The Ising Model of a Ferromagnet

Part II Computational Project

Easter 2016

Abstract

A computational solution to the two dimensional Ising model was created, in Python, using the Metropolis algorithm. The simulations were found to be consistent with known behaviours including: A sharp decrease in the mean magnetic moment when the temperature approaches $2/ln(1+\sqrt(2))$K; lack of stable non-zero mean magnetic moment above the same temperature, and stable non-zero moments beneath it; and reduction in domain (regions of like spin) size with temperature. Which are all evidence of the predicted ferromagnetic phase transition. In addition, the energy and heat capacity as functions of temperature were simulated, and together confirmed that the phase transition is one of second order. The latter was also used to estimate the critical temperature of the phase transition as 2.26 ± 0.01K, which is consistent with the analytical result. Finally, the effect of an applied magnetic field was examined. Ferromagnetic hysteresis was observed below the critical temperature, and paramagnetism above it. The paramagnetic susceptibility was a decreasing function of temperature, and generally a non-linear function of the applied field.

1 Introduction

By considering a ferromagnet as an array of mutually interacting spins, a successful model can be built up and studied. If we further restrict the interactions to nearest neighbouring spins only, we arrive at the Ising model. In two or more dimensions, this is one of the simplest systems which exhibits a phase transition [1]. The Ising Model in two dimensions has been widely studied, and is even analyticity tractable in the limit of an infinite number of spins [2,3]. Its solutions, both analytical and numerical, have advanced the physicist's understanding not only of magnetic systems, but of many other systems which exhibit phase transitions [1,3]. In addition, the development of computational methods to solve the problem have resulted in powerful advances in Monte-Carlo Simulation techniques [4].

It is the aim of this investigation to: produce and test a computational approach, based on the Metropolis algorithm [1,4,5] implemented in Python (with NumPy, and SciPy); and accurately examine the phase transition and variation in important physical quantities (with respect to multiple changing parameters).

A brief examination of the theory used, and account of the general computational methods can be found in section 2. Results from the calculations, with discussion, are

presented in section 3, and summary conclusions in section 4. References to various texts are in section 5, and a full code listing in section 6.

2 Background & Method

2.1 Definition in Two Dimensions

Consider a square lattice which has L identical sites, with no vacancies, vertically and horizontally. Each site contains a spin state, S, which can be either -1 or 1, and each have magnetic moment μ. If each spin interacts only with it's four nearest neighbours, and the interaction energy is J, the system has energy:

$$E = -J \sum_{<i,j>} S_i S_j - \mu H \sum_{i}^{L^2} S_i. \tag{1}$$

Where H is the magnetic field applied to the lattice, and $< i, j >$ denotes nearest neighbour summation. In our calculations, periodic boundary conditions have been implemented. A complete theoretical discussion is beyond the scope of the report. However a few important results [2,3] we shall use are stated below:

$$T_O = \frac{2}{\ln(1+\sqrt{2})} \approx 2.269, \tag{2}$$

T_O is the critical temperature (generally T_c) for an infinite lattice [1], below which a spontaneously magnetised lattice (i.e. net magnetic moment) is stable. The magnetic moment per site as a function of temperature, T, is given by:

$$M = \begin{cases} (1 - \sinh^{-4}(\frac{2J}{k_b T}))^{\frac{1}{8}} & T < T_O, \\ 0 & T \geq T_O. \end{cases} \tag{3}$$

k_b is Boltzmann's constant. In addition we shall use the fluctuation-dissipation theorem [1,5] to calculate the heat capacity:

$$C = \frac{\sigma_E^2}{k_b T^2}, \tag{4}$$

σ_E^2 is the variance of the energy history of a given lattice. It can also be shown that the infinite lattice exhibits a singularity in the heat capacity (as a function of temperature, i.e. it is a second order phase transition) at T_O [3]. Thus, if we can find $C(T)$, we can estimate T_c by finding the maximal heat capacity and its corresponding temperature. In a similar way, the magnetic moment per site drops sharply around the critical temperature, and this can also be used to estimate T_c. Some difficulty lies in choosing a threshold beneath which the first T is counted. Ideally it would be zero, but fluctuations around the transition prevent this from being possible practically.

2.2 The Metropolis Algorithm

Since the number of possible microstates scales as 2^{L^2} [1] explicit calculation of such is highly impractical. However, it is possible to sample the microstates by perturbing them as follows [4]. Given an initial lattice, we can find another microstate by:

1. Pick a lattice site i, with spin S_i.
2. If we flip the spin, the change in energy is $\Delta E = 2S_i(\sum_{<j>} S_j - \mu H)$. Where $< j >$ is the sum over of the 4 nearest spins.
3. (a) If $\Delta E < 0$, the process releases energy; the flipped state is thermodynamically favourable and remains flipped. Return to 1.
   (b) If $\Delta E > 0$, the process requires energy; A thermal fluctuation is simulated.
4. A (uniform) random number is generated: $0 \leq p < 1$
   (a) If $p \leq exp(-\frac{\Delta E}{K_b T})$, the spin is flipped. Since any p with this condition will correspond to an energy $\geq \Delta E$; the thermal fluctuation is large enough to provide the energy to move to a locally unstable state.
   (b) If $p > exp(-\frac{\Delta E}{K_b T})$, the spin is unchanged; the thermal fluctuation is not large enough to perturb the system.
5. Return to 1.

Only the ΔEs need be known for the algorithm to be successful, and this reduces the necessary computation time considerably. This relies on the principle of detailed balance; when in equilibrium, the transition rates, R, from some state A to B and vice versa must obey:

$$P_A R_{AB} = P_B R_{BA}, \tag{5}$$

Where P is the probability of the subscripted state occurring [4]. In our case, we have a canonical ensemble of spins, and thus the microstate probabilities are proportional to $exp(-\frac{E}{K_b T})$, where E is the energy of a given microstate. So by equation (5) $R_{AB} \propto exp(-\frac{\Delta E}{K_b T})$, $\Delta E = E_B - E_A$. The constant of proportionality (i.e. the reverse transition rate) is defined as unity for ease of calculation, step 4 encodes this principle [4]. The implementation of this can be found in section 0 of the code (specifically the $.spin\_flip$ method).

By repeating this over many "time steps" (one time step is L^2 repetitions of the algorithm), we can create a history of the system statistics from the sampled microstates (after each time step). Various physical quantities can then be estimated from the history. However, care must be taken to ensure that the system has reached its equilibrium state (or fluctuations about such) throughout the entire (recorded) history.

2.3 Checkerboard Decomposition

The speed of the algorithm can be improved by vectorisation of the operations. A matrix is used to store the spin states. The interaction energies (i.e. the first term in (1), but for a given site) can then be calculated by simply permuting the matrix rows and columns using the *roll* function in NumPy, see the *.update_interaction_energy* method. This neatly includes the periodic boundary conditions also. In this way, a matrix of the interaction energies is created for use in steps 3 and 4. The contribution of the applied field can also be included by matrix addition.

Since we are only considering nearest neighbour interactions, we can decompose the lattice matrix into alternating sites (like the black and white squares of a chess or draughts board). This is a specific example of "checkerboard decomposition", i.e. splitting a matrix into sub-matrices which have independent calculations [4]. Each "black" ("white") site will only ever interact with its "white" ("black") neighbours; all sites of one colour can be passed through the algorithm, before the energy matrix calculations need to be repeated. In fact all of one colour could even be done at the same time, allowing for threading/parallelisation, but this has not been implemented at this time. Once one colour is complete, the matrices are recalculated and the other colour is passed (*.lattice_flip_checkerboard* method, section 0).

2.4 Performance of the Code

A rudimentary object orientated approach was taken, mainly for the ease of creating multiple lattices with different parameters. While this makes the code a little more complex than a functional approach, it is not expected to affect performance too negatively. Only one class is defined and its methods are equivalent to their functional counterparts. Wherever possible, the code was vectorised by using NumPy functions designed to process arrays. This offers the fastest time and scalability without the use of threading/parallelisation, or change in algorithm.

Since each "time step" consists of L^2 spins being passed through the algorithm, we could expect that the computation time, τ, scales as $\mathcal{O}(L^2)$. If the simulation is run for F steps, then we would expect $\tau = \mathcal{O}(FL^2)$. So, to lowest order, we have:

$$\tau = AFL^2, \tag{6}$$

Where A is a system specific constant. It was found that τ did scale to lowest order as given by (6), up to at least 10000 steps (section 1 in the code). On a 3.2GHz core (100% load) a 50 by 50 lattice running for 10000 steps took about 132s. This gives us $A = 5.3\mu s$, so we would expect A to have an order of $1 - 10\mu s$ for a consumer grade desktop computer.

During run time it was observed that the memory usage of the Python process did not exceed $200MB$ (although sections were run individually), and was usually an order of magnitude less than this. A rough timing for each section in the code is presented in the "print()" statements within them.

2.5 A Note on Units

Equation (1) contains three unfixed parameters: J, μ, and H. By normalising the physical quantities using these parameters, the calculations are kept as general as possible. Dimensionless temperature,

$$T_n = T\frac{k_b}{J}, \tag{7}$$

T is the SI temperature. In the program we have set $J = k_b \times 1\text{K}$, so T and T_n are numerically equivalent. Dimensionless energy,

$$E_n = \frac{E}{J}, \tag{8}$$

Since $J = k_b \times 1\text{K}$, each E_n is equivalent to $k_b \times 1\text{K}$. Dimensionless Moment,

$$m_n = \frac{m}{\mu}, \tag{9}$$

where m is the SI moment. We have set μ to a unit value in the code, so these are numerically equivalent (in any case, m_n is equivalent to the spin of a single site). Dimensionless heat capacity,

$$c_n = \frac{c}{k_b}, \tag{10}$$

c is the SI heat capacity. Finally, the dimensionless magnetic field,

$$H_n = \frac{H\mu}{T_0 k_b}. \tag{11}$$

3 Results and Discussion

3.1 Errors

3.1.1 Random

In most cases, random errors in calculated quantities were estimated from the standard error in the mean from sample histories. Two exceptions to this are: measured values of T_c, whose errors were estimated as the interval between sampled temperatures (since the heat capacity may have peaked in the range (T_{i-1}, T_{i+1})); and errors in quantities passed to (mathematical) functions, which were estimated using the calculus [6].

By recording long histories of the measured quantities, we can average out fluctuations about equilibrium. This comes at the cost of higher memory usage, and slightly higher computation times, so it can't be made arbitrarily high.

3.1.2 Systematic

Strictly speaking, the random number p should be uniform over $[0, 1]$. The NumPy function used to generate this is uniform over $[0, 1)$. But given that $p = 1$ has an infinitesimal probability of occurrence (i.e. approximately zero), this should not be significant cause for error.

The underlying pesudorandom number generator, Mersenne Twister, is well tested and has period length $2^{19937} - 1$ [7]. Even if we used a 10^9 by 10^9 lattice, and ran it for 10^9 steps, we would not come close to the period. Our generated p is thus a good approximation to a truly random p.

The smallest numbers used in the calculations (of an order k_b) are far greater than the smallest 32bit floats (default encoding in NumPy), mutatis mutandis for the larger numbers (of order no greater than 10^9 or so). So under/overflow errors are not expected.

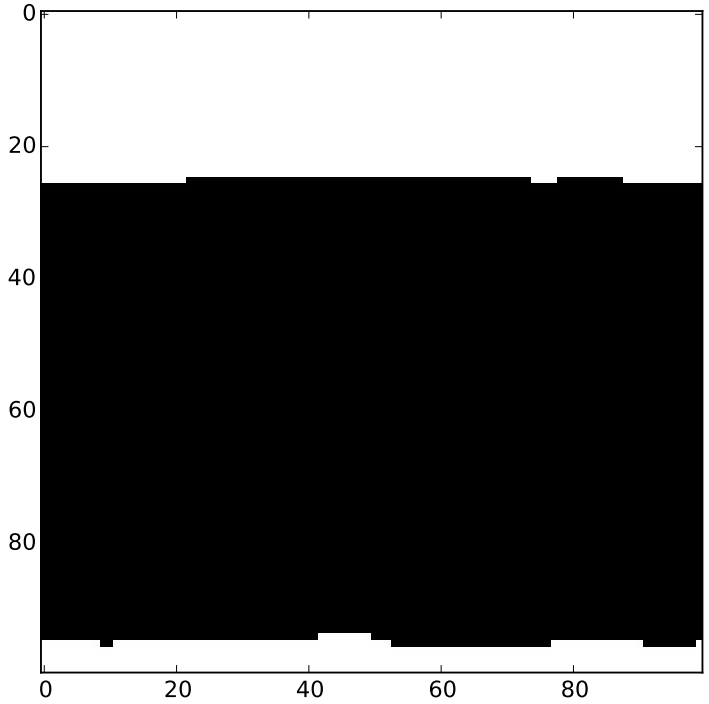


Figure 1: Binary image of a 100 by 100 lattice at $T_n = 1$, with no external field, after 800 steps from an initially random configuration. The domains (connected regions of like spin) have formed rings on the lattice torus; the domains extend through the periodic boundaries.

Occasionally, when $T_n < T_O$, domain formation like that in figure 1 occurs. The long straight edges of the domains form metastable states which change slowly, and result in extended convergence times. Flipping a square along an edge is energetically unfavourable, and since T is relatively low so are the chances of random flipping. Eventually, one domain will tend to grow into the other(s) but this takes time and can be avoided. Mitigating formation of such domains is accomplished by setting the initial configuration to a pre magnetised state (ie all spins -1 or 1) before running the simulation. Since the lattice is already close to its stable state, the formation of domains like this is unlikely.

3.2 Preliminary Tests

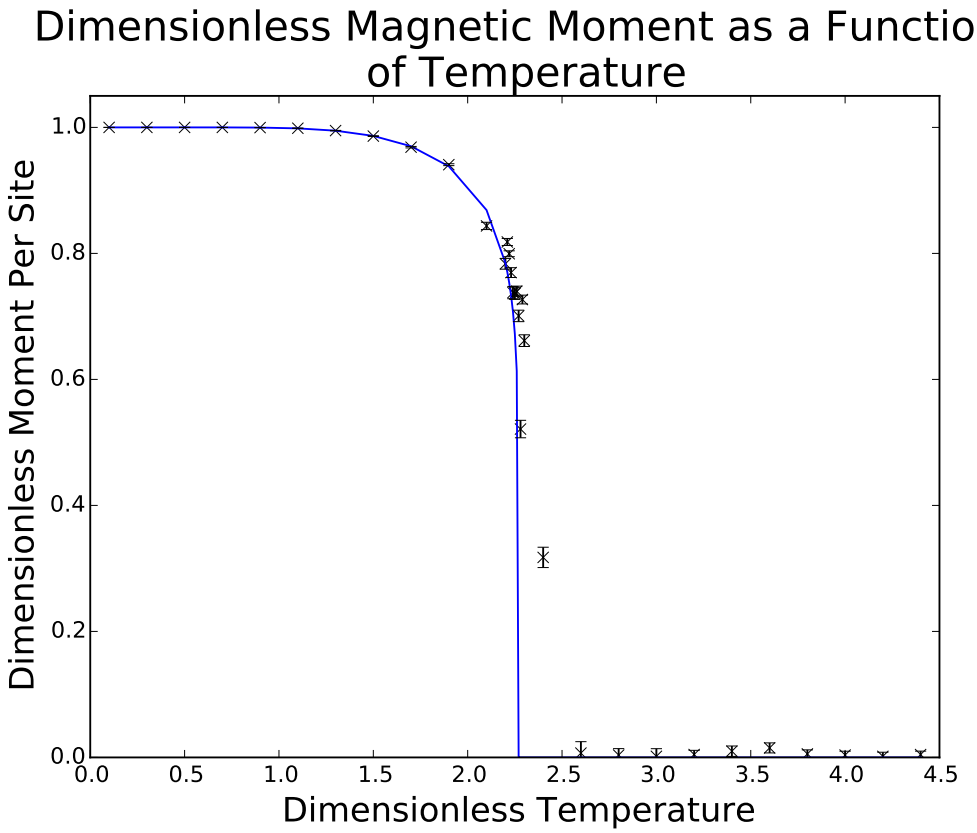
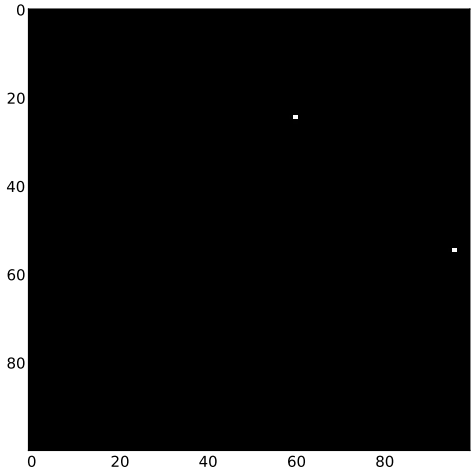
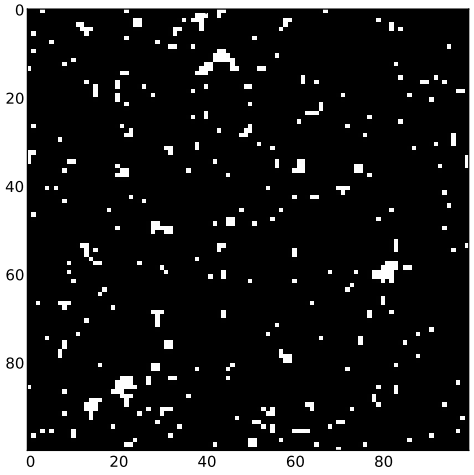


Figure 2: The exact result of equation (3) is plotted as a blue line. The simulated data are plotted as black points, which seem to be consistent with a phase transition at T_O. These data were obtained from simulations of 32 by 32 lattices. The magnitude is plotted as the direction is random with no applied field.

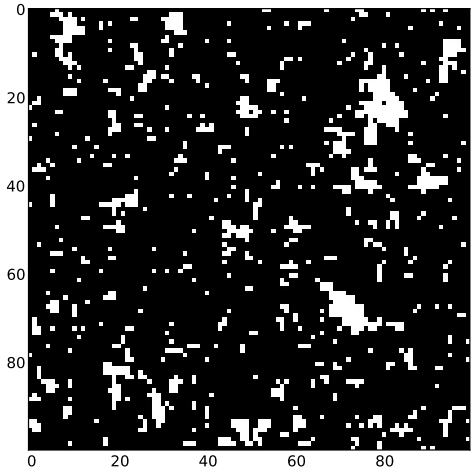
A comparison of an exact result with the simulation is graphed in figure 2, and was produced by section 2 of the code. A reasonable agreement with the theory is displayed, and approach to $T_c = T_O$ is demonstrated. Around T_O some inconsistencies are clear, but these are likely due to a combination of: finite size effects, the exact results take $\lim_{\to\infty} L$; and violent fluctuations about equilibrium for $T_n \approx T_O$ [1,4]. Setting the threshold at 0.6, and 0.4 yields values of $T_c = 2.28 \pm 0.01$ and 2.4 ± 0.1 respectively. Both are just over one deviation from T_O. The estimations are not helped with the somewhat arbitrary choice of threshold, and the slow convergence to equilibrium near T_O [1].



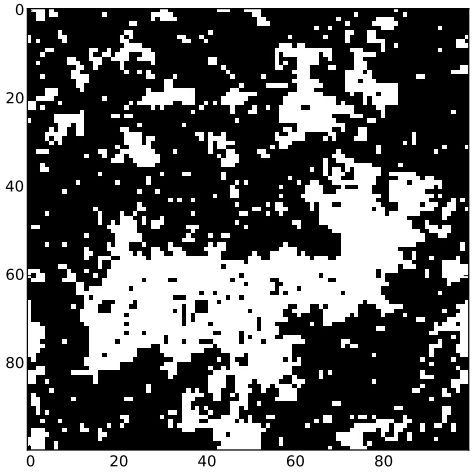
(a) 1



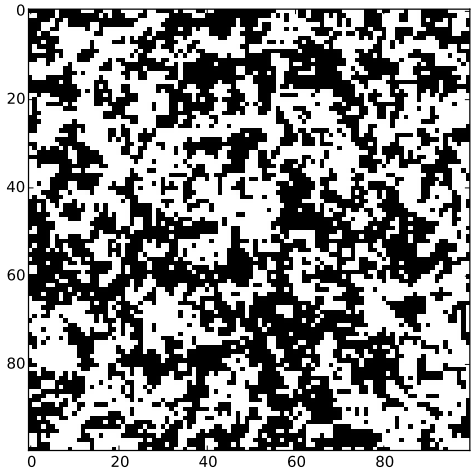
(b) 2



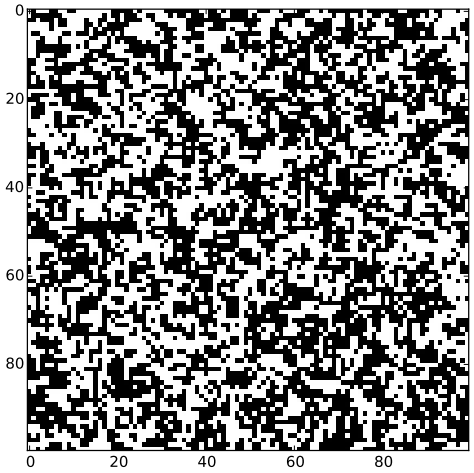
(c) 2.2



(d) 2.3



(e) 3



(f) 5

Figure 3: A 100 by 100 lattice is pre-magnetised and heated. After 1000 steps at the given T_n (after the letter), a binary image is taken before moving onto the next T_n (in alphabetical order).

Section 3 of the code is used to generate the images in figure 3. Large domain formation is stable in (a-c), and a spontaneous magnetisation is clear from the sheer size of some domains. Small domains are still present in (d), however there are approximately an equal number of -1 and 1 spin domains, giving no spontaneous magnetisation. This is consistent with (d) being slightly above T_0. Images (e) and (f) have no spontaneous magnetisation. Domain size clearly decreases with temperature, eventually devolving into randomness. All these results are consistent with the known behaviour of the Ising model [1,4].

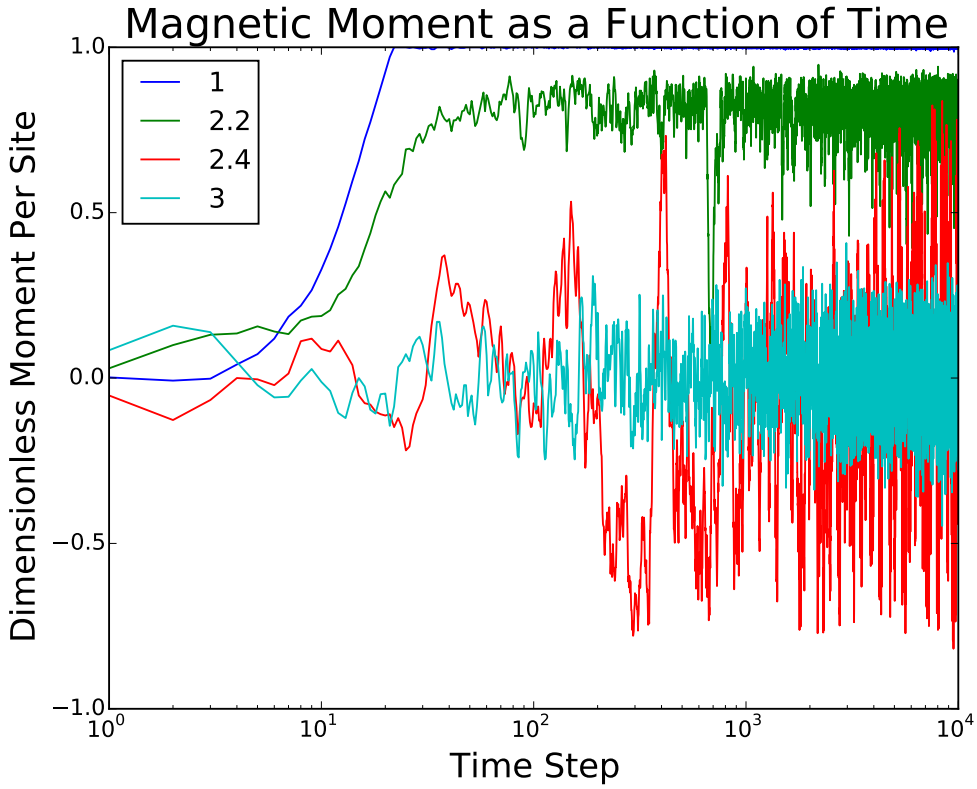
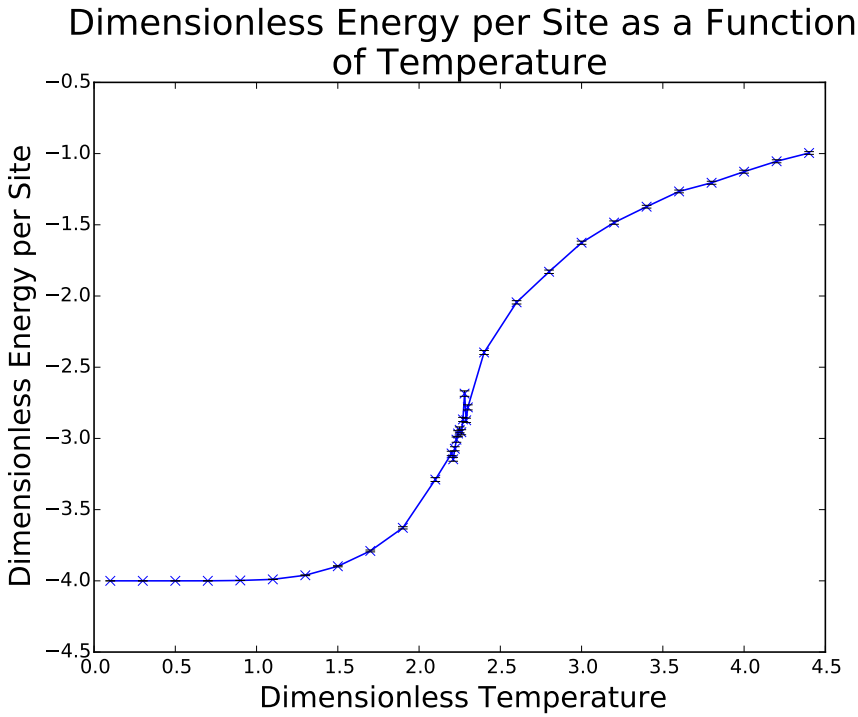


Figure 4: A set of 32 by 32 lattices at various values of T_n (see legend) run for 10000 steps, with no applied field.

The data in figure 4 were generated by section 4 of the code. Again we have results consistent with the theory: for $T_n < T_O$ the lattice converges to a non-zero mean magnetic moment, whereas $T_n > T_O$ show only fluctuations about zero. Violent fluctuations can be seen for the case $T_n = 2.4$, which could be the cause of inaccuracies. The figure also shows most cases tend to equilibrate by 100 steps, but 1000 steps is sufficient for any case. A history going back at least 100 steps (estimated from the widest fluctuations) is needed to ensure that the fluctuations don't bias any quantities estimated from the history.

With all these results there is no reasonable doubt that the code is consistent with known behaviours, and can now be used to examine more complex effects.

3.3 Energy and Heat Capacity



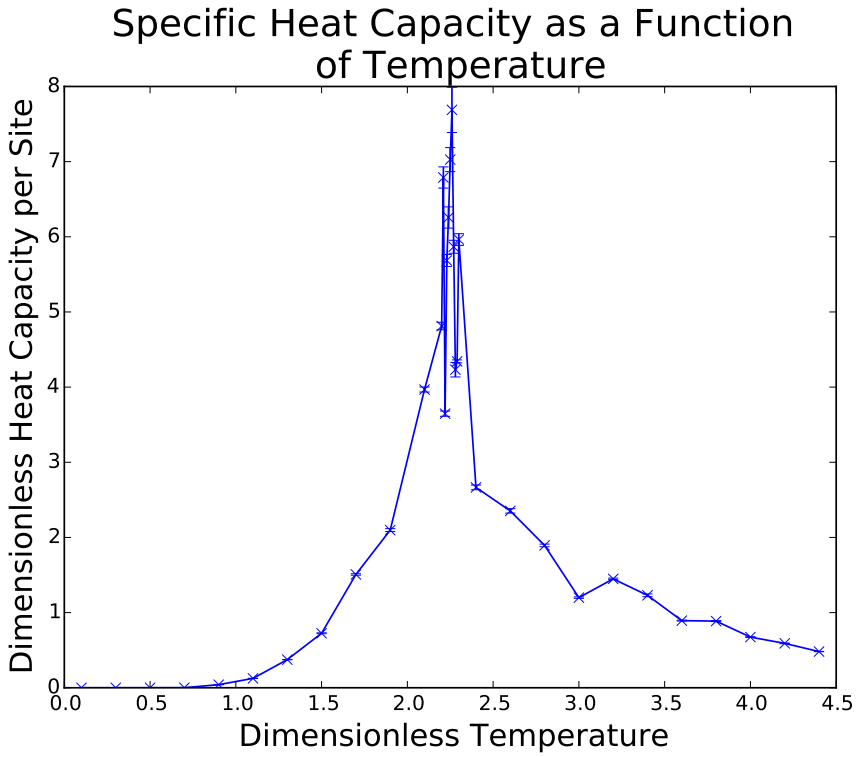


Figure 5: Energy based statistics have been graphed using the same code and conditions as figure 3. Note the verticality of the energy curve near T_O and the corresponding peak in the heat capacity.

The spike in heat capacity suggests that it is a 2nd order transition, which is known to be true [1]. The heat capacity arrays used in figure 5 can also be used to estimate the critical temperature. Passing them to the *estimate_Tc* function yields a value of:

$$T_c = 2.26 \pm 0.01, \tag{12}$$

This is consistent with T_O to within a standard deviation of the estimate. The inaccuracy (2.27 is closer) is probably caused by large fluctuations in the heat capacity near T_O, leading to overestimates of the heat capacity not at the T_c. Near T_O we can see that the heat capacity (as well the other quantities) do vary significantly, and multiple runs of the program show that this region is not always self consistent.

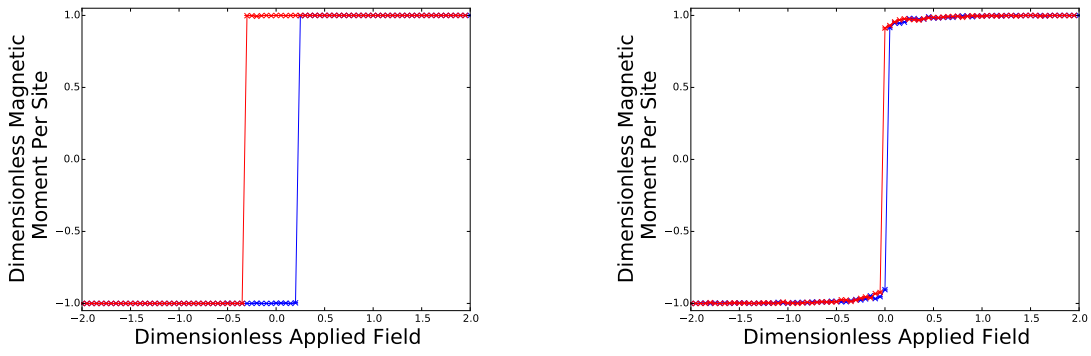
3.4 Application of a Magnetic Field

In code section 5 the application of a changing magnetic field is examined on a 32 by 32 lattice running for 250 steps at each value of the field. The results, at different temperatures, are in figure 6.

Hysteresis of the mean magnetic moment is observed for lattices with $T_n < T_O$, (a) and (b). Since a magnetised state is stable in the absence of a field, a fully magnetised state becomes metastable when an external field is applied in the opposite direction. Although, globally, field aligned spins are more favourable, ΔE for flipping an individual spin may still be positive while the interaction energy dominates the expression for ΔE (section 2.2). The area between the increasing and decreasing curves is reduced by as T_n increases, while the fluctuations at a given field increase. Higher temperatures increase the probability of a spin flipping, which accounts for a weaker H_n required to reverse the metastable spins. Above T_0 a net magnetic moment is unstable, so no hysteresis occurs.

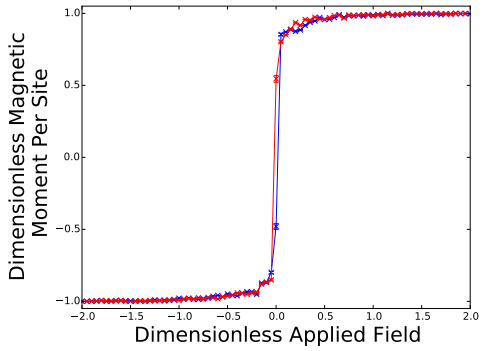
In (f) the system behaves linearly and paramagnetically, but closer to the critical temperature the behaviour is non-linear (c-e) and paramagnetic. However, all the curves are approximately linear before the magnetic moments are saturated (i.e. all at ± 1) and increasing the temperature does increase the field required for saturation. Thus (c-f) can be reconciled when seen as a curve which is being "pulled" horizontally from its ends; (f) only appears linear because the fields that were sampled are not sufficient to saturate the moments, as thermal effects are leading to higher probabilities of a spin flipping against the applied field.

If we define the (dimensionless) susceptibility (χ) by $M_n = \chi H_n$, we can make the following observations: $\chi(H_n)$ is non-linear and sigmoid in form; and $\chi(T_n)$ is a decreasing function.

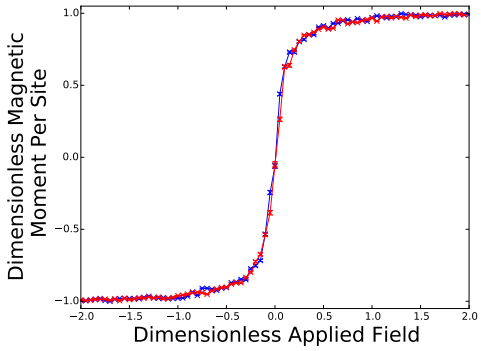


(a) 1

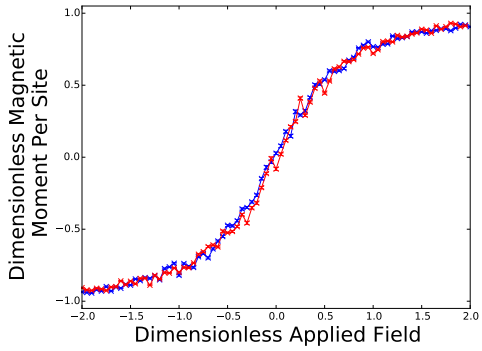
(b) 2



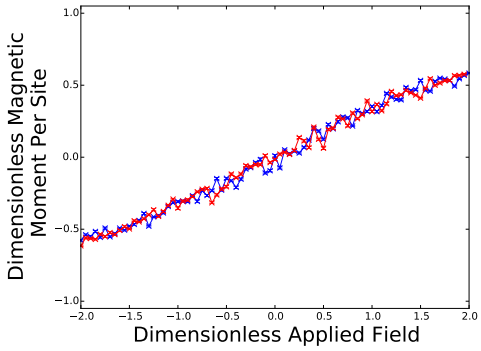
(c) 2.4



(d) 3



(e) 5



(f) 10

Figure 6: Increasing H_n points are blue, while decreasing H_n points are red, the T_n of the lattice is printed after the letter. Note the rapid changes from one state to another in (a) and (b), while (c-e) don't change as quickly. (a) and (b) are clearly ferromagnetic (remnant moments when $H_n = 0$), while (c-f) are non-ferromagnetic (no remnant moment).

4 Conclusion

By implementing the Metropolis algorithm in Python, a working model of the two dimensional Ising model was created. This model was tested against known facts, and found to be consistent with them. These tests included: the magnetic moment as a function of temperature; and the existence of a phase transition as seen in the domain binary images, the magnetic moment as a function of time, and the lack of remnant magnetic

moment above T_O; and the dimensionless critical temperature was measured 2.26 ± 0.01 (equivalent to Kelvin), which is consistent with the exact result.

Application of an external magnetic field was also studied. Magnetic moment as a function of applied field was found to exhibit ferromagnetic hysteresis below the critical temperature, and paramagnetic behaviour above it. Generally, the paramagnetic susceptibility is seen to be non-linear and sigmoid in form, and is a decreasing function of temperature.

5 References

1. Blundell S.J., Blundell K.M., 2006. *Concepts in Thermal Physics*. Oxford University Press.

2. Yang C.N., 1952. *The Spontaneous Magnetization of a Two Dimensional Ising Model*. Physical Review Letters.

3. Baxter. R.J., 1982. *Exactly Solved Models in Statistical Mechanics*. Academic Press.

4. Landau D.P., Binder K., 2009. *A Guide to Monte-Carlo Simulations in Statistical Physics*. Cambridge University Press.

5. Buscher D., 2016. *Part II Computational Physics Projects*. Department of Physics, University of Cambridge.

6. Hughes I.G., Hase T.P.A., 2010. *Measurements and their Uncertainties: A Practical Guide to Modern Error Analysis*. Oxford.

7. Matsumoto M., Nishimura T., 1998. *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*. Keio University.

6 Appendices

6.1 Code Listing

```
# -*- coding: utf-8 -*-
#
#Part II physics computational project 2015-2016
#"The Ising Model of a Ferromagnet"
#
#Python: 3.4.3
#NumPy: 1.9.2
#SciPy: 0.15.1


import numpy as np
from scipy import constants
import matplotlib.pyplot as plt
import datetime


#Important/frequently used constants are defined globally
KB = constants.value("Boltzmann constant")
ONSANGER_T = 2/np.log(1 + np.sqrt(2))

###############################################################################
#SECTION 0
#The model is implemented in the class definition, and a few functions
#are defined for various uses
```

```
25
26
27  class SpinLattice(object):
28      """2D Ising model of a simple magnetic system.
29
30      Uses a model lattice of simple (up or down) spins to model the magnetic
31      using a behaviour nearest neighbour interaction model. Periodic boundary
32      conditions have been implemented.
33      """
34
35      def __init__(self, height, width, temperature, applied_field,
36                   magnetic_moment, NN_interaction_energy, initial_spins = 0,
37                   history = 30):
38          """Constructs the lattice variables.
39
40          Notes:
41              height and width must both be even, an error is raised otherwise.
42              All quantities are defined in SI.
43
44          args:
45              height (int): number of rows in the lattice.
46
47              width (int): number of columns in the lattice.
48
49              temperature (float): the usual physical quantity. Must be > 0.
50
51              applied_field (float): the (H) magnetic field.
52
53              magnetic_moment (float): the magnetic moment of an individual
54                                       spin site.
55
56              NN_interaction_energy (float): Interaction energy of nearest
57                                             neighbour spins.
58
59              initial_spins: The initial lattice will be random if anything other
60                             than -1 or 1. In that case all the spins are all
61                             -1 or 1 respectively.
62
63              history (int): number of most recent values for stats, e.g. energy,
64                             used in calculations, e.g. of heat capacity.
65          """
66
67
68          if (height%2 == 0) and (width%2 == 0):
69              self.shape = (height, width)
70          elif (height%2 == 1) or (width%2 == 1):
71              raise ValueError("'height' and 'width' must be even")
72          else:
73              raise TypeError("'height' and 'width' must be (even) integers")
74
75
76          #these constants have their usual meanings
77          self.N = self.shape[0]*self.shape[1]
78          self.T = temperature
79          self.beta = 1/(temperature*KB)
80          self.H = applied_field
81          self.mu = magnetic_moment
82          self.J = NN_interaction_energy
83
84          #creates an initial lattice
85          if initial_spins == 1:
86              self.lattice = np.ones(self.shape, dtype = np.int8)
87          elif initial_spins == -1:
88              self.lattice = -1*np.ones(self.shape, dtype = np.int8)
89          else:
90              lattice = 2*np.random.randint(2, size = self.shape) - 1
91              self.lattice = lattice.astype(np.int8)
92
93          #these variable have their usual meanings
94          self.moment = 0
95          self.interaction_energies = 0
96          self.lattice_energy = 0
97          self.heat_capacity = 0
98
99          #creates arrays to hold the ".s" most recent values
100         self.s = history
101         self.energy_history = np.array([])
102         self.moment_history = np.array([])
103         self.heat_capacity_history = np.array([])
104
105         #calls the update method to calculate some of the quantities
106         self.update_all()
107
108         #"time_step" is incremented when height*width points lattice points
109         #have been tested
110         self.time_step = 0
111
112
113     def time_increment(self):
114         self.time_step += 1
115         return None
116
117
118     def update_summed_moment(self):
119         """Calculates and updates the total moment"""
```

```
120             self.moment = self.mu * np.sum(self.lattice)
121             return None
122 
123 
124         def update_interaction_energy(self):
125             """Calculates and updates the array of interaction energies
126 
127             Permuting the lattice to find the nearest neighbours is quick, and
128             implicitly incorporates the periodic boundary conditions.
129             """
130 
131             lefts = np.roll(self.lattice, 1, 1)
132             rights = np.roll(self.lattice, -1, 1)
133             tops = np.roll(self.lattice, 1, 0)
134             bottoms = np.roll(self.lattice, -1, 0)
135             I = lefts + rights + tops + bottoms
136             I *= self.lattice
137 
138             self.interaction_energies = -self.J * I
139             return None
140 
141 
142         def update_energy(self):
143             """Calculates and updates the energy of the lattice"""
144 
145             hamiltonian = np.sum(self.interaction_energies)
146             hamiltonian -= self.H * self.moment
147 
148             self.lattice_energy = hamiltonian
149             return None
150 
151 
152         def update_heat_capacity(self):
153             """Calculates the heat capacity
154 
155             Uses the fluctuation-dissipation theorem on ".energy_history", only
156             when the energy_history is full. The lattice must be in equilibrium for
157             all of ".energy_history" for this to be accurate.
158             """
159 
160             if len(self.energy_history) == self.s:
161                 sigma_E = np.std(self.energy_history)
162                 c = KB*self.beta*self.beta
163                 c *= sigma_E*sigma_E
164                 self.heat_capacity = c
165             else:
166                 pass
167             return None
168 
169 
170         def update_E_history(self):
171             """Takes the current energy and adds it to the history"""
172 
173             if len(self.energy_history) < self.s:
174                 self.energy_history = np.append(self.energy_history,
175                                                 self.lattice_energy)
176             else:
177                 self.energy_history = np.append(self.energy_history,
178                                                 self.lattice_energy)
179                 self.energy_history = np.delete(self.energy_history, 0)
180             return None
181 
182 
183         def update_m_history(self):
184             """Takes the current moment and adds it to the history"""
185 
186             if len(self.moment_history) < self.s:
187                 self.moment_history = np.append(self.moment_history,
188                                                 self.moment)
189             else:
190                 self.moment_history = np.append(self.moment_history,
191                                                 self.moment)
192                 self.moment_history = np.delete(self.moment_history, 0)
193             return None
194 
195 
196         def update_c_history(self):
197             """Takes the current heat capacity and adds it to the history"""
198 
199             if len(self.heat_capacity_history) < self.s:
200                 self.heat_capacity_history = np.append(self.heat_capacity_history,
201                                                        self.heat_capacity)
202             else:
203                 self.heat_capacity_history = np.append(self.heat_capacity_history,
204                                                        self.heat_capacity)
205                 self.heat_capacity_history = np.delete(self.heat_capacity_history,
206                                                        0)
207             return None
208 
209 
210         def update_all(self):
211             """Calls the "update" methods in such an order as to ensure that
212             each are calculated correctly.
213             """
214             self.update_summed_moment()
```

```
215             self.update_m_history()
216
217             self.update_interaction_energy()
218             self.update_energy()
219             self.update_E_history()
220
221             self.update_heat_capacity()
222             self.update_c_history()
223             return None
224
225
226         def spin_flip(self, row, column):
227             """Tests a given spin for flipping, and flips accordingly
228
229             Implementation of the Metropolis Hastings algorithm (for a point).
230
231             The ".interaction_energies" attribute must be correct/updated before
232             this method is called.
233             """
234
235             delta_E = -2*self.interaction_energies[row, column]
236             delta_E += 2 * self.mu * self.H * self.lattice[row, column]
237
238             if delta_E <= 0:
239                 #if flipping releases energy, then it occurs
240                 self.lattice[row][column] *= -1
241             else:
242                 #if flipping requires energy, then we simulate a thermal
243                 #fluctuation (which provides the energy) using a Monte-Carlo method
244                 p = np.random.rand()
245                 boltzman_p = np.exp(-delta_E*self.beta)
246
247                 if p <= boltzman_p:
248                     self.lattice[row][column] *= -1
249                 else:
250                     pass
251
252             return None
253
254
255         def lattice_flip_checkerboard(self):
256             """Tests and flips two sets of alternate lattice positions.
257
258             This is a simple example of Checkerboard Decomposition; by splitting
259             the lattice into 'black' and 'white' squares, we can loop over all of
260             one 'colour' without having to update the interaction energies.
261
262             The 'colour' of a particular spin is found using modular arithmetic.
263
264             NB the dimensions of the lattice must be even for this to work.
265             """
266
267             #ensures that the interaction energies are accurate, in most cases
268             #this call may be redundant
269             self.update_interaction_energy()
270             for i in range(self.shape[0]):
271                 for j in range(self.shape[1]):
272                     if i%2 == j%2:
273                         self.spin_flip(i, j)
274                     else:
275                         pass
276
277             #The interaction energies must be updated now, as the nearest neighbour
278             #spin states may have changed
279             self.update_interaction_energy()
280             for i in range(self.shape[0]):
281                 for j in range(self.shape[1]):
282                     if i%2 != j%2:
283                         self.spin_flip(i, j)
284                     else:
285                         pass
286
287             #updates the rest of the system variables
288             self.update_all()
289             self.time_increment()
290
291             return None
292
293
294         def advance_time(self, steps, data = False, timing = False):
295             """Calls the ".lattice_flip_checkerboard()" method "steps" times.
296
297             If "data" is True, then the energy and mean magnetic moment are
298             calculated and returned as functions of time.
299             If "timing" is true, then the elapsed time will be printed every 10%
300             (ish) of the steps.
301             """
302
303             if timing:
304                 start = datetime.datetime.now()
305
306             if data:
307                 self.update_all()
308                 t = np.array([self.time_step])
309                 energy_t = np.array([self.lattice_energy])
```

```
310
311             mean_moment = (self.moment/(self.N))
312             m_t = np.array([mean_moment])
313
314             for i in range(steps):
315                 self.lattice_flip_checkerboard()
316
317                 t = np.append(t, self.time_step)
318                 energy_t = np.append(energy_t, self.lattice_energy)
319
320                 mean_moment = self.moment/(self.N)
321                 m_t = np.append(m_t, mean_moment)
322
323                 if timing and ((i + 1)%(steps//10) == 0):
324                     print(i+1)
325                     now = datetime.datetime.now()
326                     print("elapsed time -- ", now - start)
327
328             return np.array([t, energy_t, m_t])
329
330         else:
331             for i in range(steps):
332                 self.lattice_flip_checkerboard()
333
334                 if timing and ((i + 1)%(steps//10) == 0):
335                     print(i+1)
336                     now = datetime.datetime.now()
337                     print("elapsed time -- ", now - start)
338
339             return None
340
341
342 def binary_plot(title, lattice):
343     """Plots a given lattice as a binary image"""
344     plt.figure()
345     plt.imshow(lattice, cmap='Greys',  interpolation='none')
346     plt.savefig("%s.pdf" % title, bbox_inches="tight")
347     plt.close()
348     return None
349
350
351 def estimate_Tc(temperatures, heat_caps):
352     """Estimates the critical temperature by considering heat capacity
353
354     For the 2D models exact solution, it can be shown that a singularity occurs
355     in the heat capacity at the critical temperature. By entering (ordered)
356     arrays of "heat_caps" as a function of "temperatures", Tc can be estimated
357     from the maximal heat capacity.
358     """
359
360     Tcrit = temperatures[np.argmax(heat_caps)]
361     return Tcrit
362
363
364 def estimate_Tc2(temperatures, magnetisation, threshold = 0.5):
365     """Estimates the critical temperature
366
367     When the (dimensionless mean) magnetisation falls bellow threshold,
368     the corresponding temperature is an estimate for the critical
369     temperature. "magnetisation" and "temperatures" must be mutually ordered
370     arrays. This function is often wildly inaccurate.
371     """
372
373     for i in range(len(magnetisation)):
374         if magnetisation[i] <= threshold:
375             return temperatures[i]
376         else:
377             pass
378
379     print("No critical temperature was found; try changing the threshold")
380     return None
381
382
383 def mag_func(temperatures, J = KB):
384     """Defines the magnetisation function (per site) for use before Tc"""
385
386     beta = np.power((temperatures*KB), -1)
387     m = np.sinh(2*beta*J)
388     m = np.power(m, -4)
389     m = 1 - m
390     m = np.power(m, (1/8))
391     return m
392
393 def exact_magnetisation(temperatures, J = KB):
394     """Returns the magnetisation (per site) as a function of temperature
395
396     Defined for SI, however the interaction energy "J" defaults to KB. So
397     Kelvin is equivalent to the T*J/KB dimensionless temperature.
398     """
399
400     m_of_T = np.piecewise(temperatures, [temperatures < ONSANGER_T,
401                                          temperatures >= ONSANGER_T],
402                                          [mag_func, 0], (J,))
403     return m_of_T
404
```

```
405
406 ###############################################################################
407 #The following sections contain specific examples which use the SpinLattice
408 #class. In general, we have set J = interaction energy = KB (ie the numerical
409 #value), such that the temperature in Kelvin is equivalent to units of J/KB.
410 #For similar reasons the magnetic moment has been set to 1 so that the
411 #calculated moments are normalised.
412 print("This script will perform simulations of the 2D Ising Model.")
413 print("The plots generated are saved to disk on creation.")
414 print("Some important results are printed in the console. \n")
415 ###############################################################################
416 #SECTION 1
417 #toy calculations are performed for performance analysis.
418
419 print("Performance analysis of a 32 by 32 lattice for 10000 steps is starting")
420 test = SpinLattice(32, 32, 1, 0, 1, KB)
421 test.advance_time(10000, False, True)
422 print("End \n")
423
424 print("Performance analysis of a 50 by 50 lattice for 10000 steps is starting")
425 test = SpinLattice(50, 50, 1, 0, 1, KB)
426 test.advance_time(10000, False, True)
427 print("End \n")
428
429
430 ###############################################################################
431 #SECTION 2
432 #In this section, the energy, mean magnetisation, and heat capacity are
433 #investigated as a function of temperature. The initial lattices are
434 #essentially pre magnetised, this is to avoid equilibrium convergence issues
435 #if large straight edge domains form below the critical temperature
436 #
437 #The equilibrium values are estimated from the means of the relevant histories
438 #and their errors from the standard errors in the means (sdt/sqrt(n))
439
440 temperatures = np.array([])
441 temperatures = np.append(temperatures, np.linspace(0.1, 2.1, 11))
442 #More points are generated close to the critical point, as the curves change
443 #rapidly here
444 temperatures = np.append(temperatures, np.linspace(2.2, 2.3, 11))
445 temperatures = np.append(temperatures, np.linspace(2.4, 4.4, 11))
446
447 energy_T = np.array([])
448 E_errs = np.array([])
449 heatcap_T = np.array([])
450 C_errs = np.array([])
451 meanmag_T = np.array([])
452 m_errs = np.array([])
453
454
455 print("The statistics, as functions of temperature, are being calculated:")
456 print("This took about 4mins on a single 3.2GHz core (100% load)")
457 print("Start", datetime.datetime.now())
458
459 for T in temperatures:
460     #longer history for accuracy
461     latt = SpinLattice(32, 32, T, 0, 1, KB, 1, 100)
462     latt.advance_time(1000, False, False)
463
464     Enorm = np.mean(latt.energy_history)/(KB*latt.N)
465     E_err = np.std(latt.energy_history)/(np.sqrt(latt.s)*KB*latt.N)
466
467     Cnorm = np.mean(latt.heat_capacity_history)/(KB*latt.N)
468     C_err = np.std(latt.heat_capacity_history)/(np.sqrt(latt.s)*KB*latt.N)
469
470     mnorm = np.abs(np.mean(latt.moment_history)/latt.N)
471     m_err = np.std(latt.moment_history)/(np.sqrt(latt.s)*latt.N)
472
473
474     energy_T = np.append(energy_T, Enorm)
475     E_errs = np.append(E_errs, E_err)
476
477     heatcap_T = np.append(heatcap_T, Cnorm)
478     C_errs = np.append(C_errs, C_err)
479
480     meanmag_T = np.append(meanmag_T, mnorm)
481     m_errs = np.append(m_errs, m_err)
482
483     print(T, "K is Done")
484
485 #provides two estimate of the critical temperature
486 critical_T = estimate_Tc(temperatures, heatcap_T)
487 critical_T2 = estimate_Tc2(temperatures, meanmag_T)
488 print("Critical Temperature Estimate (heat capacity)", critical_T)
489 print("Critical Temperature Estimate (magnetic moment)", critical_T2)
490 print("End", datetime.datetime.now(), "\n")
491
492
493
494 plt.figure()
495 plt.plot(temperatures, energy_T, color = "b", marker="x")
496 plt.xlabel("Dimensionless Temperature", fontsize=18)
497 plt.ylabel("Dimensionless Energy per Site", fontsize=18)
498 plt.title("Dimensionless Energy per Site as a Function \n of Temperature",
499           fontsize=22)
```

```
plt.errorbar(temperatures, energy_T, yerr = E_errs, ecolor='black', marker='',
             linestyle = "None")
plt.savefig("Energy_v_Temp.pdf", bbox_inches='tight')
plt.close()


plt.figure()
plt.plot(temperatures, heatcap_T, color = "b", marker="x")
plt.xlabel("Dimensionless_Temperature", fontsize=18)
plt.ylabel("Dimensionless_Heat_Capacity_per_Site", fontsize=18)
plt.title("Specific_Heat_Capacity_as_a_Function_\n_of_Temperature",
          fontsize=22)
plt.errorbar(temperatures, heatcap_T, yerr = C_errs, marker='',
             linestyle = "None")
plt.savefig("Heat_Capacity_v_Temp.pdf", bbox_inches='tight')
plt.close()


plt.figure()
plt.plot(temperatures, meanmag_T, linestyle = "None", color = "k", marker="x")
plt.plot(temperatures, exact_magnetisation(temperatures))
plt.xlabel("Dimensionless_Temperature", fontsize=18)
plt.ylabel("Dimensionless_Moment_Per_Site", fontsize=18)
plt.ylim([0, 1.05])
plt.title("Dimensionless_Magnetic_Moment_as_a_Function_\n_of_Temperature",
          fontsize=22)
plt.errorbar(temperatures, meanmag_T, yerr = m_errs, ecolor='k', fmt='',
             marker='', linestyle = "None")
plt.savefig("Mean_Magnetic_Moment_v_Temp.pdf",  bbox_inches='tight')
plt.close()
################################################################################


################################################################################
#SECTION 3
#A few binary images of a pre magnetised lattice being "heated" are collected

temps = np.array([1, 2, 2.2, 2.3, 3, 5])

print("Making_some_images")
print("This_took_about_10mins_on_a_single_3.2GHz_core_(100%_load)")
print("Start", datetime.datetime.now())

latt = SpinLattice(100, 100, 1, 0, 1, KB, 1)
for T in temps:
    latt.T = T
    latt.beta = 1/(T*KB)
    latt.advance_time(1000)

    #odd name to avoid problems with decimal points in filenames
    name = "Binary_Lattice_" + str(int(1000*T)) + "mK"
    binary_plot(name, latt.lattice)

print("End", datetime.datetime.now(), "\n")

################################################################################


################################################################################
#SECTION 4
#The normalised magnetic moment is considered as a function of
#time for a few different temperatures. A random spin initial lattice is used.

print("The_moments,_as_functions_of_time,_are_being_calculated")
print("This_took_about_4_mins_on_a_3.2GHz_core_(100%_load)")
print("Start", datetime.datetime.now())

latt1    = SpinLattice(32, 32, 1, 0, 1, KB, 0)
latt2_2 = SpinLattice(32, 32, 2.2, 0, 1, KB, 0)
latt2_4 = SpinLattice(32, 32, 2.4, 0, 1, KB, 0)
latt3    = SpinLattice(32, 32, 3, 0, 1, KB, 0)

dat1     = latt1.advance_time(10000, True)
dat2_2   = latt2_2.advance_time(10000, True)
dat2_4   = latt2_4.advance_time(10000, True)
dat3     = latt3.advance_time(10000, True)

print("End", datetime.datetime.now(), "\n")

plt.figure(4)
plt.semilogx(dat1[0], dat1[2], label = "1")
plt.semilogx(dat2_2[0], dat2_2[2], label = "2.2")
plt.semilogx(dat2_4[0], dat2_4[2], label = "2.4")
plt.semilogx(dat3[0], dat3[2], label = "3")
plt.xlabel("Time_Step", fontsize=18)
plt.ylabel("Dimensionless_Moment_Per_Site", fontsize=18)
plt.title("Magnetic_Moment_as_a_Function_of_Time", fontsize=22)
plt.legend(loc = "best")
plt.savefig("Magnetic_Moment_t_T.pdf", bbox_inches='tight')
plt.close()

################################################################################


################################################################################
```

```
#SECTION 5
#An applied magnetic field is examined here
#Let us define the normalised magnetic field H_n = H*mu/(ONSAGER_T*KB)
#Unlike in (most) previous sections, the changing magnetic field is examined on
#the same lattice

temps = np.array([1, 2, 2.4, 3, 5, 10])
H_n_up = np.linspace(-2, 2, 81)
H_n_down = H_n_up[::-1]
H_n_down = np.delete(H_n_down, 0)

#we have defined mu=1 so it will not appear here
H_fields_up = ONSANGER_T*KB*H_n_up
H_fields_down = ONSANGER_T*KB*H_n_down

print("A magnetic field is being applied to the lattice")
print("This took about 19mins on a single 3.2GHz core (100% load)")
print("Start", datetime.datetime.now())

for T in temps:

    moments_up = np.array([])
    m_up_errs = np.array([])
    moments_down = np.array([])
    m_down_errs = np.array([])

    latt = SpinLattice(32, 32, T, 0, 1, KB, 0, 100)
    for H_field in H_fields_up:
        latt.H = H_field
        #Only a short history is required for accurate moments, so few steps
        #are required
        latt.advance_time(250)

        moments_up = np.append(moments_up, latt.moment/latt.N)
        m_up_err = np.std(latt.moment_history)/(np.sqrt(latt.s)*latt.N)
        m_up_errs = np.append(m_up_errs, m_up_err)

    for H_field in H_fields_down:
        latt.H = H_field

        latt.advance_time(250)

        moments_down = np.append(moments_down, latt.moment/latt.N)
        m_down_err = np.std(latt.moment_history)/(np.sqrt(latt.s)*latt.N)
        m_down_errs = np.append(m_down_errs, m_down_err)

    plt.figure()
    plt.plot(H_n_up, moments_up, color = "b", marker="x")
    plt.errorbar(H_n_up, moments_up, yerr = m_up_errs, ecolor='b', fmt='',
                 marker='', linestyle = "None")

    plt.plot(H_n_down, moments_down, color = "r", marker="x")
    plt.errorbar(H_n_down, moments_down, yerr = m_down_errs, ecolor='r',
                 fmt='',
                 marker='', linestyle = "None")
    plt.xlabel("Dimensionless Applied Field", fontsize=24)
    plt.ylabel("Dimensionless Magnetic \n Moment Per Site", fontsize=24)
    plt.ylim([-1.05, 1.05])

    name = "Applied field " + str(int(1000*T)) + "mK"
    plt.savefig("%s.pdf" % name, bbox_inches='tight')
    plt.close()

    print(T, "K is Done")

print("End", datetime.datetime.now())
print("All the calculations are now complete")
################################################################################
```