Can a machine discover the Landau-Lifshitz equation?

Maurice Dijsselbloem

August 15, 2021

Supervisor: Dr. Huaiyang Yuan
Supervisor: Prof. dr. R.A. Duine
Institute for Theoretical Physics
Utrecht University

Abstract

In recent decades, machine learning methods have seen increased interest by physicists to solve physics problems. In this thesis we will discuss two different machine learning methods and apply them to predict the evolution of the magnetisation direction described by the Landau-Lifshitz equation. We find that the machine learning methods are successful at predicting the evolution of the magnetisation direction within reasonable error, and with one method we will find that it is possible to extract the dynamical equations from the training data. In the future we would like to apply these methods to more complicated systems like the colinear and antiferromagnetic systems. Furthermore, the methods could be improved to work with the Landau-Lifshitz-Gilbert equation.

Contents

1 Introduction

Machine learning has become an important research field in the last decades. Machine learning is different from classical programming where the rules and data are known up front. With machine learning we don't know the rules at first, but we only have the data and the answers, while the machine is supposed to figure out the rules. These rules could for example also be physical equations. Therefore, machine learning also has a huge potential for physics problems. In this thesis we will apply machine learning techniques to the magnetisation of a solid. The magnetisation direction will be determined by the Landau-Lifshitz equation. In spintronics the Landau-Lifshitz equation and Landau-Lifshitz-Gilbert equation are important equations. They have many applications in for example magnetisation switching or manipulating magnetic textures [1]. Previous work has applied machine learning to magnetisation, by trying to find the ground states of magnetic systems [2, 3]. It has not yet been applied to predicting the magnetisation evolution or finding the dynamic equations. Therefore, the main goal of this research is to see if a machine can learn to predict the magnetisation direction described by the Landau-Lifshitz equation and if it actually learns the equation or if it learns something else. By studying the simple cases first, we hope to apply the same methods to more complex systems, where the machine learning method may be quicker at calculating the magnetisation evolution. To investigate this, we use two different machine learning techniques that we will discuss. We will find that the machine is indeed able to learn to predict the evolution of the magnetisation direction with the first technique. With the second technique we will find that it is able to extract the dynamical equations from the time series data. The remainder of this thesis is structured as follows. We will first discuss some of the theory of magnetism and machine learning. Then after that, we will discuss the results of two different machine learning methods.

2 Magnetism

2.1 Magnetic interactions

Magnetic materials consist of many smaller regions of magnetic spins. For these materials we can define a vector field $\mathbf{M}(\mathbf{r}, t)$ [1]. This vector field is the magnetisation direction at place $\mathbf{r}$ and is dependent on time t. Below the Curie temperature, the direction of the magnetisation is the only relevant degree of freedom. Later we will only use the unit vector $\mathbf{m}(\mathbf{r}, t) = \mathbf{M}(\mathbf{r}, t)/M_s$, where M_s is the saturation magnetisation, which is the point where all spins would align exactly. Neighbouring magnetic regions can interact and influence each other through anistropy fields. They can also be influenced by an external magnetic field H_{ext}. The relaxation of magnetisation towards its equilibrium states is determined by damping mechanisms.

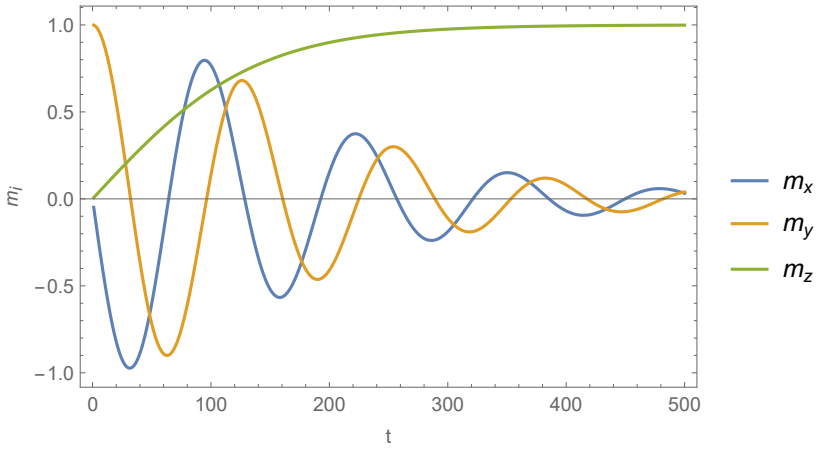

Figure 1: Magnetisation vector for the x, y and z direction over time for a magnetic element starting in the y-direction and magnetic field in the z-direction. It is visible that over time the magnetic element aligns with the magnetic field.

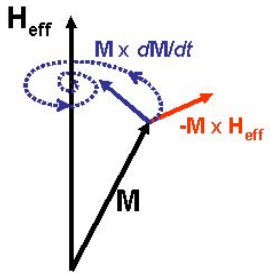

Figure 2: The precession of the magnetisation direction described by the Landau-Lifshitz-Gilbert equation [4].

2.2 Dynamic equations

The Landau–Lifshitz (LL) equation is a differential equation that describes the motion of a magnetic spin in three dimensions

$$\frac{\partial \mathbf{m}}{\partial t} = -\mathbf{m} \times \mathbf{H} + \alpha \mathbf{m} \times (\mathbf{H} \times \mathbf{m}), \tag{1}$$

where $\mathbf{m}$ is the magnetisation vector (m_x, m_y, m_z), such that $m_x^2 + m_y^2 + m_z^2 = 1$. $\mathbf{H}$ is the effective magnetic field and α is a damping factor that depends on the material. A numerical solution of this equation is shown in figure 1. An example of the precession of the magnetisation direction is shown in figure 2. The LL equation was proposed in 1935 [5] by Landau and Lifshitz. Anisotropies can arise in the effective magnetic field. In this case the effective magnetic field is for example $\mathbf{H} = H_{ext}\hat{e}_z - K_x m_x \hat{e}_x + K_z m_z \hat{e}_z$. Here H_{ext} is the external

field in the z-direction, and K_x, K_z are two positive constants determining the anisotropy. If there are anisotropies like this, the precession of the spin is an ellipsoid instead of a spiral. This equation is accurate for small damping factors α. Later the equation was improved by Gilbert to be more accurate for larger damping terms [6]. This is the Landau–Lifshitz-Gilbert (LLG) equation

$$\frac{\partial \mathbf{m}}{\partial t} = -\mathbf{m} \times \mathbf{H} + \alpha' \mathbf{m} \times \frac{\partial \mathbf{m}}{\partial t}. \tag{2}$$

The LLG equation is very similar to the LL equation. The difference is the $(\mathbf{H} \times \mathbf{m})$ term that is replaced with $\frac{\partial \mathbf{m}}{\partial t}$ and a different damping constant α'.

2.3 Magnetic textures

Magnetic materials can have microscopic structures. One example of such a structure are skyrmions [7, 8]. A Hamiltonian with which skyrmion structures can arise is

$$\mathcal{H} = -\sum_{i<j} J\mathbf{S}_i \cdot \mathbf{S}_j - \sum_{i<j} \mathbf{D}_{ij} \cdot (\mathbf{S}_i \times \mathbf{S}_j) - \sum_i \mathbf{B} S_i^z. \tag{3}$$

Here J is the exchange interaction, $\mathbf{D}_{ij}$ is the Dzyaloshinskii-Moriya vector and B is the magnitude of the magnetic field in the z-direction. Possible magnetic structures for this Hamiltonian are the ferromagnetic, Skyrmion and Spiral states shown in figure 4. A phase diagram is known in which regions for B and $\mathbf{D}$ the three different structures exists [7].

3 Machine Learning

There are multiple ways to make a machine 'learn'. We will discuss two methods in this project. One of these methods is by using neural networks. The second is a regression method.

3.1 Neural networks

A neural network is a grid of neurons that can communicate as shown in figure 3. The circles are the neurons of the network and the arrows are the weights of the model [9]. Furthermore, each neuron can have a bias. There are different kind of layers shown here. The input layer is the layer that takes the data you want to learn something from as input. The input data can be anything, for example the colour values of the pixels of an image or any other kind of data. For this layer it is important that you always normalise the data to be in between (0,1) in order to have the most optimal results. Secondly, we can have any number of hidden layers. In the example of figure 3 there is a single densely connected hidden layer. This neural network is densely connected if each neuron in each row is connected with an arrow to each neuron in the row in front and back. Another kind of hidden layer we will use in this project is

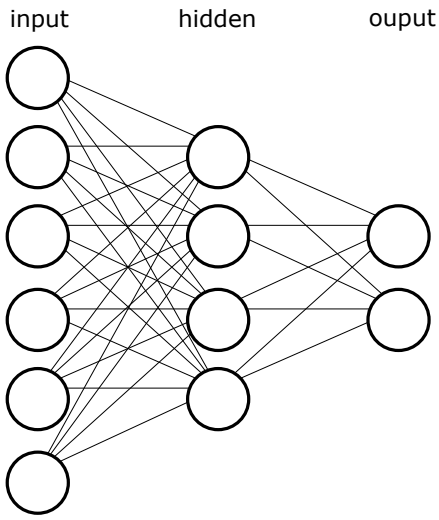

Figure 3: Neural network with one input layer, one hidden layer and one output layer. Circles present neurons and lines a connection between neurons.

a layer with LSTM (Long Short Term Memory) neurons. These neurons work significantly better at prediction tasks than normal neurons [10]. The last layer is the output layer. If you want the neural network to predict a single value, you can have just one neuron for the output layer. As another example, if you want your neural network to classify states, you can have multiple output neurons for each state. Depending on the activation function that you use, the value of each output neuron can be the probability that a system is in a certain state. The weights of the neural network can be optimised. This is called the training process. As a formula each arrow in figure 3 looks like:

$$output = f(\mathbf{w} * input + \mathbf{b}), \tag{4}$$

where $f(x)$ is the activation function, $\mathbf{w}$ is a matrix containing the weights and $\mathbf{b}$ is a vector containing biases. We call the input data $\mathbf{X}$ and the data from the output layer y. The goal of the training process is to have all connections in the network have the right weights, such that the network can make good predictions. The network changes these weights according to a loss function. The loss function calculates the difference between the real answer and answer from the neural network. At first the weights are chosen completely randomly, so the network is not expected to perform any better than just randomly guessing. In order to find the most optimal values for the weights and biases, we need an optimiser. Often, this is a gradient descent based optimiser like Adam [11]. At each training step, the loss is calculated and then each weight is changed by a bit. Then the loss is calculated again, and the weights are moved in the opposite direction of increased loss. You can run the training data multiple

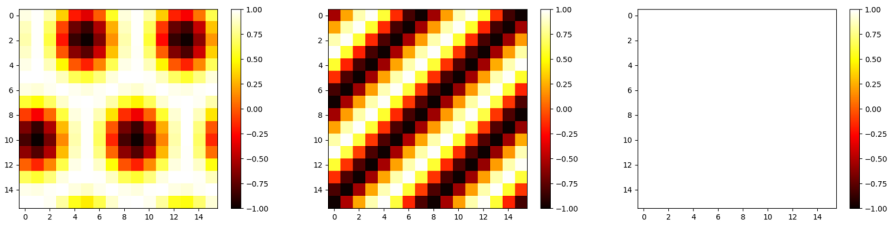

Figure 4: z-component of the magnetisation direction of Skyrmion (a), Spiral (b) and ferromagnetic (c) phases. This are the three possible phases for the Hamiltonian in equation 3.

times through the neural network in the training process. The number of times you do this is the number of epochs. Multiple epochs might make the model perform better, but with too many epochs the model can overfit, making it less accurate.

3.2 Example: Skyrmions

As an example of what machine learning can do, we first look at network that can recognise skyrmion phases. We study a system where three phases can exist. A pure ferromagnetic phase, where all particles are pointing in the same direction. A skyrmion phase and a spiral phase. This system is described by the Hamiltonian in equation 3.

The phases are calculated using Monte Carlo simulations (See appendix A). These simulations consist of a $N \times N$ grid of particles, where $N = 16$ and each particle has a spin in three dimensions. Only nearest neighbour interactions are taken into account and periodic boundary conditions are used. From the phase diagram in earlier research [7] the values of the Dzyaloshinskii-Moriya vector and magnetic field that are needed for each phase are used. The temperatures in the system are gradually lowered so that it can equilibrate. These Monte Carlo simulations generate almost perfect phases that can be used to train a Feed-Forward Network (FNN). The network that we use has every lattice site as input, one densely connected hidden layer of 64 neurons and an output layer of 3 neurons. The Sigmoid function is used as an activation function for the first layer:

$$S(x) = \frac{1}{1 + e^{-x}}, \tag{5}$$

and the Softmax function for the output neurons [12, 13]. Furthermore, as a loss function, the 'sparse categorical crossentropy' is used. This network is built using the TensorFlow library for Python. Each output neuron corresponds to one of the possible phases. 300 states of each phase type are used as training data. After the network is trained, it can be used to classify data it has never seen before. On this testing data, the model reaches almost 100% accuracy, which is not surprising, given the states are almost pure. If you introduce noise

into the model or use mixed states, it gets more interesting to see what the model does. However, we will not do that in this thesis.

3.3 Identification of Dynamic equations (SINDy)

A way to extract the dynamical equations from time-series data is the SINDy (Sparse Identification of Nonlinear Dynamical Systems) algorithm [14, 15]. This algorithm is quite different from the Neural network approach, but nevertheless also machine learning. For the calculations in this project, we use the PySINDy package for Python [16]. This algorithm assumes the equation can be written in the form

$$\frac{\partial \mathbf{x}(\mathbf{t})}{\partial t} = \mathbf{f}(\mathbf{x}(t)), \tag{6}$$

where $\mathbf{x}(\mathbf{t})$ is the state of the system at time t and $\mathbf{f}(\mathbf{x}(t))$ is a nonlinear function determining the dynamical equations. This condition is true for the LL equation 1. The training data is then arranged as

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^T(t_1) \\ \mathbf{x}^T(t_2) \\ \vdots \\ \mathbf{x}^T(t_3) \end{bmatrix}, \tag{7}$$

and similar for $\dot{\mathbf{X}}$. The data for $\dot{\mathbf{X}}$ can be either given explicitly if it is known or calculated numerically. Furthermore, we define a library $\mathbf{\Theta}(\mathbf{X})$ containing all the possible polynomial functions up to a certain order

$$\mathbf{\Theta}(\mathbf{X}) = [1, \mathbf{X}, \mathbf{X}^{P_2}, \dots], \tag{8}$$

where X^{P_n} contains all $n-th$ order polynomials. These polynomials are all the possibilities of functions that are present in equation 6. Finally, the following equation is constructed:

$$\dot{\mathbf{X}} = \mathbf{\Theta}(\mathbf{X})\mathbf{\Xi}. \tag{9}$$

Here $\mathbf{\Xi}$ is a sparse matrix containing the coefficients of the equations, determining which terms in the polynomial library are relevant. These coefficients can be found using a sparse regression method. For our simulations we use the 'Sequentially thresholded least squares' (STLSQ) algorithm. Once the coefficients are found the dynamical equations can be constructed. To score the accuracy of the found equations we use the R^2 score given by

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y}_i)^2}, \tag{10}$$

where y_i is the true value of the i-th sample, $\hat{y}_i$ the predicted value of the i-th sample and $\bar{y}_i = \frac{1}{n}\sum_{i=1}^{n} y_i$. With this scoring system a value of 1.0 is the best possible score giving a perfect prediction [17].

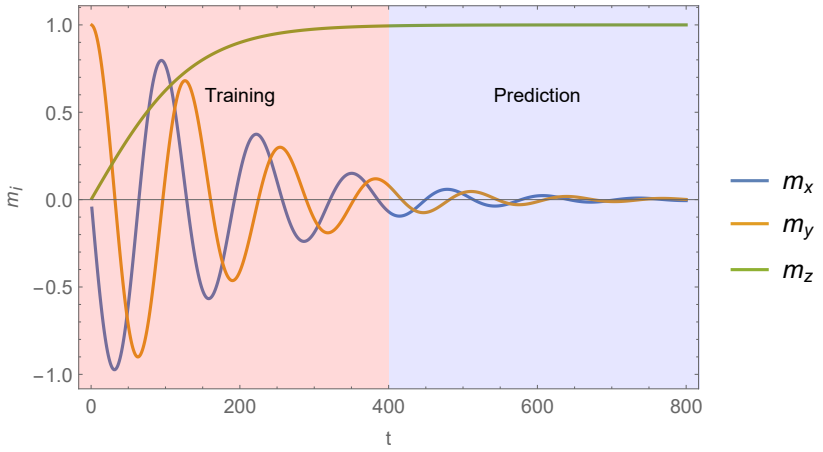

Figure 5: Magnetisation for the x, y and z direction over time for a magnetic element starting in the y-direction and magnetic field in the z-direction. It is visible that over time the magnetic element aligns with the magnetic field.

4 Predicting the magnetisation evolution

We want to feed the machine a limited set of training data. We chose to take the first 400 timesteps of magnetisation evolution as training data and we want the machine to predict the next timesteps as shown in figure 5. Relevant parameters we will use are the damping constant of $\alpha = 0.15$ and for the anisotropies we use the constants $K_x = 0.2$, $K_z = 0.3$. The external magnetic field $H_{ext} = 1.0$. For the neural network, we use timestep $\Delta t = 0.05$ for the integration. In a typical magnetic system, this corresponds to a magnetic field of 1T for the external field and an anisotropy field of 0.2/0.3T in the x, z-direction. $\Delta t = 0.05$ corresponds to about 0.3ps.

4.1 Data generation

The data that is used for the learning process, is calculated numerically using the LL equation (equation 1) and the Runge-Kutta integration method [18]. So as training data $N = 400$ discrete time steps are used. Test data is then also calculated in the same way for the next 600 time steps. We start the magnetisation in our training data in the y-direction. The Runge-Kutta algorithm works

by calculating four separate terms k_1, k_2, k_3, k_4:

$$\begin{aligned} k_1 &= \Delta t f(\mathbf{x}, t), \\ k_2 &= \Delta t f(\mathbf{x} + \frac{1}{2}k_1, t + \frac{1}{2}\Delta t), \\ k_3 &= \Delta t f(\mathbf{x} + \frac{1}{2}k_2, t + \frac{1}{2}\Delta t), \\ k_4 &= \Delta t f(\mathbf{x} + k_3, t + \Delta t), \end{aligned}$$

where $f(\mathbf{x}, t)$ is, for our training data, the right-hand side of equation 1. The next time step is then calculated by adding the terms in the following way:

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \tag{11}$$

4.2 Sliding window method

A method to let the machine learn the data is the sliding window method [19]. This method uses a window of width w time steps to train and uses the next time step as the output it should predict. This window 'slides' from the beginning to the end of the data. For example, if we have a window width of $w = 3$, the first two training data points look like:

$$\mathbf{X}(1) = \{f(t_1), f(t_2), f(t_3)\}, y(1) = f(t_4), \tag{12}$$

$$\mathbf{X}(2) = \{f(t_2), f(t_3), f(t_4)\}, y(2) = f(t_5). \tag{13}$$

With this training method, there are w input neurons needed and one output neuron for the prediction. After the training process, the window can go on to predict the time steps it doesn't know yet. A downside of this method is that after you're a window length w of time steps after the original data, the machine only uses data that is has predicted itself to predict the further values. If there is any error in this predicted data, the error will only get worse for the new predicted data. One could extend the y-values to have multiple timestep predictions at the same time. This way not just the correlations between x-values are learned, but also correlations between y-values. We will not do this in this thesis.

4.3 Used network

We will use two different neural networks. Both networks have an input layer of w neurons and an output layer with 3 neurons (One each for the x, y, z-coordinates of the prediction). The first network has two hidden dense layers with $3w$ neurons while the second network has a single LSTM layer with $6w$ neurons. For the first network $w = 65$, and for the second network $w = 31$, unless stated otherwise. The first network will be used for the isotropic case, whereas the second network will be used for the for anisotropic case.

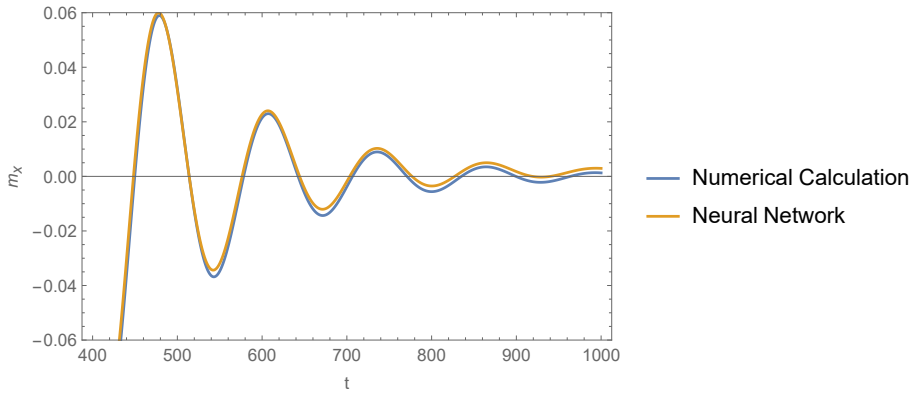

Figure 6: Prediction of the x-component of the magnetisation direction in the isotropic case using the neural network. Blue line is the exact solution, while the yellow line is the prediction. We see that the predicted solution follows the exact solution with a slight offset.

4.4 Results & Discussion

In figure 6 and 7 we see how the neural network performs on the isotropic LL equation. For the x-coordinate we see that the period of the oscillations is learned correctly. However, there is a slight offset for the equilibrium. For the z-coordinate we see that, in this case, the equilibrium is also estimated wrong by about 0.8% (Although this error may vary). We also see some oscillation that isn't present in the numerical solution. Although the machine is able to learn some of the features of the LL equation, it doesn't learn them all exactly right.

Next, we look at the anisotropic case, for which we use the LSTM hidden layer. The results are shown in figures 8 and 9. We see that the neural network is better able to predict the x-coordinate. If we look at the z-component, we see that here also the prediction has a slight offset in the equilibrium. As said earlier, the norm of the magnetisation should remain constant ($m_x^2 + m_y^2 + m_z^2 = 1$). We look at the norm of the predicted data, to see if the neural network learns this physical property. This is shown in figure 10. We see that the norm is not equal to 1 throughout the prediction, so it seems like the neural network does not learn this physical constraint.

One might wonder how the number of neurons affects the prediction quality. We let the hidden layer have $f * w$ LSTM neurons, where f is varied from 1 to 9. The results are shown in figure 11. In this range, we see that the number of neurons doesn't seem to matter that much for the error in the prediction.

Another property for the neural network discussed earlier is the window width w. It is reasonable to assume there is an optimal width for the neural network. To see how the width affects the prediction quality, we train the network for various widths and calculate the mean average error. For the FFN

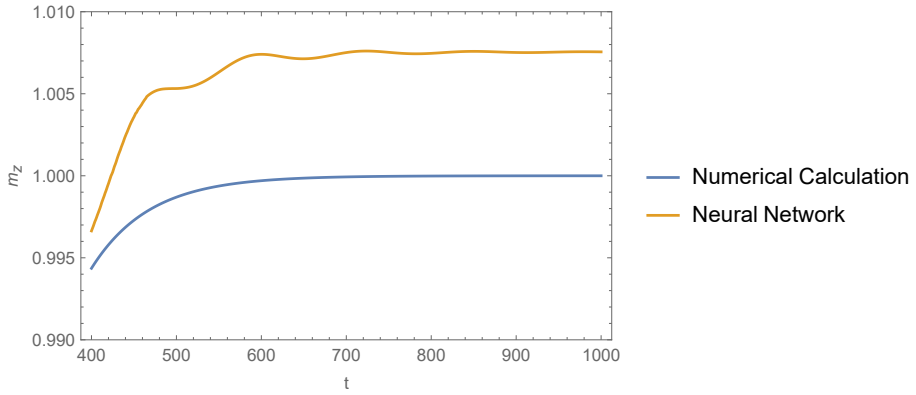

Figure 7: Prediction of the z-component of the magnetisation direction in the isotropic case using the neural network. Blue line is the exact solution, while the yellow line is the prediction. We see that the predicted solution follows the exact solution with a slight offset.

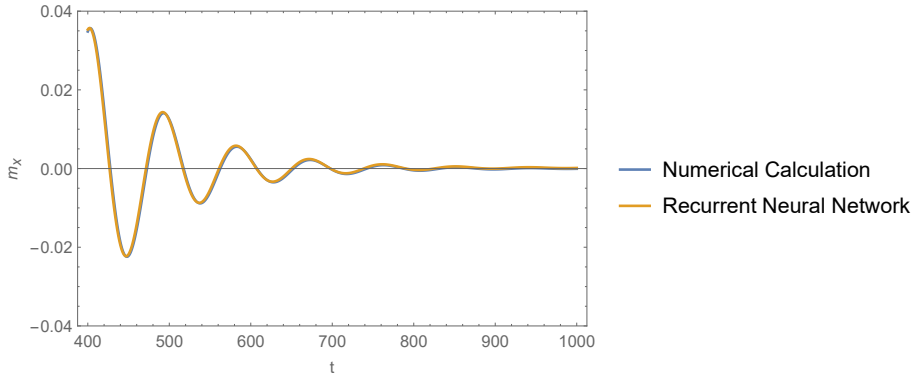

Figure 8: Prediction of the x-component of the magnetisation direction in the anisotropic case using the neural network. Blue line is the exact solution, while the yellow line is the prediction. We see that the predicted solution follows the exact solution with a slight offset.

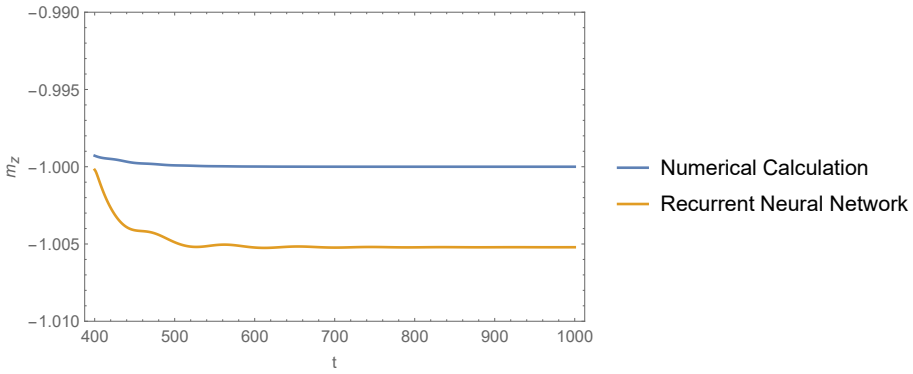

Figure 9: Prediction of the z-component of the magnetisation direction in the anisotropic case using the neural network. Blue line is the exact solution, while the yellow line is the prediction. We see that the predicted solution follows the exact solution with a slight offset.

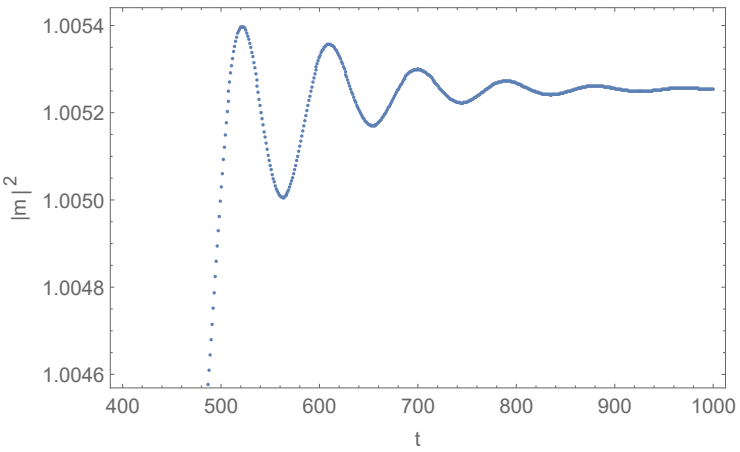

Figure 10: Evolution of the norm of the magnetisation direction over time in the anisotropic case. The norm should remain 1, but the neural network does not seem to learn this constraint.

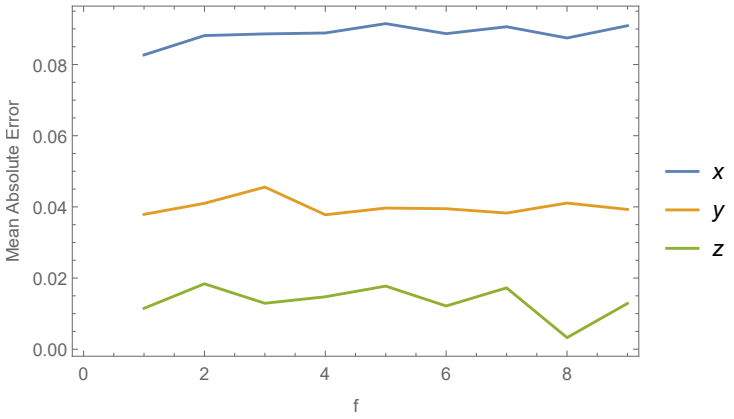

Figure 11: The mean absolute error in the prediction of the isotropic case as a function of $f * w$ neurons in the hidden layer, where w is the window width, and f is a factor. We see that for the prediction quality, the amount of hidden layer neurons doesn't matter much in this range.

network the results are shown in figure 12. We see that the prediction quality gets worse for about $w < 40$ and $w > 100$.

Finally, it is important to make sure we are not overfitting. To make sure we're not using too many epochs, we plot the mean absolute error against the number of epochs. The result for the FFN network is shown in figure 13. We see that there appears to be some kind of minimum for the x, y coordinates. However, this is not visible for the z-coordinate. After 9 epochs, the prediction quality doesn't seem to significantly improve anymore. The prediction quality also doesn't seem to get worse with more epochs, so we're not overfitting.

5 Dynamic equations

In the previous chapter we discussed the neural network method for machine learning. Unfortunately, it is not obvious what exactly the network learns to predict the evolution. It is also not possible to extract the actual dynamical equations. In this chapter we will discuss the SINDy method with which it is possible to get the dynamical equations.

5.1 Training

The same method to generate training data is used as for the neural network in chapter 4. We will also use the same constants for the damping and anisotropies, but we will vary the timestep Δt.

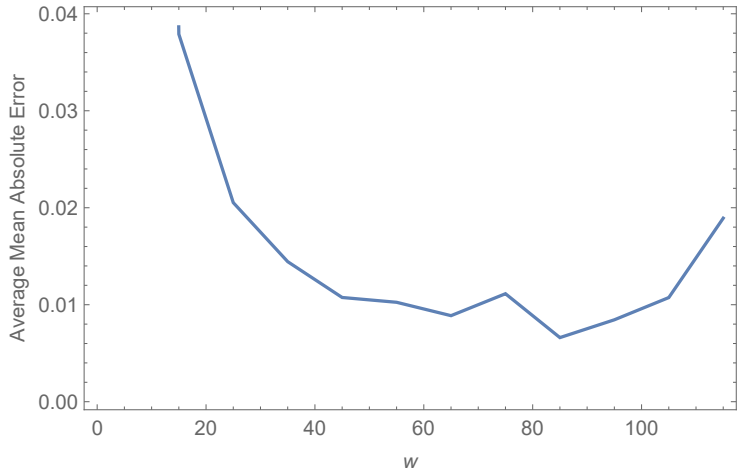

Figure 12: The mean absolute error in the prediction of the isotropic case as a function of the window width w, where w is also the amount of input neurons. There seems to be a minimum in the error between 60 and 80.

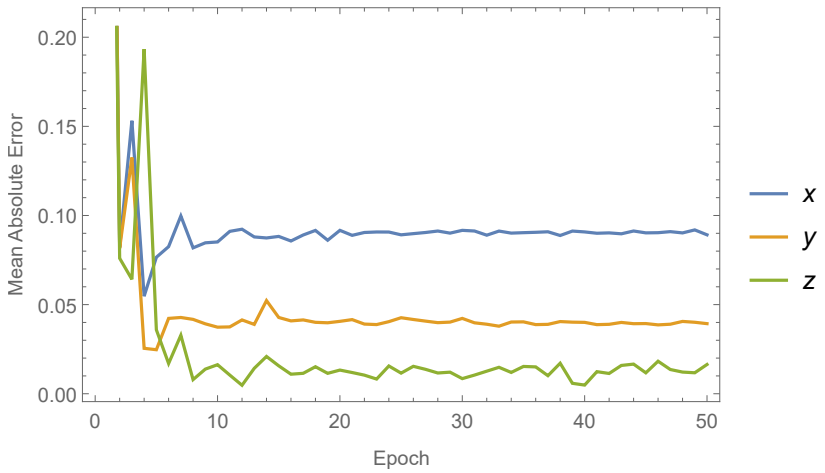

Figure 13: The mean absolute error in the prediction of the isotropic case as a function of the amount epochs that the neural network has been trained. The number of epochs stands for the number of times the neural network has seen the training data. After 10 epochs the error doesn't seem to change much anymore.

5.2 Results & Discussion

First, we look at the isotropic case. In figure 14 we see the R^2 error as a function of dt. We see a drop in the prediction quality at around $dt \approx 0.005$. The most optimal learned equations are shown in table 1. We see that the learned equations are the same as the exact equations with no error. In figure 15 we see the R^2 error for the anisotropic case with no damping ($\alpha = 0$). In this case we see the prediction quality drop off at around $dt \approx 0.05$. The same is shown in figure 16, but with damping $\alpha = 0.15$. The drop off in prediction quality happens here with a much lower dt. For the case with damping and $K_x = 0$ or $K_x = 0.2$, we show the most optimal learned equation in tables 2 and 3 respectively. In this case there is an error in the learned equations. We observe that the error can be written in the form of a factor times $(1 - m_x^2 - m_y^2 - m_z^2)$. Which should be equal to zero. Therefore, it is likely that the error comes from a differentiation or integration error and only has a very small impact on the prediction quality of the learned equation.

Now we want to know how robust the algorithm is to noise. We add a Gaussian noise with factor η to the training data and compare the prediction accuracy. This Gaussian noise may imitate the measurement noise that is present in real-world measurements of the magnetisation orientation. The results are shown in figure 17. The case with no damping is more resilient to the noise in the system. The case with damping doesn't recognise the dynamics anymore with noise levels higher than $\eta = 10^{-4}$. We did the same for the neural network and compare the results in figure 18. We see that the neural network is much better at dealing with noise, where the accuracy is still good at noise levels of $\eta = 10^{-2}$.

Table 1: Comparison between the exact dynamic equations and the best learned equations for an isotropic magnet.

Exact equation	Learned equation	Error
$\partial_t m_x = m_y + 0.15 m_x m_z$	$\partial_t m_x = m_y + 0.15 m_x m_z$	0
$\partial_t m_y = m_x + 0.15 m_y m_z$	$\partial_t m_y = m_x + 0.15 m_y m_z$	0
$\partial_t m_z = -0.15(m_x^2 + m_y^2)$	$\partial_t m_z = -0.15(m_x^2 + m_y^2)$	0

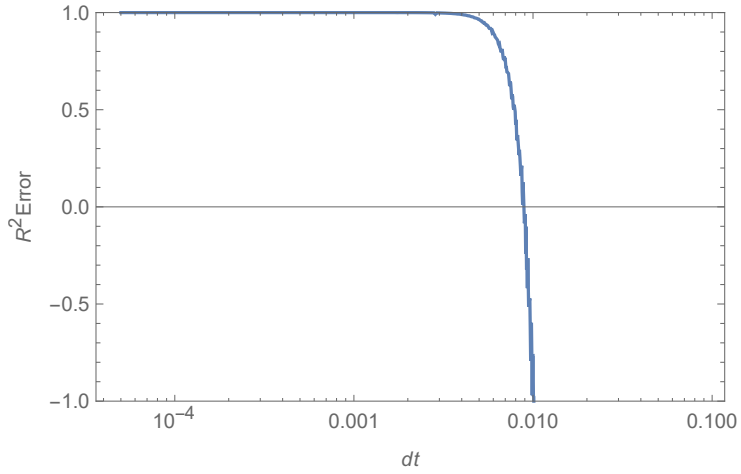

Figure 14: R^2 error (closer to one is better) for the isotropic case with damping $\alpha = 0.15$. We see that the prediction accuracy drops at timesteps above $\mathrm{d}t \approx 0.005$.

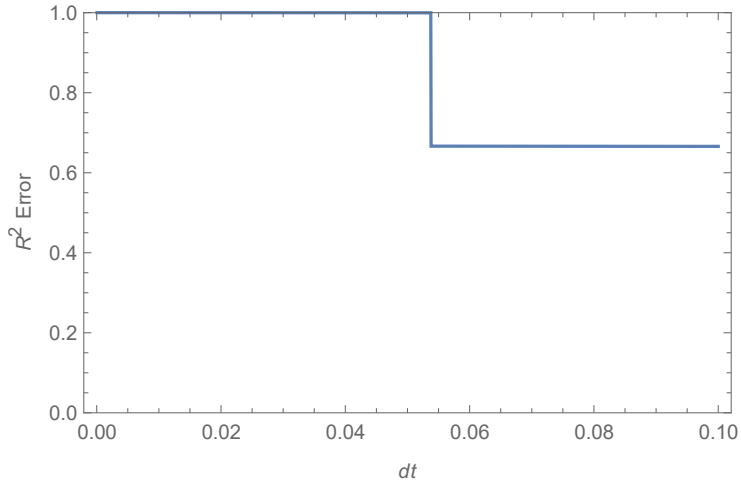

Figure 15: R^2 error (closer to one is better) for the anisotropic case with no damping ($\alpha = 0$) as a function of integration timestep $\mathrm{d}t$. We see that the prediction accuracy drops at timesetps $\mathrm{d}t \approx 0.05$.

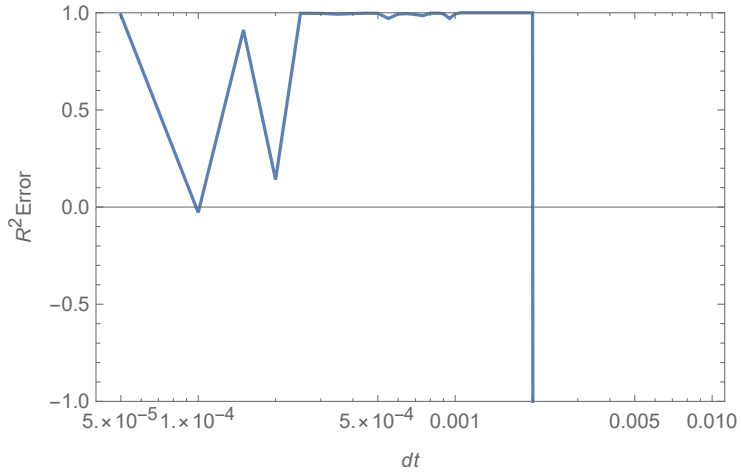

Figure 16: R^2 error (closer to one is better) for the anisotropic case with damping $\alpha = 0.15$ as a function of integration timestep dt. We see that the prediction accuracy drops at timesteps above d$t \approx 0.02$. In this case, we also see that for timesteps below d$t \approx 4 * 10^{-4}$, the accuracy drops.

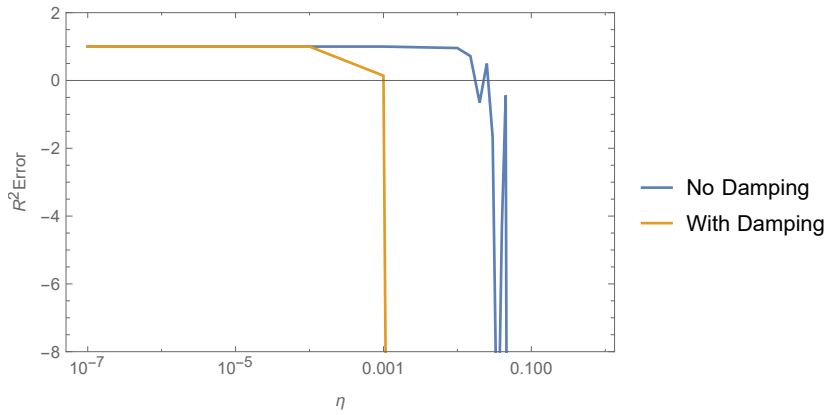

Figure 17: R^2 error (closer to one is better) for the anisotropic case with and without damping as a function of noise η added to the training data. We see that the case without damping holds up better against the noise, but both have a sharp drop off where the prediction becomes unreliable.

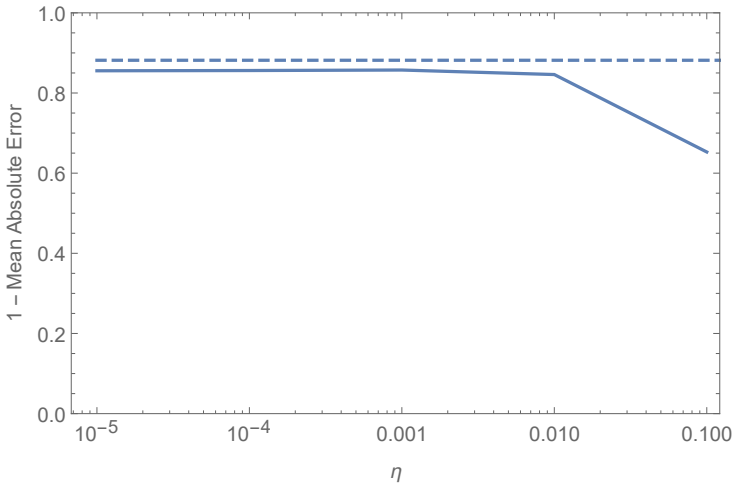

Figure 18: One minus the mean absolute error (closer to one is better) for the neural network for the anisotropic case as a function of noise η added to the training data (dotted line is for the case with no noise $\eta = 0$). We see that the neural network can handle approximately two orders of magnitude more noise as compared to the SINDy algorithm.

Table 2: Comparison between the exact dynamic equations and the best learned equations for an anisotropic magnet. $k_x = 0$, $k_z = 0.3$, $\alpha = 0.15$

Exact equation
$\partial_t m_x = m_y - 0.3 m_y m_z - m_x m_z (0.15 - 0.045 m_z)$ $\partial_t m_y = m_x - 0.3 m_x m_z + m_y m_z (0.15 - 0.045 m_z)$ $\partial_t m_z = (m_x^2 + m_y^2)(0.15 - 0.045 m_z)$
Learned equation
$\partial_t m_x = m_y - 0.3 m_y m_z - m_x m_z (0.15 - 0.045 m_z) + 0.25 m_y (1 - m_x^2 - m_y^2 - m_z^2)$ $\partial_t m_y = m_x - 0.3 m_x m_z + m_y m_z (0.15 - 0.045 m_z) - 0.25 m_x (1 - m_x^2 - m_y^2 - m_z^2)$ $\partial_t m_z = (m_x^2 + m_y^2)(0.15 - 0.045 m_z)$
Error
$0.25 m_y (1 - m_x^2 - m_y^2 - m_z^2)$ $-0.25 m_x (1 - m_x^2 - m_y^2 - m_z^2)$ 0

Table 3: Comparison between the exact dynamic equations and the best learned equations for an anisotropic magnet. $k_x = 0.2$, $k_z = 0.3$, $\alpha = 0.15$

Exact equation
$\partial_t m_x = 0.03 m_x m_y^2 + m_y(1 - 0.3 m_z) + m_x(-0.15 + 0.075 m_z) m_z$
$\partial_t m_y = -0.03 m_x^2 m_y + m_x(-1 + 0.5 m_z) + m_y(-0.15 + 0.045 m_z) m_z$
$\partial_t m_z = -0.2 m_x m_y + m_x^2(0.15 - 0.075 m_z) + m_y^2(0.15 - 0.045 m_z)$
Learned equation
$\partial_t m_x = 0.03 m_x m_y^2 + m_y(1 - 0.3 m_z) + m_x(-0.15 + 0.075 m_z) m_z$
$+0.25 m_y(1 - m_x^2 - m_y^2 - m_z^2) - 0.026 m_x * (1 - m_x^2 - m_y^2 - m_z^2)$
$\partial_t m_y = -0.03 m_x^2 m_y + m_x(-1 + 0.5 m_z) + m_y(-0.15 + 0.045 m_z) m_z$
$-0.25 m_x(1 - m_x^2 - m_y^2 - m_z^2)$
$\partial_t m_z = -0.2 m_x m_y + m_x^2(0.15 - 0.075 m_z) + m_y^2(0.15 - 0.045 m_z)$
Error
$0.25 m_y(1 - m_x^2 - m_y^2 - m_z^2) - 0.026 m_x * (1 - m_x^2 - m_y^2 - m_z^2)$
$-0.25 m_x(1 - m_x^2 - m_y^2 - m_z^2)$
0

6 Conclusion & Outlook

In this thesis we have studied if a machine can learn to predict the magnetisation direction described by the Landau-Lifshitz equation. We have shown that the machine can learn to predict the magnetisation evolution using neural networks, both in isotropic and anisotropic cases. Furthermore, we have used the SINDy algorithm to extract the dynamic equations from the training data. In future research we would like to study more complex magnetic systems with antiferromagnetic properties or colinear systems. We think that in these cases, the computation time of the neural network approach might be shorter than computation time of methods that are currently used to find the magnetisation direction. In very complicated systems that take a long time to calculate using micromagnetic simulations, this could be a benefit. We have tried to apply a two antiferromagnetic spins configuration to the models, but we found that the SINDy method isn't suitable anymore for these more complex systems. This is because the equations get a lot of terms. However, our neural network might be suitable after some optimisations. Finally, we would also like to test the algorithms on the LLG equation or add extra torque terms to the LLG equation that are caused by an electric current.

References

[1] M. D'Aquino (2005), URL `http://www.fedoa.unina.it/148`.

[2] A. Kovacs, J. Fischbacher, H. Oezelt, M. Gusenbauer, L. Exl, F. Bruckner, D. Suess, and T. Schrefl, Journal of Magnetism and Magnetic Mate-

rials **491**, 165548 (2019), URL https://doi.org/10.1016/j.jmmm.2019.165548.

[3] A. Kovacs, L. Exl, A. Kornell, J. Fischbacher, M. Hovorka, M. Gusenbauer, L. Breth, H. Oezelt, D. Praetorius, D. Suess, et al., *Magnetostatics and micromagnetics with physics informed neural networks* (2021), 2106.03362.

[4] C. Becker, *Sketch of the damped precession of the magnetization m under the influence of an effective field heff* (2006), URL https://commons.wikimedia.org/wiki/File:Damped_Magnetization_Precession.jpg.

[5] L. LANDAU and E. LIFSHITZ, in *Perspectives in Theoretical Physics* (Elsevier, 1992), pp. 51–65, URL https://doi.org/10.1016/b978-0-08-036364-6.50008-9.

[6] T. Gilbert, IEEE Transactions on Magnetics **40**, 3443 (2004), URL https://doi.org/10.1109/tmag.2004.836740.

[7] I. A. Iakovlev, O. M. Sotnikov, and V. V. Mazurenko, Physical Review B **98** (2018), URL https://doi.org/10.1103/physrevb.98.174411.

[8] V. K. Singh and J. H. Han, Physical Review B **99** (2019), URL https://doi.org/10.1103/physrevb.99.174426.

[9] F. Chollet, *Deep learning with Python* (Manning Publications Co, Shelter Island, New York, 2018), ISBN 9781617294433, oCLC: ocn982650571.

[10] S. Hochreiter and J. Schmidhuber, Neural Computation **9**, 1735 (1997), URL https://doi.org/10.1162/neco.1997.9.8.1735.

[11] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization* (2017), 1412.6980.

[12] J. Han and C. Moraga, in *Lecture Notes in Computer Science* (Springer Berlin Heidelberg, 1995), pp. 195–201, URL https://doi.org/10.1007/3-540-59497-3_175.

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning* (The MIT Press, Cambridge, Massachusetts, 2016), chap. 6.2.2.3, Adaptive computation and machine learning, ISBN 9780262035613.

[14] S. L. Brunton, J. L. Proctor, and J. N. Kutz, Proceedings of the National Academy of Sciences **113**, 3932 (2016), URL https://doi.org/10.1073/pnas.1517384113.

[15] S. Roy and D. Rana, *Machine learning in nonlinear dynamical systems* (2020), 2008.13496.

[16] B. M. de Silva, K. Champion, M. Quade, J.-C. Loiseau, J. N. Kutz, and S. L. Brunton, *Pysindy: A python package for the sparse identification of nonlinear dynamics from data* (2020), 2004.08424.

[17] A. C. Cameron and F. A. Windmeijer, Journal of Econometrics **77**, 329 (1997), URL https://doi.org/10.1016/s0304-4076(96)01818-0.

[18] W. H. Press, ed., *Numerical recipes: the art of scientific computing* (Cambridge University Press, Cambridge, UK ; New York, 2007), chap. 17.1, 3rd ed., ISBN 9780521880688.

[19] T. G. Dietterich, in *Lecture Notes in Computer Science* (Springer Berlin Heidelberg, 2002), pp. 15–30, URL https://doi.org/10.1007/3-540-70659-3_2.

A Monte Carlo simulations

With Monte Carlo simulations it is possible to find the ground state of a system. The Monte Carlo simulations in this thesis work as follows. We take a 2D grid of $N \times N$ particles that have a 3D orientation. On this grid we take a random particle and slightly change its orientation. Then, the energy difference is calculated over the particle nearest neighbours from before and after the move using the Hamiltonian in equation 3. The chance that the move gets accepted is then $P = e^{-\frac{dE}{k_B T}}$, where dE is the energy difference, k_B is the Boltzmann constant and T is the temperature of the system. Periodic boundary conditions are used, which means that the particle in position $(1, 1)$ is also a nearest neighbour of the particles in position $(N, 1)$ and $(1, N)$. This prevents any effect a boundary could have on the simulations. Over the course of the Monte Carlo simulation, we gradually lower the temperature of the system. This also means the chance to accept a move gets smaller. Because it's generally desirable to have an acceptance rate of 50%, the movement of a particle from its initial configuration gets smaller as the temperature decreases.