Features of a 2D Ising Model simulated with Markov chain Monte Carlo methods

Jake Skelton

10th September 2021

Abstract

In this paper, we present observations of a two-dimensional Ising model, simulated with the Metropolis algorithm, a Markov chain Monte Carlo method. Square lattices are used, with periodic boundary conditions and nearest-neighbour spin-spin interactions. Considerations of Markov chain convergence and time correlations are taken into account, and the variation of magnetisation and heat capacity with temperature is observed, for a range of lattice sizes and applied magnetic fields. Hysteretic phenomena in the Ising model under an applied field are also investigated. In addition, we make an estimate of the critical temperature of an infinite 2D lattice, $2.263 \pm 0.013\ J/k_B$. Onsager's analytical result of $2/\ln(1+\sqrt{2})\ J/k_B$ differs from our estimate by only 0.2%, and lies within error.

1 Introduction

Ferromagnetism is vital to much of our modern technology, but a full description of the phenomenon entails all the usual problems with treating quantum many-body interactions, and so there is great value in simple models which retain some of the characteristic features of ferromagnetism. One of the simplest such models that still offers useful insights is the *Ising model*.

The Ising model is a scheme for modelling ferromagnetic materials that consists of a lattice of spin sites which can point either up or down relative to some global axis. These spins interact with one another, pairwise, and the effects of an applied field can also be included. In this work, we consider only nearest-neighbour spin interactions, although next-nearest-neighbour and global spin interaction incarnations do exist.

Ising models can be considered in 1, 2, 3, or n dimensions, and with any lattice tiling. However, analytical solutions only exist for the 1D and 2D cases, and the latter is notoriously complicated [**Landau1980**]; higher dimensions require the employment of mean-field theory or other approximations. A key element of real ferromagnets that a good model ought to capture is the *order-disorder phase transition*. Below its Curie temperature T_c, a ferromagnet exhibits spontaneous magnetisation; above T_c it is unmagnetised. In fact, such a transition is absent from the 1D Ising model [**Onsager1944**, **Blundell2009**]. For this reason, and in order to have analytic results with which to compare, this work focuses on a two-dimensional Ising model, on a square lattice for simplicity.

Following this brief introduction, the ensuing sections of this paper describe an Ising model lattice whose time evolution is simulated according to the Metropolis algorithm, an example of a Markov chain Monte Carlo (MCMC) method. Section 2 outlines why an MCMC approach is appropriate, as well as other requisite computational physics. Section 3 presents the key elements of our implementation of the Metropolis algorithm and some data on the code's performance. Section 4 describes the qualitative features of the model: its time evolution, equilibrium magnetisation and heat capacity, and behaviour under an applied field. We also present a quantitative prediction of the critical temperature of an infinite lattice, and discuss the implications and explanations of the features observed. Finally, section 5 offers concluding remarks.

2 Background and approach

2.1 Ising canonical ensemble

Following the description of the general Ising model given in the introduction, the Hamiltonian for such a system is written [**TSPhandout**]

$$E = -J \sum_{\langle i,j \rangle} s_i s_j - H \sum_i s_i \tag{1}$$

where $s_i \in \{\pm 1\}$ is the spin at the ith lattice site, J is the spin-spin interaction energy, and H is the applied magnetic field strength. J is taken to be positive - spin alignment lowers the system energy. The first sum is over all pairs of nearest neighbours, and the second is over all lattice sites. In this expression we are using an idealised system of units in which Planck's constant, the magnetic permeability, etc. do not feature. In the more explicit case of a square lattice of size $N \times N$, where sites may be assigned coordinates (x, y), $x \in [0, N)$, $y \in [0, N)$, the Hamiltonian takes the form

$$E = -\frac{J}{2} \sum_{x,y} \sum_{\substack{x',y' \in \\ \mathrm{nn}(x,y)}} s_{xy} s_{x'y'} - H \sum_{x,y} s_{xy}. \tag{2}$$

where the set $\mathrm{nn}(x, y) = \{(x, y \pm 1), (x \pm 1, y)\}$ includes all the nearest neighbours of (x, y). The factor of $1/2$ in front of the first sum ensures nearest-neighbour 'bonds' are not double counted. In the remainder of this paper, we shall take $J = 1/2$ in order to have H and T as the only free parameters. In these units, the maximum energy of a lattice with $H = 0$ is, conveniently, equal to N^2.

As mentioned, there exists an analytic theory for the 2D Ising model of infinite extent due to Onsager [**Onsager1944**], and many sources present important results of this theory [**Landau1980**] and the mean-field approximation [**chaikin_lubensky_1995**, **Plischke2006**]. We do not pursue either here, except to mention Onsager's prediction for the temperature at which the order-disorder phase transition occurs; in our system of units, this is $T_c = 1/\ln(1 + \sqrt{2})$. This prediction will be important in section 4.

Like Onsager, we consider a *canonical ensemble* of square lattices each connected to a reservoir at temperature T, in which the probability of a given spin configuration $\{s_i\}$ being realised is

$$P(\{s_i\}) = \frac{1}{Z} e^{-E(\{s_i\})/T}. \tag{3}$$

Z is the partition function

$$Z(T, J, H) = \sum_{\{s_i\}} e^{-E(\{s_i\})/T} \tag{4}$$

which depends only upon macroscopic parameters (note that we have taken $k_B = 1$).

2.2 Markov Chain Monte Carlo

In assuming a canonical ensemble we have created an *ergodic* system, which is ideally suited to treatment as a Monte Carlo Markov chain [**Press2007**]. This procedure involves letting the Markov chain (the spin lattice) explore state space (the space of all possible spin configurations) by means of taking discrete steps (altering the spin lattice), or not, according to a probability distribution. In our case, the choice of probability distribution is obvious: a Boltzmann factor encasing the energy of a spin configuration change, $\exp(-E(\{s_i\} \to \{s_i\}')/T)$.

We do not prove that such a scheme creates an ergodic Markov chain, but it does, ultimately because a Boltzmann probability satisfies *detailed balance* [**Press2007**]. Essentially, what we are doing is using MCMC to bypass the need to evaluate the partition function (4).

Once the Markov chain reaches equilibrium, its ergodic nature allows us gain information about the *target distribution*, (3) by averaging outputs over time. Determining when equilibrium is reached is an issue of MCMC convergence, or *burn-in* [**Gilks1995**, **Press2007**]. A few techniques and metrics have been established for diagnosing convergence [**Cowles1996**, **Gilks1995**], but many authors recommend simply *looking* at the output of an MCMC simulation as the most infallible approach. We describe our methods in section 4.1, after an outline of of some of our programming decisions in the following section.

3 Implementation

3.1 Preliminaries

The code for this work was written in Python, chosen for its flexibility and excellent graphical packages. Some light object-oriented programming was employed in order to keep the code tidy. A `class` was defined to hold characteristic information about a single lattice - its temperature, applied field, etc., and to wrap routines that were used very often throughout the work and across multiple files. The header for the class instantiation method is shown in listing 1, see appendix A.2 for more details.

```
class lattice:
    def __init__(self, sidelength, magfield,
                 temp, uniform=False,
                 seed=None):
```

Listing 1: Lattice class instantiation. `sidelength`, `magfield`, and `temp` correspond to the parameters N, H, and T respectively. `uniform` controls whether the lattice starts in a random configuration, or with all spins pointing in the same direction.

The general workflow was to create one or more `lattice` objects, then use class methods to evolve them in time and collect statistics in the process - in particular, the mean spin of the lattice, and the lattice energy, for each time step. The statistics were pro-

cessed into secondary results (after discarding data from unconverged Markov steps) and all relevant information from one such simulation run was stored in a `numpy` '.npz' file - a collection of named arrays stored as binary. The plotting and data analysis routines were kept separate from data generation - data to plot was assembled from the .npz files when needed. All the plots in this paper were made using `matplotlib`.

Unfortunately, the fast vectorised routines of Python's `numpy` package are unsuitable for Markov chain calculations, owing to the conditional statement at each step, whose outcome cannot be found ahead of time. An individual Markov chain cannot benefit from parallel computing either, its evolution is a fundamentally serial process (though several independent Markov chains are a different story). This creates a challenge because, like most interpreted languages, Python's `for` loops are incredibly slow. The solution adopted here was to use Cython, a language extension to Python [**cython**], which allows the creation of compiled modules that make use of static typing.

3.2 Implementing the Metropolis algorithm

In the section 2.2, we were rather vague about what alterations to the spin lattice are permitted in each Markov step, but the code must be explicit. At each time step, we loop over all sites in the $N \times N$ spin lattice and, at each, find the energy required to flip the spin, $E(s_{xy} \to -s_{xy})$ (which may be negative). If this energy, in a Boltzmann factor, exceeds a random number in $[0, 1]$ then the spin is flipped, otherwise, it is left alone. The explicit form of the energy is easy to calculate from (2),

$$E(s_{xy} \to -s_{xy}) = s_{xy} \left(\sum_{\text{nn}} s_{x'y'} + 2H \right). \qquad (5)$$

Note that this means negative-energy flips always occur. It should be clear that adding expressions like (5) for every site in the lattice gives the energy of changing the spin configuration, wholesale. Hence, this prescription accomplishes the algorithm in section 2.2.

There is a significant omission in the above scheme: what happens to spins at the edges of the square? To avoid complicated edge effects, and to better compare with the theoretical results using infinite lattices, in this work we use *periodic boundary conditions*. In doing so, each edge of the square is identified with its opposite, and every site then has four nearest neighbours*.

Putting this together, below is a condensed version of the main loop used in our program. This loop is contained in the Cython code in appendix A.1.

```
def metrohaste(int numsteps, int[:,:] s,
                double H, double T, RNG):

cdef int N = s.shape[0]
cdef double[:,:,:] p = RNG.uniform(0, 1,
                        size=(numsteps,N,N))
for t in range(numsteps):
    for i in range(N):
        for j in range(N):
            deltaE = (s[(i-1)%N, j] +
                      s[(i+1)%N, j] +
                      s[i, (j-1)%N] +
                      s[i, (j+1)%N] +
                      2*H)*s[i,j]
            # Multiply s by -1 if
            # inequality true, else by 1
            s[i,j] *= (-1)**(exp(-deltaE/T)
                            > p[t,i,j])

return np.asarray(s)
```

Listing 2: Metropolis algorithm loop. `p` is an array of random numbers against which to compare the Boltzmann factor, `s` is the spin lattice. Note the use of modular arithmetic for periodic boundary conditions.

There are a few things to explain in listing 2. Firstly, the use of modular arithmetic when referencing elements of the spin array: `s[(i-1)%N, j]`. This is to achieve periodic boundary conditions. Secondly, `p` is a statically-typed Cython *memoryview*. Memoryviews are just a form of array that is slightly faster to operate on than a `numpy` array [**cython**]. Into the array are put random numbers using a `numpy.random.Generator` object `RNG`.

3.3 Random numbers

In this work we use the pseudo-random PCG64 generator [**oneill:pcg2014**], which is the default for `numpy`. For the advantages of PCG over the classic Mersenne twister of C `stdlib.h` fame, we refer the reader to [**oneill:pcg2014**]. Although `numpy.random` does not require it, our code creates explicit random number generator (RNG) objects. The reason is twofold. Firstly, it allows the use of one global seed for a whole batch of independent Markov chains, which makes results replicable (for plotting, for example). Secondly, creating daughter RNGs from one seed is a *thread-safe* method for obtaining random numbers, that does not clash with parallel processing.

3.4 Parallel processing

A set of independent Markov chains, unlike a single one, are ideal candidates for parallelising, and their output data can be collated and averaged. We implemented parallel processing in the Python standard

*The author finds the analogy with *Pac-Man* to be helpful.

library module `multiprocessing`, paying heed to the fact that parallel computing is only worthwhile for long calculations; otherwise, the time to spawn new threads outweighs the performance gain of multiple processors.

3.5 Performance

The two primary measures taken to improve performance, use of Cython and `multiprocessing`, have already been mentioned. Aside from this, we used obvious tricks such as using `numpy` over native Python wherever possible. In listings 3 and 4 we reproduce some tests of the impact of the steps taken to improve performance. Cython is over 100 times faster than native Python, but the performance improvement using `multiprocessing` is more modest - it halved the time for a calculation with 100 Markov chains. As mentioned, this benefit dissolves for shorter routines so parallelism was only employed for the longer data collection runs in this work.

```
In [1]: N = 32
In [2]: numsteps = 1000
In [3]: RNG = np.random.default_rng(100)
In [4]: l = lattice(N, 0.0, 1.0)
In [5]: %timeit metrohaste_slow(numsteps,
                    l.spins, l.H, l.T, RNG)
8.07 s ± 129 ms per loop
In [6]: %timeit metrohaste(numsteps, l.spins,
                    l.H, l.T, RNG)
51.2 ms ± 488 µs per loop
```

Listing 3: Cython performance test. `metrohaste_slow` is implemented in standard Python, whereas `metrohaste` is implemented in Cython. The performance difference is dramatic: the same operation is performed 160 times faster in the Cython code.

```
In [1]: N = 100
In [2]: numchains = 4
In [3]: numsteps = 10000
In [4]: %timeit lattice.stepforward_l(latlist,
                    numsteps, parallel=False)
20.4 s ± 270 ms per loop
In [5]: %timeit lattice.stepforward_l(latlist,
                    numsteps, parallel=True)
9.34 s ± 706 ms per loop
```

Listing 4: Parallel processing test. The same procedure is executed twice as fast when Markov chains can be forked into different threads.

4 Observations and results

This section gives a detailed summary of the observations and quantitative investigations of the Ising model Markov chains. We begin with the first impressions, testing if the code behaves as it should, and a consideration of burn-in (that is, the convergence of the Markov chain). Next, the behaviour of lattices at various temperatures and of various sizes, but with no applied field is investigated. In particular, we look for the location of a ferromagnetic phase transition. Finally, we make a brief enquiry into the behaviour of lattices under the effects of a magnetic field. Before going any further, below is a brief glossary of symbols used throughout the paper, for reference.

N^2	Number of sites in a lattice
H	Applied magnetic field
T	Reservoir temperature
T_c	Critical (Curie) temperature
s	Spin
M	Lattice magnetisation
E	Lattice energy
C	Heat capacity
t_b	Burn-in time
Overbar, $\bar{}$	Quantities averaged over space
Angle brackets, $\langle\rangle$	Quantities averaged over time

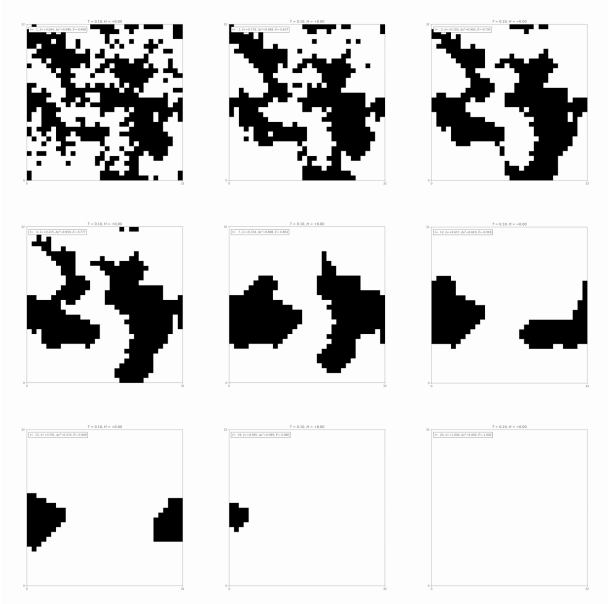

Fig. 1: Approach to equilibrium of a 32x32 lattice, starting from a random distribution, at $T = 0.1$. Black pixels represent spin down lattice sites, and white pixels up. The time gap between these images is not uniform: reaching a structure of 'islands' from a random distribution happens quickly, then the process slows as domains' perimeter/area ratio becomes smaller. This burn-in happened over the course of only 20 time steps. An animation of this process is included with this report, as 'anim1.mp4'

4.1 Initial investigations

Animations of spin lattices as they evolve in time were examined first. Representative samples are shown in figures 1 and 2; black squares correspond to spins pointing down, and white squares to those pointing up. It became quite obvious that behaviour can be separated into two regimes: at low temperatures ($T \sim 0.5$), lattices settle to a uniform spin

distribution (with some thermal noise) - either all up or all down. At high temperatures ($T >\sim 1.5$), the spin configurations look convincingly random, as white noise[†]. The temperatures involved do not change substantially with lattice size. Recall that in the units used here, Onsager's critical temperature is $T_c = 1/\ln(1+\sqrt{2}) \approx 1.1346$. These observations are unsurprising enough, from a physical point of view, to constitute a check that there were no glaring bugs in the code. A further physical check that the code satisfied is that, at low temperatures, applying some $H > 0$ causes all the spins to settle pointing up, and vice versa for $H < 0$.

At temperatures nearer 1.1, as depicted in figure 2, the configurations seen are much more complex. Large patches of uniform spin coexist with more heterogeneous areas. The lattices here embody two hallmarks of a second-order phase transition, namely *structure on all length scales* and dynamics dominated by *fluctuations* [**TSPhandout**, **Blundell2009**].

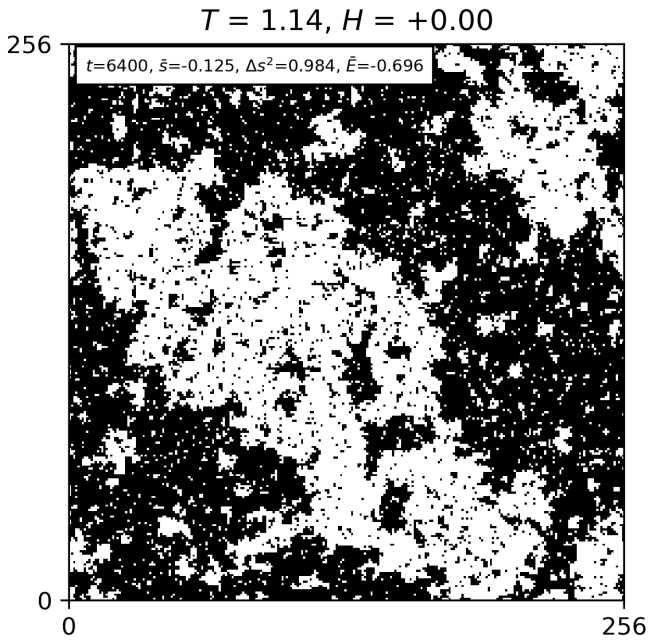

Fig. 2: View of a 256x256 lattice at equilibrium near the critical temperature. The classic characteristic of critical phenomena, *structure at all length scales* can be made out quite well (although this simulation does of course have hard limits on such scales). An animation of this process is included with this report, as 'anim2.mp4'

Not all low-temperature lattices relax to a uniform distribution quickly. Those started from a random configuration can form very interesting 'band' structures. Keeping the periodic boundary conditions in mind, they are best thought of as a neck of spin up (say) on a torus of spin down, forming two distinct regions. These bands, which are not straight, are observed to travel parallel to their length. A static image of these structures would not be very illuminating, but an animation is included with this report as 'anim3.mp4'. The author tentatively proposes these structures as an example of *solitons* - they meet the definition in [**chaikin_lubensky_1995**].

In any case, these structures are *metastable* - they can persist for a long time (thousands of time steps in some cases) but then relax very quickly to a uniform configuration; for an illustration see figure 3. Physically, this is easily explained by the lattice not having enough thermal energy to anneal, but it presents a barrier to acquiring information from equilibrium spin configurations. These metastable states exemplify the issue of MCMC burn-in mentioned in section 2.2.

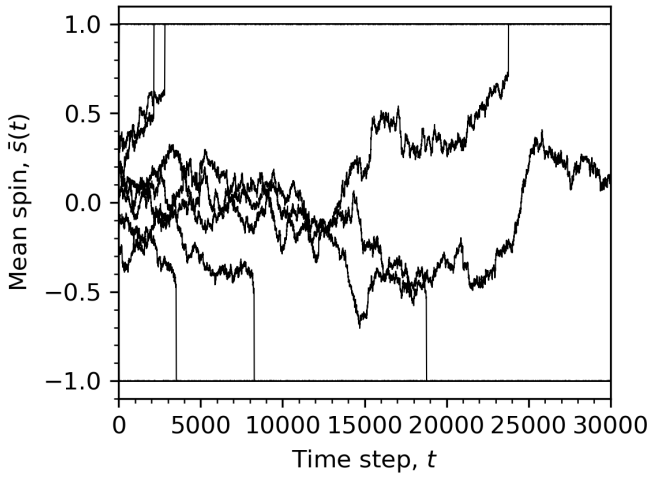

Fig. 3: Mean spin (magnetisation per atom) of 30 lattice Markov chains over time, at $T = 0.3$, starting from a random spin configuration. Most of the chains converge relatively quickly (within 1000 steps), but a minority encounter a metastable state. This plot shows six such metastable chains, one of which fails to converge in 30,000 steps. Notice that the transition from metastability to equilibrium can happen very quickly.

4.2 Determining burn-in

As mentioned previously, there are a few popular diagnostics for estimating whether a Monte Carlo Markov chain has converged, but [**Cowles1996**], a review of such methods, is somewhat pessimistic:

> [Many theoretical statisticians] conclude that all such diagnostics are fundamentally unsound. Many researchers in other areas where MCMC methods have been used for many years (e.g., physics and operations research) have also reached this conclusion.

In this work, we used a two-pronged approach; for temperatures below some (arbitrary) cutoff, the lattice was assumed to have converged when the mean

[†]With the plot format chosen, the comparison with terrestrial television 'static' is rather apt.

spin, $\bar{s}(t)$, was equal to ±1 to within some bound. For temperatures above the cutoff, convergence was taken as the period after the cumulative time-average of $\bar{s}(t)$ had fallen below some bound. In the end, the most important arbiter of the quality of these diagnostics is whether they agree with a picture of the Markov chains' output over time. Such a picture is shown in figure 5.

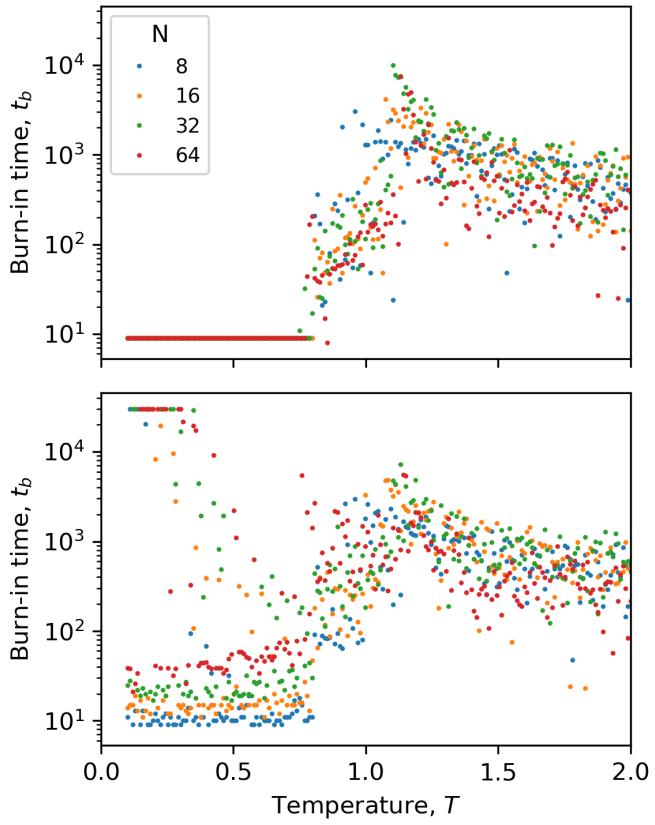

Fig. 4: Burn-in times for lattice Markov chains of various sizes as a function of temperature, starting from top) a uniform spin lattice, and bottom) a random starting configuration. Points to note are: start state makes little difference to lattices above T_c; burn-in can take as several thousand steps near T_c; and metastability has a dramatic effect on burn-in at low temperatures - many chains did not converge in the 30,000 steps over which the simulation ran. It must be pointed out that the burn-in determination algorithm used was different (as explained in section 4.2) either side of $T = 0.8$ (chosen arbitrarily), and this explains the discontinuity here, but not the overall trend.

For the purpose of choosing a simulation run time that includes sufficient post-convergence data, it is worth mapping the variation of burn-in time, t_b with temperature T and lattice size N. Such data is shown in figure 4, with both uniform and random initial configurations. The takeaways are that t_b is unaffected by start state for $T > T_c$; a uniform start state eliminates metastability and shortens burn-in below T_c; besides metastable states, the longest burn-in occurs near T_c; and finally the variation with N is marginal. These investigations informed data collection for our subsequent work.

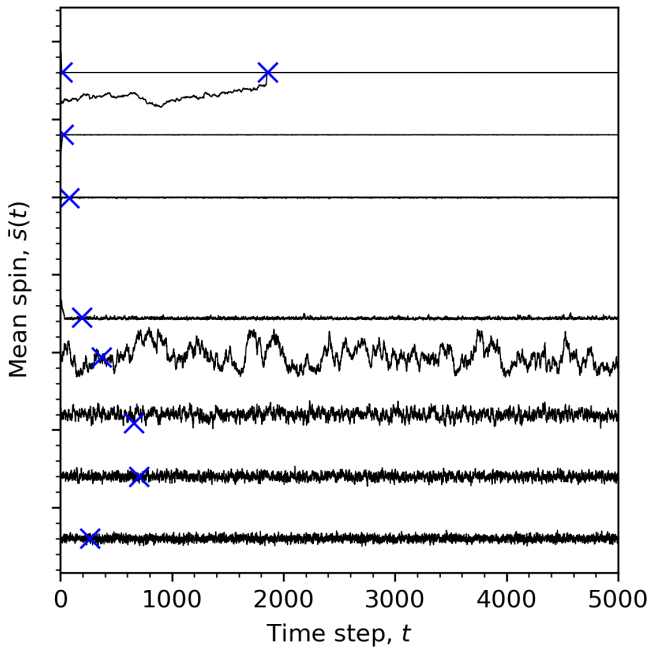

Fig. 5: Mean spin of eight lattices over time, and their algorithmically-determined burn-in times. The first lattice is at $T = 0.2$, and the temperature is incremented by 0.2 thereafter. The zero-spin position of each lattice is offset from the previous by -2. The point after which the lattices are taken to have burnt in are marked with blue crosses. Note that for useful data it is better to exclude some early burnt-in-points than to erroneously include pre-converged data.

4.3 Temperature variation

Calculating useful time averages of Markov chain outputs requires a consideration of not just convergence, but also time *correlation*. If a Markov chain has not 'forgotten' its history over an averaging interval, the average will contain little information about the target distribution - (3) in our case. The correlations can be quantified by computing the autocovariance of the magnetisation, $\kappa_M(\tau)$.

$$\kappa_M(\tau) = \langle M(t+\tau) - \langle M \rangle \rangle \langle M(t) - \langle M \rangle \rangle \qquad (6)$$

where $M = N^2\bar{s}$ is the magnetisation and angle brackets denote a time average. The decorrelation time, τ_e, is the interval over which the autocorrelation $\kappa_M(\tau)/\kappa_M(0)$ drops to $1/e$, and is a measure of the 'memory' of the Markov chain. The variation of decorrelation time with temperature and N is shown in 6; the divergence near T_c is striking, even on a logarithmic scale. This divergence is an example of *critical slowing down* **[Wolff1989]**.

The autocovariance of the magnetisation is equivalent to the autocorrelation of the *magnetisation fluctuations*. This is significant because, under the Wiener-Khinchin theorem, the power spectral density of fluctuations (in any quantity) is given by the Fourier transform of their autocorrelation. Thus, computing κ_M gives one a route to the noise spec-

trum of the model. We do not pursue this any further here, but more details are given in [**Landau1980**, **Blundell2009**, **chaikin_lubensky_1995**].

With these data informing appropriate lengths for averages, more details of the Ising model phase transition were established. Figure 7 shows the variation of time-averaged magnetisation, and heat capacity with temperature for a range of lattice sizes.

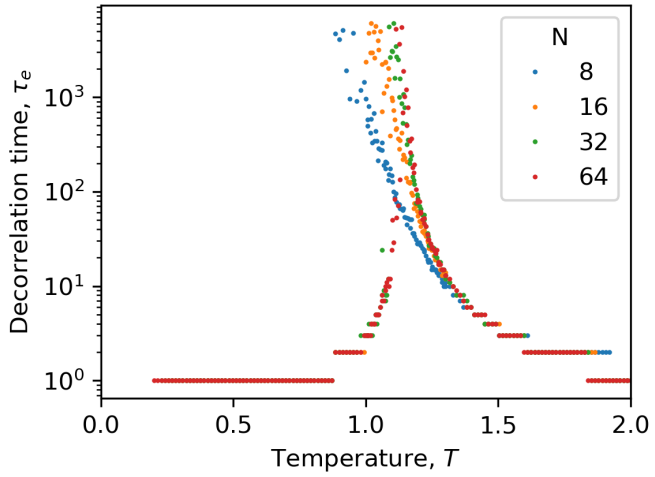

Fig. 6: Decorrelation time (the characteristic time over which magnetisation autocovariance decays) as a function of temperature for four lattices of varying size. The Markov chains that produced these data were all run for 30,000 time steps. It is obvious that there is significant time-correlated behaviour near the critical temperature for all lattice sizes, and this has implications for the size of a useful time-average at such temperatures. Either side of this critical temperature, there is remarkably little time correlation (not to be confused with low temperature spatial correlation).

The data for magnetisation reinforce the qualitative observations described in section 4.1: the equilibrium magnetisation is $\pm N^2$, corresponding to a uniform spin configuration, below the critical temperature. Above it, a disordered configuration is favoured, with $\langle M \rangle = 0$. The relatively smooth variation of $\langle M \rangle$ with T marks this out as a second-order phase transition.

The heat capacity C was obtained, for each lattice, using the fluctuation dissipation theorem [**Blundell2009**]; since C is the response function linking lattice energy with temperature, the relevant fluctuations are those of energy:

$$C(T) \equiv \frac{\partial E}{\partial T} = \frac{\langle \Delta E^2 \rangle}{T^2}; \quad \Delta E(t) = E(t) - \langle E \rangle. \qquad (7)$$

The results are shown in the lower panel figure 7. The general profile is in good agreement with the literature [**Plischke2006**] - C peaks at the critical temperature - but the singularity predicted by Onsager's results and mean-field theory is not evident. The peaks do become more severe with increasing N, however. This observation, and the variation of T_c with N - quite clear in the plot, are consequences of *finite-size scaling* [**Plischke2006**]. The canonical relationship is

$$T_c(N) = T_c(\infty) + aN^{-1/\nu} \qquad (8)$$

for some system-specific parameters a and ν.

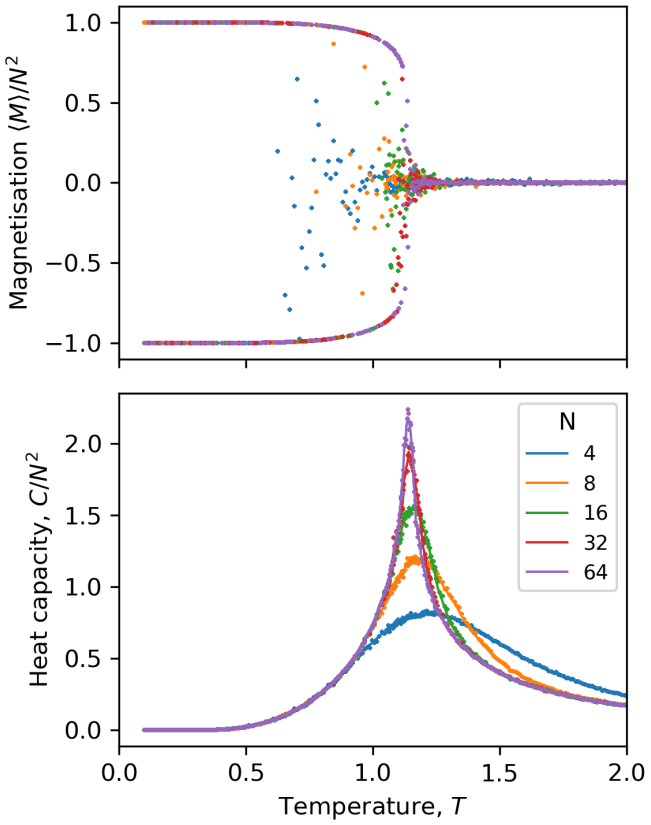

Fig. 7: The magnetisation and heat capacity of five lattices of different size as a function of reservoir temperature. Both quantities have been normalised by number of sites. A rolling average has been added to the lower plot to aid the eye. The symmetry-breaking transition to magnetic order is clear in the magnetisation plot. Notice that that the peaks in heat capacity happen at broadly the same temperature as the order/disorder transition, but that variation between lattice sizes is easier to distinguish in the lower plot.

Our data to examine this relationship are shown in figure 8. The critical temperatures, for each N, were taken as the locations of the largest heat capacity data point. This crude approach was moderated by taking the next highest points, to the left and right, as the error in the $T_c(N)$ estimate. Corresponding error bars are plotted in the figure. All these data were then fed into a non-linear least squares fitting algorithm (`optimize.curve_fit` of the `scipy` package). The estimate for the critical temperature of an infinite lattice thus produced, accounting for error in determining $T_c(N)$ and in the fitting, was 1.132 ± 0.006. The theoretical value (in the units used here) is 1.1346..., within error and representing a difference of 0.2% from our estimate.

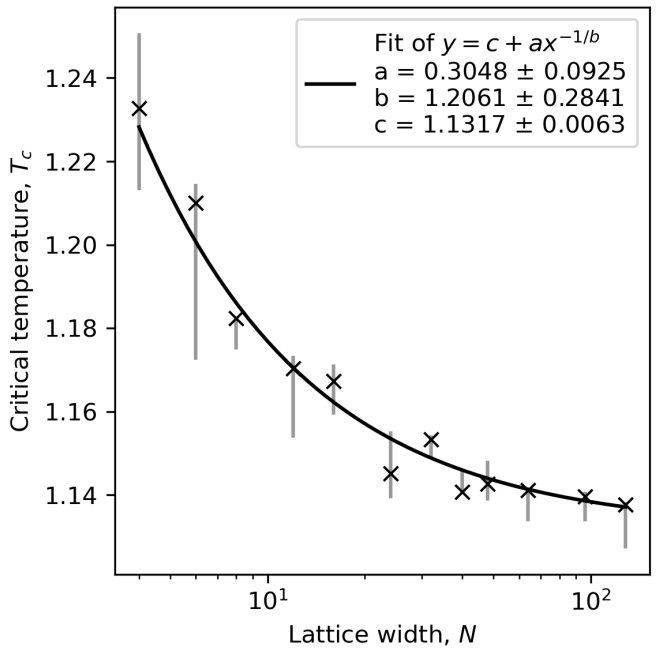

Fig. 8: Variation of the critical temperature with lattice size, plotted on a linear versus logarithmic scale. The theoretical relationship (8) was fitted to the numerical heat capacities, taking into account the error in their determination. The critical temperature for an infinite lattice corresponds to parameter *c*.

4.4 Non-zero magnetic field

Finally, the behaviour of the Ising model under an applied field was investigated. It was already noticed (section 4.1) that the equilibrium, low temperature magnetisation aligns with the field direction, as expected. However, the temperature above which the spin configuration appears random is altered by non-zero H; applying a stronger field causes the transition region to become larger, and shifts the inflection point of the magnetisation to higher temperatures. These effects are illustrated, for the magnetisation and heat capacity, in figure 9.

To probe the well-known ferromagnetic phenomenon of *hysteresis* - dependence of a system's dynamics on its past, code was written allowing the applied field to take a different value at each time step. Figure 10 shows the results of changing H sinusoidally over the interval $[0, 3\pi]$. Hysteresis is well evident at low temperatures - the MH curve has the classic form **[chikazumi2009]** and encloses a large area. The lower plots show the system energy, and the observation that this continually decreases for the coldest lattice illustrates that, in a hysteretic cycle, energy is transferred *to* the reservoir. This is because the driving field and the magnetisation are out of phase, and the area of the MH plot is another giveaway. This area is seen to shrink as temperature increases, until, at temperatures above $T_c(H)$, the MH relationship is approximately linear. Of course, at very high temperatures, $M = 0$.

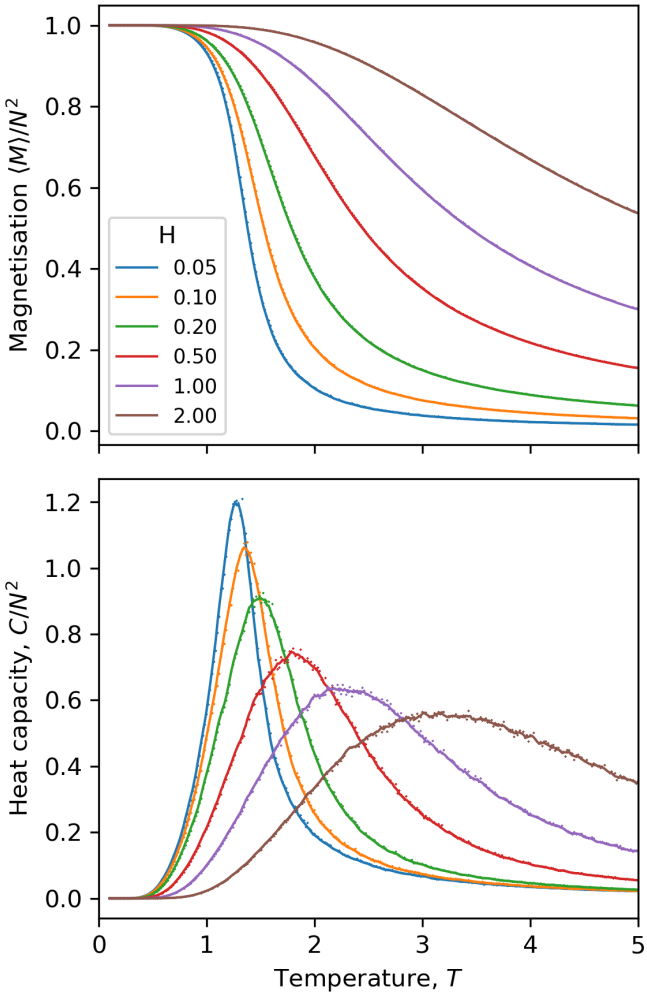

Fig. 9: The magnetisation and heat capacity of a 32×32 lattice, under five different strengths of applied field, as a function of reservoir temperature. Both quantities have been normalised by number of sites, ad a rolling average has been added to aid the eye. Note the smoothing of the transition region with increased field strength, and the increase in critical temperature.

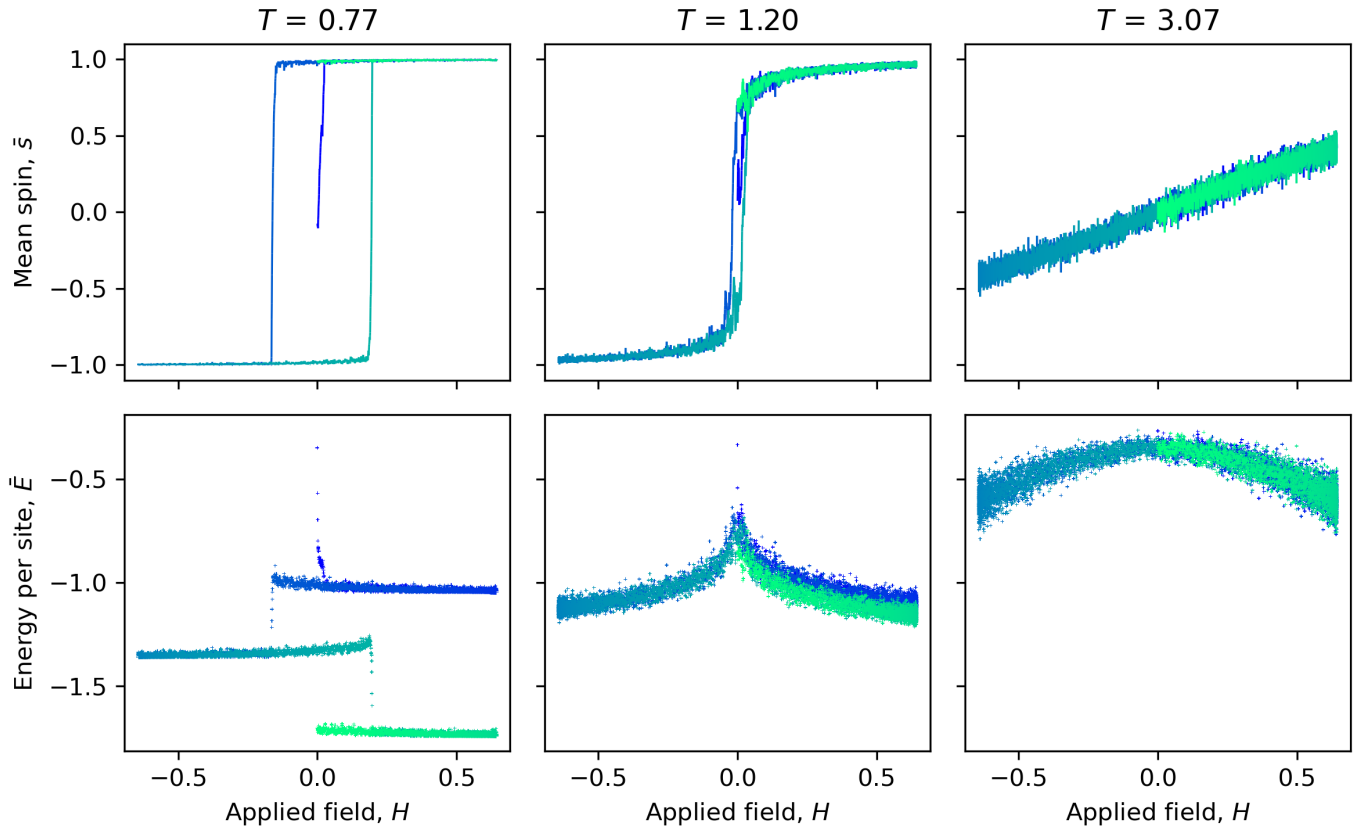

Fig. 10: Plots of mean spin and lattice energy against applied field for a 32×32 lattice, at three temperatures, showing hysteresis. Mean spin plots are on the top row, and energy plots the bottom. The plot has been coloured such that it gets lighter as time progresses; the applied field was modulated sinusoidally with a cycle length of 3π over 10,000 steps (representing an maximum change of $\Delta H = 6 \times 10^{-4}$ between steps). The behaviour in the three temperature regimes is wildly different; hysteresis is most apparent at low temperatures, whereas $\bar{s}$ appears to change linearly with H at high temperatures.

5 Conclusion

This paper has presented observations of a square Ising model simulated with the Metropolis algorithm, a Markov chain Monte Carlo method. After considering the implications of Markov chain convergence and correlations in the output, results for equilibrium magnetisation and heat capacity as a function of temperature were obtained, without an applied field. These results are in good qualitative agreement with the literature, and with simple thermodynamic arguments. The variation of critical temperature with lattice size, which is very apparent in our data, was used to extract a quantitative prediction from the simulations, of the critical temperature for an infinite 2D lattice, $T_c(\infty)$. Our estimate was $T_c(\infty) = 1.132 \pm 0.006$. In the units used in this work, Onsager's analytic result is $1/(\ln(1+\sqrt{2})) \approx 1.1346$. This differs from our estimate by 0.2% and lies within error.

We also made a brief survey of the behaviour of the model under an applied magnetic field. It was found that the region over which the order-disorder transition takes place broadens, and the critical temperature increases, with increased field strength. Cycling the field sinusoidally produces hysteretic effects at temperatures below the critical temperature.

There remain unexplored features of this model; had time permitted, we would have sought to investigate the frequency spectrum of energy and magnetisation fluctuations, the behaviour of the magnetic susceptibility in the transition region, and the effect of alternatives to the Metropolis algorithm on critical slowing-down.

Word count = 2956

A Code

The numerical code used in this work was spread over two files, `markov_chain_monte_carlo.pyx` and `comp_project_0_js2443.py`. The contents of these are listed below. In addition, three extra files with names of the form `comp_project_'n'_js2443.py` contain plotting code.

A.1 Metropolis algorithm - Cython code

```
import numpy as np
cimport cython
from libc.math cimport exp

@cython.boundscheck(False)
@cython.nonecheck(False)
def metrohaste(int numsteps, int[:,:] s, double H, double T, RNG):
    """
    A fast, Cython-based method to carry out one timestep of the Metropolis-
    -Hastings algorithm, a Markov chain Monte Carlo method. The lattice
    is stepped through in a left-to-right scanning fashion.

    Parameters
    ----------
    numsteps : int
        The number of Metropolis-Hastings timesteps to perform.
    s : square numpy array of ints
        The spin lattice, must be square and of type int.
    H : float
        The applied magnetic field, with dimensions (energy)/(spin).
    T : float
        The heat bath temperature, with dimensions of (energy).
    RNG : numpy.random generator object
        Seeded random number generator. Passing this to the function allows
        results to be repeated starting from a mother seed.

    Returns
    -------
    s_new : numpy array
        The updated spin lattice, ater 'numsteps' timesteps.

    """

    cdef Py_ssize_t N_ind = s.shape[0]
    cdef int N = s.shape[0]

    cdef double deltaE
    cdef double[:,:,:] p = RNG.uniform(0, 1, size=(numsteps,N,N))
    cdef Py_ssize_t t, i, j

    for t in range(numsteps):
        for i in range(N_ind):
            for j in range(N_ind):
                deltaE = (s[(i-1)%N, j] + s[(i+1)%N, j] +
                          s[i, (j-1)%N] + s[i, (j+1)%N] + 2*H)*s[i,j]
                s[i,j] *= (-1)**(exp(-deltaE/T) > p[t,i,j])

    return np.asarray(s)

@cython.boundscheck(False)
@cython.nonecheck(False)
def metrohaste_stats(int numsteps, int[:,:] s, double H, double T, RNG):
    """
    A fast, Cython-based method to carry out one timestep of the Metropolis-
    -Hastings algorithm, a Markov chain Monte Carlo method. The lattice
    is stepped through in a left-to-right scanning fashion. This version
    outputs statistics of the lattice over time, and is designed to called
    to output the entire length of a simulation [in practice this means it
    has to generate random numbers timestep-by-timestep, to conserve memory].
```

```
    Parameters
    ----------
    numsteps : int
        The number of Metropolis-Hastings timesteps to perform.
    s : square numpy array of ints
        The spin lattice, must be square and of type int.
    H : float
        The applied magnetic field, with dimensions (energy)/(spin).
    T : float
        The heat bath temperature, with dimensions of (energy).
    RNG : numpy.random generator object
        Seeded random number generator. Passing this to the function allows
        results to be repeated starting from a mother seed.

    Returns
    -------
    s_new : numpy array
        The updated spin lattice, ater 'numsteps' timesteps.
    sbars : numpy array
        A 1D array of the lattice's mean spin at each timestep.
    energies :
        A 1D array of the lattice's total energy at each timestep.
    """
    cdef Py_ssize_t N_ind = s.shape[0]
    cdef int N = s.shape[0]
    cdef double deltaE, E_t = 0
    cdef int count = 0, sqcount = 0
    cdef double[:,:] p = np.zeros((N, N))
    cdef double[:] sbars = np.zeros(numsteps, dtype=float)
    cdef double[:] energy = np.zeros(numsteps, dtype=float)
    cdef Py_ssize_t t, i, j

    for i in range(N_ind):
        for j in range(N_ind):
            E_t -= (0.25*(s[(i-1)%N, j] + s[(i+1)%N, j] +
                    s[i, (j-1)%N] + s[i, (j+1)%N]) + 2*H)*s[i,j]
    energy[0] = E_t

    for t in range(numsteps):
        count = 0
        sqcount = 0
        p = RNG.uniform(0, 1, size=(N,N))
        for i in range(N_ind):
            for j in range(N_ind):
                deltaE = (s[(i-1)%N, j] + s[(i+1)%N, j] +
                          s[i, (j-1)%N] + s[i, (j+1)%N] + 2*H)*s[i,j]
                if (exp(-deltaE/T) > p[i,j]):
                    s[i,j] *= -1
                    E_t += deltaE
                count += s[i,j]
                sqcount += s[i,j]*s[i,j]
        sbars[t] = <double>count/<double>(N*N)
        energy[t] = E_t

    return (np.asarray(s), np.asarray(sbars), np.asarray(energy))

@cython.boundscheck(False)
@cython.nonecheck(False)
def metrohaste_vect(int numsteps, int[:,:] s, double[:] H, double[:] T, RNG):
    """
    A fast, Cython-based method to carry out one timestep of the Metropolis-
    -Hastings algorithm, a Markov chain Monte Carlo method. The lattice
    is stepped through in a left-to-right scanning fashion. This version
    outputs statistics of the lattice over time, like metrohaste_stats, but
    takes temperature and magnetic field arguments as vectors of length
    'numsteps', in order to cycle the magnetic field or anneal the lattice
    according to a predefined trajectory.
```

```
    Parameters
    ----------
    numsteps : int
        The number of Metropolis-Hastings timesteps to perform.
    s : square numpy array of ints
        The spin lattice, must be square and of type int.
    H : float
        The applied magnetic field, with dimensions (energy)/(spin).
    T : float
        The heat bath temperature, with dimensions of (energy).
    RNG : numpy.random generator object
        Seeded random number generator. Passing this to the function allows
        results to be repeated starting from a mother seed.

    Returns
    -------
    s_new : numpy array
        The updated spin lattice, ater 'numsteps' timesteps.
    sbars : numpy array
        A 1D array of the lattice's mean spin at each timestep.
    energies :
        A 1D array of the lattice's total energy at each timestep.
    """
    cdef Py_ssize_t N_ind = s.shape[0]
    cdef int N = s.shape[0]
    cdef double deltaE, E_t = 0, T_t = T[0], H_t = T[0]
    cdef int count = 0, sqcount = 0
    cdef double[:,:] p = np.zeros((N, N))
    cdef double[:] sbars = np.zeros(numsteps, dtype=float)
    cdef double[:] energy = np.zeros(numsteps, dtype=float)
    cdef Py_ssize_t t, i, j

    for i in range(N_ind):
        for j in range(N_ind):
            E_t -= (0.25*(s[(i-1)%N, j] + s[(i+1)%N, j] +
                    s[i, (j-1)%N] + s[i, (j+1)%N]) + 2*H_t)*s[i,j]
    energy[0] = E_t

    for t in range(numsteps):
        T_t = T[t]
        H_t = H[t]
        count = 0
        sqcount = 0
        p = RNG.uniform(0, 1, size=(N,N))
        for i in range(N_ind):
            for j in range(N_ind):
                deltaE = (s[(i-1)%N, j] + s[(i+1)%N, j] +
                          s[i, (j-1)%N] + s[i, (j+1)%N] + 2*H_t)*s[i,j]
                if (exp(-deltaE/T_t) > p[i,j]):
                    s[i,j] *= -1
                    E_t += deltaE
                count += s[i,j]
                sqcount += s[i,j]*s[i,j]
        sbars[t] = <double>count/<double>(N*N)
        energy[t] = E_t

    return (np.asarray(s), np.asarray(sbars), np.asarray(energy))
```

A.2 A class to wrap Ising lattices

```
import numpy as np
import markov_chain_monte_carlo as mcmc
import multiprocessing as multi
import time
from functools import partial
import matplotlib.pyplot as plt
import plotaesthetics

class lattice:
```

```
    """
    A simple class to wrap the important aspects of a lattice of Ising model
    spins - number of lattice sites, applied field, temperature etc. Also
    includes methods to step the lattice forward in time according to the
    Metropolis-Hastings algorithm and to calculate statistics.
    """
    def __init__(self, sidelength, magfield, temp, uniform=False,
                 seed=None):
        """
        Initialise the lattice. Stores the characteristics passed as
        arguments but also calculates the mean spin (mean magnetisation per
        site) and variance in spin.

        Parameters
        ----------
        sidelength : int
            Square root of the number of lattice sites in the square lattice.
        magfield : float
            Applied magnetic field (permeability equals unity), with
            dimensions (energy)/(spin).
        temp : float
            Heat bath temperature, with dimensions of (energy) (k_B := 1).
        uniform : bool, optional
            Whether to initialise the lattice with spins all pointing up.
            Otherwise, a random lattice is generated. The default is False.
        seed : int
            Seed for the random number generator assigned to this lattice,
            which is used in the 'stepforward' method to evolve the lattice in
            time.

        Returns
        -------
        None.

        """
        self.time = 0
        self.N = sidelength
        self.H = magfield
        self.T = temp
        self.rng = np.random.Generator(np.random.PCG64(seed))

        if uniform == 'checkerboard':
            ## For testing purposes
            self.spins = (-1)**(np.sum(np.indices((self.N, self.N)),
                                       axis=0)%2)
        if uniform == 'dynamic':
            ## uniform for T < 1, random for T > 1
            if self.T < 1.0:
                self.spins = ((-1)**self.rng.integers(0,2)*
                              np.ones((self.N, self.N), dtype=int))
            else:
                self.spins = (-1)**self.rng.integers(0, 2,
                                        size=(self.N, self.N), dtype=int)
        elif uniform:
            self.spins = ((-1)**self.rng.integers(0,2)*
                          np.ones((self.N, self.N), dtype=int))
        else:
            self.spins = (-1)**self.rng.integers(0, 2, size=(self.N, self.N),
                                                 dtype=int)
        self.E = self.updateE()

    def __str__(self):
        """
        Create a toy string version of the spin lattice when the argument to
        'print'; prints a blank space where s = -1, and a unicode square where
        s = +1.
        """
        string = ''
        for i in range(self.N):
```

```
            for j in range(self.N):
                string += '\u25a0 ' if (self.spins[i,j] + 1) else '  '
            string += '\n'
        return string

    def stepforward(self, numsteps=1, desiredout='spins'):
        """
        A Cython-enhanced method to carry out the Metropolis-Hastings
        algorithm on the spin lattice, for one timestep.
        This function is dependent on the 'metrohaste' import, which in turn
        is dependent on a successfully compiled
        'markov_chain_monte_carlo.pyx'. If a link in this chain does
        not work, please use 'stepforward_slow', but there will be a
        performance penalty of around 100x.

        Parameters
        ----------
        numsteps : int, optional
            Number of timesteps to move. Only the final spin lattice is
            returned for numsteps > 1. The default is 1.
        desiredout : string, optional
            One of 'stats' and 'spins'. If 'stats' is passed, the mean and
            variance of the spins at each timestep is returned; if 'spins' is
            passed, the lattice is skipped to its final state and the
            array of spins returned. The default is 'spins'.

        Returns
        -------
        Either the tuple ('sbars(t)', 'energies(t)', burn-in time) or the
        final spin lattice after 'numsteps'.

        """
        s = self.spins
        if desiredout == 'stats':
            out = mcmc.metrohaste_stats(numsteps, s, self.H, self.T, self.rng)
            self.spins = out[0]
            self.E = out[3][-1]
            if self.time == 0:
                burn = self.findburnins_highT(out[1])
            else:
                burn = np.nan
            self.time += numsteps
            return (out[1], out[2], out[3], burn)
        else:
            self.spins = mcmc.metrohaste(numsteps, s, self.H, self.T, self.rng)
            self.time += numsteps
            return self.spins

    @staticmethod
    def stepforward_l(latlist, numsteps=1, parallel=False):
        """
        As for 'stepforward', but now a list of lattices should be passed.
        This allows for vectorisation of statistical calculations over an
        ensemble of Markov chains (burn-in times in particular).

        Parameters
        ----------
        latlist : iterable
            list, tuple or numpy array of lattice objects.
        numsteps : int, optional
            See 'stepforward'. The default is 1.

        Returns
        -------
        See 'stepforward. Now numpy arrays of the usual returns are
        returned, with the first index corresponding to the particular
        Markov chain.

        """
```

```
        numchains = len(latlist)
        sbars = np.zeros((numchains, numsteps))
        Es = sbars.copy()
        if (latlist[0].N*numsteps > 100000 and numchains > 1) and parallel:
            print('Parallelism invoked')
            funcs = [partial(mcmc.metrohaste_stats, numsteps,
                             l.spins, l.H, l.T, l.rng) for l in latlist]
            outs = parallelise(funcs)
            for m in range(numchains):
                latt = latlist[m]
                out = outs[m]
                latt.spins = out[0]
                sbars[m], Es[m] = out[1:3]
                latt.E = Es[m,-1]
        else:
            for m in range(numchains):
                latt = latlist[m]
                out = mcmc.metrohaste_stats(
                    numsteps, latt.spins, latt.H, latt.T, latt.rng)
                latt.spins = out[0]
                sbars[m], Es[m] = out[1:3]
                latt.E = Es[m,-1]
        Ts = np.array([l.T for l in latlist])
        burns = lattice.findburnins(sbars, Ts, bound=1/(latlist[0].N))

        return(sbars, Es, burns)

    @staticmethod
    def stepforward_vect(latlist, H, T, numsteps=1):
        """
        As for 'stepforward', but now a list of lattices should be passed.
        This allows for vectorisation of statistical calculations over an
        ensemble of Markov chains (burn-in times in particular).

        Parameters
        ----------
        latlist : iterable
            list, tuple or numpy array of lattice objects.
        numsteps : int, optional
            See 'stepforward'. The default is 1.

        Returns
        -------
        See 'stepforward. Now numpy arrays of the usual returns are
        returned, with the first index corresponding to the particular
        Markov chain.

        """
        numchains = len(latlist)
        sbars = np.zeros((numchains, numsteps))
        Es = sbars.copy()
        if latlist[0].N*numsteps > 100000 and numchains > 1:
            print('Parallelism invoked')
            funcs = [partial(mcmc.metrohaste_vect, numsteps,
                             latlist[m].spins, H[m,:], T[m,:], latlist[m].rng)
                             for m in range(numchains)]
            outs = parallelise(funcs)
            for m in range(numchains):
                latt = latlist[m]
                out = outs[m]
                latt.spins = out[0]
                sbars[m], Es[m] = out[1:3]
                latt.E = Es[m,-1]
        else:
            for m in range(numchains):
                latt = latlist[m]
                out = mcmc.metrohaste_vect(
                    numsteps, latt.spins, H[m,:], T[m,:], latt.rng)
                latt.spins = out[0]
```

```
            sbars[m], Es[m] = out[1:3]
            latt.E = Es[m,-1]
    burns = lattice.findburnins_highT(sbars, bound=1e-2/latlist[0].N)

    return(sbars, Es, burns)

def updateE(self):
    """
    Manually re-compute the total energy of the lattice, using the
    array of spins, at the current time.
    """
    s = self.spins
    self.E = -np.sum((0.25*(np.roll(s, 1, axis=0) + np.roll(s, -1, axis=0)+
                            np.roll(s, 1, axis=1) + np.roll(s, -1, axis=1))
                      + 2*self.H)*s)
    return self.E

@staticmethod
def findburnins(sbars, Ts, Tthresh=0.8, bound=1e-2, binsize=10):
    """
    A hybrid method to determine burn-in that takes an input of mean spins
    over time, 'sbars', and splits it according to the corresponding
    lattice temperatures 'Ts'. Temperatures higher than 'bound' have their
    spins passed to 'findburnins_highT'. Lower temperatures are operated on
    by this function, which examines when the mean spin - split into time
    averaged chunks - gets closer to +/- 1 than bound.

    Parameters
    ----------
    sbars : numpy array
        m x n array of mean spin over n numsteps for m lattices.
    Ts : numpy array
        Array of length m containing corresponding lattice temperatures.
    Tthresh : float, optional
        The temperature boundary at which to assign the lattices to
        different algorithms. The default is 0.8.
    bound : float, optional
        The user-chosen value that (|equilibrium spin| - 1) should not
        be expected to exceed. The default is 1e-2.
    binsize : int, optional
        The size of the chunks over which 'sbars' should be time-averaged.
        This helps mitigate the effect of thermal noise. The default is 10.

    Returns
    -------
    burns : numpy array
        Array with the same dimensions as 'Ts' - the burn-in time for each
        lattice.

    """

    try:
         numchains = sbars.shape[0]
         numsteps = sbars.shape[1]
    except IndexError as e:
        print(e)
        numchains = 1
        numsteps = sbars.shape[0]
        sbars = np.reshape(sbars, (1, numsteps))
        Ts = np.reshape(Ts, (1, 1))

    burns = (numsteps-1)*np.ones(numchains, dtype=int)
    lowTinds = np.arange(numchains, dtype=int)[Ts < Tthresh]
    highTinds = np.arange(numchains, dtype=int)[np.logical_not(Ts<Tthresh)]

    binmean = np.cumsum(sbars -
                        np.hstack((np.zeros((numchains, binsize)),
                                   sbars[:,:-binsize])), axis=1)/binsize
    if highTinds.size:
```

```
            burns[highTinds] = lattice.findburnins_highT(sbars[highTinds,:],
                                                         bound=bound/100)
        chains_satis = np.any(np.abs(np.abs(binmean[lowTinds,:]) - 1) < bound,
                              axis=1)
        burnslow = burns[lowTinds]      ## Juggling array definitions to modiy
                                        ## arrays in place
        burnslow[chains_satis] = np.argmax(
            np.abs(np.abs(binmean[lowTinds,:]) - 1) < bound,
            axis=1)[chains_satis]
        burns[lowTinds] = burnslow

        return burns

    @staticmethod
    def findburnins_highT(sbars, bound=5e-4):
        """
        An alternative to method to determine the burn-in time that handles
        thermal fluctuations better but cannot diagnose low temperature
        metastability. This method focuses on finding when the cumulative
        average of mean spins,
        $$ \frac{1}{t}\sum_{t'=0}^{t} \frac{\bar{s}(t')} $$
        falls below a user-defined 'bound'. This is observed to discriminate
        quite accurately between relatively short burn-ins.

        Parameters
        ----------
        sbars : numpy array
            Either a 1D array of mean spins at each timestep for a single
            lattice, or a 2D array with the first index denoting which lattice
            the mean spins correspond to.
        bound : float, optional
            The user-chosen value to which the average should relax before the
            label 'bured-in' applies. The default is 1e-3.

        Returns
        -------
        burns : int or numpy array of ints
            The determined burn-in time(s) of the supplied lattice(s).

        """
        try:
             numchains = sbars.shape[0]
             numsteps = sbars.shape[1]
        except IndexError as e:
            print(e)
            numchains = 1
            numsteps = sbars.shape[0]
            sbars = np.reshape(sbars, (1, numsteps))

        cmean = np.cumsum(sbars, axis=1)/np.arange(1, numsteps + 1)
        burns = np.argmin(np.abs(np.abs(np.diff(cmean, axis=1)) - bound),
                          axis=1)
        return burns


    @staticmethod
    def autocorrelation(sbars, burns):
        """
        Computes the autocorrelation of all the rows in 'sbars', discarding
        those values before burn-in. The different burn-ins for each row
        means that a loop over the number of Markov chains has to be
        performed, unforunately; np.correlate is used at each iteration,
        which is fast because it uses FFT behind the scenes.
        The output of this function is formatted so that if 'sbars' is a
        MxN array, 'autocorrs' will be a Mx(2N+1) array; this is quite
        familiar for FFTs.

        Parameters
        ----------
```

```
        sbars : numpy array
            1D or 2D array of mean spins over time. Time should evolve along
            the second axis.
        burns : numpy array
            1D numpy array of the same length as sbars' first axis, containg
            the burn-in times for each Markov chain.

        Returns
        -------
        autocorrs : numpy array
            Autocorrelation of each of the row vectors of 'sbars', with shape
            as described in the header. Cutting off the parts of each spin-
            -over-time vector pre burn-in leads to autocorrelation vectors of
            differing sizes (2*(numsteps-burnin)+1 to be specific), so each
            autocorrelation is symmetrically padded with zeros.
        decorrtimes : numpy array
            Array of decorrelation times (when autocorrelation first dips
            below 1/e), of the same shape as 'burns'.
        """
        try:
            numsteps = sbars.shape[1]
            numchains = sbars.shape[0]
        except IndexError:
            numsteps = sbars.size
            numchains = 1
            sbars = sbars.reshape(1, numsteps)
            burns = burns.reshape(1)

        gsteps = numsteps - burns
        autocovs = np.zeros((numchains, 2*numsteps+1))
        for m in range(numchains):
            dev = sbars[m,burns[m]:] - np.mean(sbars[m,burns[m]:])
            out = np.correlate(dev, dev, mode='full')
            autocovs[m,:] = np.pad(out, (burns[m]+1,burns[m]+1),
                                   'constant', constant_values=(0,0))

        autocorrs = (autocovs.T/autocovs[:,numsteps]).T
        ## Finds first instance where |autocorrelation| < 1/e
        decorrtimes = np.argmax(np.abs(autocorrs[:,numsteps:]) - np.exp(-1) < 0,
                                axis=1)

        return autocorrs, decorrtimes



    def heatcapacity(Ts, Es, burns):
        """
        A very simple function that computes the lattice heat capacity
        according to the fluctuation dissipation theorem. Makes use of the
        function 'reducedvar' to discard energies pre burn-in.
        """
        varEs = reducedvar(Es, burns, axis=1)
        C = varEs/Ts**2
        return C


def threadedfunc(pipe, func):
    """
    Very simple function to change a return command to a send-over-pipe
    command for the argument 'function'.
    """
    out = func()
    if out is not None:
        pipe.send((out))

def parallelise(functions):
    """
    Function utilising multiprocessing to execute several 'functions' in
    parallel. Returns all of the arguments of the functions as a tuple.
```

```
    Spawning pipes and processes can take about a second per process, so
    there is no point parallelising routines that take ~1 sec.

    Parameters
    ----------
    functions : iterable of callables
        List or tuple of functions to execute, must be of the form func()
        (no arguments), so use functools.partial if necessary.

    Returns
    -------
    returns : tuple
        Tuple of returns of all the 'functions' (which may themselves be
         tuples).

    """
    n = len(functions)
    pipes = []
    threads = []
    for i in range(n):
        pipes.append(multi.Pipe())
        threads.append(multi.Process(name='func %d'%(i), target=threadedfunc,
                                     args=(pipes[i][1], functions[i])))
        threads[i].start()
    try:
        returns = list(range(n))
        stillgoing = list(range(n))
        while len(stillgoing):
            for i in stillgoing:
                if pipes[i][0].poll(0.1):
                    returns[i] = pipes[i][0].recv()
                    pipes[i][0].close()
                    threads[i].join()
                    stillgoing.remove(i)
            time.sleep(0.01)
        return returns
    except Exception as e:
        for i in range(n):
            threads[i].close()
            pipes[i][0].close()
        raise(e)
```