Metropolis algorithm for the 2D Ising model

Nicolas Yan (ndy21)

April 2020

Abstract

A Metropolis algorithm, implemented in Python, is used to study thermodynamic properties of the 2D Ising model, with attention to finite-size scaling of critical behaviour and the critical temperature, including mean magnetisation and heat capacity. Dynamical features of the algorithm are also analysed: relaxation time, auto-correlation.

1 Introduction

In this report, I detail my investigations into a Metropolis algorithm applied to the 2D Ising model. I raise the computational challenges that came up during the investigation and present my results, both quantitative and qualitative. The Metropolis algorithm allows equilibrium information to be determined, but also gives the system probabilistic dynamics which are also studied.

The remainder of this section gives additional theoretical background on the problem. Section 2 describes my general approach. Section 3 will then go through each task with details on the investigations conducted. Section 5 contains appendices.

1.1 The Ising model

The Ising model is a simple model for a ferromagnet. It is of particular physical interest because it is analytically tractable (Onsager solution) while still displaying interesting thermodynamic behaviour.

There are N^2 degrees of freedom in the model, where N is the side length of a square grid of spins s_i. The Hamiltonian is

$$E = -J \sum_{\text{nn } i,j} s_i s_j - \mu H \sum_{i=1}^{N^2} s_i,$$

J being the interaction energy and H the applied magnetic field. The boundary conditions are periodic. The grid is part of a canonical ensemble and connected to a heat bath of temperature T.

A key feature of the Ising model is that it displays a phase transition at $T_{ons} = 2/\log(1+\sqrt{2})$. Because of symmetry, the mean magnetisation M is always 0, but $|M| > 0$ at subcritical temperatures due to spontaneous symmetry breaking.

H is measured in units of $1/\mu$ and temperatures in units of J/k_B.

1.2 The Metropolis algorithm

The statistics of any canonical ensemble are encoded in the probability distribution

$$P(i) = \frac{e^{-\beta E_i}}{Z}$$

where $\beta = 1/k_B T$. Although the numerator is easy to calculate, Z is much more difficult. The *Metropolis algorithm* is a method for generating a set of samples which approaches the distribution $P(i)$ from a function $f(i) \propto P(i)$, such as the numerator in the above expression, as the number of samples goes to infinity. Each new sample generated is called a "step" and the set of samples be interpreted as a random walk through the phase space of possible microstates.

Alternatively, this stepping can be interpreted as a way of probabilistically time-evolving an initial state. The probability arises from the uncertain state of the heat bath to which the system is coupled. It can be assumed that the two interpretations are equivalent – the dynamical evolution of any particular state will explore the full phase space with the same probability density as that predicted in the canonical ensemble (ergodicity).

2 Methodology

At the outset of the project, I thought it important to be systematic in approaching each task, especially due to the open-ended nature of the investigation. I established the following "protocol":

- Preliminary investigation, with qualitative testing. This helped determine the range of the parameters for the bulk of the task.
- Writing the data generation and analysis code in Python, including running small scale testing.
- Data generation and saving. This sometimes included letting the program run overnight.
- Data analysis.
- Plotting results.

Details about the program structure and approach to code and data re-use can be found in Appendix A. The key points are that I separated out the general-purpose code into a package called `ising`, which contains multiple modules

handling data generation, analysis, and plotting separately. This separation made it possible to revise my analysis methods and gave me more flexibility, as I didn't need to wait for data to regenerate every time I made a change.

I will give general comments on computational considerations in the remainder of this section.

2.1 Data generation

2.1.1 Ensembles

In the Metropolis algorithm, simple time evolution should in theory explore the full distribution of states. However, this fails at subcritical temperatures, especially for larger grids, because of "non-equilibrium" effects where the initial conditions matter: spontaneous symmetry breaking and domain formation. This motivated me to generate *ensembles* of simulations with given parameters to better sample the phase space and get around long-timescale correlations of microstates across Metropolis time steps.

By generating ensembles, I was also able to get quantitative errors by taking the ensemble standard deviation, since each system in the ensemble could be assumed to be independent, unlike different time steps of the same simulation.

The parameters associated with each ensemble are: the temperature, which I labelled with the variable `b` for thermodynamic β; the number of systems `sysnum` in the ensemble; the initial probability distribution labelled by `p`; the applied magnetic field `h`; and the number of iterations calculated `iternum`.

2.1.2 Computational aspects of simulation

The actual simulation code stored in the `simulator` module can be found in the appendix. Note that the code allows for spatially varying temperatures and applied fields (controlled by `const_b` and `const_h`), but this was not used. I also had to allow for a time-varying magnetic field for hysteresis loop calculations – this is handled in the `datagen` module.

Across the project I made extensive use of `numpy` arrays. Unfortunately, during the generation phase, I was unable to find a way to vectorise the code and take advantage of the associated performance increase. This was because I had to pick out N^2 spins one after the other for each time step, so a for loop was unavoidable.

2.2 Data analysis

The `thermo` module listed in the appendix collects some functions common to each task, but much of the analysis is delegated to the main task files. I used vectorised `numpy` functions, but this was mainly for brevity and code readability than performance. As a rule, I favoured readability over performance for the

analysis code, since this would take at most a few minutes to execute. This was in contrast to the data generation code, which I would usually let run overnight.

Importantly, throughout my code, I had to keep track of what indices corresponded to what dimension across multiple modules and files – to minimise confusion I was liberal with comments, avoiding unexplained "magic numbers".

2.3 Plotting

I used the `matplotlib` package for all plotting.

One particularly useful plotting device was to animate the spins as a function of time, with the help of the `matplotlib.animation` module. By animating an entire ensemble side by side (a "mosaic" animation), I could get a representative sample of the dynamics of the simulation. This was useful for troubleshooting and to gain qualitative insight. See the "plotter" module provided alongside the report for details.

3 Investigations

3.1 Equilibrium behaviour

Throughout the investigation, I used the `plotter.animate_mosaic( )` function to animate ensembles with various parameters as an aid to visualisation. In this section I will qualitatively describe the equilibrium behaviour I observed. Please see the accompanying animation, `equilibrium.avi`.

Above a critical temperature T_c, the spins quickly "forget" about any structure in the initial conditions and rapidly reach an equilibrium dominated by small-scale fluctuations or noise. As T is lowered to near T_c, these fluctuations become larger and larger. Below the critical temperature, the equilibrium state is for all the spins to align (save for very small local fluctuations). However, if the initial conditions are not fully aligned, metastable domains may form which temporarily impede progress towards true equilibrium, especially at colder temperatures. (See Section 3.8 for details.)

These factors are influenced by the grid size. Near T_c, the grids will oscillate between mostly black to mostly white on a relatively long timescale; these are critical fluctuations. For an infinite grid, this would only occur exactly at $T = T_c$, but I observed this flipping occurring at a wide range of temperatures for small grids (e.g.: $N = 5$). This *finite-size scaling* becomes relevant in Section 3.3 and is investigated quantitatively in Section 3.6.

3.2 Relaxation time

I propose a way to quantitatively determine the relaxation time – the time it takes to reach equilibrium. I then implement this calculation for a range of temperatures.

3.2.1 Condition for equilibrium

If we interpret the Metropolis stepping as some kind of time evolution, where the probabilistic behaviour is due to the coupling with a heat bath in an uncertain microstate, then recall the condition for an ensemble to be at equilibrium, which is that

$$\frac{d\rho}{dt} = 0$$

where ρ is the ensemble's density distribution over the microstate phase space.

Then for any test function f defined over that phase space,

$$\frac{d\langle f \rangle_e}{dt} = 0.$$

In other words, the ensemble average of any test function must not change at equilibrium. This suggests a numerical test for equilibrium: select a test function, calculate its derivative, and measure when this derivative goes below a certain cutoff.

The first candidate test function I considered was the magnetisation, but the symmetry between spin up and spin down made this a poor choice, as it always averaged out to zero for randomised initial spins. Instead, I used the square magnetisation.

To calculate the time derivative, I first calculated the step-to-step difference (np.diff) of the values, but due to the finite size of the ensemble, this was extremely noisy, especially for initially random spins. To get around this, I smoothed out the signal with a rolling average, with a heuristically determined window of 100 steps.

See appendix for code.

3.2.2 Ensemble size

As a non-equilibrium measurement, the exact initial conditions mattered greatly. The time taken to reach equilibrium varied greatly from run to run due to the initial conditions and the randomness of the Monte-Carlo method. This proved to be a computational problem, as it meant I needed to generate large ensembles of grids to get reliable results, but this had to be balanced against the computation time. In the end, I opted to use ensembles of 100 grids at a time, sized 30x30.

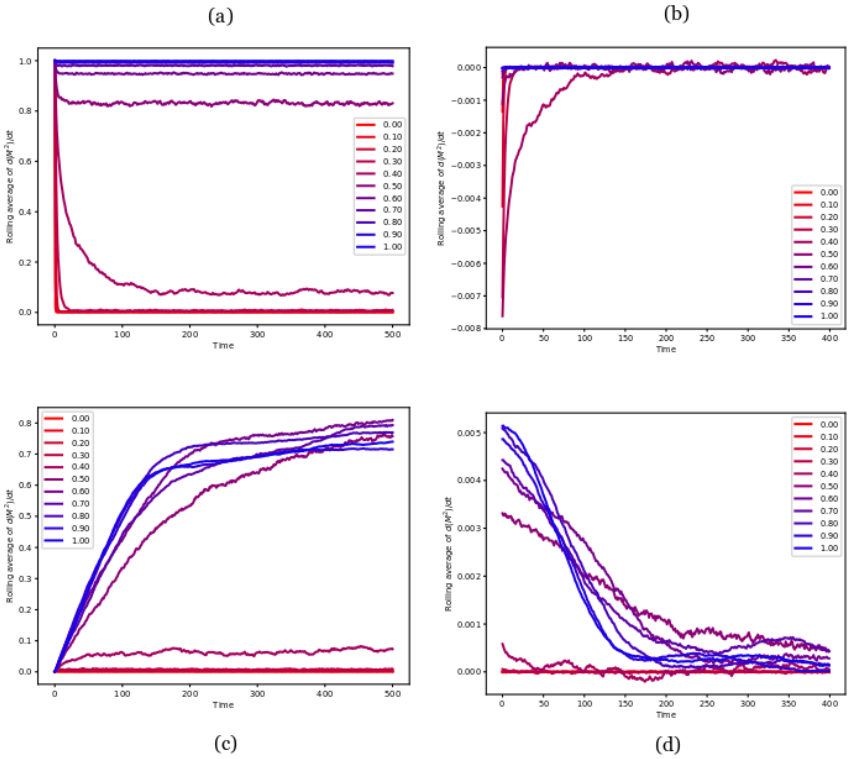

Figure 1: **Square magnetisation and smoothed derivatives versus time.** (a) init. aligned square magnetisation. (b) init. aligned derivatives. (c) init. random square magnetisation. (d) init. random derivatives.

3.2.3 Domains and cutoff

For subcritical temperatures and initially random conditions, convergence to equilibrium proceeds in two stages. At first, the magnetisation increases quickly. This corresponds to the fine-grained detail of the initial conditions being smoothed out. But as described in Section 3.8, domains form, and the magnetisation plateaus, increasing only very slowly. To save on computation time, I decided to consider the domained states as being "at equilibrium". This led me to set the cutoff to be 0.001 per Figure 1(d).

3.2.4 Results

Figure 2 shows that at high temperatures equilibrium is generally reached quicker. This makes sense as low temperatures slow down evolution. Initially random spins have many small-scale fluctuations, so it stands to reason that they would be closer to supercritical equilibrium, and vice-versa for subcritical and initially aligned spins.

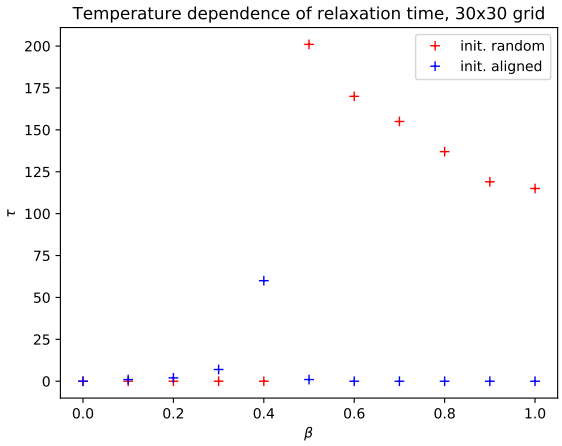

Figure 2: **Temperature dependence of relaxation time.** In red, the relaxation time for initially random spins. In blue, for initially aligned. Random spins are closest to supercritical equilibrium, aligned spins to subcritical. The relaxation time peaks close to criticality.

This information was useful in subsequent tasks as it informed the amount of iterations to trim off the beginning of each simulation to get equilibrium data.

3.3 Auto-correlation

I calculate the auto-correlation of the magnetisation at equilibrium at various T and N. This is used as a way of quantifying the timescale of fluctuations in equilibrium.

3.3.1 Data generation

For this task, I had to produce large amounts of equilibrium data with different parameters. I used initially aligned spins, per the results of the relaxation time investigation, randomising whether $M(t = 0) = \pm 1$.

In deciding how much data to generated, I considered two factors. First, in order for the auto-correlation data to be useful, the number of iterations should be many times the maximum time lag τ_{max}. Second, the number of systems in the ensemble must be suitably large so that a sample error can be calculated at each τ.

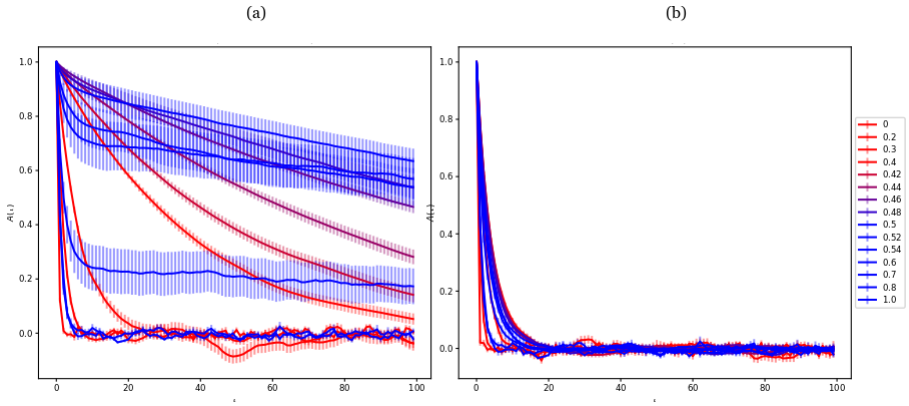

Figure 3: **Auto-correlation of (a) M and (b) $|M|$ at a small grid size.** At low temperatures, due to finite-size scaling, the grids fluctuate between the two equilibria produced by symmetry breaking. Thus $\langle M \rangle_t = 0$ even as $\langle |M| \rangle_t > 0$, producing the "long tails" in (a).

3.3.2 Data analysis

Originally, I directly calculated the auto-correlation of the magnetisation, but I noticed that it flattened out at large time lags for small grids at certain temperatures, rather than decaying exponentially (see for example Figure 3). By generating some mosaic animations, I determined that this was due to the finite-size scaling described in Section 3.1. (This animation can be viewed at the end of `equilibrium.avi`.)

This phenomenon isn't relevant at supercritical temperatures since there is no long-range order, nor at very low temperatures since there is no magnetisation-flipping (fluctuations become too small to overcome the free energy barrier). But at intermediate-low temperatures, the grid oscillates between two equilibria (M adopts a bimodal distribution across time even for a single system), and the auto-correlation ends up capturing that variation instead of the local fluctuation about a single equilibrium.

Although interesting, this phenomenon prevented me from extracting a useful τ_e (e-folding time lag) from the data, which was important in informing the choice of averaging time in later investigations. To get around this issue I took the auto-correlation of the *absolute* magnetisation.

I calculated τ_e by just finding the first index which went under $1/e$. A more accurate method would have been to fit an exponential decay curve or take logarithms and carry out a linear regression, but I felt this method was good enough to get a general idea of the fluctuation timescale.

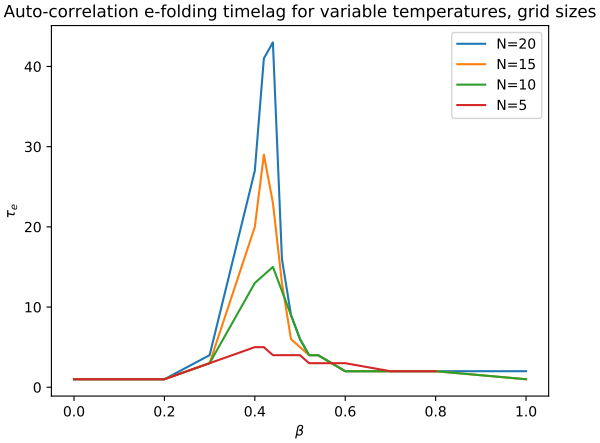

Figure 4: **Auto-correlation e-folding time τ_e versus temperature at different grid sizes.** Larger grid sizes leave more room for larger and longer-lived critical fluctuations.

3.3.3 Results

It turns out that the timescale of fluctuations is longer for larger grids and peaks near criticality. For infinite grids, critical fluctuations are scale invariant, so τ_e should go to infinity at T_c. In this model, however, the scale of fluctuations is limited by the size of the grid. This explains both observations.

3.4 Magnetisation

In this task the mean magnetisation at equilibrium is measured at various temperatures. This reveals the expected phase transition and magnetisation branching.

This was a comparatively simple task against the previous two. I re-used data from the auto-correlation measurement. I decided to keep only $N = 30$ for expediency.

3.4.1 Results

The first plot (Figure 5) is a scatter plot of the time-averaged magnetisation of every system in every ensemble (i.e.: every temperature value). We can clearly see the bimodal distribution at subcritical temperatures, with large fluctuations around the critical temperature. See the accompanying animation, `mags.avi`.

Due to the symmetry, the ensemble average of M is always zero, so to produce the second plot (Figure 6) I calculated the time-averaged square magnetisation

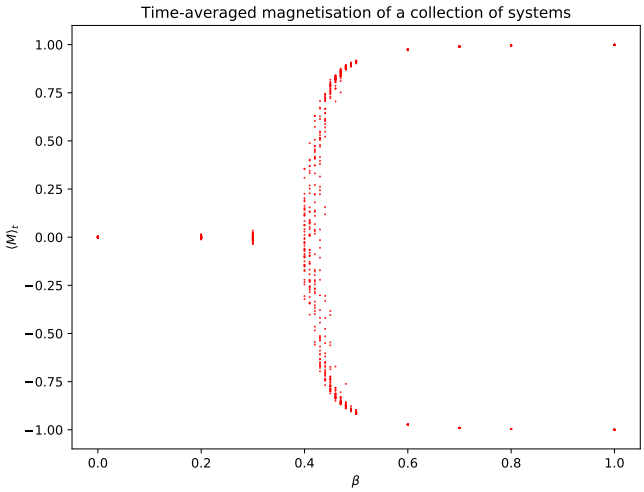

Figure 5: **Branching of magnetisation around criticality.** As the temperature is lowered from super to subcritical, the magnetisation branches to $M = \pm 1$.

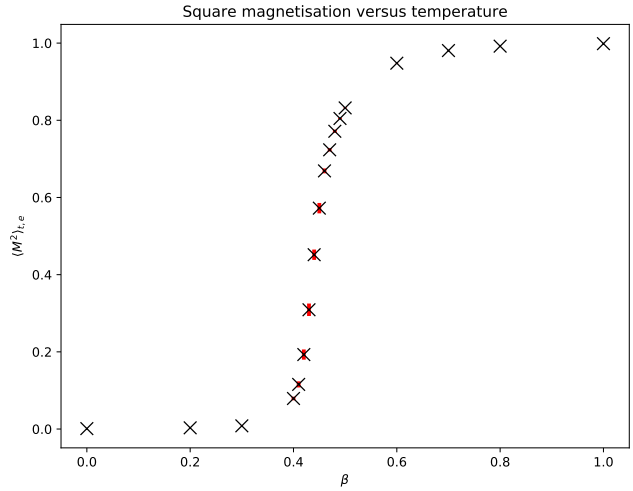

Figure 6: **Square magnetisation versus temperature at $N = 30$.** Standard error in the mean (ensemble mean) shown in red.

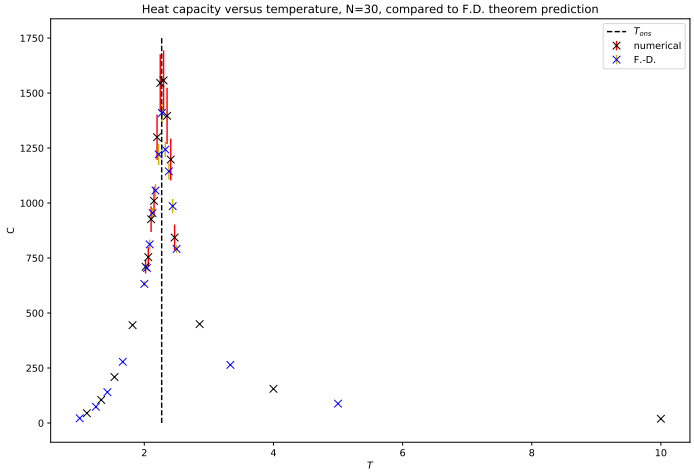

Figure 7: **Comparison of heat capacity calculations: numerical differentiation versus via Fluctuation-Dissipation theorem.** The heat capacity peaks near the Onsager critical temperature T_{ons}. Both calculations broadly agree in the shape of the chart.

of each system before ensemble-averaging.

If N were smaller, it would become easier for even very subcritical grids to undergo magnetisation flipping, so I would expect the critical temperature to become even blurrier. Conversely, as N goes to infinity, the region of criticality would become sharper and sharper until we recover the non-differentiability at T_{ons}.

3.5 Heat capacity

I measured the heat capacity versus temperature by calculating the mean energy while varying T and numerically differentiating. I also calculated the fluctuations in energy with time, and then used the fluctuation-dissipation theorem (F.-D. theorem) to get another calculation of the heat capacity.

I obtained the energy values by using the same data set as in the previous two tasks and writing `ising.thermo.energy( )`. Using `np.roll( )` allowed me to vectorise this calculation. The relevant modules are `thermo` and `main_energy` and are detailed with comments in the appendix.

Both methods produce similar graphs (Figure 7) with a large increase in the heat capacity near the critical temperature, slightly higher than the Onsager prediction. Although at every particular temperature, the two solutions agree

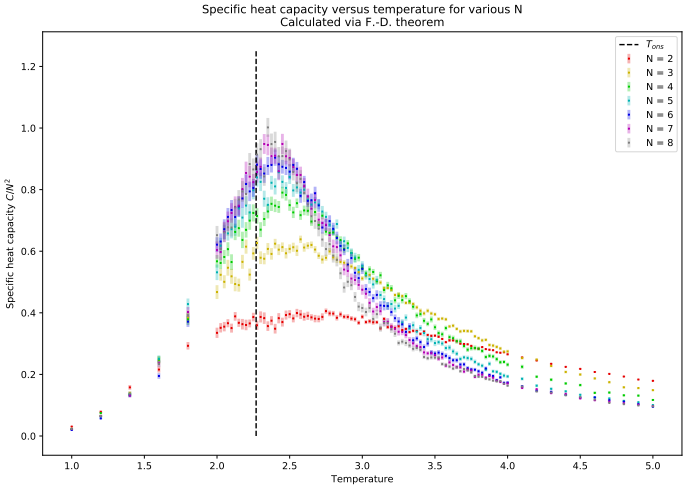

Figure 8: **Specific heat capacity versus temperature for small grid sizes displaying finite-size scaling.** The peak in C is spread out as N becomes smaller, and the maximum drifts to larger temperatures. This is due to finite-size scaling.

within 1σ, the F.-D. calculation appears to systematically underestimate the true value. However, it is much easier to calculate and produces much smaller errors. This could either be because the fluctuations are more consistent across the ensemble, or because the numerical differentiation requires taking a finite difference which amplifies the relative error.

3.6 Critical temperature and finite-size scaling

Inspired by the previous section's result, I decided to use the F.-D. method to obtain critical temperature measurements for a range of Ns.

For this task, I wanted to be able to impose a maximum relative error on the critical temperature. Therefore, unlike the previous tasks, I did not properly separate out the data generation from data analysis. Instead, each ensemble was simulated in chunks of 50 steps. At the end of each chunk, the heat capacity and error were calculated. This process was repeated until the heat capacity was found to a suitable relative error. The code is listed in the appendix under `main_scaling.py`.

Initially, I had tried to calculate the energy as in the previous section and numerically differentiated. But it was difficult to get a critical temperature measurement because of the error. This led me to adopt the F.-D. method.

In order to get the critical temperature from this data, I would ideally have fitted a model curve, but in the absence of a theoretical curve, I tried to find the maximum and its error by hand. Unfortunately, the data was not accurate enough to draw conclusions about the functional form of $T_c(N)$.

On the figure (Figure 8), at each N, the curve peaks somewhere near the Onsager temperature. Due to finite-size scaling, this peak becomes less pronounced as N decreases, and the maximum drifts to higher and higher temperatures.

3.7 Hysteresis

At low temperatures, the Ising model exhibits hysteresis. By incorporating a time-varying magnetic field, I was able to generate a hysteresis loop for various temperatures.

For this task, I first generated an equilibrium ensemble by setting the applied field to be zero and simulating an ensemble from aligned initial conditions for 200 iterations. I then trimmed off this initial simulation. I then introduced a sinusoidal time-varying applied field with a period of 200 by updating the `h` value passed onto `ising.simulator.iterate( )` at each time step. The ensemble-averaged magnetisation was calculated and its error evaluated at each time step and plotted against the corresponding value of `h`.

high temps: no hysteresis. low temps: hysteresis. even lower temps: no hysteresis again, because the field isn't strong enough to flip it. interesting!

As expected, at high temperatures, there is no hysteresis, and M and H are proportional. At low temperatures, a hysteresis loop forms, with M lagging behind H. Interestingly, however, at even lower temperatures, the hysteresis loop became very poorly defined and the errors on M became very large. This was because the applied H was not strong enough to overcome the free energy barrier at those lower temperatures, or was not applied for long enough, so the systems in the ensemble did not all flip as H reached its maximum magnitude. See the accompanying animation, `hysteresis.avi`.

3.8 Spatial features

This section details some qualitative aspects of the spatial distribution of spins in various situations that I observed through animations.

3.8.1 Domains

At low temperatures, the equilibrium behaviour should be for all the spins to align, as this is the lowest energy configuration. However, animations show that this does not happen. Instead, equilibrium is only reached locally: spins align all up or all down in different domains.

Why does this occur? If we consider two neighbouring regions, one all spin up and one all spin down, the spins in the middle of both regions are entirely stable –

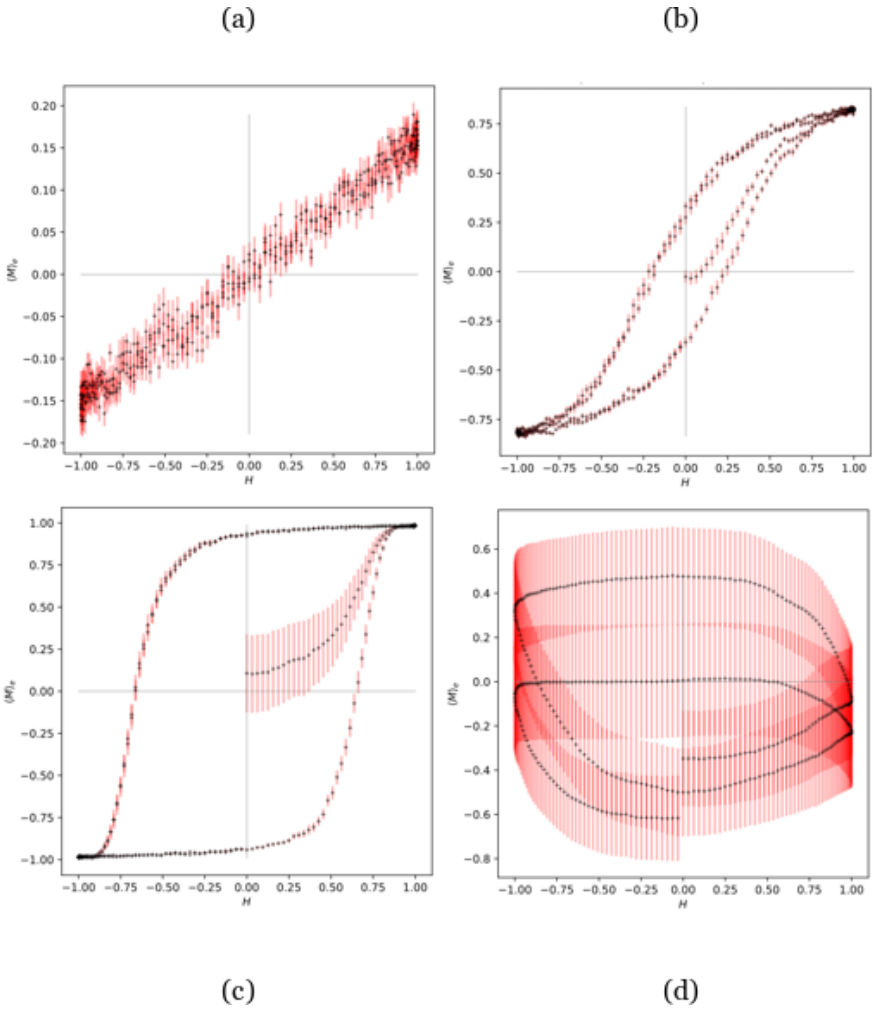

Figure 9: **Magnetisation versus oscillating applied field at various temperatures, showcasing hysteresis,** $N = 20$**.** Red: error bars. (a) $\beta = 0.1$, high temperature; no hysteresis. (b) $\beta = 0.3$ and (c) $\beta = 0.5$, low temperatures displaying increasing hysteresis. (d) $\beta = 1.0$, very low temperature – applied field fails to overcome free energy barrier.

small fluctuations away from the dominant spin will immediately be suppressed. It is only on the interface between the domains that small perturbations are likely to persist and allow one domain to expand against another.

Small-scale noise effectively consists of many domains, so the interface length : domain area ratio is large. Such noise is therefore quickly relaxed in favour of larger-scale variations, similar to the rapid decay of higher frequency modes in solutions to the diffusion equation.

Interestingly, perfectly horizontal and vertical domains are much more stable than diagonally aligned domains. This is because a fluctuation "sticking out" from such a domain is surrounded by 3 opposing spins. Contrast this to a 45 degree domain wall, where a fluctuation only has 2 opposing spins acting against it.

3.8.2 Nucleation during hysteresis

In animating the hysteresis task, I noticed that at low temperatures, the flipping of the spins occurred from a small number of "nucleation sites" and spread outwards from those, rather than flipping uniformly, creating interesting patterns. I suspect this is because fluctuations are rare at low temperatures, but when they do occur, the large applied field causes them to begin easily spreading.

3.8.3 Perspectives for further study

The spatial dynamics of the model would be interesting to study in further depth. Quantitative analysis of the spatial features could include:

- Calculating the spatial auto-correlation. This is a kind of convolution, so the grids could be studied in Fourier space, giving insight into the lengthscale of fluctuations.
- Studying the distribution of time auto-correlations of the spin at a particular point on the grid. This could be used to study domain stability.
- Quantitative study of domain shape. I would be interested to see a quantitative method of distinguishing horizontal/vertical domains from diagonal domains, and comparing their stability. This could be done by counting the number of aligned versus opposing neighbours for each spin and correlating that to the time taken before that spin flips.

4 Conclusion

In this project, I studied the Ising model for finite grids using the Metropolis algorithm to numerically sample the microstate phase space. I looked at qualitative and quantitative equilibrium properties of the Ising model proper, such as the magnetisation, critical temperature, heat capacity, and hysteresis effects, and studied how these varied with temperature and grid size. I also looked at

non-equilibrium properties arising due to Metropolis timestep-to-timestep correlations, like the relaxation time and auto-correlation. For qualitative analysis I relied heavily on animations which clearly showed the dynamics arising from the sampling method. By repeating the calculations over an ensemble of simulations with identical parameters, I obtained quantitative errors on many of the results.

5 Appendices

5.1 Appendix A: Code structure

Noting the importance of code re-use and program structure, I decided to write the `ising` package which contains general purpose code not specific to any task. I split this code up into readable-sized modules as much as possible. The actual main task files with the task-specific code were separated off into modules in the `tasks` or code root folder (the `main_*` files).

Here is a list of all modules and a brief description of what they do:

- /ising: Ising package.
  - datagen.py: Object-oriented interface for generating data using simulator.py, and saving/loading datasets to file.
  - loadingbar.py: Prints loading bars.
  - plotter.py: Allows plotting spin grids and producing animations using matplotlib.
  - simulator.py: Simulation code.
  - thermo.py: Common analysis functions.
- /data: Data storage.
- /results: Results (plots and animations).
- /tasks: Contains some modules with functions used by the main_*.py modules.
- main_*.py: Main (executable) code files for the tasks.

5.1.1 Organisation of the data

Early in the investigation, I wrote the module `datagen`, which contains an object-oriented interface to generate, save, and load data. Having this flexible foundation proved to be extremely useful during testing. The `Ensemble` class allowed me to quickly produce ensembles of simulations with identical parameters (temperature, grid size, number of iterations...). The `DataSet` class saves and loads groups of ensembles to file, while storing metadata on the ensemble

parameters in a json file, minimising confusion. Using this framework, I was able to easily re-use and re-purpose data across different tasks.

Intermediate calculation data was stored in the `/data` directory, and results (animations and plots) in `/results`. All these have been removed from the submitted work, so some of the code in the main task files will not be able to run unless all the data is re-generated.

In the end, I produced around 17.5GB of data.

5.2 Appendix B: Core code listing

5.2.1 simulator.py

```
"""Simulator for the 2D Ising model using a Metropolis method"""

import numpy as np
import numpy.random as npr
from pathlib import Path

from . import loadingbar


nwxs = np.newaxis


def new_grid(grid_shape, p=0.5):
    """
    Create new random initial state

    grid_shape: (int, int)
    p: float -- percentage of spin up
    """

    assert 0 <= p <= 1

    if type(grid_shape) is int:
        grid_shape = (grid_shape, grid_shape)

    return 2 * (npr.rand(*grid_shape) > p) - 1


def new_ensemble(grid_shape, sysnum, p=0.5, identical=False, randflip=False):
    """
    Create a new statistical ensemble of initial states

    grid_shape: (int, int)
    sysnum: int -- number of independent systems in the ensemble
    p: float -- percentage of spin up

    RETURNS: (sysnum, Nx, Ny)-array
    """

    assert 0 <= p <= 1

    if type(grid_shape) is int:
        grid_shape = (grid_shape, grid_shape)

    if identical:
        a = 2 * (npr.rand(*grid_shape) > p) - 1
        ret = np.repeat(a[nwxs, ...], sysnum, axis=0)
    else:
        ret = 2 * (npr.rand(sysnum, *grid_shape) > p) - 1

    if randflip:

        ret *= (-1)**npr.randint(0, 2, size=(sysnum, 1, 1))
```

```
    return ret


def _rand_flip_spin(spins, b, h, Nx, Ny, const_h=True, const_b=True):
    """
    Helper function for iterate()

    [!] modifies spins array in-place
    """

    if not const_h:
        h = h[i, j]
    if not const_b:
        b = b[i, j]

    # Pick out one random spin
    i = npr.randint(0, Nx)
    j = npr.randint(0, Ny)

    # Calculate Delta E
    dE = (
        spins[(i + 1) % Nx, j] +
        spins[i - 1, j] +
        spins[i, (j + 1) % Ny] +
        spins[i, j - 1] +
        h
    ) * spins[i, j]

    # Choose whether to flip it or not!
    if np.exp(-2 * b * dE) > npr.rand():
        spins[i, j] *= -1


def iterate(spins, b=1, h=0, inplace=False, const_h=True, const_b=True):
    """
    Step through one iteration on the spins.

    spins: int (Nx, Ny)-array
    b: float -- kinetic/temperature parameter, b = J/kT
    h: float -- field parameter, h = muH/J

    RETURNS: int (Nx, Ny)-array
    """

    if not inplace:
        # I'm going to be modifying spins in-place, so first I make a copy
        # of the spins grid.
        spins = np.copy(spins)

    Nx, Ny = spins.shape

    for k in range(spins.size):
        _rand_flip_spin(spins, b, h, Nx, Ny, const_h, const_b)

    return spins
```

```
def iterate_ensemble(ensemble, b=1, h=0, const_h=True, const_b=True):
    """
    Step through one iteration on an ensemble.
    """

    ensemble = np.copy(ensemble)

    sysnum = ensemble.shape[0]

    for k in range(sysnum):
        iterate(ensemble[k], b, h, inplace=True,
                const_h=const_h, const_b=const_b)

    return ensemble
```

5.2.2 thermo.py

```
"""Determining thermodynamic properties from Ising model simulation"""

import numpy as np


def magnetisation(a):
    """
    Calculate mean magnetisation

    a: (..., Nx, Ny)-array
    RETURNS: (...,)-array
    """

    return np.mean(a, axis=(-1, -2))


def square_mag(a):
    """
    Convenience function to calculate the square of magnetisation
    """

    return np.mean(a, axis=(-1, -2))**2


def energy(a, h=0):
    """
    Calculate energy of a grid

    Takes a (..., Nx, Ny)-array
    Returns a (...,)-array
    """

    # Each nearest-neighbour pair is either bottom-top or left-right.
    # To sum over every possible nearest-neighbour pair, generate
    # all bottom-top pairs and left-right pairs separately and add
    b = np.roll(a, 1, axis=-1)
    c = np.roll(a, 1, axis=-2)

    return -np.sum((b + c + h) * a, axis=(-1, -2))
```

```
def autocovariance(samples, maxtau=None, axis=-1, rem_dc=True):
    """
    Calculate the auto-correlation of a sampled function of time

    samples: float (iternum,)-array
    maxtau: int <= iternum
    rem_dc: bool -- whether to remove the DC component of the sampling
    """

    iternum = samples.shape[axis]

    if maxtau is None:
        maxtau = samples // 2
    else:
        assert type(maxtau) is int and maxtau <= iternum

    if rem_dc:
        samples = np.copy(samples - np.mean(samples, axis=axis))

    autoc_values = []
    for tau in range(maxtau):

        # Annoyingly, there's no way to pick an arbitrary axis using
        # slice notation so I have to use np.take

        # This is equivalent to s1 = samples[:,:,:-tau,:]; s2 = ..
        # with :'s in every axis but one
        s1 = np.take(samples, range(0, iternum - tau), axis=axis)
        s2 = np.take(samples, range(tau, iternum), axis=axis)
        autoc_values.append(np.mean(s1 * s2, axis=axis))

    return np.stack(autoc_values, axis=axis)


def autocorrelation(samples, maxtau=None, axis=-1):

    autocov = autocovariance(samples, maxtau, axis)
    return autocov / np.take(autocov, [0], axis=axis)


def rolling_average(values, window, axis=-1):
    """Take rolling average of array over axis"""

    # This solution is a bit messy because np.convolve only likes
    # 1d arrays and because slicing is messy if you need to take
    # values over an arbitrary axis

    # Basic idea is: the difference in the cumulated sum before and
    # after the window divided by the window is the average

    cs = np.cumsum(values, axis=axis)
    l = values.shape[axis]
    cs1 = np.take(cs, range(0, l - window), axis=axis)
    cs2 = np.take(cs, range(window, l), axis=axis)

    return (cs2 - cs1) / window
```

```
def isflat(testfunc, ensemble, timescale, tolerance, absolute=True):
    """
    Determines when and if a test function is ~constant over time
    over an ensemble

    testfunc: callable -- must take a microstate
    ensemble: datagen.Ensemble
    tolerance: float
    absolute: bool -- whether to interpret tolerance as absolute or relative
        defaults to True, i.e.: absolute tolerance

    RETURNS: (ensemble.iternum, - timescale)-array
    """

    iternum = ensemble.iternum

    # First, we calculate the ensemble average of the quantity
    # as it varies over time
    ens_avgs = np.array(ensemble.ensemble_avg(testfunc))

    # Next, we look at variations in that ensemble average over time
    diffs = np.diff(ens_avgs)

    # Take a rolling average of the changes over the timescale
    smoothed_diffs = rolling_average(diffs, timescale)

    # If the smoothed difference is less than the tolerance,
    # the test function can be said to be ~constant.
    #
    # i.e.: if
    if absolute:
        isflats = smoothed_diffs < tolerance
    else:
        smoothed_avgs = rolling_average(ens_avgs, timescale)
        isflats = smoothed_diffs / smoothed_avgs < tolerance

    return isflats
```

5.2.3 relaxation.py

Only key computational parts listed.

```
def generate(datapath, grid_size=30, sysnum=100, maxiternum=500):
    """Generate the data"""

    print("Generating data\n")

    datapath = Path(datapath)

    init_aligned_dataset = datagen.DataSet(datapath / "init_aligned")
    init_random_dataset = datagen.DataSet(datapath / "init_random")

    # Shared parameters
    h = 0

    # Generate initially aligned simulations
```

```
    p = 1.0

    for k in range(11):

        b = 0.1 * k
        print(f"aligned b = {b}")
        ens = datagen.Ensemble(grid_size, sysnum, p, b, h, identical=False)
        ens.simulate(maxiternum, reset=False, verbose=True)
        init_aligned_dataset.add_ensemble(ens, save=True)

    # Generate initially randomised simulations
    p = 0.5

    for k in range(11):

        b = 0.1 * k
        print(f"random b = {b}")
        ens = datagen.Ensemble(grid_size, sysnum, p, b, h, identical=False)
        ens.simulate(maxiternum, reset=False, verbose=True)
        init_random_dataset.add_ensemble(ens, save=True)

    print("Finally saving...")
    init_aligned_dataset.save()
    init_random_dataset.save()
    print("Done generating!\n")



def analyse(datapath, rolavg_window=100):
    """Analyse and return magnetisation data"""

    datapath = Path(datapath)

    datasets = [datagen.DataSet(datapath / "init_aligned"),
                datagen.DataSet(datapath / "init_random")]

    datasets[0].load()
    datasets[1].load()

    # Square magnetisation data, indexed by:
    # [dataset, ensemble][time]
    # dataset -- 0: aligned, 1: random
    mags = np.array([
        np.array([
            # ensemble.asarray(): [time, system, Nx, Ny]
            # -> magnetisation(...): [time, system]
            # -> np.mean(..., axis=-1): [time]
            np.mean(thermo.magnetisation(ensemble.asarray()), axis=-1)
            for ensemble in dataset.ensembles
        ])
        for dataset in datasets
    ])

    sqmags = mags**2
    diffs = np.diff(sqmags, axis=-1)
    smoothed_diffs = thermo.rolling_average(diffs, rolavg_window, axis=-1)
```

```
    return mags, sqmags, diffs, smoothed_diffs
```

5.2.4 main_energy.py

```
print("Calculating")


# for k, ens in enumerate(dataset.ensembles):

#     print(end=".")
#     ener_arr = thermo.energy(ens.asarray())  # indexed by time, system
#     energies.append(np.mean(ener_arr, axis=0))
#     flucts.append(np.std(ener_arr, axis=0))

# print()

# # These guys are saved to file indexed as [ensemble, system]
# energies = np.stack(energies, axis=0)
# flucts = np.stack(flucts, axis=0)
# np.save(datapath / "energies.npy", energies)
# np.save(datapath / "flucts.npy", flucts)


bs = np.load(datapath / "bs.npy")
energies = np.load(datapath / "energies.npy")
flucts = np.load(datapath / "flucts.npy")

sysnum = energies.shape[1]

# Best estimates and error obtained by averaging over each ensemble
est_energies = np.mean(energies, axis=1)
err_energies = np.std(energies, axis=1) / np.sqrt(sysnum)

# Take midpoints of each b value and associated temperatures
# These are the arguments of dE_db calculated below
midbs = (bs[1:] + bs[:-1]) / 2
Ts = 1 / midbs

# Then calculate dE/db and use chain rule to get
# heat capacity = caps = dE/dT = db/dT dE/db = -b**2 dE/dB
dE_db = np.diff(est_energies) / np.diff(bs)
caps = -midbs**2 * dE_db

# Now for everyone's favourite: error propagation!
# Since the steps in b are pretty big, it's clear that the dominant
# source of error is the random error across the ensemble, not
# numerical errors. So we can just use regular error propagation
# as applied to the finite-difference derivative formula,
#
#                       f(b) - f(a)
#    df/dx ((b+a)/2) ~= -----------
#                          b - a
#
# b and a are known exactly, so it's just the numerator error we need

rel_err_caps = (np.sqrt(err_energies[1:]**2 + err_energies[:-1]**2)
```

```
                / np.abs(np.diff(est_energies)))

err_caps = caps * rel_err_caps

# Second method for calculating the heat capacity, using the
# fluctuation dissipation theorem

# A bit confusing - we have fluctuations which are estimated using
# a standard deviation, and then we have errors on those fluctuations
# which are also obtained using a standard deviation. But the former
# is obtained with std over time and the latter with std over ensemble.
est_flucts = np.mean(flucts, axis=1)
err_flucts = np.std(flucts, axis=1) / np.sqrt(sysnum)

# Note "kb = 1" because of the units we chose to measure temp. with
fd_caps = est_flucts**2 * bs**2
fd_caperrors = 2 * est_flucts * bs**2 * err_flucts  # basic err. prop.
```

5.2.5 main_scaling.py

```
def find_heat_capacity(N, T, tol, sysnum=20):
    """Find heat capacity to specified tolerance"""

    # Basic working:
    # Create ensemble with given parameters.
    # Keep simulating, periodically calculating heat capacity using f-d
    # Error is estimated by avging over ensemble.
    # As soon as the heat capacity is found to a suitable tolerance,
    # return that value.

    relaxtime = 150
    maxtime = 5000
    checktime = 50
    sysnum = 100

    print(f"Finding heat capacity, N={N}, T={T:.2f}, {sysnum} systems\n")

    b = 1 / T
    ensemble = datagen.Ensemble(N, sysnum, p=1, b=b, h=0, randflip=True)

    # initial simulation to reach equilibrium
    ensemble.simulate(relaxtime + 10)
    ensemble.trim_init(relaxtime)

    total_iterations = 10

    done = False

    while total_iterations < maxtime and not done:

        print(end=f"Simulating {total_iterations} -> "
              f"{total_iterations + checktime}: ")

        ensemble.simulate(checktime, reset=False)

        arr = ensemble.asarray()
        energies = thermo.energy(arr)
```

```
        flucts = np.std(energies, ddof=1, axis=0)
        est_fluct = np.mean(flucts)
        err_fluct = np.std(flucts, ddof=1) / np.sqrt(sysnum)

        # Error propagation!

        est_cap = est_fluct**2 * b**2
        err_cap = 2 * err_fluct * est_fluct * b**2

        rel_err = err_cap / est_cap

        print(
            f"|err_caps / cap| = |{err_cap:.3f} / {est_cap:.3f}| = {rel_err:.3f}")

        if rel_err < tol:
            done = True

        total_iterations += checktime

    if not done:
        print(f"Exceeded max time of {maxtime}")

    return est_cap, err_cap
```

5.2.6 main_hysteresis.py

```
def get_hysteresis_loop(b, N, iternum, sysnum, hparams, anim=False):
    """
    Compute and return magnetisation v time values

    RETURNS: mags -- (iternum, sysnum)-array
             hs -- (iternum,)-array
    """

    a, p = hparams["maxh"], hparams["period"]
    hs = a * np.sin(2 * np.pi * np.arange(0, p) / p)

    # In this case, domain walls probably actually lead to
    # interesting behaviour, so set p = 0.5

    # Initially the applied field is zero, but we'll turn it on
    # once the relaxation time has elapsed
    ensemble = datagen.Ensemble(N, sysnum, p=0.5, b=b, h=np.zeros(1))

    ensemble.simulate(relaxtime + 1)
    ensemble.trim_init(relaxtime)

    ensemble.hs = hs
    ensemble.simulate(iternum - 1, reset=False, verbose=True)

    if anim:

        print("animating:")
        fig, _, _ = plotter.animate_mosaic(
            ensemble, timestamp=True,
            saveas=resultspath / f"N{N}-b{b:.2f}.mp4", verbose=True
```

```
    )
    plt.close(fig)

arr = ensemble.asarray()
mags = thermo.magnetisation(arr)

return np.resize(hs, iternum), mags
```