Physics 381/382/504 Lab Handout

David Stewart
Date modified: Spring 2019
Prior Version: Jed Thompson Spring 2017

February 19, 2019

1 Introduction

The Ising model is one of the simplest physical models that exhibits a phase transition. It consists of a d-dimensional[1] square lattice of "particles," with a "spin" (either up or down) and a magnetic moment that aligns with its spin. The particles (also known as lattice sites) interact with each other through a nearest-neighbor interaction of strength J and are coupled to an external magnetic field with strength B. For a lattice of side-length N, we then have the Hamiltonian:

$$H = -J \sum_{\langle i,j \rangle} \sigma_i \sigma_j - B \sum_i \sigma_i \tag{1}$$

where $\sigma_i = \pm 1$ denotes the i-th spin state and $\langle i, j \rangle$ denotes a sum over nearest neighbors (in 2-dimensions there are 4 nearest neighbors). Analytic solutions to the Ising model are usually considered in the limit $N \to \infty$.

For $J > 0$, the spins tend to align, but at finite temperature, thermal fluctuations will prevent them from doing so. For $d \geq 2$, the system exhibits a phase transition at zero applied field and a critical temperature T_c, such that for $T > T_c$, the system is dominated by thermal fluctuations and no overall magnetization is observed, while for $T < T_c$, the spins tend to align and the system develops a spontaneous magnetization. You will simulate the Ising model using *Markov Chain Monte Carlo* (MCMC) techniques, and one of your first tasks will be to measure T_c.

A collection of exact results for the Ising model is presented at [1]. For a good introduction to Monte Carlo sampling techniques in general, see [2].

[1]We will soon specialize to the case $d = 2$, but the same questions you will investigate here can be asked for the Ising model in any number of dimensions. For $d = 1, 2$ and $d \geq 4$, it turns out the Ising model is exactly solvable, meaning all the quantities you will compute from simulation can be solved for analytically. The case $d = 3$ is an area of current research.

2 Overview of Statistical Mechanics

This section is not intended to be a complete exposition of statistical mechanics, but merely to remind you of a few of the basic ideas and definitions. For a much more thorough reference, consult any textbook on thermodynamics or statistical mechanics, such as [3].

The most important prediction of statistical mechanics for us is the Gibbs distribution, which states that the probability of finding a macroscopic system in a microscopic state of energy E is given by:

$$P(\text{state}) \propto e^{-\frac{E}{k_B T}} \tag{2}$$

where k_B is Boltzmanns constant and T is the temperature of the system. In principle this makes it easy to compute various macroscopic properties of the system. For example, if we index the possible microstates by i and their energies by E_i then the expected value of some observable property (like the overall energy of the system) is given by averaging over all the microstates with weights given by Equation 2. Denoting the thermodynamic average at a temperature T by $\langle \cdot \rangle_T$ (we will occasionally drop the T but it should always be understood that the thermodynamic average is temperature-dependent) and defining the thermodynamic beta $\beta = 1/(k_B T)$ we have:

$$\langle E \rangle_T = \frac{\Sigma_i E_i e^{-\beta E_i}}{\Sigma_i e^{-\beta E_i}} \tag{3}$$

where the denominator is a normalization factor to deal with the fact that Equation 2 is only a proportionality, not an equality.

Now there is a very slick way to repackage this in terms of the *partition function* $\mathcal{Z}$, defined as

$$\mathcal{Z} = \sum_i e^{\beta E_i} \tag{4}$$

It is clearly a function of temperature because of the explicit dependence on β, but it is also implicitly a function of other external variables, such as applied magnetic field in the 2-D Ising model because each factor of E_i depends on these variables. Now many thermodynamic quantities can be realized as simple derivatives of the partition function. For example:

$$\langle E_T \rangle = \frac{1}{\mathcal{Z}} \sum_i E_i e^{-\beta E_i} = \frac{1}{\mathcal{Z}} \left(-\frac{\partial}{\partial \beta} \sum_i e^{-\beta E_i} \right) = \frac{1}{\mathcal{Z}} \left(-\frac{\partial}{\partial \beta} \mathcal{Z} \right) = -\frac{\partial}{\partial \beta} \ln \mathcal{Z} \tag{5}$$

since we assume each E_i does not depend on temperature.

The partition function can also be used to derive various useful relations among thermodynamic quantities. In this experiment, you will be interested in computing specific heat c_V and magnetic susceptibility χ, defined as:

$$c_V = \frac{\partial \langle E \rangle_T}{\partial T}, \quad \chi = \frac{\partial \langle M \rangle_T}{\partial B} \tag{6}$$

where M is the magnetization (per site) of the system and B is an externally-applied magnetic field. Trying to compute these in this way would be difficult because of the large errors associated with derivatives of numeric quantities, but using the partition function we can find more useful forms for them. We begin with c_V :

$$c_V = \frac{\partial \langle E \rangle_T}{\partial T}, c_V = \frac{\partial \langle E \rangle_T}{\partial T} = \frac{\partial \langle E \rangle_T}{\partial \beta}\frac{\partial \beta}{\partial T} = -\frac{\beta}{T}\frac{\partial \langle E \rangle_T}{\partial \beta} \tag{7}$$

$$= \frac{\beta}{T}\frac{\partial}{\partial \beta}\left(\frac{1}{\mathcal{Z}}\frac{\partial}{\partial \beta}\mathcal{Z}\right) \tag{8}$$

$$= \frac{\beta}{T}\left(\frac{1}{\mathcal{Z}}\frac{\partial^2}{\partial \beta^2}\mathcal{Z} - \frac{1}{\mathcal{Z}^2}\left(\frac{\partial \mathcal{Z}}{\partial \beta}\right)^2\right) \tag{9}$$

Note that the second term in the second factor in Equation 9 is simply $\langle E \rangle_T^2$ while the first term is:

$$\frac{1}{\mathcal{Z}}\frac{\partial^2}{\partial \beta^2}\mathcal{Z} = \frac{1}{\mathcal{Z}}\frac{\partial^2}{\partial \beta^2}\sum_i e^{-\beta E_i} = \frac{1}{\mathcal{Z}}\sum_i E_i^2 e^{-\beta E_i} = \langle E^2 \rangle_T \tag{10}$$

so we obtain a more useful formula for c_V:

$$c_V = \frac{\beta}{T}\left(\langle E^2 \rangle_T - \langle E \rangle_T^2\right) \tag{11}$$

A virtually identical derivation gives a similar formula for χ:

$$\chi = \beta\left(\langle M^2 \rangle_T - \langle M \rangle_T^2\right) \tag{12}$$

Throughout this experiment, you should use Equations 11 and 12 to compute c_V and χ respectively.

The final thermodynamic quantity we will care about here is the *correlation length* ξ. In any system at a finite temperature there will be thermodynamic fluctuations, and often we are interested in the rough size of these fluctuations. To probe this, we look at spatial correlation functions, which measure how closely two different locations in our system are correlated. For the Ising model, we will be interested in the correlation function $R(x) \equiv \langle \sigma(0)\sigma(x) \rangle$, which measures how likely it is that two spins separated by a distance x (or, in our simulation number of lattice site) are aligned. Since the lattice is translation-invariant, we can always translate so one of the lattice sites are are considering is the origin). If $\langle \sigma(0)\sigma(x) \rangle \sim 1$ then the two sites are highly correlated, while if $\langle \sigma(0)\sigma(x) \rangle \sim 0$ then the two sites are uncorrelated. (If $\langle \sigma(0)\sigma(x) \rangle \sim -1$, the two sites are highly correlated but are likely to be anti-aligned rather than aligned.)

It is generally true that spatial correlation functions die off exponentially with some characteristic length scale ξ:

$$\langle \sigma(0)\sigma(x) \rangle \sim e^{-x/\xi} \tag{13}$$

where ξ is known as the correlation length and will depend on temperature. For $x \ll \xi$, the correlation function will typically have some power-law behavior with some characteristic exponent typically written as:

$$\langle \sigma(0)\sigma(x) \rangle \sim \frac{1}{|x|^{d-2+\eta}}, \quad x \ll \xi \tag{14}$$

where d is the dimension of the system. In this lab we will be interested in finding the correlation length of the system at various temperatures and also in finding the exponent near the phase transition.

3 Overview of Numerical Methods

This section is intended as an extremely brief overview of why numerical methods are necessary for this problem and how they work. I will skip most of the details since they are discussed quite well (and in the context of the Ising model) in [2].

In principle the rules of statistical mechanics are very straightforward. If we want to compute the thermodynamic average of some variable $\mathcal{O}$, we simply use the Gibbs distribution:

$$\langle \mathcal{O} \rangle_T = \frac{1}{\mathcal{Z}} \sum_i \mathcal{O}(i) e^{-i\beta E_i} \tag{15}$$

In practice this is very difficult, since we need to sum over all the different microstates of the system, and any macroscopic system has an incredibly large number of possible microstates. Even a very simple system, such as a 20×20 lattice of spins , has $2^{20^2} \sim 10^{120}$ possible states, substantially more than the number of particles in the universe, so actually computing the thermodynamic average is impossible. Instead, we need to find a way to sample the Gibbs distribution in a way that gives a good approximation to the true thermodynamic average. This is where numerical methods come in.

In order to make our numerical approximations, we first note that we dont have access to the actual Gibbs probability distribution, given by

$$P(\text{state}) = \frac{1}{\mathcal{Z} e^{-\beta E(\text{state})}} \tag{16}$$

since we do not know $\mathcal{Z}$ We can, however, easily compute $e^{-\beta E}$ for any given state, so we have access to a function which is proportional to the actual probability distribution we want to sample. This turns out to be enough. At this point there are several different numerical techniques we can use, such as *importance sampling, rejection sampling,* and the *Metropolis algorithm* (all of these are others are discussed in [2]). In this experiment you will use the last of these, and so that is all we will discuss here.

Before we continue however, it is worth understanding why the most naive sampling method, *uniform sampling*, does not work. Imagine trying to compute a thermodynamic average by selecting n random states, computing $\mathcal{Z}$ based only on these states, and then computing $\mathcal{Z}^{-1}\Sigma_{j=1}^{n}\mathcal{O}(j)e^{-\beta E_j}$ as an approximation of $\langle\mathcal{O}\rangle_T$. The key problem with this is that although the number of states is very large, both $\mathcal{Z}$ and the thermodynamic average are typically dominated by a much smaller set of states, known as the *typical set*. If our uniform sample doesnt happen to contain any of the states in the typical set, we will come up with a wildly inaccurate estimate for $\langle\mathcal{O}\rangle_T$. So just how much smaller does the typical set tend to be? In the case of the Ising model with N^2 sites near the critical temperature, it turns out that the typical set is quite small. In order to hit it even once you need to draw on average $2^{N^2/2}$ samples.[2] For $N = 20$, this is roughly 10^{60} samples. Not quite the number of particles in the universe, but large enough that this should convince you that we need something more sophisticated than uniform sampling.

The basic idea of Metropolis sampling is to start somewhere in the state space and then explore the space in a way that tends to keep you near the areas of high probability density, since these are the areas that will dominate the thermodynamic average. It can be shown that over a long period of time this procedure gives an accurate sample of the underlying distribution, but we will not do so here. Instead, we will focus on how the algorithm works and what problems it can run into.

Metropolis sampling falls into a class of numerical techniques known as Markov Chain Monte Carlo (MCMC). This is because it explores the probability distribution by building up a Markov chain of samples. At each step, it only uses information about its current position to determine where it should move next. Over time, it will generate a set of samples x(t) that will give an accurate value for thermodynamic averages. So how does it do this?

At each step t, a tentative step x' is generated (i.e. from the given state of the system, a spin is chosen to potentially be flipped). Then we compute

$$a = \frac{\exp\big(-\beta E(x')\big)}{\exp\big(-\beta E(x^{(t)})\big)} \tag{17}$$

where $E(x')$ and $E(x^{(t)})$ are the energies of the system in the trial state and current state respectively. If $a \geq 1$ then the trial state is more probable than the one we are currently in (and thus closer to the typical set), and so we accept it and set $x^{(t+1)} = x'$. If $a < 1$ then the trial state is less probable than the one we are currently in, but that doesnt mean we should immediately rule it out. Instead, we want to explore the probability space, and so we accept the trial state with probability a. Importantly, if we reject the trial state we set $x^{(t+1)} = x^{(t)}$, so it is possible to have the same state for several steps in a row if that state is much more likely than the others close to it.

Clearly nearby samples are correlated, but over time the Metropolis algorithm will explore the state space and then a good approximation for $\mathcal{Z}$ and the thermodynamic

[2] Deriving these results uses some probability theory that we do not have here. [2] has slightly more information, but for good comprehension, you are recommended to use a book on information theory, such as [4]

average can be computed by:

$$\mathcal{Z} \approx \sum_t e^{-\beta E(x^{(t)})} \tag{18}$$

$$\langle \mathcal{O}_T \rangle \approx \frac{1}{\mathcal{T}} \sum_t \mathcal{O}\left(x^{(t)}\right) \tag{19}$$

where $\mathcal{T}$ is the total number of samples generated by the run. In the second equation, note the absence of $\mathcal{Z}$ and the Gibbs weight $e^{-\beta E}$. This is because the Metropolis algorithm automatically generates more samples in the regions of high probability density in precisely a way that eliminates the need for these factors.

Now that we have some idea of these techniques, it is worth pointing out a few things. First, although we have denoted our sample series above as $\{x^{(t)}\}$, you should not interpret this as a time series. This is a walk through the probability space of the system, and although it is sometimes helpful to think about your experiment as simulating a spin lattice for some finite time period, you must remember that this isnt actually what you are doing. Second, the Metropolis algorithm must be given the chance to fully explore the probability space, otherwise the final average will be significantly biased by the initial state you set the system in. Third, although the algorithm will eventually find the typical set, it may take a while to do so. Once it finds the typical set the simulation is said to have *converged*, but before this happens, all the samples from the probability distribution are being drawn from low-probability areas dictated by your choice of initial conditions. For this reason, it is usually recommended that you allow the system to burn in for several thousand steps before you start collecting data.

For more information on how to implement the Metropolis algorithm in the case of the Ising model, see Section 5. For tips on how to better achieve convergence, see Appendix A.

4 Theory and Goals

From here on, it is easiest to choose a simple unit system. We first set $J = 1$, which means all energies will be measured in units of J. We then set Boltzmanns constant, $k_B = 1$, and since k_B has units of energy over temperature, this amounts to measuring temperature in units of energy and thus in units of J. With these choices, all quantities in the problem become unitless, and we will assume this unit system throughout the rest of this handout. We also specialize to the case $d = 2$.

The exact critical temperature of the 2D Ising model is given by:

$$T_c = \frac{2}{\ln\left(1+\sqrt{2}\right)} \approx 2.269 \tag{20}$$

Your first task is to simulate the Ising model and determine T_c numerically (with an uncertainty). A general characteristic of critical points is that many thermodynamic quantities in the system show nonanalytic behavior (discontinuities, divergences, etc.), and T_c can be determined by fitting theoretical functions to the behavior of these quantities in the regime near the critical point.

You should use at least three different system state variables (specific heat, magnetization, susceptibility, and correlation length) to determine T_c. Each experimental value for T_c should have an uncertainty of $\lesssim 5\%$, and your report should contain a full description and justification for your statistical analysis.

Your next task is to compare the expected theoretical forms for the heat capacity, magnetization, and magnetic susceptibility to a numerical calculation. Since the Ising model is solvable in two dimensions, there are known exact results for these that are valid at all temperatures. For example, in the $N \to \infty$ limit, the enrgy per lattice site as a function of temperature, $E(T)$ is given by:

$$E(T) = -2\tanh\left(\frac{1}{T}\right) - \frac{\sinh^2(2/T) - 1}{\sinh\big(2/T\big)\cosh 2/T}\left[\frac{2}{\pi}K_1(\kappa) - 1\right] \tag{21}$$

where,

$$\kappa = 2\frac{\sinh\big(2/T\big)}{\cosh^2(2/T)} \tag{22}$$

$$K_1(\kappa) = \int_0^{\pi/2} \frac{\mathrm{d}\phi}{\sqrt{1 - \kappa^2 \sin^2\phi}} \tag{23}$$

Here $K_1(\kappa)$ is a complete elliptic integral of the first kind and can be numerically evaluated. The average magnetization per site has a much simpler form. For $T > T_c$, we have $\langle M \rangle = 0$, and for $T < T_c$ we have:

$$|\langle M \rangle| = \Big(1 - [\sinh\big(2/T\big)]^{-4}\Big)^{1/8} \tag{24}$$

Knowing these formulas, it is possible to compute specific heat (c_V) and magnetic susceptibility (χ) as:

$$c_V = \frac{\partial \langle E \rangle_T}{\partial T}, \quad \chi = \frac{\partial \langle M \rangle_T}{\partial B} \tag{25}$$

Needless to say, these formulas are unwieldy. Luckily, your analysis will be made easy by restricting your attention to the vicinity of the critical temperature. It is a general result of the theory of critical phenomena that for temperatures T near T_c, virtually all the system state quantities of interest scale as some power law in the *reduced temperature* t:

$$t = \frac{T - T_c}{T_c} \tag{26}$$

The exponents in these power-law relations are known as *critical exponents*, and they are of principle interest, since they are are characteristics of the critical point itself, not

Crit Exp.	Definition	Theor. Values
α	$c_V \propto \lvert t\rvert^{-\alpha}$	0
β	$\lvert M\rvert \propto \lvert t\rvert^{\beta}$	1/8
γ	$\chi \propto \lvert t\rvert^{-\gamma}$	7/4
δ	$\lvert M\rvert \propto \lvert B\rvert^{1/\delta}$	15
ν	$\xi \propto \lvert t\rvert^{-\nu}$	1
η	$\langle\sigma(0)\sigma(x)\rangle \propto \lvert x\rvert^{-d+1-\eta}$	1/4
Parameter	**Name/definition**	
c_V	specific heat	
M	magnetization	
χ	magnetic susceptibility	
ξ	correlation length	
t	$(T-T_c)/T$	

Table 1: Theoretical values of some of the critical exponents in the $d = 2$ Ising model as $N \to \infty$. In general, α, γ, and ν are well-defined both above and below T_c, and the Renormalization Group (RG) approach to critical phenomena implies that in most cases (including the 2-D Ising model), they should be the same both above and below T_c. Note that β is only well-defined above T_c. The value of 0 given for α means that c_V will not truly behave as a power law in $|t|$ but rather $c_V \propto \log(t)$ near T_c.

the Ising model in particular. In other words, there are several different systems you can write down with different degrees of freedom and different Lagrangians but which share some critical behavior and critical exponents near a phase transition.[3] The critical exponents you will be interested in are defined as follows, and Table 4 gives exact values for these exponents in the 2-D Ising model.

$$c_V \propto |t|^{-\alpha} \tag{27}$$
$$\chi \propto |t|^{-\gamma} \tag{28}$$
$$\xi \propto |t|^{-\nu} \tag{29}$$
$$|M| \propto |t|^{\beta} \tag{30}$$

where ξ is the correlation length and β is not to be confused with the thermodynamic $\beta = 1/(k_B T)$. In the above, it must be noted that the expression for $|M|$ only makes sense below T_c. Further, near T_c (and within the correlation length), the spatial spin correlation function $R(x)$ has the form:

$$R(x) = \langle \sigma(0)\sigma(x) \rangle \propto \frac{1}{|x|^{d-2+\eta}} \tag{31}$$

where $d = 2$ is the dimensionality of the system. Finally, for T slightly above T_c, the magnetization with an applied (static, homogeneous) magnetic field B has the form:

$$|M| \propto |B|^{1/\delta} \tag{32}$$

You should be able to use your simulated data to (1) verify that the above quntities truly do have power-law scaling behavior near T_c and (2) find the critical exponents discussed above. At minimum you should find α, β, γ, ν, and η, but if you have time, finding δ is also fairly simple.

The general theory of critical phenomena also yields the following relations between the various critical exponents, where again d is the dimension of the system:

$$\text{Rushbrooke's, Identity:} \quad \alpha + 2\beta + \gamma = 2 \tag{33}$$
$$\text{Widom's Identity:} \quad \delta - 1 = \gamma/\beta \tag{34}$$
$$\text{Josephson's Identity} \quad 2 - \alpha = \mathrm{d}\nu \tag{35}$$
$$\text{Fisher's Identity:}\gamma = (2 - \eta)\nu \tag{36}$$

These imply that even though there are apparently six independent critical exponents listed above, there are only two independent exponents.[4] Once you have computed the

[3] This is a highly nontrivial phenomenon, known as *critical universality*, and the reasons for it were only uncovered in the 1960s and 1970s, principally by Leo Kadanoff, Michael Fisher, and Ken Wilson. For their work they jointly received the Wolf prize in 1980, and for his work Wilson was awarded the Nobel Prize in Physics in 1982.

[4] All this is essentially a consequence of the fact that you only need to tune two parameters (i.e. temperature to T_c and external applied magnetic field B to 0) to reach the critical point.

above critical exponents, verify that your system obeys these identities to within your experimental precision.

So far, everything in this section has assumed the infinite-size limit $N \to \infty$. Since this is a simulation-based experiment and you only have access to a finite-size computer and a finite amount of time, you will be simulating only a finite-size grid [5]. In this case, you will only have finitely-many degrees of freedom (namely the N^2 lattice sites), and it is generally true that any system with only a finite number of degrees of freedom cannot have a true phase transition.[5] Luckily for us, such systems can come close to a phase transition, but there will still be some residual finite-size effects that will be most pronounced very near the critical temperature. For example, in a finite-size system, the specific heat, magnetic susceptibility, etc. cannot truly diverge, but instead will obtain some maximum value that will depend on the grid size N.

This behavior sets in when the system "realizes" it is finite, which happens roughly when the correlation length is on the order of the lattice size: $\xi \sim N$. Since $\xi \propto |t|^{-\nu}$, this gives an effective value t_{eff} at which $|t_{\text{eff}}|^{-\nu} \sim N$, and we expect that the various thermodynamic quantities will have their divergences cut off near this temperature. Thus we expect that the maximum value of ξ for example, instead of being infinite at T_c, will be given roughly by:

$$\xi(T = T_c) \sim (N^{-1/\nu}) \sim N^{\gamma/\nu} \tag{37}$$

This analytic method is known as *finite size scaling*, and provides another probe of the critical exponents. By running your experiment at several different grid sizes, you can extract / and compare it with your results from above. You can run a similar analysis for c_V. All of this finite-size analysis is optional, but if you have time it is highly recommended. It should help give you a sense for how what initially appears to be an obstacle (the fact that you have to limit yourself to a finite-size grid) can actually be used as a tool.

The above discussion has been fairly theoretical. We will now move into how to actually simulate the Ising model. If, after reading through that procedure, you still have questions about extracting any of the above quantities, check Appendix B, although be aware that you will likely need to experiment with things yourself. Getting a Monte Carlo simulation of the Ising model running and producing reliable results with good statistics will occupy the bulk of your time at first, and you will likely need to let your program run for an extended period of time in order to collect enough usable data to get reasonable precision on the various quantities you will measure.

[5]This is fairly easy to see. A system with only finitely-many degrees of freedom only has finitely-many states it can possibly be in, and so the partition function must necessarily be analytic. An analytic partition function means that all macroscopic thermodynamic properties (various derivatives of the partition function) must be analytic, and thus a phase transition, characterized by non-analytic behavior in one or more thermodynamic quantities, is impossible.

Summary of Requirements:

This simply restates the requirements for your experiment. All of the following should be included in your report (regardless of whether it is your formal or informal report).

- Compute T_c from at least three different system state variables (specific heat, magnetization, magnetic susceptibility, and correlation length). Each fit should have a precision of $\lesssim 5\%$ and and you should fully explain and justify the statistical analysis that determines the uncertainty in your values.
- Verify power-law scaling behavior of c_V, $|M|$, (beneath T_c), χ, xi, and $\langle\sigma(0)\sigma(x)\rangle$. For the 2-D Ising model, note that c_V actually has logarithmic scaling rather than power-law scaling.
- Compute (with uncertainties) the critical exponents α, β, γ, nu, and η. Compare them to the theoretical values in Table 1.
- Verify that to within your experimental precision the critical exponents that you find satisfy the known scaling relations given in Equations 33, 34, 35, and 36. If you did not find δ you do not need to verify Equation 34.

If you have completed all of the above, consider experimenting with the some of the following:

- Experiment with varying the external applied magnetic field B and compute the critical exponent δ.
- Investigate the finite-size scaling behavior of your system. Extract γ/ν using this method as discussed above.

5 Code and Algorithm

Base code is available for you to get you started on this lab. The purpose of this code is to implement a 2D *Markov Chain Monte Carlo* (MCMC) algorithm and allow to you interface with the code to calculate the desired physical properties listed in the previous section.

The code generates a grid of size $N_{\text{sites}} = N \times N$ lattice locations/sites, where N is an input parameter. This "lattice" is selected with periodic boundary conditions – i.e. the bottom row "wraps around" to be next to the top row, as does the left column to be next to the right column. In effect, this places the lattice on a torus.

For any configuration of spins, function calls can calculate the following from the grid using the nearest neighbor interaction strength (J) and magnetic field strength (B).[6]

[6] Note that the code hard codes the value of J to 1. This is for the simplicity of coding, though in your lab write-up you will need to explain what this does to the units and values of B and T).

1. Average energy per site: H/N_{sites}, where H is calculated by Equation 1.
2. Average spin per site: $\sum_i \sigma_i/N_{\text{sites}}$
3. The spacial spin correlation function: $\langle\sigma(0)\sigma(k)\rangle_T = \text{avg}(R_i(k))$ where R_i is the auto-correlation for any given row/column i, calculated by

$$R_i(k) = \frac{1}{N}\sum_j (\sigma_{(i,j)} - \bar{\sigma}_i)(\sigma_{(i,j+k)} - \bar{\sigma}_i)$$

where $\bar{\sigma}_i$ is the average of all spins in the row/column labelled by i and $j+k$ is understood to be taken to be modulo N. Note that this is only calculated for $j < N/2$ because of the periodic boundary conditions.

The lattice may be evolved using MCMC steps, in which each step does the following:

1. Select n_{s} lattice sites randomly, where $n_{\text{s}} \leq N_{\text{sites}}$.[7]
2. For each selected site i:
   (a) Determine the change in energy to the system if the site is flipped: $\Delta E_i = 2(J\sigma_i(\Sigma_j\sigma_j) + B\sigma_i)$, where j indexes the four adjacent sites.
   (b) If $E_i < 0$, the new state is more probable, so add this site to the list of sites to flip.
   (c) if $E_i > 0$, energy is required to flip the spin. The probability of having this energy is determined by the Boltzmann distribution. Therefore, add it to the list of sites to flip with the probability of $\exp\left[-\Delta E(i)/(k_B T)\right]$.[8]
3. Flip all sites selected to be flipped.

For computational efficiency, a C++ library has been written to do the above. For the sake of convenience, a C wrapper has been written that allows Python to initialize and interact with the above library. Additionally, a Python program has also been written to drive the above library and submit multiple parallel runs of different ising MCMC lattice simulations and cataloging the final grid physical results in output text files.

[7] Note that in the pure MCMC algorithm, only a single site are selected for every step with $n_{\text{s}} = 1$. However, for efficiency, it turns out that it's good enough to pick a small percentage of N_{sites} and flip them. Be careful though. If you flip too few spins per step your simulation will take too long to get good results, and if you flip too many, the state will be too volatile and will not give a good walk throughout the probability distribution. The code default is $n_{\text{s}} \approx N_{\text{sites}} \times 10\%$.

[8] In context of this literature, "chance to flip" means the chance of the probability distribution falling above a random number uniformly selected from the range of 0 to 1. Therefore, the program chooses "flip" if random number $x \in (0,1)$ is less than $\exp(-E_i/T)$.

You are free to use the provided code at your convenience. Typically, your coding for this lab will consist of using Python to generate appropriate run conditions and sample outputs of the lattice at various steps. How you do this is largely up to you. In general, you are advised to start with smaller lattices, and smaller numbers of steps, and move to larger and longer runs when you have good code and are seeking more precision in your results (or perhaps are studying the effects of lattice size and run times on results).

The available code is posted on Github at `https://git.yale.edu/phys382L/ising2D_code`. If you wish, you can download the code to your computer, compile the C++ library and use it locally. The Yale High Performance Computing Group (HPC) has also put the code onto the Grace cluster and will (or has already) generated a group folder for this class, so that you can *ssh* into Grace and work from there. This will allow you to use many nodes concurrently on the Grace cluster and run longer jobs than may be convenient on your own computer. You are referred to the GitHub link for more documentation. Note that there is also a pure python implementation of the Ising model, which you are also welcome to use. It is anticipated that this would be used primarily only when compiling the C++ library is troublesome.

5.1 Practical Introduction to `ising.py`

You are referred to the `github readme` file for a listing of the various code files. This subsection gives a little practical advice on how to get started using the code. A full code listing of `ising.py` is given in Appendix D.

Code Requirements

The `ising.py` requires `python3` with the following modules:

- `numpy`: for various array and matrix functions
- `matplotlib`
- `multiprocessing`
- `scipy`

Running the code and input options

Issue the following command on the command line:

```
python3 ising.py [keyword1]:[value1] [keyword2]:[value2] ... [keywordN]:[valueN]
```

The keywords and values fill a dictionary which controls various options when running the code. In `ising.py` this dictionary is named `inp` (for "inputs") and the defaults are set in the function `set_input`. The input options and defaults are listed in Table 2.

keyword	default value	use
`N`	10	ising lattice size = `N`×`N`
`n_steps`	10000	number of steps in the MCMC algorithm
`n_analyze`	5000	num. steps to calc $\langle E \rangle$, σ_E, $\langle M \rangle$, σ_M
`flip_perc`	0.10	ratio of lattice sites that may flip per step
`plots`	`False`	make plots at end of run
		note: cannot use with `multiprocess`
`use_cpp`	`True`	use the C++ library
`multiprocess`	`False`	use multiple cores concurrently
`skip_prog_print`	`False`	skip printing progress
`print_inp`	`False`	echo `inp` to command line
temperature array options		
note: temperature array is `[t_min,t_max)` with step size of `t_step`		
`t_min`	`1.6`	minimum temperature
`t_max`	`3.6`	maximum temperature
`t_step`	`0.1`	step size between `t_min` and `t_max`
		note: progress can only printed without `multiprocess`
options used in annealing		
`t_top`	4.0	tempature from which to anneal
`n_burnin`	2000	steps after annearling prior to `n_analyze`
`B`	0	constant mag. field
options used for result printout		
`dir_out`	`data`	name of output directory
`date_output`	`False`	prepend output dir with `YYYMMDD-HHMMSS`
`file_prefix`	empty string	prefix for output files
`t_min`	`1.6`	minimum temperature

GRAMMAR FOR INPUT VALUES

value	command-line input	notes
`True`	`t` or `true`	case insensitive
`False`	`f` or `false`	case insensitive
integer	integer	
float	decimal number	if there's no decimal point, defaults to integer
string	string	default value if none of above

Table 2: Default command line options for `ising.py`

Functions in `ising.py` to Modify

- `make_B_generator(inp, T_final):`

  This function makes a python generator that returns a value for the magnetic field each time it is called. That generator's control flows from the top to the first `yield`. At that statement, it returns the `yield` value and pauses until it is called again with the `next` method. At that point it continues from the `yield` statement until the next `yield` statement, and so forth. It is called once per step of the Ising Lattice. Therefore, you should use this function to modify the evolution of the magnetic field through all `inp['n_steps']` of the simulation.

- `make_T_generator(inp, T_final):`

  This function is completely analogous to `make_B_generator`. Modify it to determine the temperature annealing.

- `set_input(cmd_line_args):`

  Modify this function if you wish to add new input parameters that can be modified from the command line. You may find it equally convenient to simply modify the code without new input parameters.

- `run_ising_lattice(inp, T_final, skip_print=False):`

  This is where the actual body of the ising simulation is run. See in particular lines 194 to 222. It is likely that you may want to use some more sophisticated sampling than just averaging the final `n_steps` values at the end of a run for determining some parameters/exponents.

- `plot_graphs(data):`

  You will most likely not modify this code. These are *very* simplistic plots just to get you started. You will probably make your plots from the output csv (comma separate variable) files. This has been successfully done with many different software packages, such as Matplotlib, MATLAB, and Excel, to name just a few.

- `print_results(inp, data, corr):`

  You probably won't modify this code, except if you want to update the overly simplistic spin correlation output (the `corr` in the function) to be values averaged over many steps.

- **remaining functions:**

  You are welcome, of course, to modify or copy any of the remaining code. This section is just to point out where you will likely spend your first efforts. It is anticipated that you can successfully ignore the remaining code. If this proves wrong, and you are feeling particularly helpful, you can also make pull requests to the GitHub (https://git.yale.edu/phys382L/ising2D_code) repository to improve the code.

Where the code could improve

There are two major points in which the code could be improved. You are welcome to take a stab at either.

1. Implement a different algorithmic step for the Ising lattice that will be more resilient against critical slowing. As you will find out in this lab, the parameters you calculate are meaningful at the critical temperature T_c. However, as you approach the T_c, the evolution of the Ising Lattice slows down, which is known as "critical slowing down" [6]. There are some promising algorithms, such as the Swendsen-Wang and Wolff algorithms, which could improve results near the critical point. See [7].

2. Much simpler than the first point, make the code calculate the spin correlations for many steps and average them for the final values, instead of taking the value at the single final step alone.

6 Errata and Errata Reporting

The base files for this document are posted online at https://git.yale.edu/phys382L/ising2D_handout. Feedback under the "Issues" and, from perhaps more proactive users, "Pull request" sections of the repository constitute the errata for this work. You are invited to provide constructive feedback to the same. For code errors and improvements, please submit likewise to the https://git.yale.edu/phys382L/ising2D_code repository.

References

[1] http://www.scholarpedia.org/article/Ising_model:_exact_results

[2] D. J. C. MacKay, "Introduction to Monte Carlo Methods", Kluwer Academic Press (1998). Canvas link

[3] R. K. Pathria and P. D. Beale, "Statistical Mechanics" 3rd ed., Elsevier (2011).

[4] D. J. C. MacKay, "Information Theory, Inference, and Learning Algorithms", Cambridge Univ. Press (2003).

[5] T. Paine (1776).

[6] P.C. Hohenberg and B.I. Halperin, Rev. Mod. Phys. 49 (1977) 435.

[7] R. H. Swendsen and J.-S. Wang, "Cluster Monte Carlo Algorithms", Physica A 167 (1990) 565-579. science direct link

Appendix A: Tips for Convergence

As discussed in Section 3, it is important that the system converges at a given temperature before you start using that data to compute thermodynamic averages. Above T_c, convergence is easy to obtain, and a few thousand burn-in steps should be more than enough to ensure it. At low temperatures, however, the system has far less energy available to move around in the probability landscape, and so its easy for it to get stuck in a local minimum. This will often manifest itself as the development of large magnetic domains, macroscopic regions of your lattice that all have the same spin, but that do not have the same spin as their surroundings. Interestingly, normal magnetic materials have several different magnetic domains on a mesoscopic scale, but we want to avoid them in this experiment. What follows are a few different suggestions of ways to help improve convergence at low temperatures, ranging from the extremely brute force to the slightly-more-clever.

1. Its often very easy to check whether your system has converged or not by looking at its magnetization. At low temperatures we expect that all the spins should be roughly aligned, but if there are macroscopic domains present, then the average magnetization will be much closer to zero. A telltale sign for non-convergence is thus an absolute value for average magnetization that is much lower than the data at nearby temperatures would suggest. One very brute force option is thus to test whether your system has converged after each temperature run, and simply rerun it at that temperature if it hasnt. For a reasonably large range of parameters, this works alright because the chances of getting stuck in a local minimum of the Gibbs distribution and failing to converge isnt too high, so you may only need to rerun a few runs. Still, at very low temperatures this becomes extremely time-intensive.
2. Changing the percentage of accepted flips that you make each step can sometimes help achieve convergence. If the percentage is too low, the MCMC can only make very small steps in the Gibbs distribution, and so it takes a while for it to get out of a local minimum. If the percentage is too high, each step will be far too large and the system wont be able to park itself at the global minimum once it reaches it.
3. Most of the problems obtaining convergence stem from the fact that when you randomly initialize the spin matrix you can initialize it in a local minimum of the Gibbs distribution and at low temperatures the system doesnt have enough

energy available to get out of this initial area its stuck in. One way to combat this is to start the system at a much higher temperature than the one you are trying to measure it at. By starting it at a temperature above the critical point and then cooling it slowly down to the desired temperature you let it have more energy in the beginning to explore the whole probability landscape before slowly finding and settling into the global minimum. This technique is known as *simulated annealing.*

   If you are implementing this, be careful not to bias the system! Once it has reached the desired temperature you should let it sit there for several thousand additional burn-in steps before collecting data so you can be sure it isnt still suffering from any of the effects of starting it at a higher temperature.

4. Since macroscopic magnetic domains are a significant problem when trying to obtain convergence, consider biasing your system from the start so one spin (either up or down) is favored, thus discouraging any domains of the other spin from forming. This can be done by subjecting your system to a small external magnetic field. Note though that applying a magnetic field changes the critical temperature and messes with other system state variables, so you should probably do something similar to the simulated annealing technique discussed above: Start your system with a small external magnetic field which you slowly cool to zero (or your actual desired magnetic field if youre doing a run at nonzero field). Then, let the system burn in for several thousand steps before starting to collect data.

This is by no means an exhaustive list, but should give you enough ideas to start attacking any convergence problems you might have.

Appendix B: Computational Tricks

Log Plots

Many of the relationships you expect to find throughout this experiment are of the form $y = ax^b$ where a and b are unknown constants you wish to determine. Fitting to a function of this form is hard (and makes error analysis a pain), so convert it to a linear fit! Note that lny = lna + blnx, so plotting lny vs lnx turns the annoying power-law fit into a trivial linear fit, and has the added benefit of making your error analysis much easier.

Selecting the right range of data

The various critical exponent relationships summarized in Table 1 only hold near the critical temperature, so when fitting your simulated data to these relationships you should ignore data that is far from the critical temperature. Furthermore, data in the immediate vicinity of the critical temperature will suffer from incredibly large errors and potential finite size effects, so you should exclude a small window of data around the critical temperature when doing all data fits.

Finding γ

Magnetic susceptibility typically suffers from very large errors near the critical temperature, making γ one of the harder exponents to get a good lock on. If youre suffering from this problem, try the following:

First recall the definition of magnetic susceptibility, $\chi = \frac{\partial M}{\partial B}$, so if we apply a very B small magnetic field (above the critical temperature where the average magnetization would otherwise be zero), we will in general have:

$$M = \chi B \propto |t|^{\gamma} B \propto |t|^{-\gamma} \tag{38}$$

This means that in order to compute γ we can apply a small magnetic field and sweep the temperature above the critical temperature. We can then measure magnetization and fit it to a power-law. Since errors in magnetization are typically smaller than errors in χ, this can often help tame errors in γ.

If you are doing this, be sure that the magnetic field you apply isnt too large. Applying a magnetic field actually takes you away from the critical point, meaning even in the $N \to \infty$ limit there is no divergence in any of the thermodynamic quantities. As long as the field is small this shouldn't give you any large errors in γ, but be careful not to stray into the large-B regime, and dont be too concerned if the critical temperature appears to be shifting around once you apply a magnetic field.

Appendix C: Figures from Sample Student Data

What follows are a few graphs of sample student data so that you can get a sense for what your results may look like.

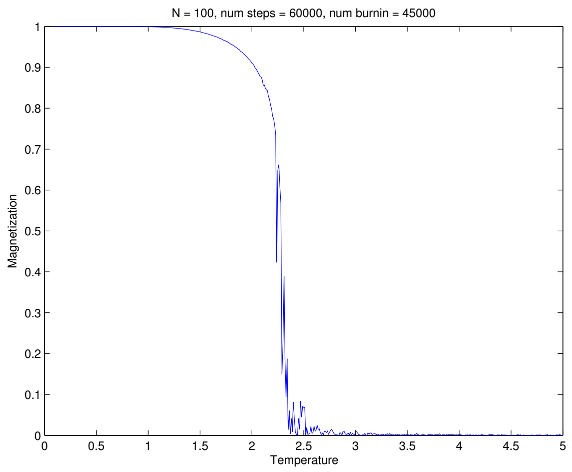

Figure 1: Average magnetization per site vs. temperature for $N = 100$. Below T_c, the magnetization spontaneously achieves a nonzero value, while above T_c it is approximately zero (although fluctuations are much larger near T_c than they are far above it.

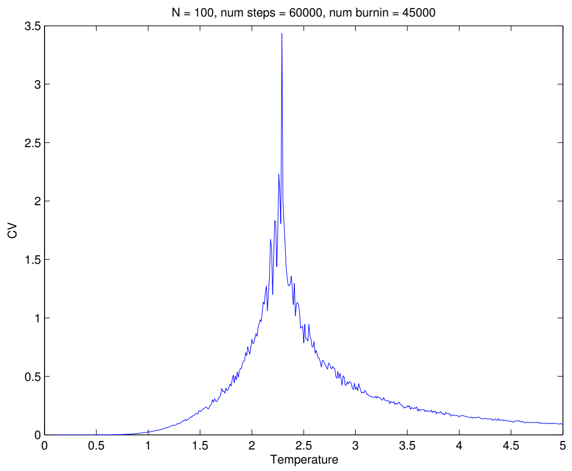

Figure 2: Specific heat vs. temperature for $N = 100$. A divergence near the critical temperature is clearly visible.

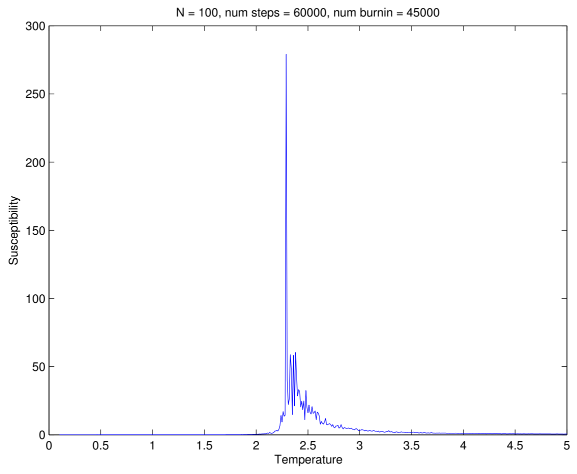

Figure 3: Magnetic susceptibility vs. temperature for $N = 100$. A divergence near the critical temperature is clearly visible.

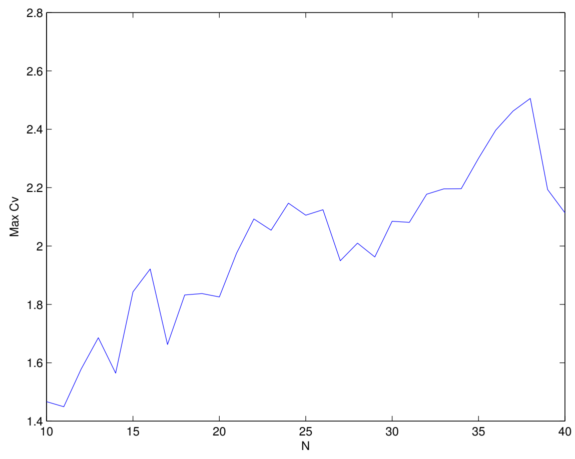

Figure 4: The maximum value of c_V obtained for a variety of grid sizes. Finite size effects are visible as the clear increase of maximum c_V with the size of the grid.

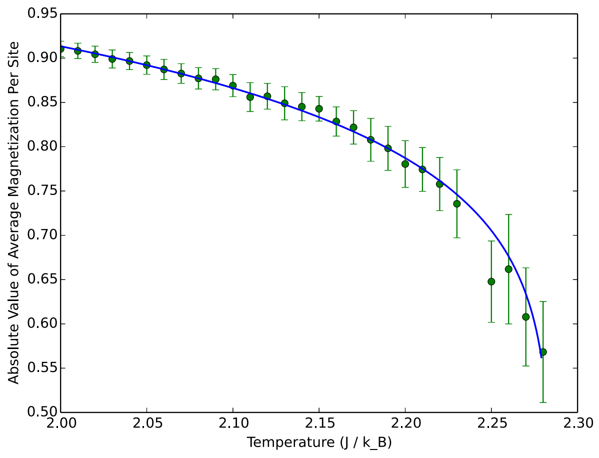

Figure 5: Magnetization vs. temperature for $N = 100$ in a region near the critical temperature. Error bars and a power-law fit are shown; the latter can be used to determine β.

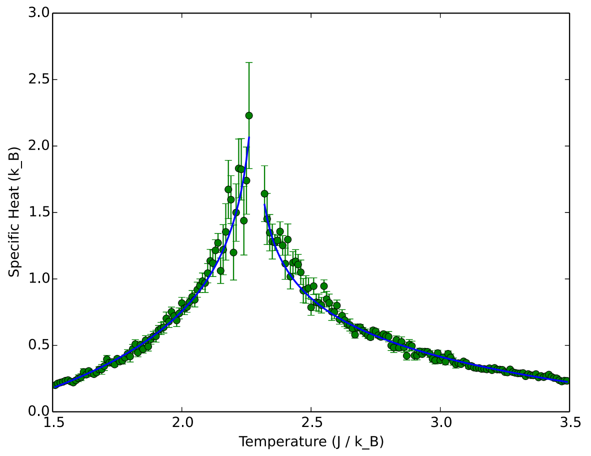

Figure 6: Specific heat vs. temperature for $N = 100$. Error bars and two separate log fits are shown (one each for above and below the critical temperature) to confirm $\alpha = 0$. Note that a window of data near the critical temperature has been excluded.

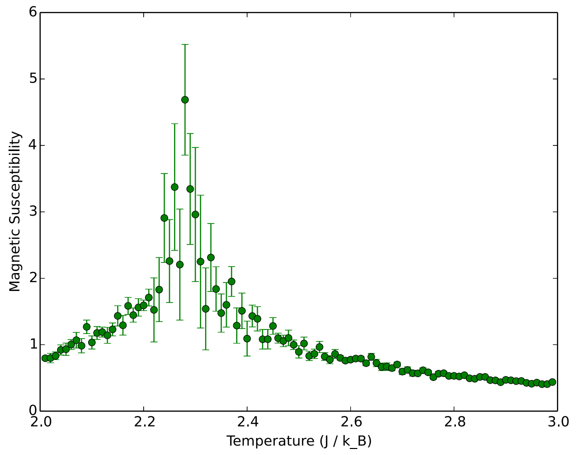

Figure 7: Magnetic susceptibility vs. temperature for $N = 100$. Error bars are shown, but the relatively large errors near T_c make it difficult to obtain γ from this data.

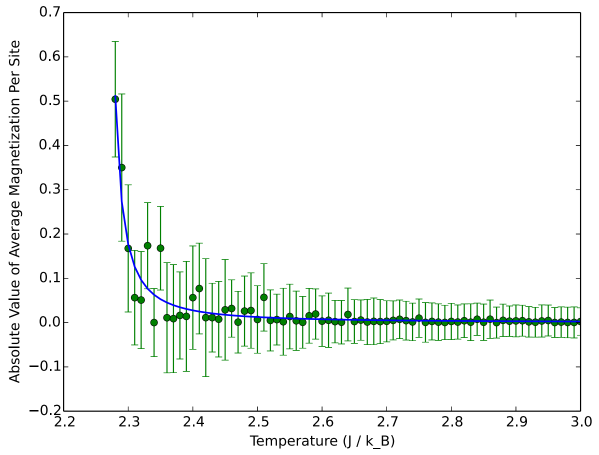

Figure 8: Magnetization vs. temperature for $N = 100$ with an applied field of $B = 0.1$ (in units of J). Error bars are shown, and the fit here was used to obtain γ.

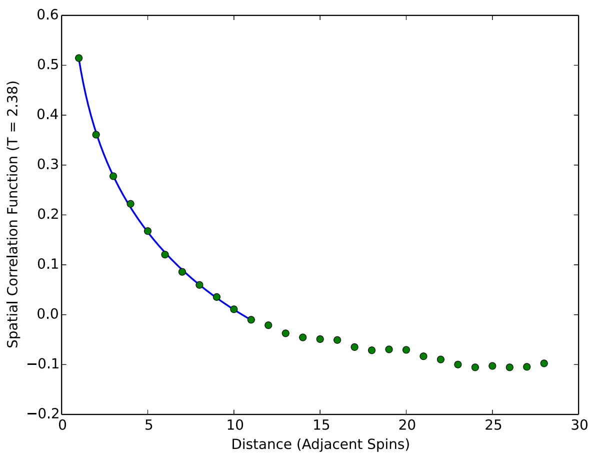

Figure 9: Spatial spin correlation function vs. distance (in units of the lattice spacing) at $T = 2.38$. A fit to the first 11 data points gives a good estimate for ξ at this temperature, as well as an estimate for ν.

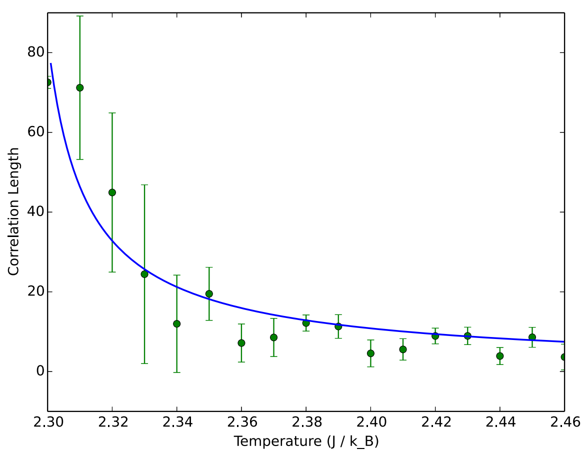

Figure 10: Correlation length ξ vs. temperature. Error bars are shown, and the fit here was used to obtain ν.

Appendix D: Code Listing of `ising.py`

```python
import sys
import os
import time
import csv
import logging
import numpy as np
import matplotlib.pyplot as plt
import multiprocessing as mp
from sys import exit, argv
from IsingLattice_python import IsingLattice as IsingLattice_py

# conditionally import python and C++ version of IsingLattice
has_cpp = False
if os.path.isfile('ising_lattice_lib.so'):
    has_cpp = True
    from IsingLattice_cpp    import IsingLattice as IsingLattice_cpp

## main program
def make_B_generator(inp, t_final=None):
    """Return a generator that makes values of B (magetic field in each step)
    note: you *should* play with this function

    default implementation: always return 0
    """
    for val in range(inp['n_steps']):
        yield inp['B']

def make_T_generator(inp, t_final):
    """Return a generator that makes values of T (temperature in each step)
    note: you *should* play with this function

    default implementation:
        start at T=t_top, and linearly decrease temperature to t_final
        hold at t_final for n_burnin
        hold at t_final for n_analyze
    """
    n_slope = inp['n_steps'] - inp['n_burnin'] - inp['n_analyze']
    if n_slope < 0:
        print('fatal␣error:␣n_steps␣-␣n_burnin␣-␣n_slope␣<␣0')
        print('terminating␣program')
        exit(2)

    # get linearly decreasing values from t_top to t_final
    for val in np.linspace(start=inp['t_top'], stop=t_final, num=n_slope):
        yield val

    for val in range(inp['n_burnin']):
        yield t_final

    for val in range(inp['n_analyze']):
        yield t_final

def set_input(cmd_line_args):
    """Parse command-line parameters into an input dictionary.
    provide default values

    note: feel free to add your own input parameters here,
          and them use them by calling them from the dictionary
          names 'inp' in the modules where they would prove useful

    input:   sys.argv
             use syntax of keyword:value on command line
    return:  dict[key] = value
    note:    Any value which can be turned into a number will be a float
             if it has a '.', otherwise it will be a float.

    """

    inp = dict()
    inp['t_min']       = 1.6     # minimum temperature
    inp['t_max']       = 3.6     # maximum temperature
    inp['t_step']      = 0.1     # step size from min to max temperature
    inp['t_top']       = 4.0     # start temperature (arbitrary; feel free to change)
    inp['N']           = 10      # sqrt(lattice size) (i.e. lattice = N^2 points
    inp['n_steps']     = 10000   # number of lattice steps in simulation
    inp['n_burnin']    = 2000    # optional parameter, used as naive default
    inp['n_analyze']   = 5000    # number of lattice steps at end of simulation calculated for averages and std.dev.
    # inp['J']           = 1.0     # **great** default value -- spin-spin interaction strength
    inp['B']           = 0.0     # magnetic field strength
    inp['flip_perc']   = 0.1     # ratio of sites examined to flip in each step
    inp['dir_out']     = 'data'  # output directory for file output
    inp['plots']       = False   # whether or not plots are generated

    inp['print_inp']   = False   # temperature option
    inp['use_cpp']     = True    # use 1 for True and 0 for False

    inp['date_output'] = False
    inp['file_prefix'] = ''
    inp['multiprocess'] = False
    inp['skip_prog_print'] = False

    for x in cmd_line_args[1:]:
        if ':' in x:
            try:
                key, val = x.split(':')
                try:
                    if '.' in val:
                        inp[key] = float(val)
                        print('%-20s'%('inp["%s"]'%key),'set to float  ',inp[key])
                    elif val.lower() == 'false' or val.lower() == 'f':
                        inp[key] = False
                    elif val.lower() == 'true' or val.lower() == 't':
                        inp[key] = True
                    else:
                        inp[key] = int(val)
                        print('%-20s'%('inp["%s"]'%key),'set to int    ',inp[key])
                except:
                    inp[key] = val
                    print('%-20s'%('inp["%s"]'%key),'set to string ',inp[key])
            except:
                print('warning: input "%s" not added to arguments'%x)
        else:
            print('ignoring command line input: %s'%x)

    if inp['print_inp']:
        print('Printed list of input keys:')
        for key in sorted(inp.keys()):
            print('%-20s'%key,' ',inp[key])
    return inp

class check_progress(object):
    """A class that will print a simple status bar of percentage of work done"""
    def __init__(self, inp, T, skip_progress=False):
        self.skip_print = skip_progress
        if self.skip_print:
            return

        self.start_time = time.time()
        self.n_steps = inp['n_steps']
        self.n_check = 1000
        if 'check_per_steps' in inp:
            self.n_check = inp['check_per_steps']
        self.fmt_print = ('%ix%i (T=%.2f) steps: %%7i/%7i, %%5.1f%%%%  '
                'run time: %%8s  est.time-to-go: %%8s'%
                (inp['N'], inp['N'], T, self.n_steps))
```

```python
        self.n_called = -1 # will progress untill n_called = n_steps
        self.check()
        # print(self.fmt_print)

        # if False:
        #     print('%ix%i IsingLattice. Finished %10i / %10i steps (%4.1f%%). '
        #          'Started '+str(time.strftime('%m-%d %H:%M:%S')))
            # dir_out += str(time.strftime("_%Y%m%d-%H%M%S"))
    def check(self, final=False):
        if self.skip_print:
            return

        self.n_called += 1

        if not self.n_called % 1000 == 0 and not final:
            return
        ratio = float(self.n_called) / self.n_steps
        if ratio == 0:
            time_pass_str = '00:00:00'
            est_str = 'n/a'
        else:
            time_pass = time.time() - self.start_time
            time_pass_str = time.strftime('%H:%M:%S', time.gmtime(time_pass));
            est_time  = (1-ratio)*time_pass/ratio
            # print('est_time ', est_time)
            est_str = time.strftime('%H:%M:%S', time.gmtime(est_time))
        if final:
            print(self.fmt_print%(self.n_called, ratio*100., time_pass_str, 'done!'))
        else:
            print(self.fmt_print%(self.n_called, ratio*100., time_pass_str, est_str), end='\r')

def run_ising_lattice(inp, T_final, skip_print=False):
    '''Run a 2-D Ising model on a lattice.
    Return three objects (each a numpy.array, which you can treat
    as identical to a numpy.array)
        M_avg:       the average magnetization of each site for each n_analyze step
        E_avg:         '........' energy '..................................'
        correlation: an (N/2-1) array of the correlation function values at the final data frame

        Note that this will use the generators from functions:
        (1) make_T_generator
        (2) make_B_generator
    '''

    time_start = time.time()

    lattice = None
    if inp['use_cpp'] and has_cpp:
        lattice = IsingLattice_cpp(inp['N'], inp['flip_perc'])
    elif inp['use_cpp'] and not has_cpp:
        print('Warning:␣although␣use_cpp␣is␣set␣to␣1,␣'
        '␣the␣shared␣library␣IsingLattice.so␣is␣not␣present.\n␣'
        '␣Therefore,␣the␣python␣implementation␣of␣Ising␣will␣be␣used.')
        lattice = IsingLattice_py(inp['N'],inp['flip_perc'])
    else:
        lattice = IsingLattice_py(inp['N'],inp['flip_perc'])

    # Make the run loop
    try: # try loop that can be interrupted by the user
        #first loop through all steps up to n_analyze
        T_generator = make_T_generator(inp, T_final)
        B_generator = make_B_generator(inp, T_final)
        n_prior = inp['n_steps'] - inp['n_analyze']

        progress = check_progress(inp, T_final, skip_print)

        for T, B, step in zip(T_generator, B_generator, range(n_prior)):
            lattice.step(T,B)
            progress.check()

        # loop through the analyze section of generators
        E_avg = []
        M_avg = []
        for T, B, step in zip(T_generator, B_generator, range(inp['n_analyze'])):
            lattice.step(T,B)
            E_avg.append(lattice.get_E())
            M_avg.append(lattice.get_M())
            progress.check()
        progress.check(True)
        spin_correlation = np.array(lattice.calc_auto_correlation())

        lattice.free_memory()
        return (
            np.array(E_avg),
            np.array(M_avg),
            np.array(spin_correlation)
        )

    except KeyboardInterrupt:
        try:
            lattice.free_memory()
        except:
            pass
        print("\n\nProgram␣terminated␣by␣keyboard.␣Good␣Bye!")
        sys.exit()

def plot_graphs(data): #T,E_mean,E_std,M_mean,M_std): #plot graphs at end
    dat = np.array(data)
    # print('data: ', dat)
    # print('x: ', dat[:,1])
    # x = dat[:,1][0]
    # print('xlist: ', x)

    plt.figure(1)
    # plt.ylim(0,1)
    plt.errorbar(dat[:,0], dat[:,1], yerr=dat[:,2], fmt='o')
    plt.xlabel('Temperature')
    plt.ylabel('Average␣Site␣Energy')
    plt.figure(2)
    plt.errorbar(dat[:,0], np.absolute(dat[:,3]), yerr=dat[:,4], uplims=True, lolims=True,fmt='o')
    plt.xlabel('Temperature')
    plt.ylabel('Aveage␣Site␣Magnetization')
    plt.show()

def get_filenames(inp): #make data folder if doesn't exist, then specify filename
    '''Generate the output file names for the EM (energy and megnetism) and SC (spin correlation) files'''
    try:
        dir_out = inp['dir_out']
        prefix  = inp['file_prefix']
        if inp['date_output']:
            dir_out += str(time.strftime("_%Y%m%d-%H%M%S"))

        if not os.path.isdir(dir_out):
            os.makedirs(dir_out)

        # file name = [file_prefix]##_EM_v#.csv if only one temperature (example: runA_4.20_EM_v0.csv)
        #             [file_prefix]##T##_EM_v#.csv if there are two temperatures (example: runA_4.2T5.3_EM_v0.csv)
        # the other file name is identical, but with "SC" (for spin correlation)) instead of EM
        if inp['t_max'] <= inp['t_min']:
            t_name = '%.2f'%inp['t_min']
        else:
            t_name = '%.2fT%.2f'%(inp['t_min'],inp['t_max'])

        # print('%s%s_SC_v%i.csv'%(prefix,t_name,v))
        v = 0
        while (os.path.isfile( os.path.join(dir_out, '%s%s_EM_v%i.csv'%(prefix,t_name,v))) or
```

```python
                os.path.isfile( os.path.join(dir_out, '%s%s_SC_v%i.csv'%(prefix,t_name,v)))):
            v += 1

        return ( os.path.join(dir_out, '%s%s_EM_v%i.csv'%(prefix,t_name,v)),
                 os.path.join(dir_out, '%s%s_SC_v%i.csv'%(prefix,t_name,v)) )

    except:
        print ('fatal: Failed to make output file names')
        sys.exit()

def print_results(inp, data, corr):
    data_filename, corr_filename = get_filenames(inp)
    with open(data_filename,'w') as f_out:
        writer = csv.writer(f_out, delimiter=',', lineterminator='\n')
        writer.writerow(['N', 'n_steps', 'n_analyze', 'flip_perc'])
        writer.writerow([inp['N'], inp['n_steps'], inp['n_analyze'], inp['flip_perc']])
        writer.writerow([])
        writer.writerow(['Temp','E_mean','E_std','M_mean','M_std'])
        for entry in data:
            writer.writerow(entry)
        # for t, e_mean, e_std, m_mean, m_std in zip(T, E_mean, E_std, M_mean, M_std):
        #     writer.writerow([t, e_mean, e_std, m_mean, m_std])

    with open(corr_filename,'w') as f_out:
        writer = csv.writer(f_out, delimiter=',', lineterminator='\n')
        writer.writerow(['N', 'n_steps', 'n_analyze', 'flip_perc'])
        writer.writerow([inp['N'], inp['n_steps'], inp['n_analyze'], inp['flip_perc']])
        writer.writerow([])
        writer.writerow(['Temp']+['d=%i'%i for i in range(1,len(corr[0])+1)])
        for entry in corr:
            writer.writerow(entry)


def run_indexed_process( inp, T, data_listener):
# def run_simulation(
#         temp, n, num_steps, num_burnin, num_analysis, flip_prop, j, b, data_filename, corr_filename, data_listener, corr_listener):
    print("Starting Temp {0}".format(round(T,3)))
    try:
        E, M, C = run_ising_lattice(inp, T, skip_print=True)
        data_listener.put(([T,E.mean(),E.std(), M.mean(), M.std()], [T,]+[x[1] for x in C]))
        # corr_listener.put([T,]+[x[1] for x in C])
        print("Finished Temp {0}".format(round(T,3)))
        return True

    except KeyboardInterrupt:
        print("\n\nProgram Terminated. Good Bye!")
        data_listener.put('kill')
        # corr_listener.put('kill')
        sys.exit()

    except:
        logging.error("Temp="+str(round(T,3))+": Simulation Failed. No Data Written")
        return False

def listener(queue, inp, data):
    '''listen for messages on the queue
    appends messages to data'''
    # f = open(fn, 'a')
    # writer = csv.writer(f, delimiter=',', lineterminator='\n')
    while True:
        message = queue.get()
        # print('message: ', message)
        if message == 'kill':
            data['data'].sort()
            data['corr'].sort()
            print_results(inp, data['data'], data['corr'])
            print ('Closing listener')
            # print('killing')
            break
        data['data'].append(message[0])
        data['corr'].append(message[1])
        # print('--------\n',data)

def make_T_array(inp):
    if inp['t_max'] <= inp['t_min']:
        return [inp['t_min'],]
    else:
        return np.arange(inp['t_min'], inp['t_max'], inp['t_step'])


def run_multi_core(inp):
    print("\n2D Ising Model Simulation; multi-core\n")
    T_array = make_T_array(inp)

    #must use Manager queue here, or will not work
    manager = mp.Manager()
    data_listener = manager.Queue()
    # corr_listener = manager.Queue()
    pool = mp.Pool(mp.cpu_count())


    # arrays of results:
```

```python
    data = {'data':[], 'corr':[]}
    # corr = []

    #put listener to work first
    data_watcher = pool.apply_async(listener, args=(data_listener, inp, data,))
    # corr_watcher = pool.apply_async(listener, args=(corr_listener, inp, corr,))

    #fire off workers
    jobs = [pool.apply_async(run_indexed_process,args=(inp,T, data_listener)) for T in T_array]

    # collect results from the workers through the pool result queue
    [job.get() for job in jobs]
    data_listener.put('kill')
    pool.close()

def run_single_core(inp):
    print("\n2D Ising Model Simulation; single core\n")
    # sequentially run through the desired temperatures and collect the output for each temperature
    data = []
    corr = []
    for temp in make_T_array(inp):
        E, M, C = run_ising_lattice(inp, temp, skip_print=inp['skip_prog_print'])
        data.append( (temp, E.mean(), E.std(), M.mean(), M.std() ) )
        corr.append([temp,]+[x[1] for x in C])

    print_results(inp, data, corr)

    if inp['plots']:
        plot_graphs(data)


if __name__ == "__main__":
    """Main program: run Ising Lattice here"""
    inp = set_input(argv)
    if inp['multiprocess']:
        run_multi_core(inp)
    if not inp['multiprocess']:
        run_single_core(inp)
```