Ch/ChE 164 Winter 2022 Final Project
The Statistical Thermodynamics of $He^3 - He^4$ Mixtures

Due Date: Friday March 15, 2024 @ 11:59pm PT

An interesting and instructive problem in statistical mechanics is the fluid-superfluid transition of He^4 at low temperatures. This transition is typically continuous (2nd-order), but becomes discontinuous (1st-order) in the presence of a critical amount of He^3 impurities. In this project, we aim to model and understand this behavior by both deriving analytical results, and implementing simulation methods.

- For a really cool visual of the transition, watch the video at this link: https://youtu.be/ 2Z6UJbwxBZI.

The system to model is described as follows:

- The system can be described as a lattice gas. From class, we know that the lattice-gas model is isomorphic with the Ising model. In this problem, we can treat the helium fluid/superfluid using a variation of the spin- 1 Ising model. We say He^4 particles can have spin $s = \pm 1$, whereas He^3 particles have spin $s = 0$. We thus interpret the problem as a "diluted" Ising model where the presence of the impurities ($s = 0$) serve only to disrupt the interactions. The fluid/superfluid transition then becomes the disorder/order transition.
- The lattice has coordination number z.
- For convenience, we label He^4 as Type **A** and He^3 as Type B.

- The Type A spins interact with each other through magnetic nearest-neighbor interactions with strength $J \geq 0$ (ferromagnetic), whereas Type B spins have no magnetic interactions and are taken simply as non-interactive impurities.
- The spins are not subjected to an external magnetic field, i.e. $h = 0$.
- We can consider the system in the constrained grand canonical ensemble, where $N_A + N_B = N$ with N fixed. In the lattice model, N replaces V since $N = V/v_0$ where v_0 is the size of a lattice site.

1

1. Deriving the free energy of the diluted Ising model.

1.1

a) Determine the characteristic variables of the system in the grand canonical ensemble.
As always, we will have the temperature as one of these. In this case, we are told that the total number of particles will replace the volume, so the second variable is N. Finally, we are interested in the chemical potentials of both particles, so we will have μ_A and μ_B as the third and fourth variables.

1.2

b) Write the Hamiltonian in terms of the spin-state of the system. Remember, only neighbors that are both Type A interact.
So we have two submissions. The first runs over all of the particles which are of the first type, and the second runs over all of the nearest neighbors of a particle. With define a spin variable τ_i which can take on talus of -1, 0, or 1, with the metal value only possible for the particle of the second type and the other two values possible for a particle of the first type. Since we are dealing with a grand canonical ensemble were the number of each particle is not fixed, but the total number must equal to N, we can write the Hamiltonian as:

$$\mathcal{H} = -\frac{1}{2} J \sum_{i=0}^{N_A} \sum_{j=0}^{z-1} \tau_i \tau_j - \mu_A N_A - \mu_B N_B \tag{1}$$

We can define a variable which tracks the average number of particles of the first type:

$$x = \frac{N_A}{N} \tag{2}$$

Since we know that $N_A + N_B = N$, we can write $N_B = N - N_A = N - xN$ and $N_A = xN$. We can then write the Hamiltonian as:

$$\mathcal{H} = -\frac{1}{2}J\sum_{i=0}^{N_A}\sum_{j=0}^{z-1}\tau_i\tau_j - \mu_A xN - \mu_B(N - xN) \tag{3}$$

We can neglect the $N\mu_B$ term since it is a constant and does not affect the minimization of the free energy and then we define the difference in chemical potentials as $\Delta\mu = \mu_A - \mu_B$. We can then write the Hamiltonian as:

$$\mathcal{H} = -\frac{1}{2}J\sum_{i=0}^{N_A}\sum_{j=0}^{z-1}\tau_i\tau_j - \Delta\mu xN \tag{4}$$

1.3

c) Define the relative magnetization m, and the Type A number fraction x :

$$m = \frac{\mathcal{M}}{N_A} = \frac{\sum_{i=1}^{N} s_i}{N_A} = \frac{\sum_{i=1}^{N} s_i}{xN}; \quad -1 \leq m \leq 1 \tag{1}$$

$$x = \frac{N_A}{N}; \quad 0 \leq x \leq 1 \tag{2}$$

Let us first consider the expression for the relative magnetization:

$$m = \frac{\sum_{i=1}^{N} s_i}{xN} \tag{5}$$

We know that only the first type of Portugal will make any contribution to the summation in the numerator, so we can change its aper bound to reflect this:

$$m = \frac{\sum_{i=1}^{N_A} s_i}{xN} = \frac{\sum_{i=1}^{xN} s_i}{xN} \tag{6}$$

This problem reduces to our original problem in the icing model if we know that we are dealing with a particle of the first type, so:

$$m = P(1) - P(-1) \tag{7}$$

so then isolating $P(1)$ we get:

$$P(1) = m + P(-1) \rightarrow P(1) = m + (1 - P(1)) \rightarrow P(1) = \frac{1+m}{2} \quad (8)$$

We plug this into find $P(-1)$:

$$P(-1) = 1 - P(1) = 1 - \frac{1+m}{2} = \frac{1-m}{2} \quad (9)$$

We just have to multiply these probabilities with the fraction of total number of particles of the first type:

$$p(1) = \frac{1+m}{2}x \quad p(-1) = \frac{1-m}{2}x \quad (10)$$

And then we know the probability of no spin is just the amount of particles of the second type:

$$p(0) = 1 - x \quad (11)$$

We can then write the average spin at a given site as:

$$\langle \tau_i \rangle = -1p(-1)+0p(0)+1p(1) = -1\frac{1-m}{2}x+0(1-x)+1\frac{1+m}{2}x = mx \quad (12)$$

Using the mean field approximation, write an expression for the average Hamiltonian in terms of m and x. In other words, eliminate all summations over the spin-state.
Taking the average of the hamiltonian, we get:

$$\langle \mathcal{H} \rangle = -\frac{1}{2}J \left\langle \sum_{i=0}^{N_A} \sum_{j=0}^{z-1} \tau_i \tau_j \right\rangle - \Delta\mu x N = -\frac{1}{2}J \sum_{i=0}^{N_A} \sum_{j=0}^{z-1} \langle \tau_i \tau_j \rangle - \Delta\mu x N \quad (13)$$

But as in our previous derivation of the icing model, we can assume that our spins are decoupled:

$$\langle \mathcal{H} \rangle = -\frac{1}{2}J \sum_{i=0}^{N_A} \sum_{j=0}^{z-1} \langle \tau_i \rangle \langle \tau_j \rangle - \Delta\mu x N \quad (14)$$

But we get the same result if we just replace the bound of the first submission with the total number of particles N:

$$\langle \mathcal{H} \rangle = -\frac{1}{2}J \sum^{N} \sum_{j=0}^{z-1} \langle \tau_i \rangle \langle \tau_j \rangle - \Delta\mu x N \quad (15)$$

Plugging in our expression for the average spin at a given site, we get:

$$\langle \mathcal{H} \rangle = -\frac{1}{2} J \sum^{N} \sum_{j=0}^{z-1} (mx)(mx) - \Delta \mu x N \tag{16}$$

Since the insides of the summations is no longer dependent on the indices, we can just multiply by a factor of $N \cdot z$:

$$\langle \mathcal{H} \rangle = -\frac{1}{2} J N z (mx)^2 - \Delta \mu x N \tag{17}$$

1.4

d) Write an expression of the Gibbs entropy in terms of m and x.
The Gibbs entropy is given by:

$$S = -kN \sum_{\tau_i} p(\tau_i) \ln p(\tau_i) \tag{18}$$

where we are summing over all possible spins. Also this is an exe extensive property, so we have to multiply by a tarter of N. So, this trust includes three terms:

$$S = -kN \left(p(1) \ln p(1) + p(-1) \ln p(-1) + p(0) \ln p(0) \right) \tag{19}$$

We can plug in our expressions for the probabilities:

$$S = -kN \left(\frac{1+m}{2} x \ln \frac{1+m}{2} x + \frac{1-m}{2} x \ln \frac{1-m}{2} x + (1-x) \ln(1-x) \right) \tag{20}$$

1.5

e) Write the variational free energy, G. In addition to the variables characteristic to the ensemble, the free energy should also depend on m and x.
The free energy is given by:

$$G = E - TS \tag{21}$$

where E is the average Hamiltonian and S is the Gibbs entropy. We need to find the average energy and the entropy of the previous problem. Plugging in gives us:

$$G = -\frac{1}{2}JNz(mx)^2 - \Delta\mu xN + NkT\left(\frac{1+m}{2}x\ln\frac{1+m}{2}x + \frac{1-m}{2}x\ln\frac{1-m}{2}x + (1-x)\ln(1-x)\right) \tag{22}$$

1.6

f) Express the free energy as a dimensionless free energy per particle, g. Define new scaled parameters.

$$g \equiv \frac{G}{NkT}, \quad \alpha \equiv \frac{zJ}{kT}, \quad \mu \equiv \frac{\Delta\mu}{kT} \equiv \frac{\mu_A - \mu_B}{kT} \tag{3}$$

Now, we want to manipulate the expression for the free energy that we just got. Since we are ingested in a free energy per particle, we know to divide by N and we also know that we can divide by kT, which has dimensions of energy, to get a dimensionless quantity:

$$g \equiv \frac{G}{NkT} = -\frac{1}{2}\alpha(mx)^2 - \mu x + \left(\frac{1+m}{2}x\ln\frac{1+m}{2}x + \frac{1-m}{2}x\ln\frac{1-m}{2}x + (1-x)\ln(1-x)\right) \tag{23}$$

where we have used the definitions of α and μ to get the final expression.

2

2. Analyzing the free energy

2.1

a) State the conditions for phase equilibrium.

Normally, for phase equilbium, we would want the same free energy per particle, chemical potential, and temperature. However, here all we have to require is that the dimensionless free energy per particle be minimzzed with respect to the variables m and x.

2.2

b) Minimize the free energy with respect to m to obtain an implicit relationship between m and x, i.e. $F(m, x) = 0$.

Given the free energy per particle for this system, we can minimize it with respect to m, to get:

$$\frac{\partial g}{\partial m} = -1.0\alpha m x^2 - \frac{x \log\left(x\left(\frac{1}{2} - \frac{m}{2}\right)\right)}{2} + \frac{x \log\left(x\left(\frac{m}{2} + \frac{1}{2}\right)\right)}{2} \tag{24}$$

We set this all equal to 0 in order to sets by the minimization condition:

$$-1.0\alpha m x^2 - \frac{x \log\left(x\left(\frac{1}{2} - \frac{m}{2}\right)\right)}{2} + \frac{x \log\left(x\left(\frac{m}{2} + \frac{1}{2}\right)\right)}{2} = 0 \tag{25}$$

Additionally we can multiply by a factor of $\frac{2}{x}$ and also combine the logarithms and move the non logarithmic term to the other side to get:

$$\log \frac{x\left(\frac{m}{2} + \frac{1}{2}\right)}{x\left(\frac{1}{2} - \frac{m}{2}\right)} = 2\alpha m x \tag{26}$$

There is a consolation of the xs inside the logarithm, so we can cancel them out and then exponent both sides:

$$\frac{\left(\frac{m}{2} + \frac{1}{2}\right)}{\left(\frac{1}{2} - \frac{m}{2}\right)} = e^{2\alpha m x} \tag{27}$$

We can then multiply both sides by the denominator:

$$\left(\frac{m}{2} + \frac{1}{2}\right) = e^{2\alpha m x}\left(\frac{1}{2} - \frac{m}{2}\right) \tag{28}$$

Multiplying through by a factor of 2:

$$m + 1 = e^{2\alpha m x}(1 - m) \tag{29}$$

This gives:

$$m\left(1 + e^{2\alpha m x}\right) = e^{2\alpha m x} - 1 \tag{30}$$

We can then isolate m:

$$m = \frac{e^{2\alpha m x} - 1}{1 + e^{2\alpha m x}} \tag{31}$$

Then we multiply rhs by a factor of $\frac{e^{-\alpha m x}}{e^{-\alpha m x}}$:

$$m = \frac{e^{\alpha m x} - e^{-\alpha m x}}{e^{-\alpha m x} + e^{\alpha m x}} \tag{32}$$

which gives:

$$m = \tanh(\alpha m x) \tag{33}$$

2.3

c) Minimize the free energy with respect to x to obtain an equation for $\mu = \mu(m, x)$.

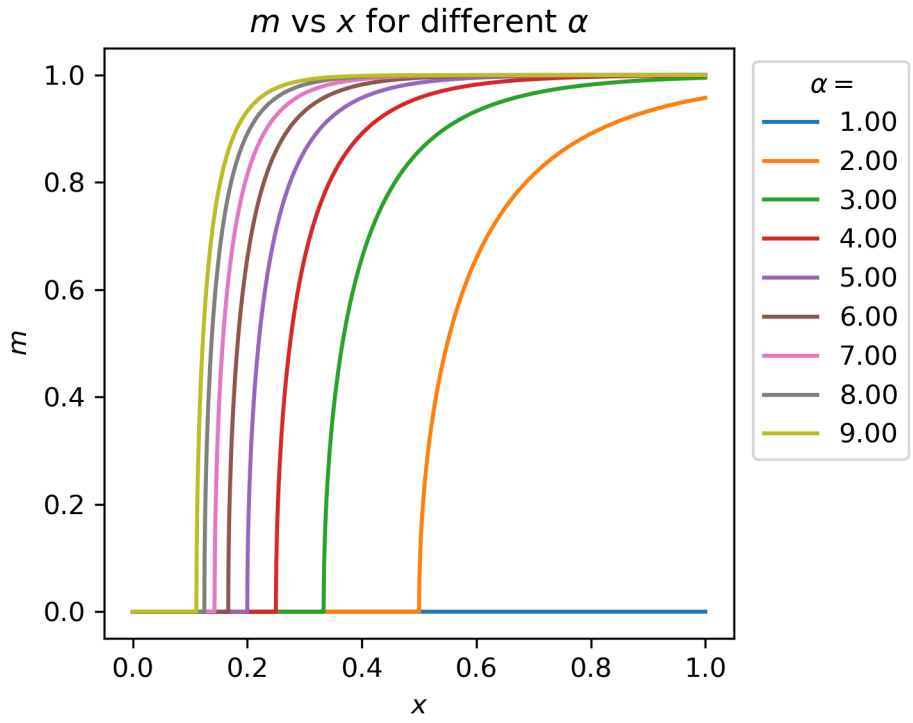
We can minimize the free energy with respect to x to get:

$$\frac{\partial g}{\partial x} = -1.0\alpha m^2 x - \mu + \left(\frac{1}{2} - \frac{m}{2}\right)\log\left(x\left(\frac{1}{2} - \frac{m}{2}\right)\right) + \left(\frac{m}{2} + \frac{1}{2}\right)\log\left(x\left(\frac{m}{2} + \frac{1}{2}\right)\right) - \log(1 - x) \tag{34}$$

We can set this equal to 0 to get and solve for μ:

$$-1.0\alpha m^2 x + \left(\frac{1}{2} - \frac{m}{2}\right)\log\left(x\left(\frac{1}{2} - \frac{m}{2}\right)\right) + \left(\frac{m}{2} + \frac{1}{2}\right)\log\left(x\left(\frac{m}{2} + \frac{1}{2}\right)\right) - \log(1 - x) = \mu \tag{35}$$

```python
from sympy import symbols, diff, ln, latex, solve

# Define symbols
m, x, alpha, mu = symbols('m x alpha mu')

# Define the expression for g
g = -1/2 * alpha * (m * x)**2 - mu * x + ((1 + m)/2)*x*ln(((1 + m)/2)*x) + ((1 - m)/2)*x*ln(((1 - m)/2)*x) + (1 - x)*ln(1 - x)

# Calculate the first derivatives
dg_dm = diff(g, m)
dg_dx = diff(g, x)

# Solve the equation dg_dx = 0 for mu to get mu as a function of m, x, and alpha
mu_solution = solve(dg_dx, mu)

print(latex(mu_solution))
```



2.4

d) With these equations in hand, plot the following:

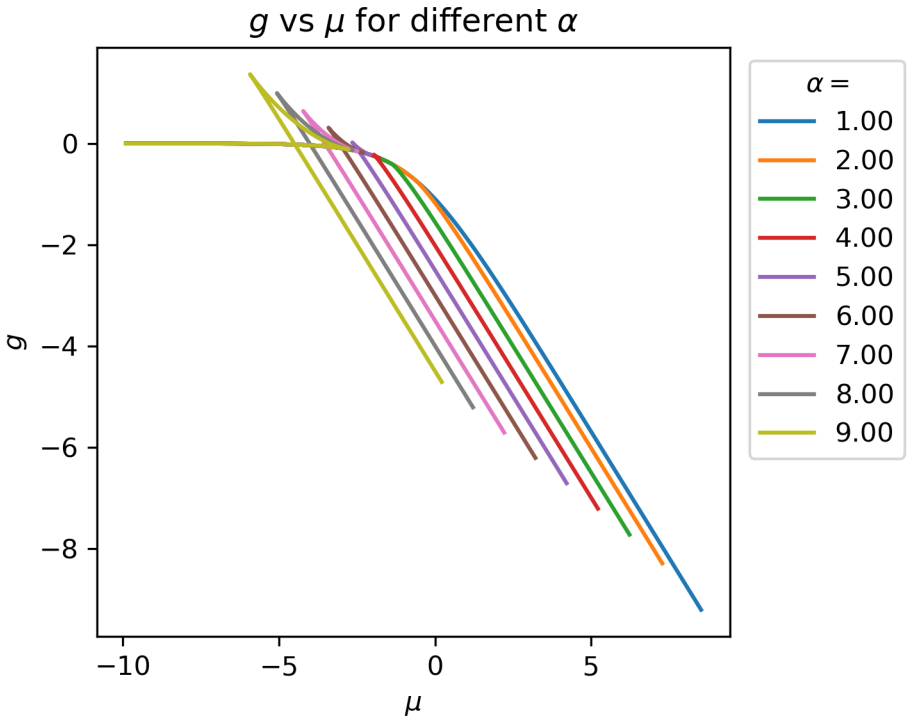
2.4.1

i. m vs x for $\alpha = \{1, 2, \ldots, 9\}$.

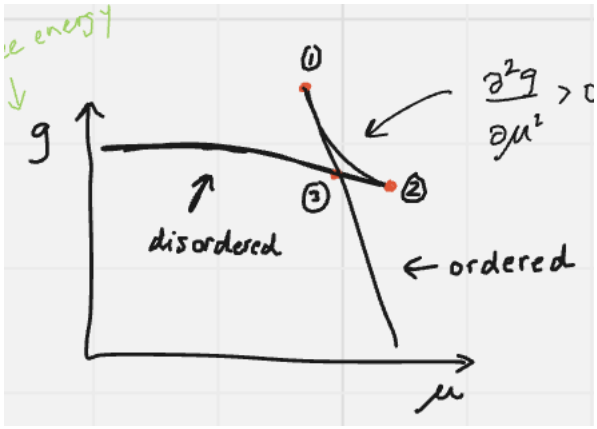
2.4.2

ii. g vs μ for $\alpha = \{1, 2, \ldots, 9\}$.

iii. For $\alpha = 9$, label the following on the g vs μ plot: (1) stable disordered state, (2) metastable disordered state, (3) stable ordered state, (4)


g vs μ for different α
0
−2
−4
−6
−8
g
−10
−5
0
5
μ
$\alpha =$
1.00
2.00
3.00
4.00
5.00
6.00
7.00
8.00
9.00

metastable ordered state, (5) unstable state, (6) spinodal of ordered state, (7) spinodal of disordered state, (8) binodal (coexistence).

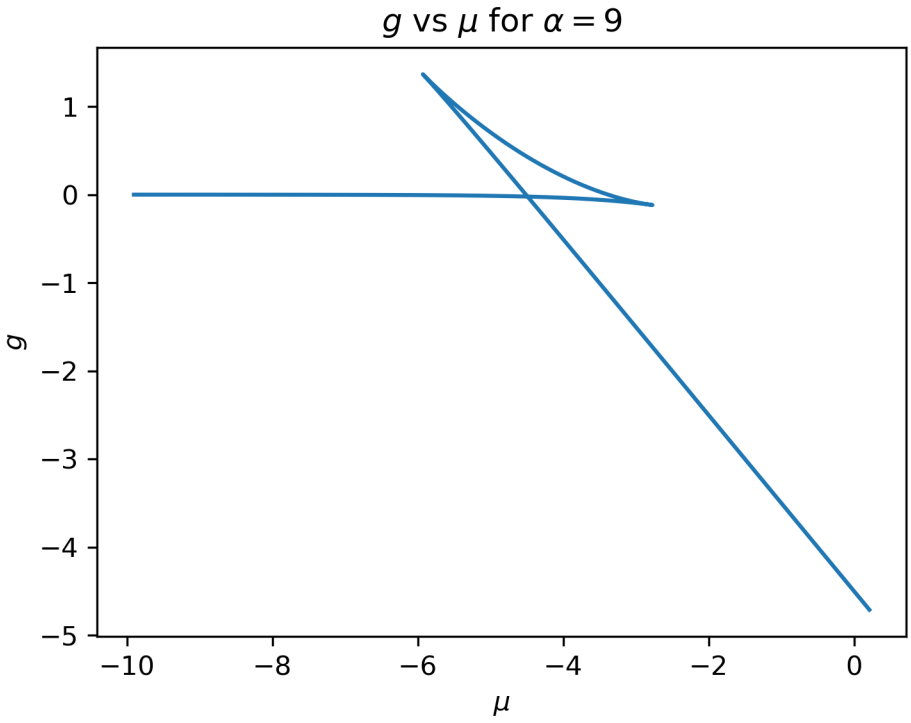


First of all, we will note that when we have disordered and ordered phases, we have $\frac{\partial^2 g}{\partial \mu^2} < 0$, or the curves are concave down. We will start by going along the disordered phase where mu starts at zero. The definition of metastability is when the previously stated thermodynamic condition is met, but the system does not have the lowest free energy. This is clearly the case between the points 2 and 3. At 2, the domain of stability for the disordered phase is reached and the system achieves the spinodal of the disordered phase. In the path between 1 and 2 there is the curve of instability. Now, we will start by considering the curve of the ordered phase. It starts off as stable, until it reaches 3, where it enters a domain of metastability (as discussed earlier) and that goes until 1, which is the spinodal of the ordered phase. In this plot, 3 shows the coexistence of the disordered and ordered phases, which is called the binodal point.

iv. Plot the phase diagram in $\alpha - \mu$ space. Use the domains $1 \leq \alpha \leq 9$ and $0 < x < 1$ to construct the phase diagram. Please use a sufficiently fine grid in each. How does the addition of impurity influence the phase transition? Does this make sense?

Note that above a certain chemical potential, the phase transition will be second order. Below this chemical potential, the phase transition is first order. You will need to obtain the spinodal(s) and binodal in the first order region, and the criticalline or λ-line in the second order region. The point which joins the two regions is called the tri-critical point.

v. Extra Credit (5 Pts): Plot the phase diagram in $\alpha^{-1} - x_B$ space, where $x_B = 1 - x$ is the type B number fraction in the system.


g vs μ for $\alpha = 9$
1
0
−1
−2
−3
−4
−5
g
−10
−8
−6
−4
−2
0
μ

3

3. 2-dimensional Monte-Carlo simulation in canonical ensemble

Define a (40×40) square lattice with $N_A = xN$ interacting spins that can take $s = \pm 1$ and $N_B = N - N_A = (1 - x)N$ non-interacting spins that can take $s = 0$.

3.1

a) Write the Hamiltonian in the canonical ensemble as a function of the spin-state of the system assuming nearest neighbor interactions. Don't forget to account for over counting.

The Hamiltonian is the same as earlier, but with the exception that here we are just working in the canonical ensemble:

$$\mathcal{H} = -\frac{1}{2} J \sum_{i=0}^{N_A} \sum_{j=0}^{z-1} \tau_i \tau_j \tag{36}$$

b) Write a program to perform Monte Carlo simulations at fixed (N, x, T). For all attempted moves, use the Metropolis-Hastings algorithm to make decisions. Use periodic boundary conditions! Your code should do the following:

i. Start from a random initial state at a given x. The interacting spins should be assigned up or down each with probability 1/2.

ii. In each MC iteration, loop over all Type A spins and attempt "flipping" moves.

iii. In each MC iteration, loop over all spins and attempt "swapping" moves with a randomly chosen spin in the immediate vicinity (orthogonally or diagonally touching). Each spin has 4 orthogonally touching spins and 4 spins that are diagonally touching. When attempting a swap, randomly choose 1 out of these 8 possibilities.

iv. Plot the total energy E vs iteration number i.

v. Plot the relative magnetization m vs iteration number i.

vi. Print a visual snapshot of both the initial and final configurations. (in MATLAB, for a matrix A, imagesc(A) will display the matrix as an image).

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors
import random

class MonteCarlo():

    def __init__(self, x, alpha, n_iter):
        self.x = x  # Fraction of interacting spins
        self.beta = alpha  # Inverse temperature
        self.ni = int(n_iter)  # Number of iterations
        self.N = 1600
        self.ndim = int(np.sqrt(self.N))  # Assuming a square lattice
        self.Na = int(self.N*x)
        self.Nb = int(self.N - self.Na)

        np.random.seed(24)

        # Initialize a two-dimensional square lattice
        state = np.zeros((self.ndim, self.ndim), dtype=int)

        # Randomly choose Na positions in the lattice for the interacting spins
        interacting_indices = np.random.choice(self.N, self.Na, replace=False)

        # Convert linear indices to 2D indices and assign s = +/- 1 to these positions
        for index in interacting_indices:
            i, j = divmod(index, self.ndim)
            state[i, j] = np.random.choice([-1, 1])









        self.oldstate = np.copy(state)
        self.newstate = np.copy(state)

        self.initial = np.copy(state)
        self.E = 0
        for i in range(40):
            for j in range(40):
                self.E += self.local_hamiltonian(i, j, self.
```

```
    oldstate)
42          self.E /= 2 #overcounting
43          self.dE = 0 #placeholder
44
45      def plot_state(self, state, n_iter, title="Lattice State"
    ):
46          plt.figure(figsize=(6, 6))
47          plt.title(title)
48          plt.imshow(state, cmap='viridis', interpolation='
    nearest')
49          plt.colorbar(label='Spin Value')
50          # Dynamically generate the filename using the
    iteration number
51          filename = f"lattice_state_{n_iter}_{self.x}.png"
52          plt.savefig(filename)
53          plt.close()  # Close the figure after saving to free
    up memory
54          return
55
56
57
58
59      def run_iter(self):
60          # first consider the flips
61          for i in range(self.ndim):
62              for j in range(self.ndim):
63                  # consider all possible flips only for the
    interacting spins
64                  if abs(self.oldstate[i][j]) == 1:
65                      self.flip(i, j)
66                      # calculate the change in energy
    associated
67                      self.dE = self.local_hamiltonian(i, j,
    self.newstate) - self.local_hamiltonian(i, j, self.
    oldstate)
68                      self.dE *= self.beta
69                      # except or reject the switch based off
    of the metropolis-hastings algorithm
70                      if self.update():
71                          self.accept()
72                      self.newstate = np.copy(self.oldstate)
73
74          # now consider the swaps
75          for i in range(self.ndim):
76              for j in range(self.ndim):
```

```
77                # no need to check whether the spins are
    interacting or not
78                neighbor = self.swap(i, j)
79                # calculate the change new energy associated
80                new_energy = self.local_hamiltonian(i, j,
    self.newstate) + self.local_hamiltonian(neighbor[0],
    neighbor[1], self.newstate)
81                old_energy = self.local_hamiltonian(i, j,
    self.oldstate) + self.local_hamiltonian(neighbor[0],
    neighbor[1], self.oldstate)
82                self.dE = new_energy - old_energy
83                self.dE *= self.beta
84                # except or reject the switch based off of
    the metropolis-hastings algorithm
85                if self.update():
86                    self.accept()
87                self.newstate = np.copy(self.oldstate)
88        return
89
90
91
92
93
94
95    def flip(self, i, j):
96        #
    ################################################################

97        # THIS FUNCTION SHOULD PERFORM A FLIP AT INDEX i AND
    j, AND    #
98        # COMPUTE THE ASSOCIATED ENERGY CHANGE WITH THiS FLIP
              #
99        #
    ################################################################

100       # changed the sign at index i, j
101       self.newstate[i][j] = -self.newstate[i][j]
102       return
103
104
105
106   def swap(self, i, j):
107       #
    ################################################################
```

```
108         # THIS FUNCTION SHOULD PERFORM A SWAP OF INDEX i AND
    j, AND    #
109         # COMPUTE THE ASSOCIATED ENERGY CHANGE WITH THiS SWAP
                #
110         #
    ##################################################################

111         # Define offsets for the 8 neighbors: top, bottom,
    left, right, and the 4 diagonals
112         offsets = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1)
    , (-1, 1), (1, -1), (1, 1)]
113         # compute their indices given the periodic boundary
    conditions, which means taking mod self.ndim
114         neighbors = [((i + di) % self.ndim, (j + dj) % self.
    ndim) for di, dj in offsets]
115         # choose a random neighbor using random.choice
116         neighbor = random.choice(neighbors)
117         # swap the spins
118         self.newstate[i][j], self.newstate[neighbor[0]][
    neighbor[1]] = self.newstate[neighbor[0]][neighbor[1]],
    self.newstate[i][j]
119         # returned the index of the neighbor that was chosen
120         return neighbor
121
122
123
124     def update(self):
125
126         #
    ##################################################################

127         # THIS FUNCTION SHOULD CHECK WHETHER OR NOT A GIVEN
    OPERATION  #
128         # SHOULD BE ACCEPTED. THIS IS YOUR METRPOLIS-HASTING
    ALGORITHM #
129         #
    ##################################################################


130
131         condition = False
132         if self.dE <= 0 or np.random.rand() <= np.exp(-self.
    dE):
133             condition = True
134         # REGARDLESS OF THE OUTCOME ABOVE, YOU STILL NEED TO
    INITILIASE
```

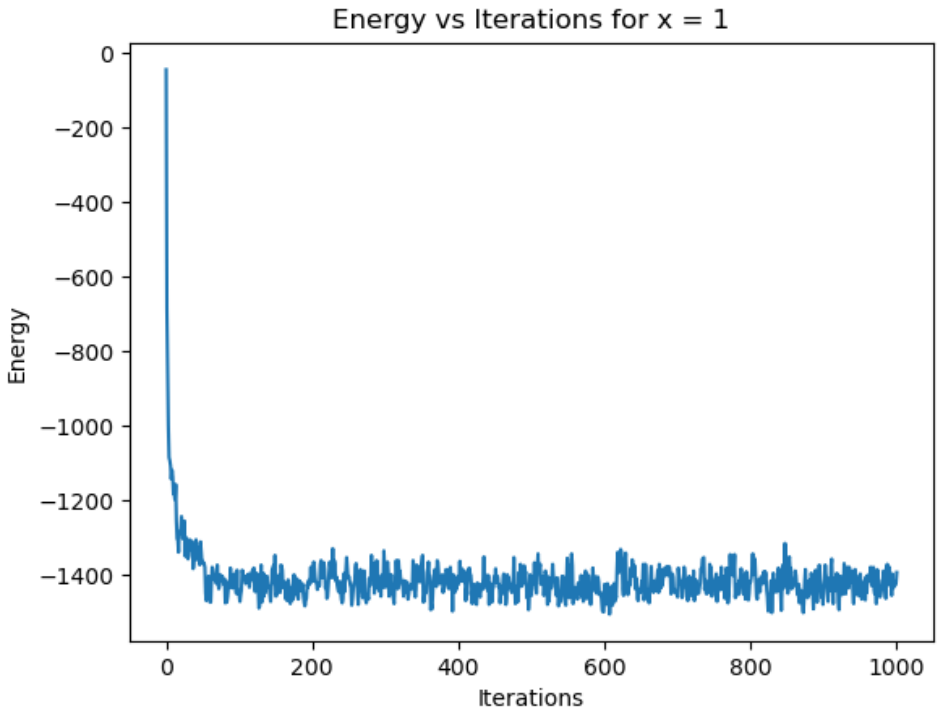
```
135         # YOUR NEW STATE FOR THE NEXT ITERATION
136         # self.newstate = np.copy(self.oldstate)
137         return condition
138 
139 
140 
141     def accept(self):
142         #
    ##################################################################
143         # THIS FUNCTION SHOULD UPDATE YOUR OLDSTATE AND
    ENERGY GIVEN   #
144         # A CHANGE WAS ACCEPTED
                $
145         #
    ##################################################################
146         self.oldstate = np.copy(self.newstate)
147         self.E += self.dE
148         return
149 
150 
151 
152     def local_hamiltonian(self, i, j, state):
153         E = 0
154         #
    ##################################################################
155         # THIS FUNCTION SHOULD CALCULATE THE CHANGE IN THE
    HAMILTONIAN #
156         # LOCALLY AROUND i and j. REMEMBER TO ACCOUNT FOR
    PERIOD BOUN- #
157         # -DARY CONDITIONS.
              #
158         #
    ##################################################################
159         # we only add a contribution to the energy if the
    spin is interacting
160         if abs(state[i][j]) == 1:
161             # Define offsets for the 4 neighbors: top, bottom
    , left, right,
162             offsets = [(-1, 0), (1, 0), (0, -1), (0, 1)]
163             # compute their indices given the periodic
    boundary conditions, which means taking mod self.ndim
```

```
164             neighbors = [((i + di) % self.ndim, (j + dj) %
    self.ndim) for di, dj in offsets]
165             # check if the neighbor is in interacting spin
166             for neighbor in neighbors:
167                 if abs(state[neighbor[0]][neighbor[1]]) == 1:
168                     E += -state[i][j] * state[neighbor[0]][
    neighbor[1]]
169         return E
170
171
172     def get_energy(self):
173         return self.E
174
175     def get_m(self):
176         #
    ##################################################################
177         # THiS FUNCTION SHOULD CALCULATE THE AVERAGE
    MAGNETISATION OF  #
178         # YOUR SYSTEM AT A GIVEN ITERATION.
                #
179         #
    ##################################################################
180         magnetization = (1/self.Na) * np.sum(self.oldstate)
181         return magnetization
182
183     def run(self):
184         self.initial = np.copy(self.oldstate)
185
186         self.ms = [self.get_m()]
187         self.Es = [self.get_energy()]
188
189         for i in range(self.ni):
190             # plot the state if it is the initial state
191             if i == 0:
192                 self.plot_state(self.oldstate, i, title='
    Initial Lattice State')
193
194             self.run_iter()
195
196
197             self.Es.append(self.get_energy())
198             self.ms.append(self.get_m())
199             # plot the state if it is the final state
```

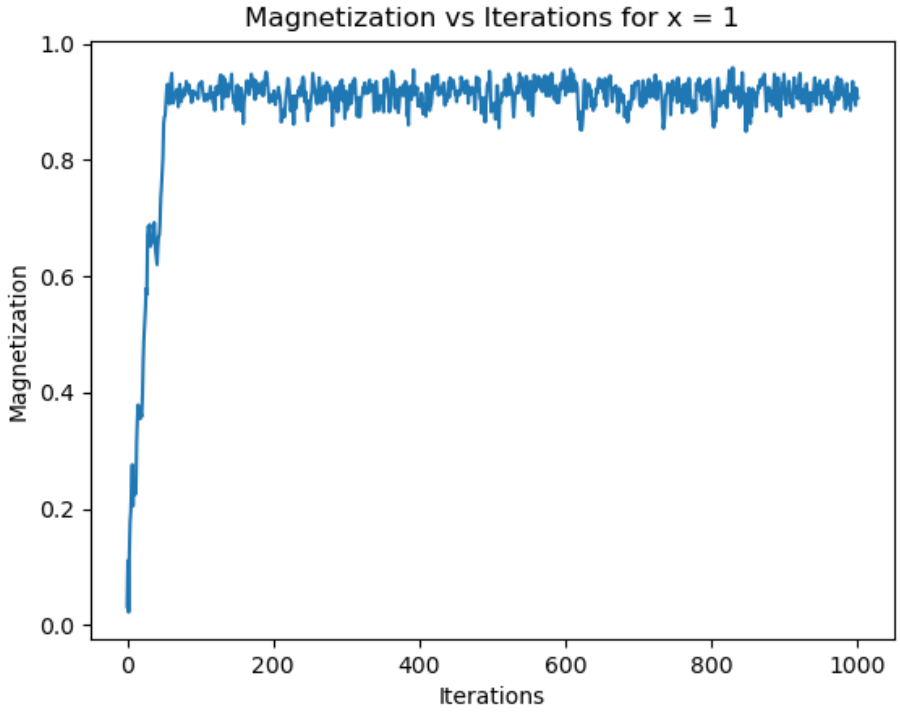
```
            if i == self.ni - 1:
                self.plot_state(self.oldstate, i, title='Final Lattice State')


        self.final = self.oldstate
def plot_magnetization_vs_iterations(x, alpha, n_iter):
    plt.figure()
    plt.plot(MC.ms)
    plt.xlabel('Iterations')
    plt.ylabel('Magnetization')
    plt.title(f'Magnetization vs Iterations for x = {x}')
    plt.savefig(f'magnetization_vs_iterations_{x}.png')
    return
def plot_energy_vs_iterations(x, alpha, n_iter):
    plt.figure()
    plt.plot(MC.Es)
    plt.xlabel('Iterations')
    plt.ylabel('Energy')
    plt.title(f'Energy vs Iterations for x = {x}')
    plt.savefig(f'energy_vs_iterations_{x}.png')
    return

x = [1, 0.75, 0.4]
alpha = [0.5, 3, 1]
n_iters = [1e3, 1e4, 1e4]

# make a dictionary for all of the trials whose entries are given in the correct order in the lists above
trials = {'x': x, 'alpha': alpha, 'n_iters': n_iters}
for i in range(3):
    MC = MonteCarlo(trials['x'][i], trials['alpha'][i], trials['n_iters'][i])
    MC.run()
    # plot the energies vs the number of iterations
    plot_energy_vs_iterations(trials['x'][i], trials['alpha'][i], trials['n_iters'][i])
    # plot the magnetization vs the number of iterations
    plot_magnetization_vs_iterations(trials['x'][i], trials['alpha'][i], trials['n_iters'][i])



# print(MC.initial)

```

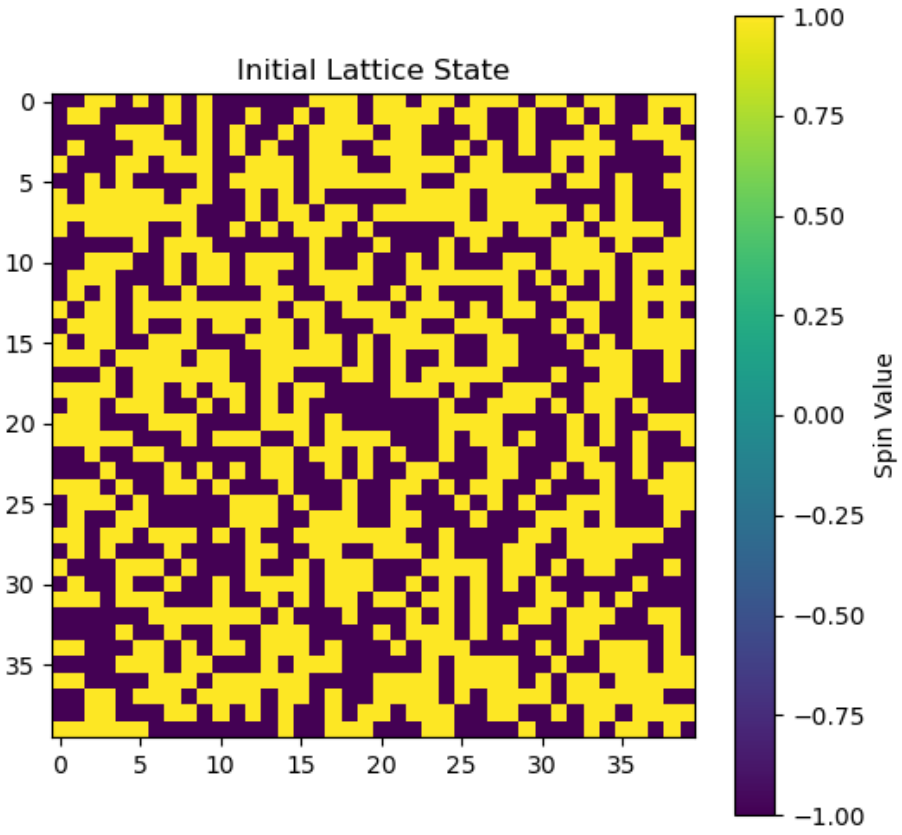
```
# print(MC.final)
# print(MC.ms)
# plot the energies vs the number of iterations
```

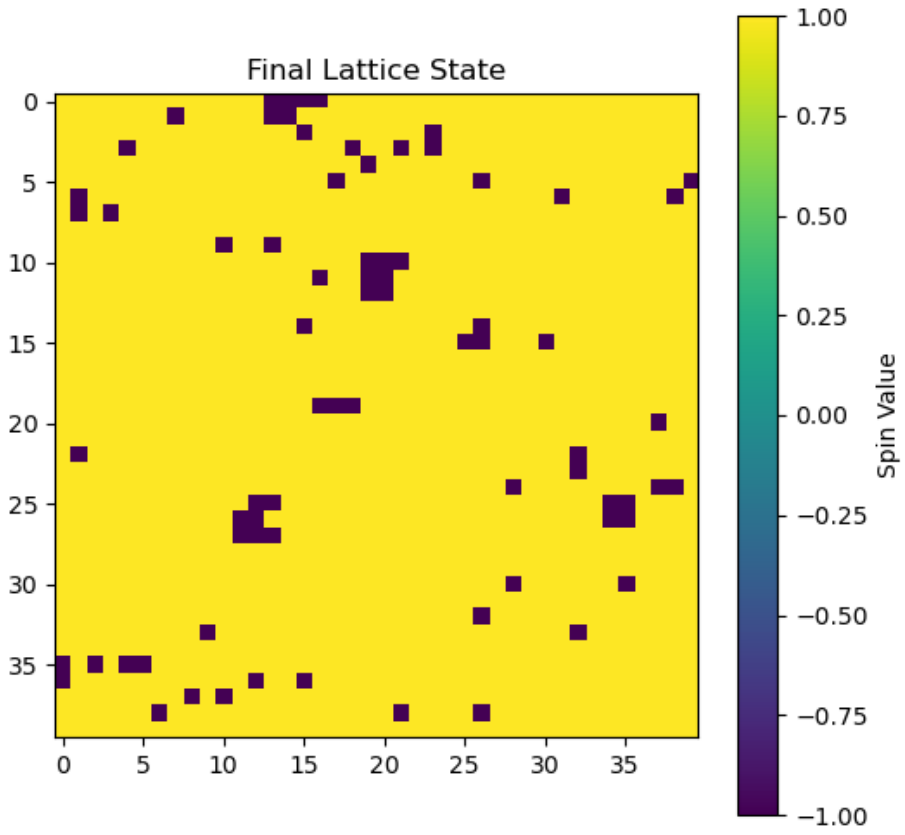
c) Run the following simulations. For each simulation provide the following: (1) E vs i plot, (2) m vs i plot, (3) snapshots of the initial and final configurations.
I will show the plots for the simulations in the order that is given.

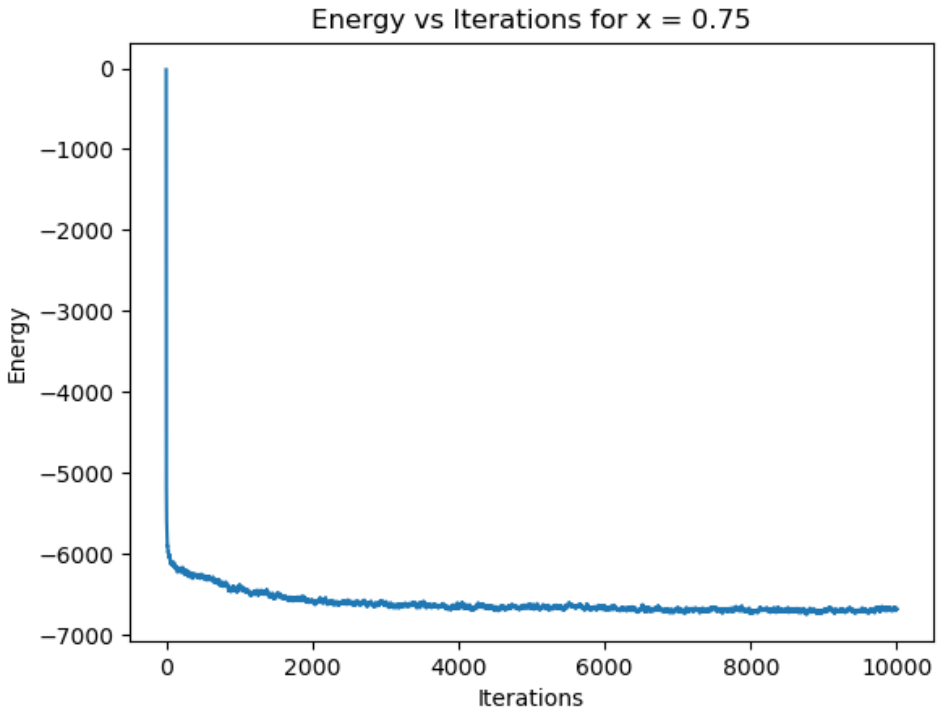


i. $x = 1, J/kT = 0.5, n_{\text{iters}} = 10^3$.

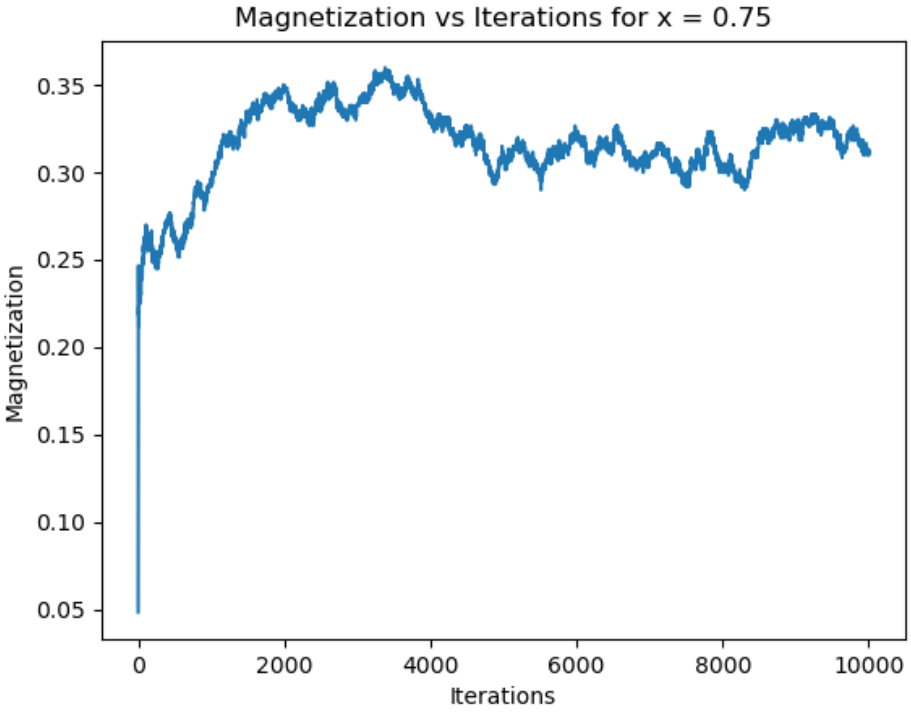

Magnetization vs Iterations for x = 1
Magnetization
Iterations
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000

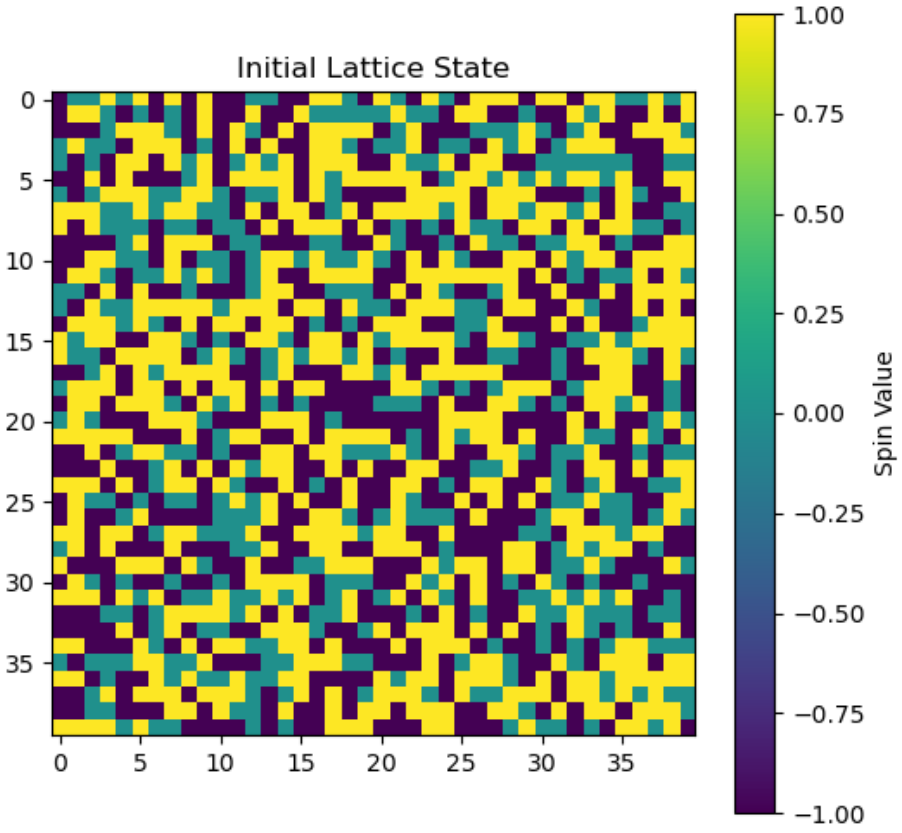

Initial Lattice State
0
5
10
15
20
25
30
35
0
5
10
15
20
25
30
35
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Spin Value

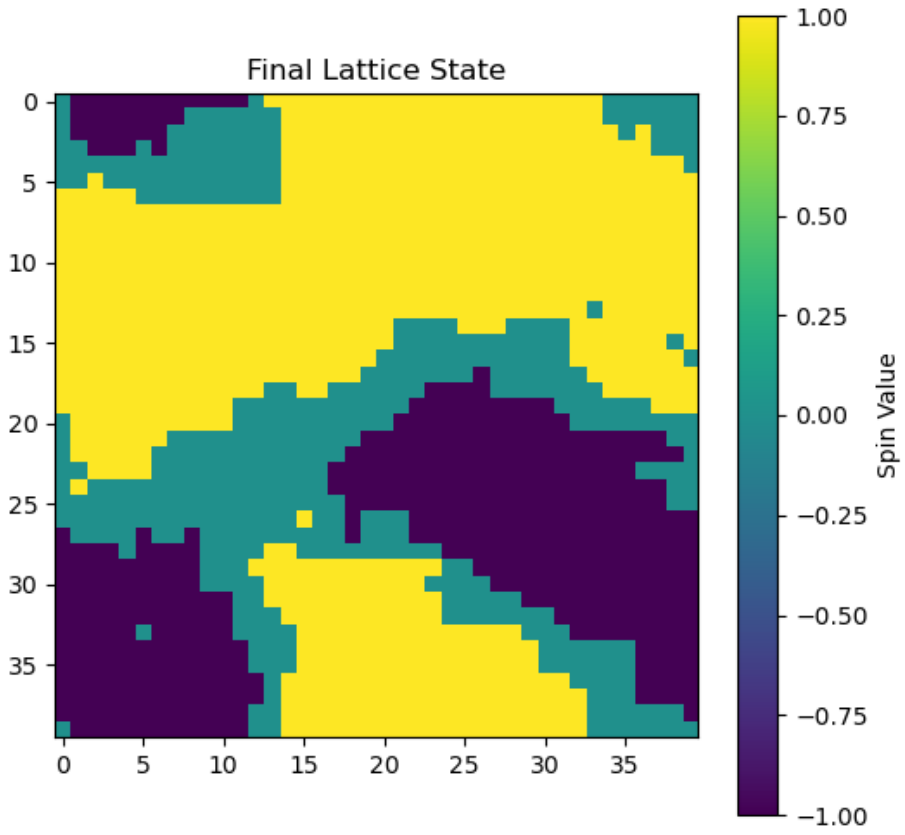

Final Lattice State
0
5
10
15
20
25
30
35
0
5
10
15
20
25
30
35
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Spin Value

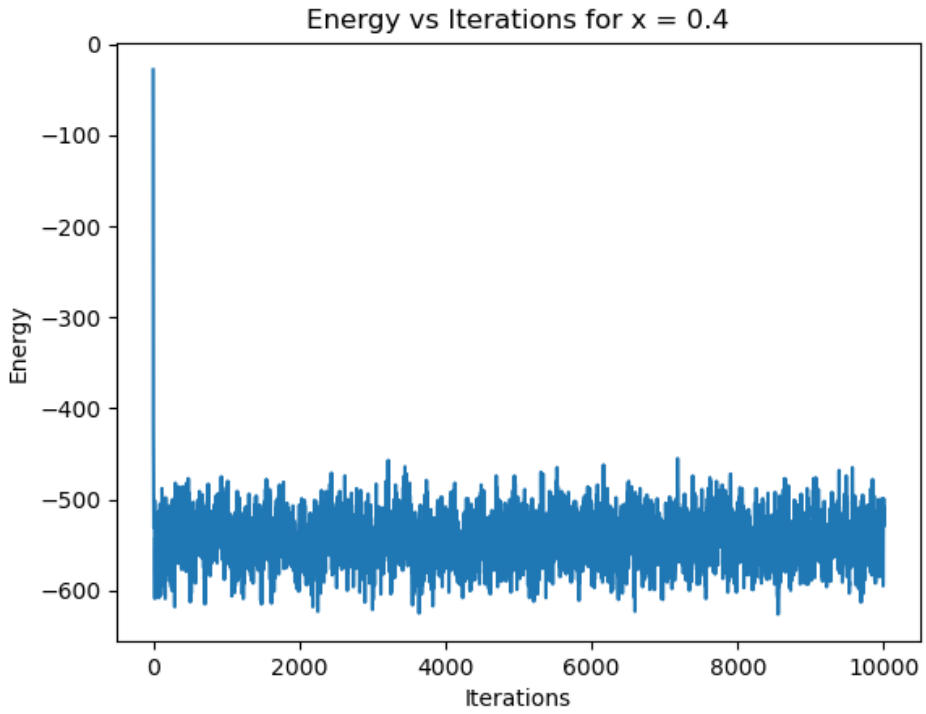


ii. $x = 0.75, J/kT = 3, n_{\text{iters}} = 10^4$.

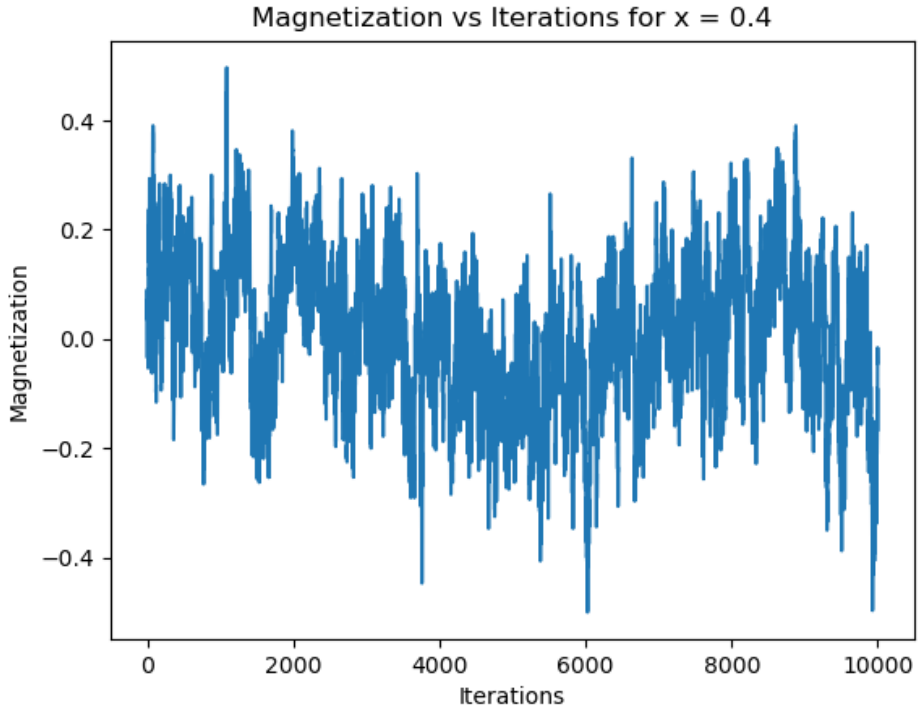

Magnetization vs Iterations for x = 0.75
Magnetization
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0
2000
4000
6000
8000
10000
Iterations

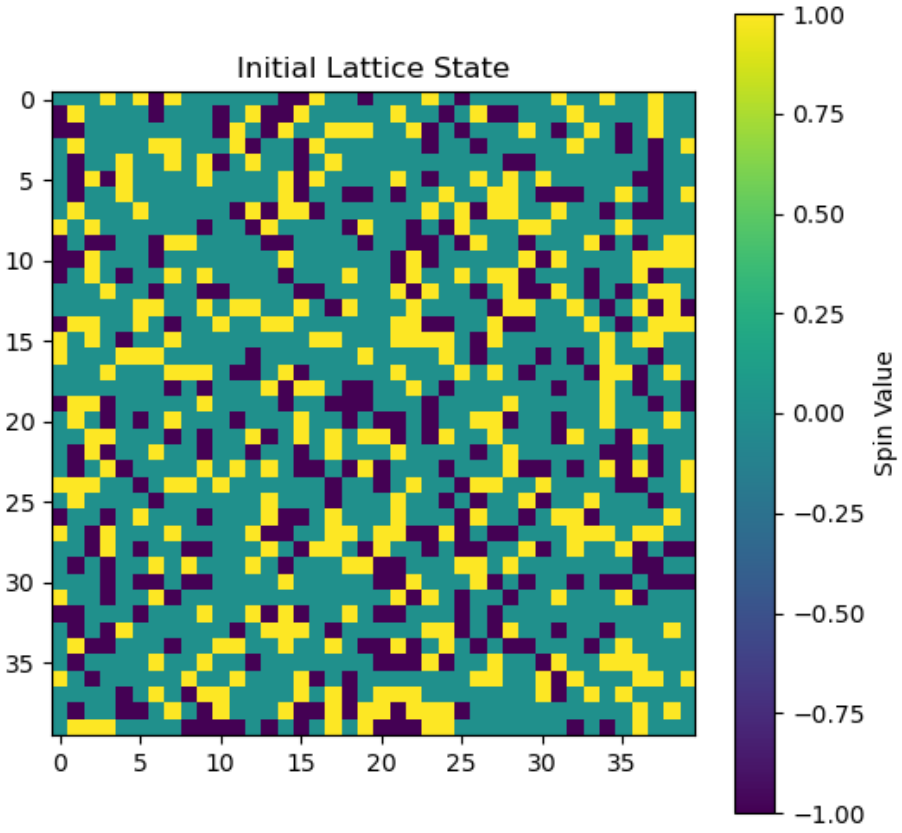

Initial Lattice State
0
5
10
15
20
25
30
35
0
5
10
15
20
25
30
35
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Spin Value

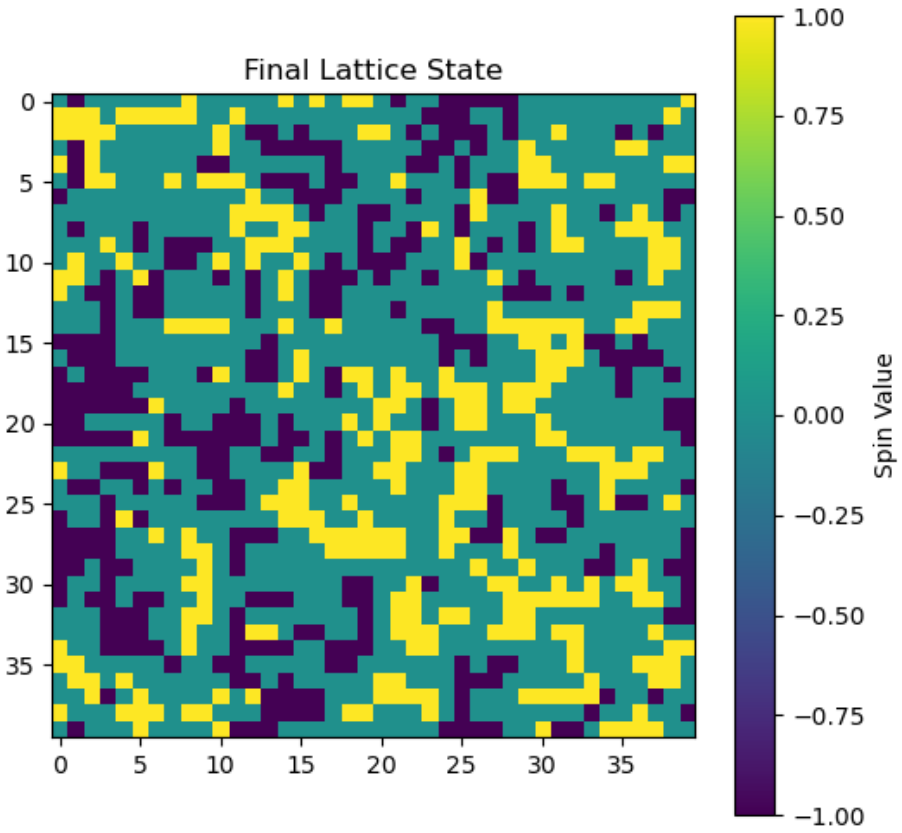

Final Lattice State
0
5
10
15
20
25
30
35
0
5
10
15
20
25
30
35
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Spin Value

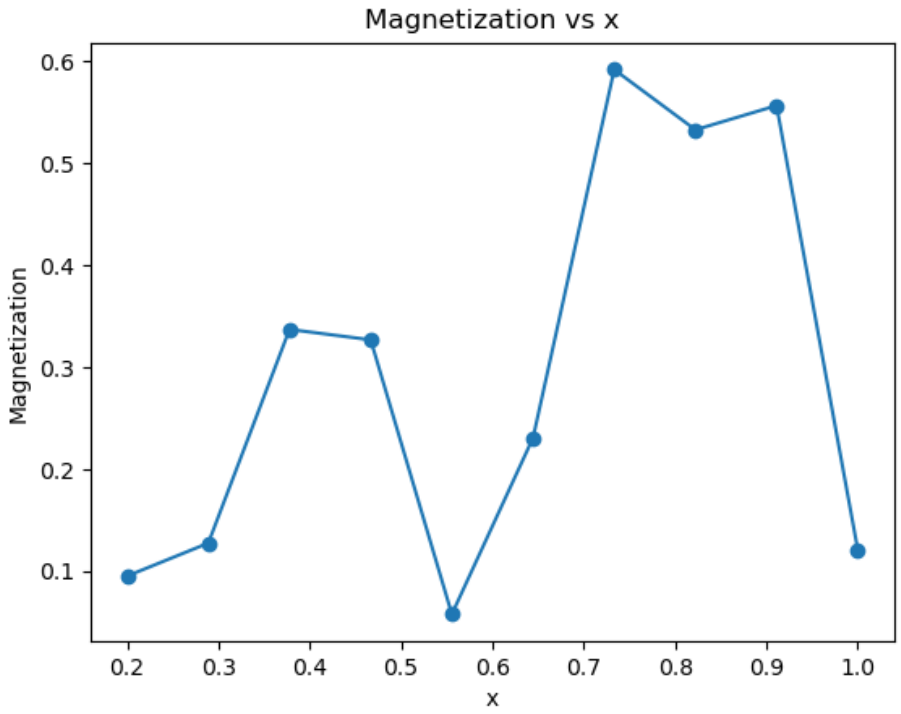


iii. $x = 0.4, J/kT = 1, n_{\text{iters}} = 10^4$.


Magnetization vs Iterations for x = 0.4
Magnetization
0.4
0.2
0.0
−0.2
−0.4
0
2000
4000
6000
8000
10000
Iterations


Initial Lattice State
0
5
10
15
20
25
30
35
0
5
10
15
20
25
30
35
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Spin Value


Final Lattice State
0
5
10
15
20
25
30
35
0
5
10
15
20
25
30
35
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Spin Value



d) Extra Credit (8 pts): Plot m vs x for $J/kT = 1$. You should use $n_{\text{iters}} \geq 10^4$ for the best results. You will need to average over the trajectory once equilibrium is reached to get an $\langle m \rangle$ for each x.

```
# Define parameters
x = np.linspace(0.2, 1, 10)
alpha = 3
n_iters = int(1e4)  # Ensure n_iters is an integer

# Initialize an empty list for plotting
x_m = []

# Loop over different values of x
for i in range(len(x)):
    MC = MonteCarlo(x[i], alpha, n_iters)
    MC.run()
    # Get the absolute value of the average value of the magnetization over the last 500 iterations
    m = np.mean(np.abs(MC.ms[500:]))
```

```
    # Store the value in a tuple like (x, m)
    x_m.append((x[i], m))

# Extract x and m values for plotting
x_values, m_values = zip(*x_m)

# Plot the information
plt.figure()
plt.plot(x_values, m_values, marker='o')  # Added marker for
    clarity
plt.xlabel('x')
plt.ylabel('Magnetization')
plt.title('Magnetization vs x')
plt.savefig('magnetization_vs_x_2.png')
```