance*, vol. 3, no. 3, pp. 125–143, 1991.

[3] M. Weigel, "Extended Phase Graphs: Dephasing, RF Pulses, and Echoes - Pure and Simple," *Journal of Magnetic Resonance Imaging*, vol. 41, no. 2, pp. 266–295, 2015.

[4] A. Loktyushin, K. Herz, N. Dang, F. Glang, A. Deshmane, S. Weinmüller, A. Doerfler, B. Schölkopf, K. Scheffler, and M. Zaiss, "MRzero–Automated discovery of MRI sequences using supervised learning," *Magnetic Resonance in Medicine*, 2021.

[5] P. K. Lee, L. E. Watkins, *et al.*, "Flexible and efficient optimization of quantitative sequences using automatic differentiation of Bloch simulations," *Magnetic Resonance in Medicine*, vol. 82, no. 4, pp. 1438–1451, 2019.

[6] N. Ben-Eliezer, D. K. Sodickson, and K. T. Block, "Rapid and accurate T2 mapping from multi–spin-echo data using Bloch-simulation-based recon-

struction," *Magnetic Resonance in Medicine*, vol. 73, no. 2, pp. 809–817, 2015.

[7] K. C. McPhee and A. H. Wilman, "Transverse relaxation and flip angle mapping: Evaluation of simultaneous and independent methods using multiple spin echoes," *Magnetic Resonance in Medicine*, vol. 77, no. 5, pp. 2057–2065, 2017.

[8] F. Bloch, "Nuclear Induction," *Physical Review*, vol. 70, no. 7-8, p. 460, 1946.

[9] B. A. Hargreaves, "Extended phase graph Matlab algorithm."

[10] J. Lamy and P. Loureiro de Sousa, "Sycomore: an MRI simulation toolkit," in *Proceedings of the ISMRM 28th Annual Meeting, Online*, vol. 4819, 2020.

[11] M. R. Kellman, E. Bostan, N. A. Repina, and L. Waller, "Physics-Based Learned Design: Optimized Coded-Illumination for Quantitative Phase Imaging," *IEEE Transactions on Computational Imaging*, vol. 5, no. 3, pp. 344–353, 2019.

[12] J. Fessler, "Michigan Image Reconstruction Toolbox," *MIRT, https://web.eecs.umich.edu/ fessler/code/index.html*.

[13] M. Uecker, F. Ong, *et al.*, "Berkeley Advanced Reconstruction Toolbox (BART)," in *Proceedings of the 23rd Annual Meeting of ISMRM, Toronto, Canada*, 2015.

[14] F. Ong and M. Lustig, "SigPy: A Python Package for High Performance Iterative Reconstruction," in *Proceedings of the ISMRM 28th Annual Meeting*, 2019.

[15] S. Rakshit and J. I. Tamir, "mri-sim-py.epg: A GPU-accelerated MRI Sequence Simulator for Differentiable Optimization and Learning," *GitHub. Note: https://github.com/utcsilab/mri-sim-py.epg*, vol. 1, 2021.

[16] S. Rakshit, K. Wang, and J. I. Tamir, "A GPU-accelerated Extended Phase Graph Algorithm for differentiable optimization and learning," in *Proceedings of the 2021 ISMRM & SMRT Annual Meeting & Exhibition, Online*, vol. 4819, 2021.

[17] R. W. Keener, *Theoretical Statistics: Topics For A Core Course*. Springer Science & Business Media, 2010.

[18] J. K. Barral, E. Gudmundson, *et al.*, "A robust methodology for in vivo T1 mapping," *Magnetic Resonance in Medicine*, vol. 64, no. 4, pp. 1057–1067, 2010.

[19] J. Tamir, *MR Shuffling: Accelerated Single-Scan Multi-Contrast Magnetic Resonance Imaging*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2018.

[20] J. I. Tamir, F. Ong, S. Anand, E. Karasan, K. Wang, and M. Lustig, "Computational MRI with physics-based constraints: Application to mul-

ticontrast and quantitative imaging," *IEEE Signal Processing Magazine*, vol. 37, no. 1, pp. 94–104, 2020.

[21] S. Sra, S. Nowozin, and S. J. Wright, *Optimization for machine learning*. MIT Press, 2012.

[22] M. E. Jerrell, "Automatic Differentiation and Interval Arithmetic for Estimation of Disequilibrium Models," *Computational Economics*, vol. 10, no. 3, pp. 295–316, 1997.

[23] G. F. Corliss, "Applications of Differentiation Arithmetic," in *Reliability in Computing*, pp. 127–148, Elsevier, 1988.

[24] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic Differentiation in Machine Learning: a Survey," *Journal of Machine Learning Research*, vol. 18, 2018.

[25] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[26] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, vol. 1. MIT Press, Cambridge, 2016.

[27] J. Bergstra, O. Breuleux, *et al.*, "Theano: A CPU and GPU math compiler in Python," in *Proceedings of 9th Python in Science Conference*, vol. 1, pp. 3–10, 2010.

[28] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: A modular machine learning software library," tech. rep., Idiap, 2002.

[29] M. Abadi, P. Barham, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.

[30] A. Paszke, S. Gross, *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, 2019.

[31] S. A. Forth and T. P. Evans, "Aerofoil Optimisation via AD of a Multi-grid Cell-Vertex Euler Flow Solver," in *Automatic Differentiation of Algorithms*, pp. 153–160, Springer, 2002.

[32] D. Casanova, R. S. Sharp, B. Christianson, and P. Symonds, "Application of Automatic Differentiation to Race Car Performance Optimisation," in *Automatic Differentiation of Algorithms*, pp. 117–124, Springer, 2002.

[33] A. Walther, "Automatic differentiation of explicit Runge-Kutta methods for optimal control," *Computational Optimization and Applications*, vol. 36, no. 1, pp. 83–108, 2007.

[34] J.-D. Müller and P. Cusdin, "On the performance of discrete adjoint CFD codes using automatic differentiation," *International Journal for Numerical Methods in Fluids*, vol. 47, no. 8-9, pp. 939–945, 2005.

[35] J. P. Thomas, E. H. Dowell, and K. C. Hall, "Using Automatic Differentiation to Create a Nonlinear Reduced-Order-Model Aerodynamic Solver," *AIAA journal*, vol. 48, no. 1, pp. 19–24, 2010.

[36] C. H. Bischof, H. M. Bücker, A. Rasch, E. Slusanschi, and B. Lang, "Automatic Differentiation of the General-Purpose Computational Fluid Dynamics Package FLUENT," *Journal of Fluids Engineering, Transactions of the ASME*, vol. 129, no. 5, pp. 652–658, 2007.

[37] A. Griewank, "Who Invented the Reverse Mode of Differentiation," *Documenta Mathematica, Extra Volume ISMP*, pp. 389–400, 2012.

[38] J. Schmidhuber, "Deep Learning in Neural Networks: An Overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.

[39] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[40] W. Falcon *et al.*, "PyTorch Lightning," *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning*, vol. 3, 2019.

[41] D. Ma, V. Gulani, *et al.*, "Magnetic Resonance Fingerprinting," *Nature*, vol. 495, no. 7440, pp. 187–192, 2013.

[42] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[43] S. Ruder, “An overview of gradient descent optimization algorithms,” *arXiv preprint arXiv:1609.04747*, 2016.

[44] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization,” *Journal of Machine Learning Research*, vol. 12, no. 7, 2011.

[45] T. Tieleman and G. Hinton, “Lecture 6.5-RMSProp: Divide the gradient by a running average of its recent magnitude,” *Coursera: Neural networks for machine learning*, pp. 26–31, 2012.

[46] J. P. Wansapura, S. K. Holland, *et al.*, “NMR relaxation times in the human brain at 3.0 tesla,” *Journal of Magnetic Resonance Imaging*, vol. 9, no. 4, pp. 531–538, 1999.

[47] J. I. Tamir, V. Taviani, *et al.*, “Targeted Rapid Knee MRI Exam using T2 Shuffling,” *Journal of Magnetic Resonance Imaging*, vol. 49, no. 7, pp. 195–204, 2019.

Vita

Somnath Rakshit was born in Bankura, West Bengal, India on 19 June 1995, the son of Dr. Gouranga Kanti Rakshit and Dr. Manjusri Rakshit. He received the Bachelor of Technology degree in Computer Science and Engineering from Jalpaiguri Government Engineering College. He then spent a year working on applying machine learning to bioinformatics and collaborating with the Centre of New Technologies, University of Warsaw, Poland. Then, he applied to the University of Texas at Austin for enrollment in their information studies program. He was accepted and started graduate studies in August, 2019.

Email address: `somnath@utexas.edu`

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.