The Ising Model of a Ferromagnet

Word Count: 2981

Abstract

This report uses a Python implementation of the Metropolis (and Wolff) algorithm to explore the two-dimensional Ising model, equipped with toroidal periodic boundary conditions. The first part of the report analyses the statistical basis of the problem (i.e. that of Markov Chain Monte Carlo (MCMC) methods and their associated error estimates), as well as the ways in which computation could be improved. The remainder of the report then covers various aspects of the computational Ising model such as: autocorrelation, hysteresis, thermodynamic response functions, domain size, and Wolff's cluster-flipping algorithm as a method of overcoming critical slowing down. The quantitive results arise in the form of universal critical exponents, which are estimated using the finite-size scaling method. These results – which all agree with theoretical values to within one standard error – are $\nu = 0.95 \pm 0.10$, $C_0/\nu = 0.51 \pm 0.08$, $\gamma/\nu = 1.78 \pm 0.07$. The critical temperature of the infinite lattice was also estimated to be $T_c(\infty) = 2.27 \pm 0.04$, agreeing favourably with the analytical result due to Onsager.

Contents

1 Introduction

In his 1924 PhD thesis [1], Ising erroneously predicted – based on his investigation into the one-dimensional model – that there were no phase transitions for any lattice dimension d with temperature $T > 0$. However, it has since been shown that abrupt phase transitions occur for lattices with dimensions $d \geq 2$. In 1944, Lars Onsager published his proof for the two-dimensional Ising model [2], containing his analytical result for the critical temperature [3] for an infinite lattice

$$\frac{k_B T_c}{J} = \frac{2}{\ln\left(1+\sqrt{2}\right)} = 2.269185\ldots \tag{1}$$

The investigation involved writing a Python program (the main module of which can be found in Appendix A) to implement the Metropolis algorithm for the two-dimensional Ising model. Various properties of the model were obtained using computational methods and compared with the theoretical expectations.

First we analyse the statistical and computational aspects of the task at hand. Section 3 then discusses the implementation of said analysis, focussing on ways in which computational speed and efficiency were improved. Section 4 explores the results of the investigation, namely; autocorrelation, statistical error, hysteresis, response functions, finite-size scaling, domain size, and finally comparing the Metropolis algorithm to Wolff's cluster-flipping algorithm.

2 Analysis

2.1 Definition of The Ising Model

Consider $N \times N$ sites on a two-dimensional square lattice where each site is occupied by a spin $s_i = \pm 1$ where the positive and negative correspond to spin-up and spin-down respectively. The energy of the system is then given by

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

Here $\langle i, j \rangle$ runs over the nearest-neighbours, J is the exchange energy, μ is the magnetic moment and H is the external field. The regime where $J > 0$ corresponds to ferromagnetism, since the spins favour alignment. There is therefore a 'competition' between the exchange energy J trying to align the spins and the thermal energy $k_B T$ trying to randomise the spins. To simplify the problem[1], we set $J = k_B = 1$. In order to avoid edge effects, the two-dimensional square lattice is mapped onto a torus $\mathbf{T}^2 = S^1 \times S^1$ by imposing suitable periodic boundary conditions[2].

[1]Thus T is measured in units of J/k_B.

[2]This is discussed further in Section 3.3.

2.2 Markov Chain Monte Carlo (MCMC)

For the finite lattices in question, we aim to measure the temporal average of a macroscopic observable $\mathcal{A}$. By the *ergodic hypothesis* – which states that all accessible microstates are equally probable over a long period of time – this takes the form of a weighted ensemble average

$$\langle \mathcal{A} \rangle = \sum_x \mathcal{A}(x)\pi(x)$$

where the sum runs over all 2^{N^2} possible microstates of the system and $\pi(x)$ reflects the probability of the system being in a particular microstate of the spin ensemble given by $x = \{s_1, s_2, \ldots, s_{N^2}\}$. Thus x is an element of the phase space $x \in \{-1, +1\}^{N^2}$. This sum is so large that even for a small lattice, it becomes computationally expensive[3]. For this reason we use a Monte Carlo approximation

$$\frac{1}{n}\sum_{t=1}^{n} \mathcal{A}(x_t) \xrightarrow{P} \langle \mathcal{A} \rangle \tag{3}$$

where $\xrightarrow{P}$ denotes convergence in probability, n is the number of Monte Carlo time steps and x_t are samples from the target distribution $\pi(x)$. To generate these samples we create a Markov Chain with a stationary distribution equal to the target distribution and undergo a process of *importance sampling.* The target distribution under investigation is the Boltzmann distribution

$$\pi(x) \sim \exp\left[-\beta E(x)\right] \tag{4}$$

where $\beta = 1/(k_B T)$ and the energy $E(x)$ is defined by the Ising model in Eq.(2). One of the advantages of the MCMC approach is that the distribution need only be known up to a normalizing factor[4].

2.3 The Metropolis Algorithm

A common algorithm for prescribing a transition rule for the Markov chain is the Metropolis-Hastings algorithm, the general form of which goes as follows:

Given the current state x_t, propose a new microstate y from some *proposal distribution* $g(y|x_t)$. Draw from $U \sim \text{Uniform}[0, 1]$ and update the state according to

$$x_{t+1} = \begin{cases} y & \text{if } U < r(y, x_t) \\ x_t & \text{otherwise} \end{cases} \tag{5}$$

[3]E.g. an 8×8 lattice requires summing over $\sim 10^{19}$ possible states.

[4]In this case the partition function Z.

where $r(y, x_t)$ is the *acceptance rule*, which takes the form

$$r(y, x_t) = \min\left(1, \frac{\pi(y)}{\pi(x_t)}\frac{g(y|x_t)}{g(x_t|y)}\right) \tag{6}$$

For the Metropolis algorithm we require that the proposal distribution is symmetric, i.e. $g(y|x_t) = g(x_t|y)$. For the Boltzmann distribution in Eq.(4), the acceptance rule becomes

$$r(y, x_t) = \begin{cases} \exp\left[-\beta(E(x_t) - E(y)\right] & \text{for } E(y) \geq E(x_t) \\ 1 & \text{for } E(y) < E(x_t) \end{cases} \tag{7}$$

3 Implementation

3.1 Randomness

It was found that computation could be sped up by initialising the random numbers required for the algorithm *outside* of the loops themselves. In one line the random numbers could be generated into an array with size corresponding to the number of loops and subsequently called within the loop using the looping index integer. To give an idea of runtime for the whole program, the code was profiled line-by-line for one sweep of a $N = 16$ lattice. This is can be found in Appendix B.

3.2 Equilibration Time

A 'burn-in' time τ_{eq} was needed in order for the system to reach an equilibrium state. Preparing the lattice[5] in a random microstate of ± 1 spin states was found to be more effective than all-up or all-down since above the critical temperature T_c equilibration time was shorter. Figure 1 shows a randomly initialised 8×8 lattice become polarized to either all-up or all-down when below T_c, as well as the exponential relationship between lattice size and number of burn-in steps to reach equilibrium. It is worth noting the difference between individual steps of the algorithm and sweeps of the lattice (i.e. N^2 steps); the latter will be referred to as a *time step* hereafter.

3.3 Vectorization

One aspect where vectorization of the code led to a significant speed-up was the use of `numpy.roll` in calculating of the total energy E of the lattice. Profiling the program showed that there was a bottleneck in the total energy function, which looped the individual site energy function over all the lattice sites. However, it was found that instead of using modulo operator `%` to impose the periodic boundary conditions, the `roll` functions could be used to operate on the array as a whole. Whilst the `roll` functions

[5] I.e. a $N \times N$ `numpy` array.

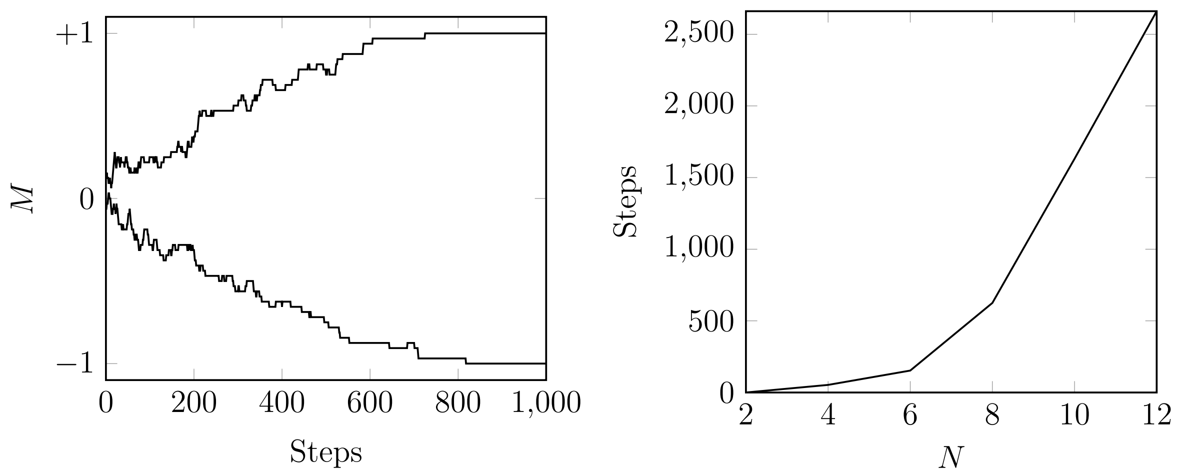

Figure 1: (Left) Plot of magnetisation per spin vs. individual steps of the Metropolis algorithm for $N = 8$. (Right) Number of individual steps to reach $|M| = 1$ vs. lattice size N. Both plots are for $T = 1$.

takes time to allocate new space, once this has been allocated the performance increases greatly since there are no cache misses from the CPU cache. Figure 2 shows the runtime of the `%` method increase exponentially as lattice size N increases, whilst the vectorized `roll` functions does not suffer this slow down[6]. Similarly, the total magnetisation M could be straightforwardly calculated by calling the `numpy.sum` function on the lattice array.

3.4 Memory Allocation

The program made use of *in-place operators* such as `+=` with a view to reducing the number of requests to the operating system to store data. For this reason, variables were allocated as 'scratch space' before the time step loop and were updated upon each iteration. Another positive of this approach was that the relevant variable could be updated only if the Metropolis algorithm accepted the proposed microstate[7]. These counters could then be averaged in one calculation at the end of the looping process.

4 Results and Discussion

4.1 Autocorrelation

Whilst MCMC methods are extremely useful for tackling problems of high analytical complexity and dimensionality, they are not without drawbacks. The principal concern

[6]At least over the lattice sizes used in the investigation.

[7]I.e. if the spin had been flipped.

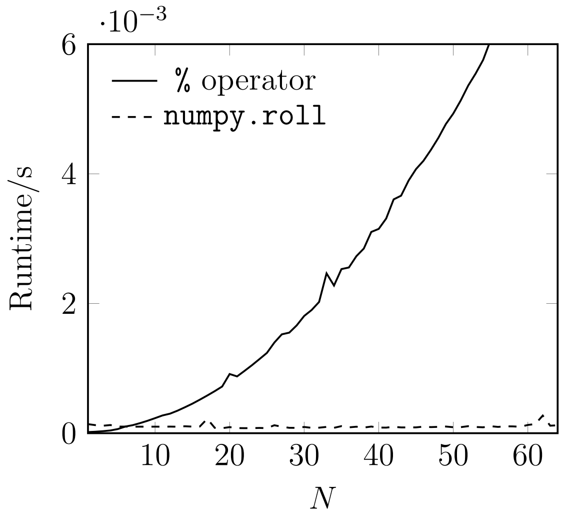

Figure 2: Plot of run time vs. lattice size N for total energy function using the modulo operator `%` and `numpy.roll`.

is that the resulting samples are often highly correlated. This leads to abnormally high variances and correspondingly inaccurate values for the thermodynamic quantities involved. The first requirement is to measure the number of time steps which is sufficient to make the thermodynamic averages robust. Since the Metropolis algorithm flips one spin site at a time, the value of an observable will be strongly correlated with the previous measurement. For example, the magnetisation takes the form

$$M = \frac{1}{N^2} \sum_{i=1}^{N^2} s_i \tag{8}$$

which changes as $\mathcal{O}(1/N)$ with each time step, hence a large number of steps are needed to take an independent sample.

To quantify this 'stickiness' [4] we use the *autocorrelation*. The lag-τ correlation for a time series $M^{(1)}, M^{(2)}, \ldots, M^{(n)}$ in equilibrium is defined as

$$a(\tau) = \mathrm{corr}(M^{(1)}, M^{(\tau+1)}) = A(\tau)/A(0) \tag{9}$$

where $A(\tau)$ is the *autocovariance* such that

$$A(\tau) = \langle M'(t)M'(t-\tau)\rangle \tag{10}$$

Here $\langle\cdot\rangle$ denotes time-averaging and $M' = M - \langle M\rangle$ is the deviation from the mean. It is observed in Figure 3 that $a(\tau)$ decays exponentially, thus we can define the *exponential autocorrelation time* τ_e such that $a(\tau) \sim \exp(-\tau/\tau_e)$.

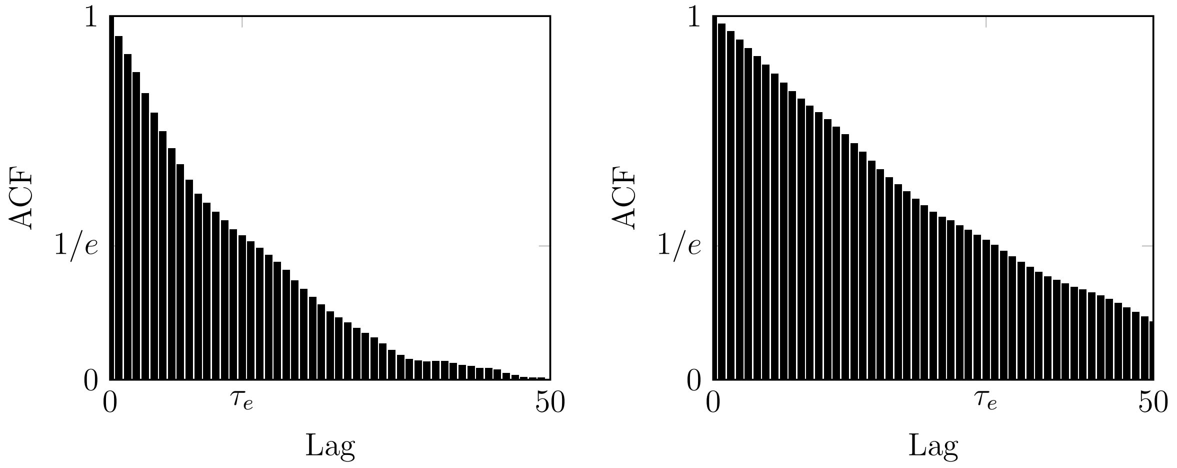

Figure 3: Autocorrelation plot for individual iterations of the Metropolis algorithm showing exponential decay, with $T = 5$ for $N = 4$ (left) and $N = 8$ (right). The plots also show the $1/e$ factor and the corresponding exponential autocorrelation time τ_e.

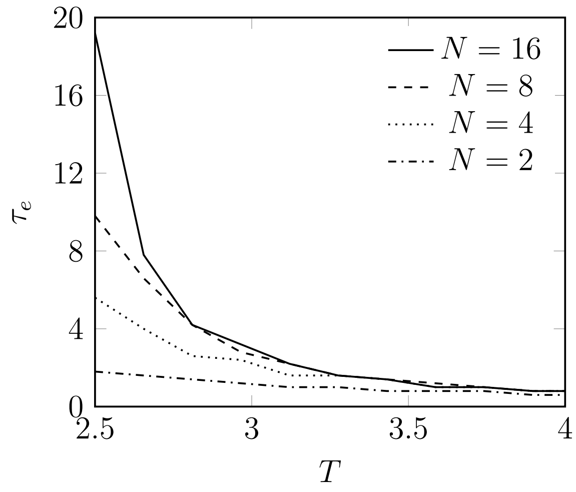

Figure 4: Plot of exponential autocorrelation time τ_e vs. temperature T for varying lattice size N. The data was generated from the total magnetisation M for the system in equilibrium and exhibits the phenomenon of critical slowing down (outlined in Section 4.7).

An autocorrelation function `acf()` was written using the `numpy.correlate` function. The 1D arrays containing the magnetisation values at each step were correlated to themselves, and subsequently normalized. This was then iterated over a range of lattice sizes N and temperatures T, starting from just above the critical temperature T_c. The variation in τ_e is shown in Figure 4. Since τ_e reflects the convergence speed of the MCMC sampler[8] it is necessary to provide a more rigorous quantitive analysis of the effect of autocorrelation on statistical error.

4.2 Statistical Error

Including the equilibration time and autocorrelation time, the averages of thermodynamic quantities thus take the form

$$\bar{\mathcal{A}} \propto \sum_{t>\tau_{eq}}^{t_f>>\tau_e} \mathcal{A}(x_t) \Longrightarrow \frac{1}{n}\sum_{t=1}^{n} \mathcal{A}(x_t) \tag{11}$$

where the mean value $\bar{\mathcal{A}}$ is an *estimator* of the theoretical value $\langle \mathcal{A} \rangle$. The estimator $\bar{\mathcal{A}}$ is a *random* variable which fluctuates around the this theoretical value for n finite time steps. The Monte Carlo error is straightforwardly given by these fluctuations so that the variance in $\bar{\mathcal{A}}$ is

$$\sigma^2_{\bar{\mathcal{A}}} = \left\langle \bar{\mathcal{A}}^2 \right\rangle - \left\langle \bar{\mathcal{A}} \right\rangle^2 \tag{12}$$

meaning that the simulation does not need to be run multiple times to obtain an error. Using the notation $\mathcal{A}(x_t) \equiv \mathcal{A}_t$ we now insert Eq.(11) into Eq.(12), separating diagonal and off-diagonal terms to yield

$$\sigma^2_{\bar{\mathcal{A}}} = \frac{1}{n^2}\sum_{t=1}^{n}\left(\left\langle \mathcal{A}_t^2 \right\rangle - \left\langle \mathcal{A}_t \right\rangle^2\right) + \frac{1}{n^2}\sum_{t\neq k}^{n}\left(\left\langle \mathcal{A}_t\mathcal{A}_k \right\rangle - \left\langle \mathcal{A}_t \right\rangle \left\langle \mathcal{A}_k \right\rangle\right) \tag{13}$$

Since the data is indeed correlated the off-diagonal term must be included. Rearranging leads to

$$\sigma^2_{\bar{\mathcal{A}}} = \frac{\sigma^2_{\mathcal{A}}}{n}\left[1 + 2\sum_{\tau=1}^{n-1} a(\tau)\left(1 - \frac{\tau}{n}\right)\right] \tag{14}$$

where again $a(\tau)$ is the autocorrelation. At this point it is useful to define the *integrated autocorrelation time* τ_{int} which reflects the statistical efficiency of the algorithm [4]

$$\tau_{int} = \frac{1}{2} + \sum_{\tau=1}^{n-1} a(\tau)\left(1 - \frac{\tau}{n}\right) \approx \frac{1}{2} + \sum_{\tau=1}^{\infty} a(\tau) \tag{15}$$

Hence, Eq.(14) can be rewritten as

$$\sigma^2_{\bar{\mathcal{A}}} = \frac{\sigma^2_{\mathcal{A}}}{n} 2\tau_{int} \tag{16}$$

[8]I.e. that second largest eigenvalue of the Markov chain transition matrix [4].

We see that the variance is increased by a factor or $2\tau_{int}$. The *effective sample size* can thus be defined as

$$n_{\text{eff}} = \frac{n}{2\tau_{int}} \leq n \tag{17}$$

which implies that the data is approximately uncorrelated for every $2\tau_{int}$ time steps. Choosing a sufficient n_{eff} does not just affect the error on the observables, but can also effect bias. We will see in Section 4.4 that the variances involved in the investigation dictate the values of heat capacity of susceptibility. The estimator for the variance of an observable[9] is given by [5]

$$\hat{\sigma}^2_{\mathcal{A}} = \sigma^2_{\mathcal{A}}\left(1 - \frac{1}{n_{\text{eff}}}\right) \tag{18}$$

It is possible to use the values for $\tau_e(N)$ derived in Section 4.1 as an approximation for τ_{int}. Summing the infinite geometric series and expanding the exponential power series gives

$$\tau_{int} \approx \sum_{\tau=0}^{\infty} e^{-\tau/\tau_e} - \frac{1}{2} = \frac{1}{1 - e^{-1/\tau_e}} - \frac{1}{2} \approx \tau_e \tag{19}$$

Thus there is a choice of whether to increase the number of time steps for large lattices thereby using an approximately uniform n_{eff} throughout, or to use a uniform number of time steps n throughout. The latter was chosen for this investigation, since the runtime of a large lattice[10] can reach multiple days for a large number ($\sim 10^6$) of time steps. Hence, getting a rougher estimate for the large lattice was preferred over none at all. For this reason, $n = 10^4$ time steps were used in the investigation, leading to the 'master' equation for the error

$$\sigma_{\bar{\mathcal{A}}} = \frac{\sigma_{\mathcal{A}}}{100}\sqrt{2\tau_e} \tag{20}$$

where τ_e is a function of N and must therefore be calculated for each lattice.

4.3 Hysteresis

In general, hysteresis refers to the dependence of a state on its history. In the case of ferromagnetism the system is said to exhibit hysteresis with respect to an external field. For convenience we set $\mu = 1$ in the Ising model definition in Eq.(2). Thus, when a large enough external field H is applied to the system and then 'switched off' (i.e. $H \to 0$) we expect there to be a residual magnetisation. This is exhibited computationally, as shown in Figure 5. For small T/T_c, there is a large 'nucleation' energy cost to flipping a spin when the system is in a homogenous polarized state. Once this nucleation barrier is broken[11], the system quickly flips its polarization.

[9]For example the heat capacity is proportional to $\hat{\sigma}^2_E/T^2$.

[10]Here 'large' is taken to mean the largest lattice in the investigation $N = 64$.

[11]I.e. the activation energy is overcome.

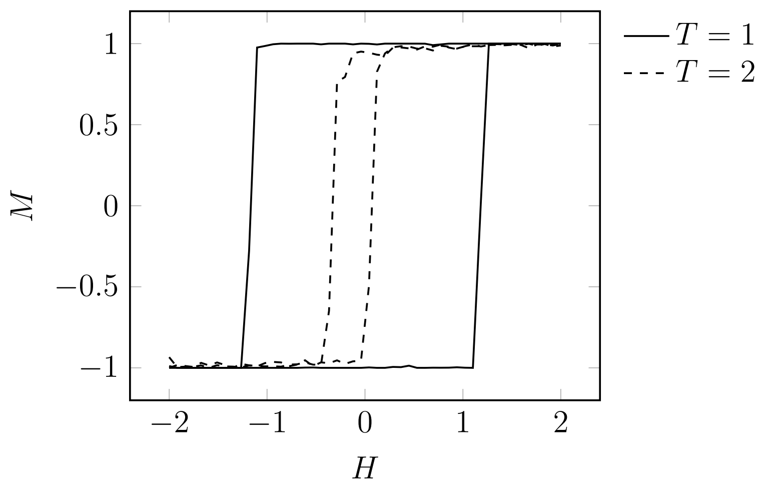

Figure 5: Plot of absolute magnetisation per spin vs. external field showing hysteresis loops ($N = 4$).

4.4 Response Functions

The thermodynamic observables relevant to the investigation are the energy E and the magnetisation M. For purposes of future calculations (see Section 4.5), intensive quantities are required, therefore E and M (shown in Figure 6) will henceforth refer to energy per spin and magnetisation per spin, unless otherwise stated. For an infinite lattice, the system undergoes spontaneously broken ergodicity for $T < T_c$, however this does not occur for the finite lattice under investigation. For this reason, the absolute magnetisation $|M|$ is used instead of M to avoid the system time-averaging to zero[12]. As the number of lattice sites approach infinity, $|M|$ tends toward the analytical value [6] for M given by

$$M(T, N \to \infty) = \begin{cases} 0 & \text{for } T \geq T_c \\ \left[4\sqrt{2}\ln\left(1+\sqrt{2}\right)\right]^{1/8}(-t)^{1/8} & \text{for } T \to T_c^- \end{cases} \tag{21}$$

Where the reduced temperature t is defined to be $(T - T_c)/T_c$. Figure 7 shows this $M \sim (-t)^{1/8}$ dependence with computational data for $N = 32$.

The corresponding response functions for E and M are the heat capacity C and the susceptibility χ respectively. Both are examples of the general *fluctuation-dissipation theorem* [7] and are defined by the following

$$C = \frac{\partial E}{\partial T} = \frac{\sigma_E^2}{k_B T^2} = \frac{\langle E\rangle^2 - \langle E^2\rangle}{k_B T^2} \tag{22}$$

[12]Since it has equal likelihood of being in the all-up or all-down state.

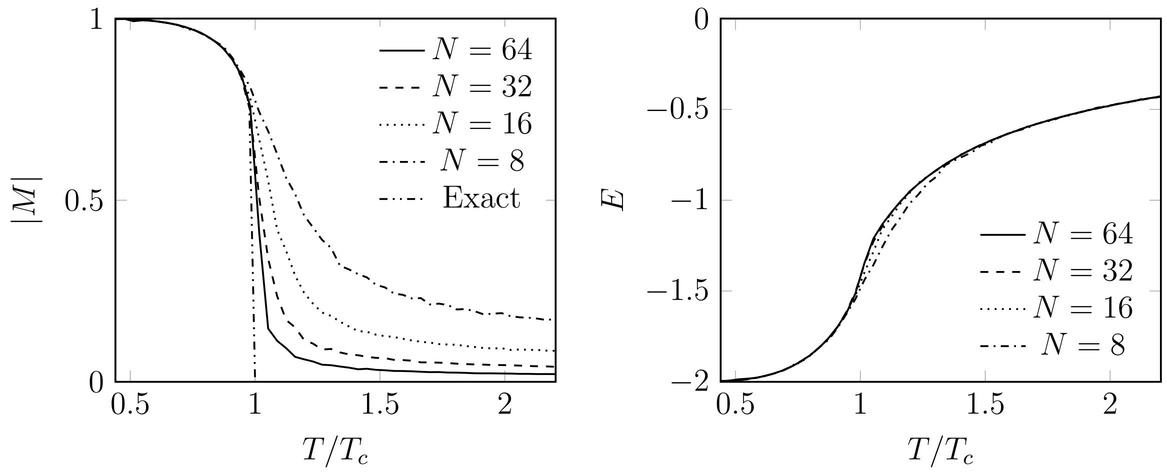

Figure 6: (Left) Absolute magnetisation per spin vs. dimensionless temperature (using Onsager's analytical value). The exact magnetisation per spin is also plotted using Eq.(21). (Right) Energy per spin vs. dimensionless temperature.

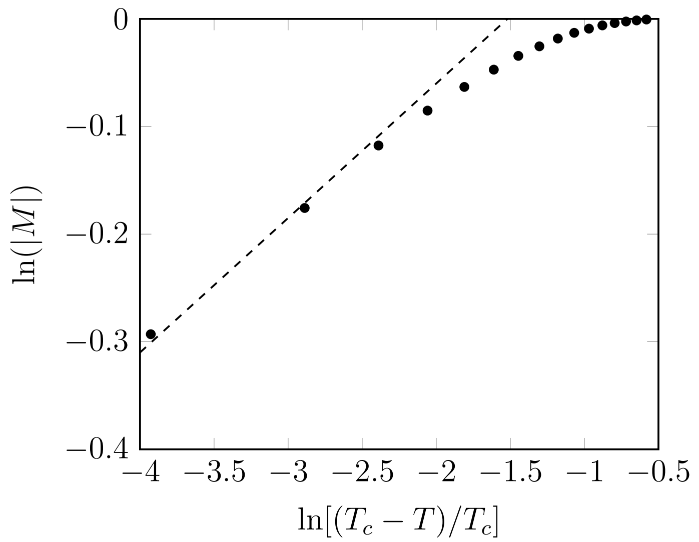

Figure 7: Logarithmic plot of absolute magnitude vs. reduced temperature for $N = 32$. A straight line with the slope's theoretical value of $\beta = 0.125$ has been fitted to the data.

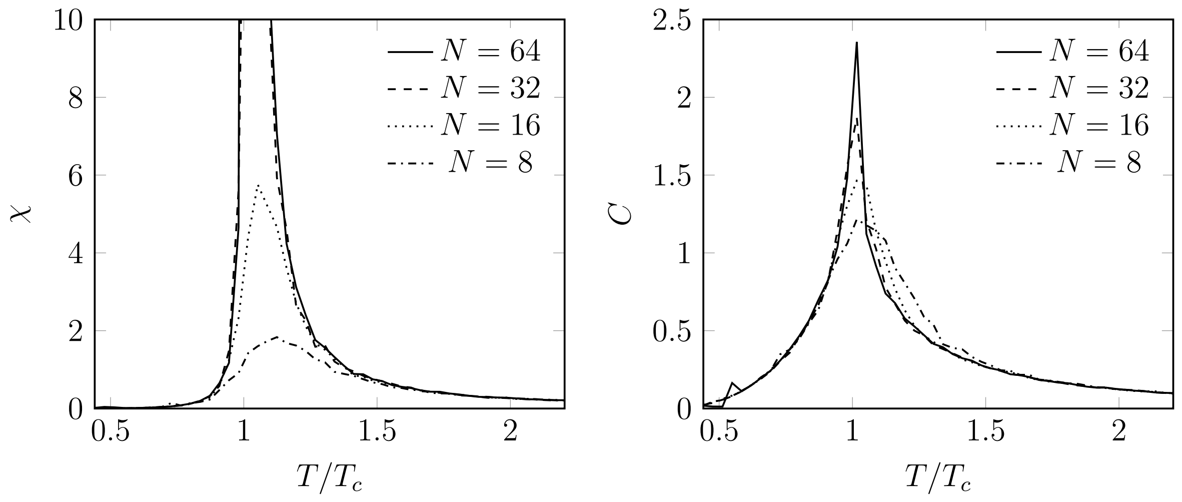

Figure 8: (Left) Plot of susceptibility per spin χ vs. dimensionless temperature T/T_c (using Onsager's analytical value) for varying lattice size N. The plot's vertical axis is truncated at $\chi = 10$ to illustrate the peaks becoming more exaggerated as N increases. (Right) Same plot as left but for heat capacity per spin C. For both plots, the errorbars have been suppressed for clarity.

$$\chi = \frac{\partial M}{\partial T} = \frac{\sigma_M^2}{k_B T} = \frac{\langle M \rangle^2 - \langle M^2 \rangle}{k_B T} \tag{23}$$

The resulting plots for Eq.(22) and (23) are shown in Figure 8. For $N \to \infty$ at $T = T_c$ the values for χ and C are expected to diverge [6]. Since the lattice size is finite, the sharp peaks around $T = T_c$ shown in the two plots agree with the theory, with the peaks becoming more exaggerated for larger N.

4.5 Finite-Size Scaling

We have already seen that computation is restricted to finite lattice sizes, and will therefore produce inaccurate values for $T_c(\infty)$. However this can be overcome by using a method called *finite-size scaling* whereby critical exponents can be estimated by measuring observables as a function of lattice size. We have already seen this for the critical exponent β in the case of magnetisation in Section 4.4. The universal critical exponents for the thermodynamic observables relevant to the investigation can be found in Table 1.

In particular, there is an equation which defines the critical exponent ν in terms of lattice size such that

$$T_c(N) = T_c(\infty) + aN^{-1/\nu} \tag{24}$$

Using the data for χ shown in Figure 8, values for $T_c(N)$ can be derived for each lattice

Exponent	Definition	Analytic
α	$C \sim \|t\|^{-\alpha}$	0
β	$M \sim (-t)^{\beta}$	0.125
γ	$\chi \sim \|t\|^{-\gamma}$	1.75
ν	$\xi \sim \|t\|^{-\nu}$	1

Table 1: Table showing the relevant universal critical exponents.

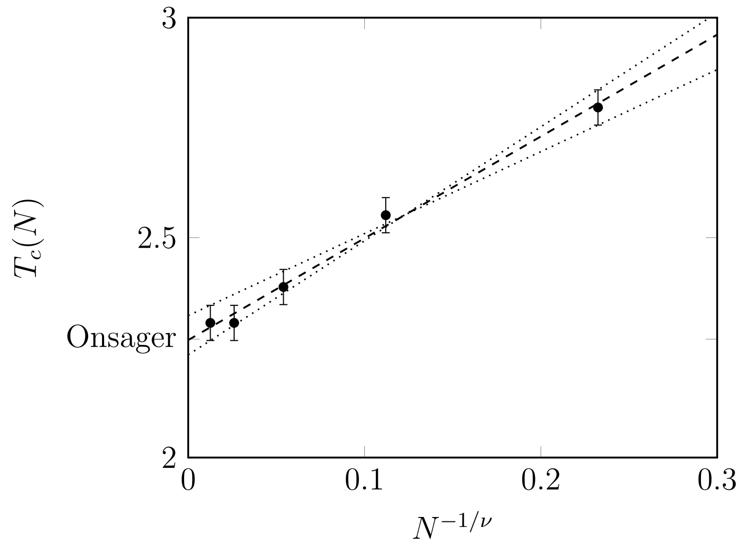

Figure 9: Plot of estimated critical temperature using susceptibility data against $N^{-1/\nu}$ using $\nu = 0.95$. The analytical value for $T_c(\infty)$ due to Onsager is also shown, as well as the maximum and minimum gradients (thin dotted line).

size. Eq.(24) suggests that the ν parameter can then be adjusted until the plot of T_c against $N^{-1/\nu}$ sits on a straight line, thus yielding an estimate for ν, as well as the y-intercept $T_c(\infty)$ and slope a. This procedure was carried out in Figure 9, giving the following estimates, which agree favourably with the theoretical values: $\nu = 0.95 \pm 0.10$, $a = 2.3 \pm 0.4$ and $T_c = 2.27 \pm 0.04$. The errors on the a and T_c values were derived from the maximum and minimum gradient, as shown on the figure. The errorbars on T_c were chosen to be half the width of the temperature T interval. The error on ν was estimated 'by eye', given the range of possible values that led to what could be considered a straight line.

We can take this method further and use our value for ν to estimate the other critical exponents in Table 1, using the fact that

$$C_{\max} \sim N^{\alpha/\nu} \tag{25}$$

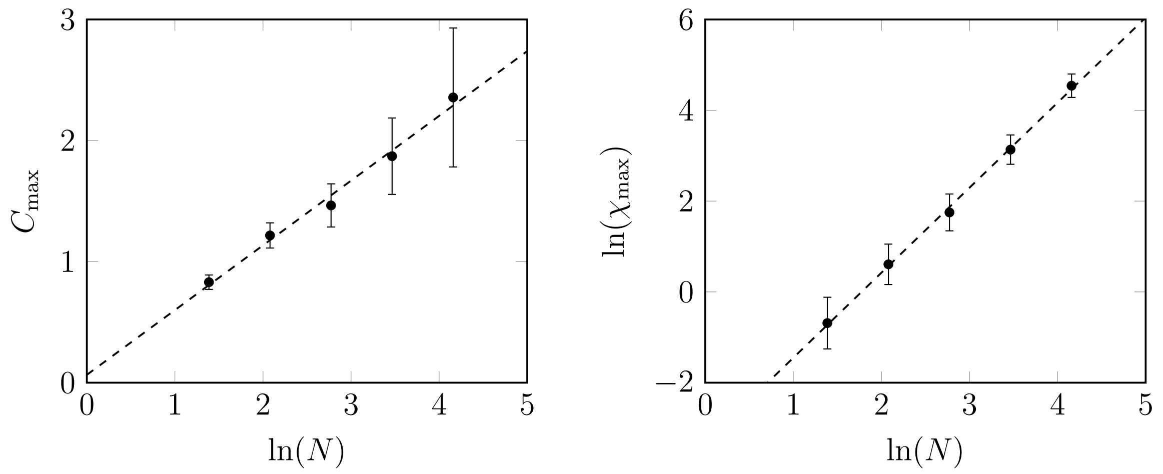

Figure 10: (Left) Plot of maximum heat capacity $C_{\max}$ value vs. $\ln(N)$. The slope provides an estimate for C_0/ν. The two largest data points were not used in the calculation due to their large errors. (Right) Plot of maximum susceptibility $\ln(\chi_{\max})$ value vs. $\ln(N)$. The slope provides an estimate for γ/ν.

$$\chi_{\max} \sim N^{\gamma/\nu} \tag{26}$$

Thus a logarithmic plot of the value of C and χ evaluated their peak against N should yield slopes of α/ν and γ/ν respectively. However, there is a slight complication with measuring the α exponent, since it is analytically equal to zero, corresponding to a logarithmic singularity since for $T \to T_c^{\pm}$ [6]

$$C \sim \ln|t| \Longrightarrow C \sim \lim_{\alpha\to 0} \frac{1}{\alpha}\left(|t|^{-\alpha} - 1\right) \tag{27}$$

For this reason, the form of Eq.(25) is in fact $C \sim C_0 \ln(N)$. This method was carried out in Figure 10, the results of which are left in the form of critical exponent ratios so that the error in the ν estimate does not dominate and lead to imprecise values. These results are $C_0/\nu = 0.51 \pm 0.08$ and $\gamma/\nu = 1.78 \pm 0.07$. Both errors were obtained using the maximum and minimum gradients of the graph[13], the errorbars of which were estimated using Eq.(20) for correlated Monte Carlo error, derived in Section 4.2. The values lie within one standard error of the theoretical values $\gamma/\nu = 1.75$ and $C_0/\nu = 0.5$.

4.6 Domain size

A domain is defined as a cluster of equal spin connected via nearest neighbours, as shown in Figure 11. The largest domain of a lattice is a quantity of interest as it is closely

[13]Excluding the $N = 32$ and $N = 64$ points in the heat capacity case.

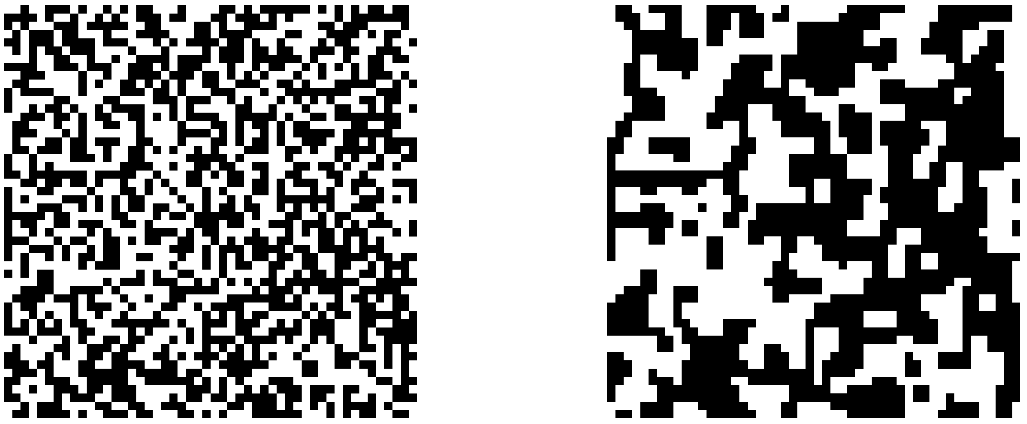

Figure 11: (Left) Randomly initialised 50×50 lattice. (Right) The same lattice after 100 time steps at $T = 1$. It can be seen that magnetic domains of the same spin are beginning to form.

linked to the correlation length ξ [6]. To calculate the largest domain size per spin D for the lattice, a connected-component labelling function was written. It was necessary to again implement toroidal periodic boundary conditions since the largest cluster may wrap around the edges. The most efficient implementation of the cluster area calculation was found to be the `scipy.ndimage.measurements` module, the results of which are shown in Figure 12. The theoretical expectation is that the correlation length diverges at the critical temperature, however since the lattice is finite this corresponds to $D = 1$. Thus, the family of curves in Figure 12 is in keeping with the theoretical expectation.

4.7 Critical Slowing Down and the Wolff Algorithm

Whilst the Metropolis algorithm used throughout is sufficient for high and low temperatures, we have seen that it suffers from highly correlated samples in the vicinity of T_c. This is known as *critical slowing down* and can be described by the following exponential relation

$$\tau \sim \left| \frac{T - T_c}{T_c} \right|^{-z\nu} \tag{28}$$

where z is known as the *dynamical critical exponent.* It is worth noting that z is not universal, but depends on the algorithm used. A large value for z corresponds to significant slowing down in the vicinity of T_c. Since $\tau_e \sim N^{z/\nu}$, fitting a straight line to a logarithmic plot of τ_e vs. N for $T = 2.5$ (i.e. close to critical temperature) yields an estimate of z.

With this in mind, Wolff [8] introduced an alternative cluster flipping algorithm[14] whereby an individual algorithm step does not just update one spin variable, but a

[14]Based on the Swendsen-Wang algorithm.

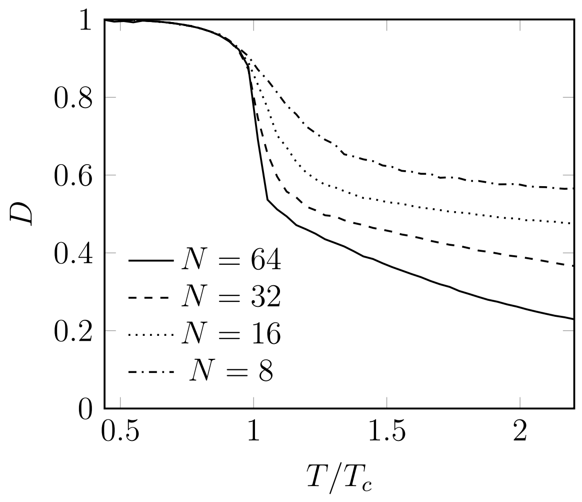

Figure 12: Plot of largest domain size per site D vs. dimensionless temperature T/T_c for varying lattice size N.

group of them. The Wolff algorithm goes as follows:

1. Choose a random spin site s_i on the lattice, thereby initialising a cluster of size one.
2. Inspect the nearest neighbour spin sites s_j. If $s_j = s_i$ then add the spin site to the cluster with probability $P(\text{add}) = 1 - e^{-2\beta J}$.
3. Repeat 2. with the added spin sites. Break when cluster is finished.
4. Flip all of the spin sites in the cluster with probability $r(y, x_t) = 1$

To highlight the difference from the Metropolis algorithm outlined in Section 2.3, we note that the proposal distributions are no longer symmetric. In the Wolff algorithm $g(x_t|y) = (1 - P(\text{add}))^m$ and $g(y|x_t) = (1 - P(\text{add}))^n$ where m and n are the number of same spin neighbouring bonds around the *domain wall* (i.e. the perimeter of the cluster) with m and n corresponding to before and after the change in microstate, respectively. In order to see from where $P(\text{add})$ originates, the *detailed balance* condition is imposed such that

$$\frac{r(x_t, y)}{r(y, x_t)}(1 - P(\text{add}))^{n-m} = e^{-\beta(E(y)-E(x_t))} \tag{29}$$

Since $E(y) - E(x_t) = 2J(n - m)$ for the Ising model, to ensure that acceptance ratio equals one for all transitions, we must then have $P(\text{add}) = 1 - e^{-2\beta J}$.

The effect of the Wolff algorithm is to explore the phase space $\{-1, +1\}^{N^2}$ at a much faster rate, thereby avoiding the critical slowing down effect. This can be seen in Figure

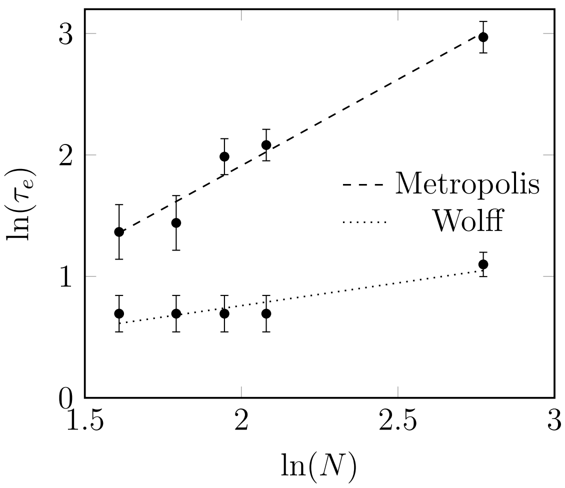

Figure 13: Plot of $\ln(\tau_e)$ vs. $\ln(N)$ for the Metropolis and Wolff algorithms at $T \approx T_c$. The slope yield an estimate for z/ν.

13, which compares the dynamical exponent z for the two algorithms in question. The estimate for the dynamical exponent for the Wolff algorithm was $z_w/\nu = 0.37 \pm 0.21$, compared to $z_m/\nu = 1.4 \pm 0.3$ for the Metropolis algorithm, where the errors were derived from the maximum and minimum gradients and errorbars estimate using ± 1 lag-τ. The value for z_m is somewhat lower than the expected value of $z_m \approx 2$, though this could be explained by the fact that only lattices up to $N = 16$ were used in the calculation, as well as the fact that $T = 2.5 \neq T_c$ was used as an approximate value. In obtaining τ_e for the Wolff algorithm, the usual time step n could not be used to compare to the Metropolis algorithm since it is temperature dependent and the number of spins flipped in the cluster varies. For this reason, the following conversion was used for the Wolff time step n_w

$$n = n_w \frac{\langle \text{cluster size} \rangle}{N^2} \tag{30}$$

5 Conclusions

It is clear that Markov Chain Monte Carlo (MCMC) methods lend themselves to simulating stochastic trajectories such as that of the two-dimensional Ising model. Using a straightforward Python implementation of the Metropolis (and later Wolff) algorithm, an estimate for the critical temperature of an infinite lattice $T_c(\infty) = 2.27 \pm 0.04$ was found to agree favourably with Onsager's analytical value. Similarly, estimates of various universal critical exponents (and ratios thereof) were found to lie within one standard error of the theoretical values. The origin of this error – along with the statistical basis of MCMC methods such as the two algorithms investigated – were also discussed, along with other aspects such as domain size and hysteresis. It is worth noting that there are alternative algorithmic approaches which were disregarded in the interest of brevity, such as Glauber dynamics, Heat Bath algorithm or even generalising the program to investigate the Potts model. These could prove fruitful to explore in future investigations.

References

[1] Ernst Ising. Beitrag zur Theorie des Ferromagnetismus. *Zeitschrift für Physik*, 31(1):253–258, February 1925.

[2] Lars Onsager. Crystal Statistics. I. a two-dimensional model with an order-disorder transition. *Phys. Rev.*, 65:117–149, Feb 1944.

[3] H. A. Kramers and G. H. Wannier. Statistics of the two-dimensional ferromagnet. part i. *Phys. Rev.*, 60:252–262, Aug 1941.

[4] Jun S. Liu. *Monte Carlo Strategies in Scientific Computing.* Springer Series in Statistics. Springer New York, New York, 2004.

[5] Wolfhard Janke. *Monte Carlo Simulations in Statistical Physics - From Basic Principles to Advanced Applications*, pages 93–166. World Scientific, 2012.

[6] Kim Christensen and Nicholas R Moloney. *Complexity and Criticality.* Imperial College Press, London, 2005.

[7] Herbert B. Callen and Theodore A. Welton. Irreversibility and Generalized Noise. *Phys. Rev.*, 83:34–40, Jul 1951.

[8] Ulli Wolff. Collective monte carlo updating for spin systems. *Phys. Rev. Lett.*, 62:361–364, Jan 1989.

A Main Code

```
import numpy as np
import matplotlib.pyplot as plt
import time
import math
from scipy import ndimage


__all__ = ['energy_required_to_flip',
           'total_magnetisation',
           'total_energy',
           'getNeighbours',
           'CClabel',
           'domain_size',
           'burn',
           'main']


def energy_required_to_flip(lattice, N, i, j):
    """
    Returns energy required to flip the spin of an
    individual spin site (i, j). Toroidal periodic
    boundary conditions are imposed.
    """
    dE = 2. * lattice[i][j] * (lattice[((i - 1) % N)][j]
                             + lattice[((i + 1) % N)][j]
                             + lattice[i][((j - 1) % N)]
                             + lattice[i][((j + 1) % N)])
    return dE


def total_magnetisation(lattice):
    """
    Returns total magnetisation of a given lattice
    """
    return np.sum(lattice)


def total_energy(lattice):
    """
    Returns total energy of a given lattice.
    Toroidal periodic boundary conditions are imposed.
    """
    energy_array = -1. * lattice * (np.roll(lattice, -1, 0)
                                  + np.roll(lattice, +1, 0)
                                  + np.roll(lattice, -1, 1)
                                  + np.roll(lattice, +1, 1))
    return np.sum(energy_array)


def getNeighbours(N, i, j):
    """
    Returns a list of neighbour coordinates tuples
    for the site (i, j). Toroidal periodic boundary
    conditions are imposed.
    """
    neighbours = []
    neighbours.append(tuple([(i + 1) % N, j]))
```

```python
    neighbours.append(tuple([(i - 1) % N, j]))
    neighbours.append(tuple([i, (j - 1) % N]))
    neighbours.append(tuple([i, (j + 1) % N]))
    return neighbours


def CClabel(lattice):
    """
    Connected-component labeling function for a given lattice
    Toroidal periodic boundary conditions are imposed.
    """
    N = len(lattice)
    labelled = np.zeros_like(lattice)
    todolist = []
    label = 1
    for i in range(N):
        for j in range(N):

            if labelled[i][j] == 0:
                labelled[i][j] = label
                todolist.append((i, j))
                # Pop spin site coordinates out of the to-do list and add
                # them to the cluster, repeating til empty.
                while todolist:
                    site = todolist.pop(0)
                    neighbours = getNeighbours(N, *site)

                    for k in [0, 1, 2, 3]:
                        if  lattice[neighbours[k]] == lattice[i][j] and \
                            labelled[neighbours[k]] == 0:
                            labelled[neighbours[k]] = label
                            todolist.append(neighbours[k])
                label += 1

    return labelled


def domain_size(lattice, plot=False):
    """
    Function which returns the largest cluster size for a given
    lattice. The boolean parameter 'plot' also can be set to
    True to plot a colourmap grid of the lattice group by domain size.
    """
    N = len(lattice)
    labels = CClabel(lattice)
    area = ndimage.measurements.sum(lattice, labels,
                                    index=np.arange(labels.max() + 1))
    out = np.abs(area[labels])

    if plot:
        plt.figure()
        plt.imshow(out, cmap='gray')
        plt.colorbar()
        plt.axis('off')
        plt.title('Lattice Grouped By Domain Size (N=%i)' % N)

    largest_cluster = out.max()

    return largest_cluster
```

```
def burn(N, nsites, lattice, T, nburn):
    """
    Function which iterates the Metropolis algorithm
    to `burn-in' the lattice to reach equilibrium.
    -----------------------------
    Parameters
    -----------------------------
    N :         Lattice size (i.e. N x N lattice)
    nsites :    Number of sites on the lattice (i.e. N^2)
    lattice :   N x N numpy array of +1 or -1 values
    T :         Temperature
    nburn :     Number of iterations to reach equilibrium
    """
    # Generate random coordinates/test values
    random_site = np.random.randint(N, size=(nburn*nsites, 2))
    boltzmann_picker = np.random.rand(nburn*nsites)

    for step in range(nburn*nsites):
        i, j = random_site[step][0], random_site[step][1]
        dE = energy_required_to_flip(lattice, N, i, j)
        if dE < 0 or math.exp(-dE / T) >= boltzmann_picker[step]:
            lattice[i][j] = -lattice[i][j]


def main(N=8, ntimesteps=10**4, Tmin=1, Tmax=5, ntemp=50):
    """
    Function which implements the Metropolis algorithm and saves
    the following observables in a .txt file under the name
    'N<lattice size>.txt' (e.g. N8.txt for an 8 x 8 lattice):
        - Temperature
        - Absolute Magnetisation
        - Energy
        - Susceptibility
        - Heat capcity
        - Largest Domain Size
    The above are all intensive quantites (i.e. per spin), and can
    be loaded using numpy.loadtxt into other scripts as a numpy array
    (and subsequently indexed for the relevant observables).
    -----------------------------
    Parameters
    -----------------------------
    N :             Lattice size (i.e. N x N lattice)
    ntimesteps :    Number of time steps (i.e. sweeps of the lattice)
    Tmin :          Minimum temperature bound
    Tmax :          Maximum temperature bound
    ntemp :         Number of intervals in the temperature loop.
    """
    nsites = N * N
    total_steps = ntimesteps * nsites

    # Temperature array to be used in loop
    T_arr = np.linspace(Tmin, Tmax, num=ntemp)

    # Initialise arrays
    Mabs_arr = np.empty(ntemp)  # Absolute magnetisation per spin
    E_arr = np.empty(ntemp)     # Energy per spin
    X_arr = np.empty(ntemp)     # Susceptibility per spin
```

```
C_arr = np.empty(ntemp)     # Heat capcity per spin
D_arr = np.empty(ntemp)    # Largest domain size per spin

# Initialisation of lattice
lattice = np.random.choice([+1, -1], size=(N, N))

temp_ind = 0        # temperature loop array indexing integer

# Temperature loop
for T in T_arr:

    # Burn in to reach equilibrium
    burn(N, nsites, lattice, T, nburn=1000)

    # Define observables
    Mabs = 0
    Msq = 0
    E = 0
    Esq = 0
    D = 0

    # Update random coordinates/test values for main loop
    random_site = np.random.randint(N, size=(total_steps, 2))
    boltzmann_picker = np.random.rand(total_steps)

    # Initialise counters
    m = total_magnetisation(lattice)
    e = total_energy(lattice)

    tstep_ind = 0  # timestep loop array indexing integer

    # Main timestep loop
    for timestep in range(ntimesteps):

        # Do a lattice sweep
        for site in range(nsites):
            i, j = random_site[tstep_ind][0], random_site[tstep_ind][1]
            dE = energy_required_to_flip(lattice, N, i, j)
            if dE < 0 or math.exp(-dE / T) >= boltzmann_picker[tstep_ind]:
                lattice[i][j] = -lattice[i][j]
                m += 2 * lattice[i][j]
                e += 2 * dE
            tstep_ind += 1

        # Update thermodynamic observables
        Mabs += abs(m)
        Msq += m * m
        E += e / 2
        Esq += (e / 2) * (e / 2)
        D += domain_size(lattice)

    # Update averages
    Mabs_av = Mabs / total_steps
    Msq_av = Msq / total_steps
    E_av = E / total_steps
    Esq_av = Esq / total_steps
    D_av = D / total_steps

    # Place averages into arrays
```

```
        Mabs_arr[temp_ind] = Mabs_av
        E_arr[temp_ind] = E_av
        X_arr[temp_ind] = (Msq_av - ((Mabs_av**2) * nsites)) / (T)
        C_arr[temp_ind] = (Esq_av - ((E_av**2) * nsites)) / (T**2)
        D_arr[temp_ind] = D_av

        temp_ind += 1
        # end of temperature loop

    # Create master array and save to file
    out = np.vstack((T_arr, Mabs_arr, E_arr, X_arr, C_arr, D_arr)).T
    np.savetxt("N%i" % N, out, header="T, |M|, E, X, C, D")


start = time.time()

if __name__ == "__main__":
    main()

print(time.time() - start)
```

B Line-by-line Profiling for a N = 16 Time Step

```
Total time: 2.24926 s
File: ising.py
Function: main at line 123

Line #      Hits         Time  Per Hit   \% Time  Line Contents
==============================================================
   123                                           @profile
   124                                           def main(N=16, ntimesteps=1, Tmin=1, Tmax=5, ntemp=1):
   125                                               """
   126                                               Implements the metropolis algorithm and saves data to file
   127                                               """
   128         1        185.0    185.0      0.0      nsites = N * N
   129         1          2.0      2.0      0.0      total_steps = ntimesteps * nsites
   130
   131                                               # Temperature array to be used in loop
   132         1        123.0    123.0      0.0      T_arr = np.linspace(Tmin, Tmax, num=ntemp)
   133
   134                                               # Initialise arrays
   135         1          5.0      5.0      0.0      Mabs_arr = np.empty(ntemp)  # Absolute magnetisation per spin
   136         1          3.0      3.0      0.0      E_arr = np.empty(ntemp)     # Energy per spin
   137         1          3.0      3.0      0.0      X_arr = np.empty(ntemp)     # Susceptibility per spin
   138         1          3.0      3.0      0.0      C_arr = np.empty(ntemp)     # Heat capcity per spin
   139         1          3.0      3.0      0.0      D_arr = np.empty(ntemp)    # Largest domain size per spin
   140
   141                                               # Initialisation of lattice
   142         1        156.0    156.0      0.0      lattice = np.random.choice([+1, -1], size=(N, N))
   143
   144         1          1.0      1.0      0.0      temp_ind = 0        # temperature loop array indexing integer
   145
   146                                               # Temperature loop
   147         2          7.0      3.5      0.0      for T in T_arr:
   148
   149                                                   # Burn in to reach equilibrium
   150         1    2239254.0 2239254.0     99.6          burn(N, nsites, lattice, T, nburn=1000)
   151
   152                                                   # Define observables
   153         1          2.0      2.0      0.0          Mabs = 0
   154         1          1.0      1.0      0.0          Msq = 0
   155         1          0.0      0.0      0.0          E = 0
   156         1          1.0      1.0      0.0          Esq = 0
   157         1          0.0      0.0      0.0          D = 0
   158
   159                                                   # Update random coordinates/test values for main loop
   160         1         31.0     31.0      0.0          random_site = np.random.randint(N, size=(total_steps, 2))
   161         1         12.0     12.0      0.0          boltzmann_picker = np.random.rand(total_steps)
   162
   163                                                   # Initialise counters
   164         1         48.0     48.0      0.0          m = total_magnetisation(lattice)
   165         1        241.0    241.0      0.0          e = total_energy(lattice)
   166
   167         1          1.0      1.0      0.0          tstep_ind = 0  # timestep loop array indexing integer
   168
   169                                                   # Main timestep loop
   170         2          4.0      2.0      0.0          for timestep in range(ntimesteps):
   171
   172                                                       # Do a lattice sweep
   173       257        174.0      0.7      0.0              for site in range(nsites):
   174       256        349.0      1.4      0.0                  i, j = random_site[tstep_ind][0], random_site[tstep_ind][1]
   175       256       1940.0      7.6      0.1                  dE = energy_required_to_flip(lattice, N, i, j)
   176       256        409.0      1.6      0.0                  if dE < 0 or math.exp(-dE / T) >= boltzmann_picker[tstep_ind]:
   177        18         24.0      1.3      0.0                      lattice[i][j] = -lattice[i][j]
   178        18         21.0      1.2      0.0                      m += 2 * lattice[i][j]
   179        18         19.0      1.1      0.0                      e += 2 * dE
   180       256        194.0      0.8      0.0                  tstep_ind += 1
   181
   182                                                       # Update thermodynamic observables
   183         1          1.0      1.0      0.0              Mabs += abs(m)
   184         1          1.0      1.0      0.0              Msq += m * m
   185         1          2.0      2.0      0.0              E += e / 2
   186         1          1.0      1.0      0.0              Esq += (e / 2) * (e / 2)
   187         1       2673.0   2673.0      0.1              D += domain_size(lattice)
   188
   189                                                   # Update averages
   190         1          1.0      1.0      0.0          Mabs_av = Mabs / total_steps
   191         1          1.0      1.0      0.0          Msq_av = Msq / total_steps
   192         1          1.0      1.0      0.0          E_av = E / total_steps
   193         1          1.0      1.0      0.0          Esq_av = Esq / total_steps
   194         1          1.0      1.0      0.0          D_av = D / total_steps
   195
   196                                                     # Place averages into arrays
   197         1          1.0      1.0      0.0            Mabs_arr[temp_ind] = Mabs_av
   198         1          1.0      1.0      0.0            E_arr[temp_ind] = E_av
   199         1          3.0      3.0      0.0            X_arr[temp_ind] = (Msq_av - ((Mabs_av**2) * nsites)) / (T)
   200         1          3.0      3.0      0.0            C_arr[temp_ind] = (Esq_av - ((E_av**2) * nsites)) / (T**2)
   201         1          1.0      1.0      0.0            D_arr[temp_ind] = D_av
   202
   203         1          1.0      1.0      0.0            temp_ind += 1
   204                                                     # end of temperature loop
   205
   206                                                 # Create master array and save to file
   207         1         51.0     51.0      0.0        out = np.vstack((T_arr, Mabs_arr, E_arr, X_arr, C_arr, D_arr)).T
   208         1       3303.0   3303.0      0.1        np.savetxt("N\%i" \% N, out, header="T, |M|, E, X, C, D")
```