HW15

白博臣、何骐多、夏营

图 1: 白博臣

图 2: 何骐多

图 3: 夏营

1 Problem1

当模拟一维 Ising 模型时，我们需要实现以下步骤：
初始化系统：设置自旋格子数，外界温度T，外加磁场h，以及初始自旋构型
计算能量：根据 Ising 模型的能量函数计算每种构型的能量。
选择随机自旋：随机选择一个自旋进行翻转。
计算翻转后的能量变化：计算翻转后系统能量与翻转之前能量的差值 Δ E。
判断是否接受翻转：如果 < 0，则接受翻转；否则，以概率 $e^{-/kT}$ 接受翻转。
重复上述步骤，直到系统达到平衡

1.1 a

对于此题，我们初始设置自旋格子数为20，外界温度为$1.0\frac{J}{k_B}$，外加磁场$h = 0$，以及初始所有自旋格子的自旋方向均为向上，即$spin = 1$。

由以下代码实现计算周期性边界的系统能量：

```
def energy(spins, h):
    n = len(spins)
    energy = 0
    for i in range(n):
        energy += -(spins[(i - 1) % n] + spins[(i + 1) % n])*spin[i] - h *
    ↪ spins[i]
    return energy/2

```

以下代码实现随机选取一个格子进行翻转并利用Metropolis-Hastings算法来判断是否接受这一翻转结果：

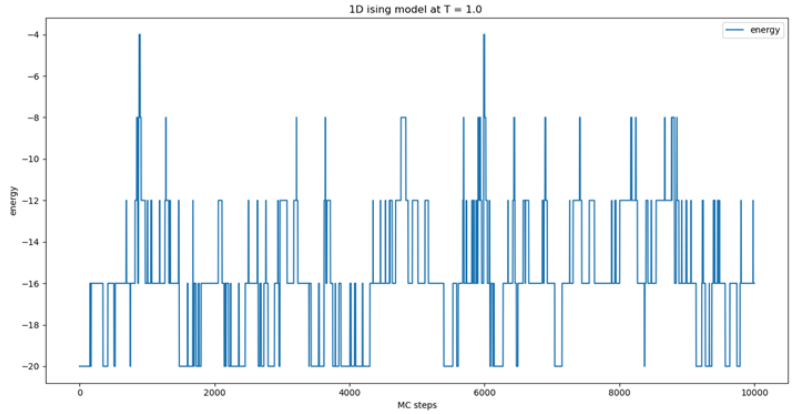
```
# 实现自旋翻转
def flip(spins, beta):
    n = len(spins)
    # 随机选取翻转的自旋格子
    i = np.random.randint(n)
    s = -spins[i]

    deltaE = 2 * spins[i] * (spins[(i - 1) % n] + spins[(i + 1) % n])
    if deltaE <= 0:
        spins[i] = s
    elif np.random.random() < np.exp(-beta * deltaE):
        spins[i] = s

    return spins

```

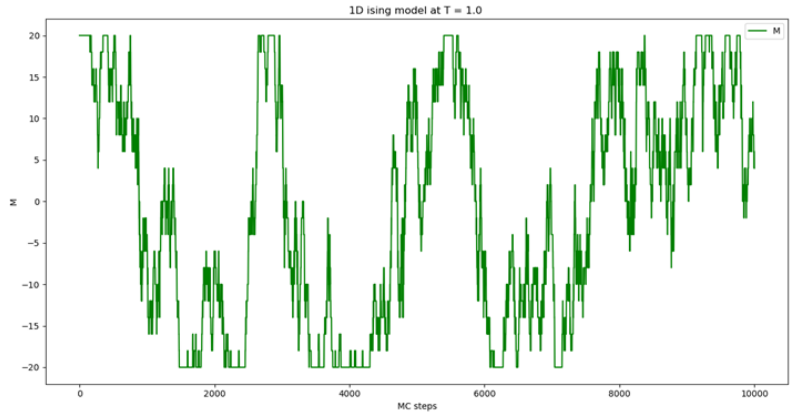
以下代码对MCMC算法模拟Ising model进行实际运算：

```
# 实现MCMC算法
def MCMC(spins, T, n, h):
    beta = 1.0 / T
    energies = np.array([])
```

```
    m = np.array([])
    for i in range(n):
        spins = flip(spins, beta)
        e = energy(spins, h)
        m = np.append(m, sum(spins))
        energies = np.append(energies, e)
    return energies, m

```

而后只需将初始设定的条件代入运算即可，结果如下：

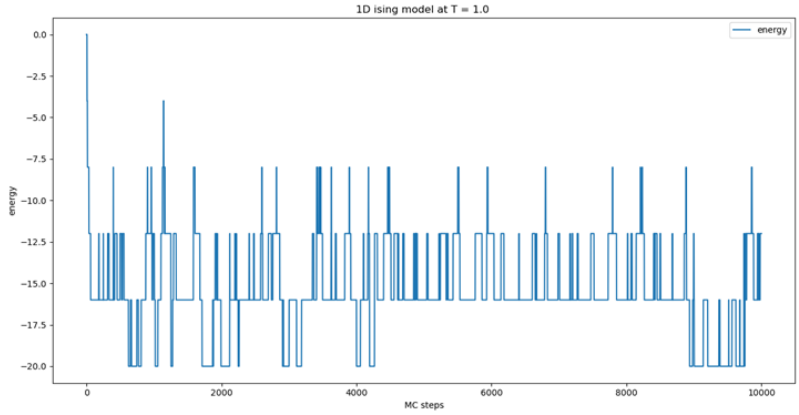


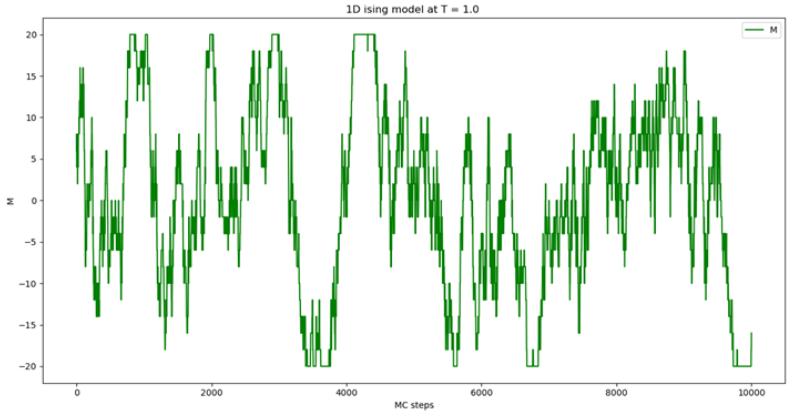


从图上我们可以大致的看出，在经历了100步左右，系统趋近于平衡状态。

1.2 b

将自旋的初始构型随机给定，即每个自旋格子的自旋方向等可能的向上或者向下。我们利用python numpy库中的random.choice([-1,1],size=N)函数来实现这一条件，其余步骤与上一问相同，有结果如下：





1.3 c

由以下代码实现对于T属于[0.5,5]区间的遍历，同时记录¡E¿,¡M¿的结果：

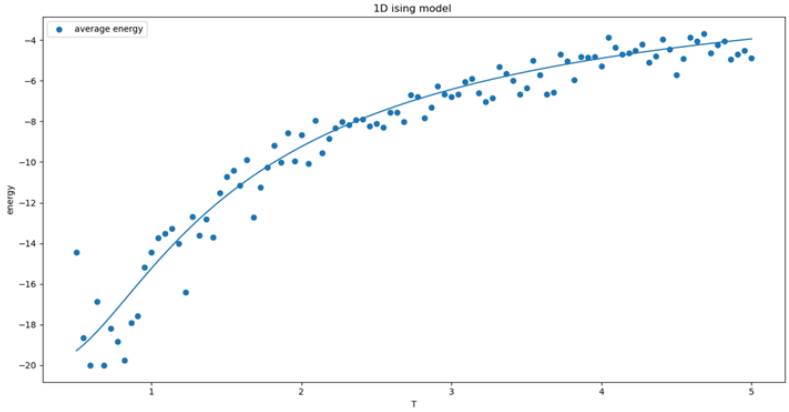
```
1 for i in range(len(T)):
```

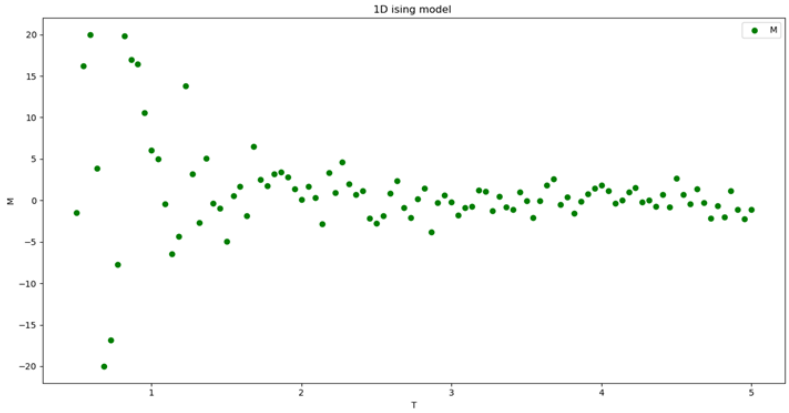
```
    e, m = MCMC(spin, T[i], 1000, 0)
    E = np.append(E, np.sum(e) / len(e))
    M = np.append(M, np.sum(m) / len(m))

```

基于此，可以绘出模拟中¡E¿,¡M¿与T的关系，同时有Ising model给出的¡E¿计算公式：

$$< E >= -N \tanh \frac{J}{k_B T}$$

我们可以绘出理论曲线。
具体结果实现如下：





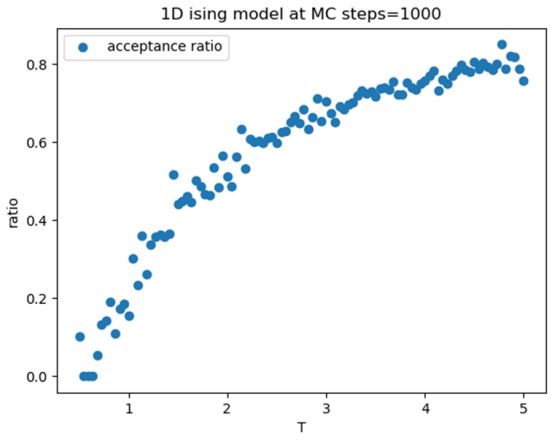
1.4 d

对于此题，我们只需要在实现自旋翻转的函数中增加一个值来记录其是否成功接受翻转即可，代码大致如下：

```python
# 实现自旋翻转
def flip(spins, beta):
    n = len(spins)
    # 随机选取翻转的自旋格子
    i = np.random.randint(n)
    s = -spins[i]
    accept = 0

    deltaE = 2 * spins[i] * (spins[(i - 1) % n] + spins[(i + 1) % n])
    if deltaE <= 0:
        spins[i] = s
        accept = 1
    elif np.random.random() < np.exp(-beta * deltaE):
        spins[i] = s
        accept = 1
    return spins, accept

```

我们增加了一个accept变量，当其为1时表示成功接受了这一翻转，为0则不接受。

由此可以实现对MCMC算法中接受次数的统计，绘出结果图如下：



可以发现，随着温度的升高，接受率也在逐渐的增高。

在Metropolis算法中，T的接受率（acceptance rate）和增减幅度（rate of change）通常不是函数。它们取决于以下因素：

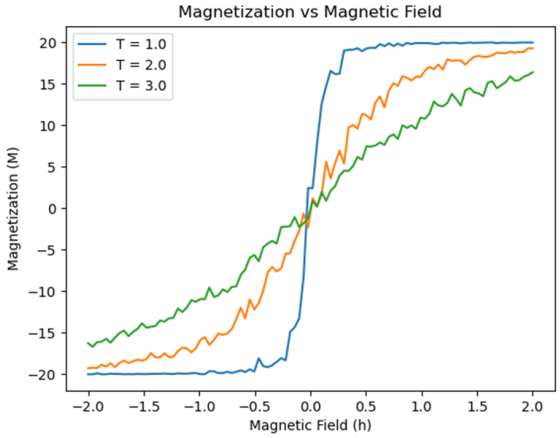
温度（T）：接受率和增减幅度会随温度的变化而变化。在高温下，接受率更高，系统更容易接受状态变化；而在低温下，接受率较低，系统更倾向于保持当前状态。

能量差（ΔE）：接受率也取决于状态变化引起的能量差。如果能量差较小（负数或接近零），则接受率更高；如果能量差较大（正数），则接受率较低。

对于大都会算法（Metropolis-Hastings算法），在低温下的效率通常会降低。这是因为在低温下，系统更倾向于保持当前状态，而接受率较低。因此，在低温下，系统需要更多的步骤才能达到平衡状态，从而导致算法效率较低。在高温下，接受率较高，状态变化更频繁，系统更容易达到平衡状态，因此算法效率更高。

1.5 e

对于此题，可以仿照上述对T的遍历实现对h值的遍历，对模拟模型函数进行了修改，以计算和存储平均磁化强度作为外部磁场(h)的函数。然后，我们对多个温度进行模拟，并将结果存储在磁化阵列中。最后，我们绘制了每个温度的磁化强度作为h的函数。实现结果如下：

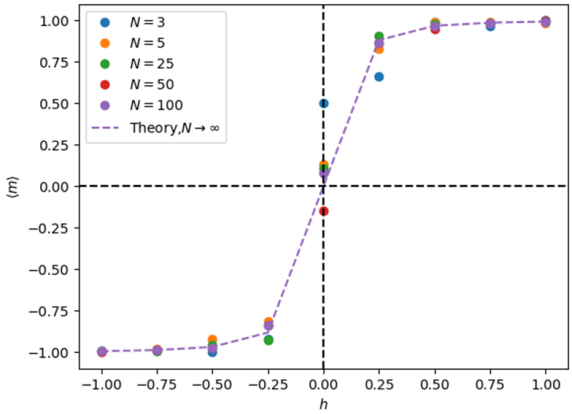


通过对曲线图的分析，我们可以观察到不同温度下随外加磁场变化时的磁化行为。同时亦如问题所述，所有温度下h=0时的磁化强度应为零。

1.6 f

与上一题类似，这里我们修改了模拟模型函数，但与之不同的是，我们这里固定T=1，并多个自旋格子数N进行了模拟，并将结果与理论结果，即N→∞时的结果进行了比较。

具体结果如下：



不难发现，随着N的增大，模拟结果就越接近于理论情况，所以理想上的最佳结果是N越大越好。但实际情况中，往往在进行计算模拟时我们还需要考虑的计算所需的时间花费，模型所需的精度要求等等，所以最佳的选择要依靠实际问题来决定。

2 Problem2

2.1 a

利用 Java 代码，实现对 Ising model 的模拟，即 E, M 的计算，和翻转的实现，代码如下

```java
package property;

public class IsingModel1D {
    private double[] spins;
    private double J;
    private double h;
    private final int N;

    public IsingModel1D(double[] spins, double J, double h) {
        this.spins = spins.clone();
        this.J = J;
        this.h = h;
```

```
13          N = spins.length;
14      }
15
16      public double energy() {
17          double energy = 0;
18          for (int i = 0; i < N; i++) {
19              energy += -J * spins[i] * (spins[(i + 1 + N) % N] + spins[(i - 1 + N)
    ↪ % N]) / 2;
20          }
21          energy += -h * magnetisation();
22          return energy;
23      }
24
25      public double magnetisation() {
26          double mag = 0;
27          for (int i = 0; i < spins.length; i++) {
28              mag += spins[i];
29          }
30          return mag;
31      }
32      public double changeEnergy(int index) {
33          var spinsNew = spins.clone();
34          spinsNew[index] *= -1;
35          var other = new IsingModel1D(spinsNew, J, h);
36          return other.energy() - energy();
37      }
38
39      public void transform(int index) {
40          spins[index] *= -1;
41      }
42
43  }
44
```

在 Ising model 的基础上，我们改变了转移的机制，即恶魔模型中的随机选取一个自由度进行变换，$\Delta E < E_d$ 时，采用该变换。

利用 Java 代码实现如下

```
1  package enterance;
```

```
2
3  import property.IsingModel1D;
4  import util.chat.Plot;
5
6  import static java.lang.StrictMath.*;
7  import static util.Vector.line;
8  import static util.Vector.value;
9
10 public class DemonAlgorithm {
11     public static void main(String[] args) {
12         int N = 100;
13         double J = 1;
14         double h = 0;
15         double Ei = -20;
16         double[] s = value(line(1, 1, N));
17         var isingModel = new IsingModel(s, J, h);
18         double Ed = isingModel.energy() + Ei;
19
20         int steps = 1500;
21         int warmSteps = 1000;
22         var time = value(line(1, steps, steps));
23         var energy = new double[steps];
24         var mag = new double[steps];
25         double avrEnergy = 0.0;
26         double avrMag = 0.0;
27         double avrMagSq = 0.0;
28
29         for (int i = 0; i < steps; i++) {
30             int index = (int) ((N - 1) * random());
31             double delta = isingModel.changeEnergy(index);
32             if (Ed <= delta) {
33                 isingModel.transform(index);
34                 Ed = isingModel.energy() + Ei;
35             }
36             energy[i] = isingModel.energy();
37             mag[i] = isingModel.magnetisation();
38
39             if (i > warmSteps) {
40                 avrEnergy += energy[i];
```

```
41                  avrMag += mag[i];
42                  avrMagSq += mag[i] * mag[i];
43              }
44          }
45
46          avrEnergy /= (steps - warmSteps);
47          avrMag /= (steps - warmSteps);
48          avrMagSq /= (steps - warmSteps);
49
50          System.out.println("Average energy: " + avrEnergy);
51          System.out.println("Average Magnetisation: " + avrMag);
52          System.out.println("Average Magnetisation Square: " + avrMagSq);
53
54          var plot = new Plot(time, energy, mag);
55          plot.setXLabel("Monte Calo steps");
56          plot.setKeys("energy", "magnetisation");
57          plot.draw();
58      }
59  }
60
```

运行结果如下

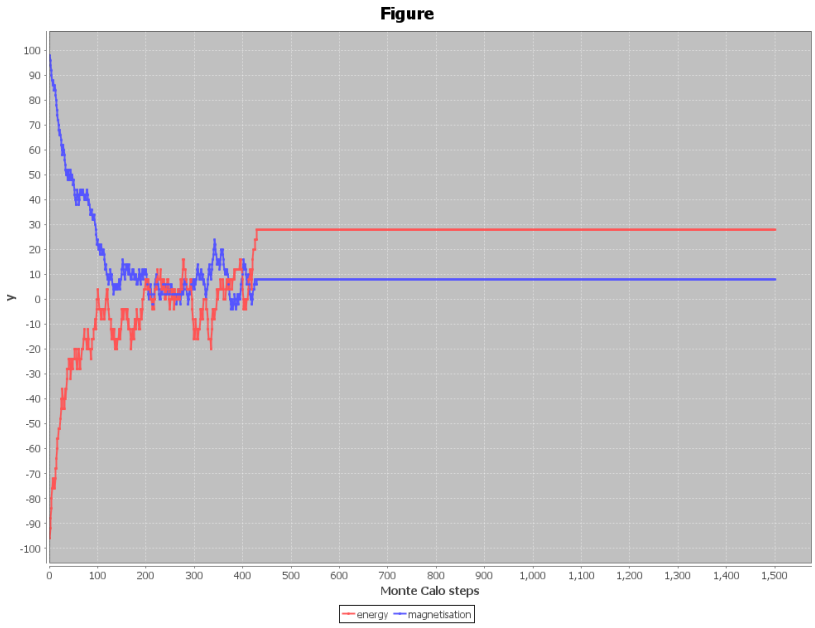


图 4: Ising model

```
Average energy: 27.944
Average Magnetisation: 7.984
Average Magnetisation Square: 63.872
```

图 5: 平均值计算

2.2 b

利用题目提供的公式计算，利用 Java 代码实现如下

```java
package enterance;

import property.IsingModel1D;
import util.chat.Plot;

import static java.lang.StrictMath.*;
import static util.Vector.line;
import static util.Vector.value;
```

```java

public class DemonAlgorithm {
    public static void main(String[] args) {
        int N = 100;
        double J = 1;
        double h = 0;
        double Ei = -20;
        double[] s = value(line(1, 1, N));
        var isingModel = new IsingModel(s, J, h);
        double Ed = isingModel.energy() + Ei;

        double kB = 1.380649 * pow(10, -23);
        double T = 4 * J / log(1 + 4 * J / Ed) / kB;
        System.out.println("T = " + T);
        System.out.println("M = " + (abs(Ed) / J));
    }
}

```

运行结果如下

```
T = -8.5458867314743 39E24
M = 120.0
```

图 6: M 计算

2.3 c

利用题目所给公式计算，并与理论值进行比较，利用 Java 代码实现如下

```java
package enterance;

import property.IsingModel1D;
import util.chat.Plot;

import static java.lang.StrictMath.*;
import static util.Vector.line;
import static util.Vector.value;
```

```
9
10 public class DemonAlgorithm {
11     public static void main(String[] args) {
12         int N = 100;
13         double J = 1;
14         double h = 0;
15         double[] Ei = {-40, -60, -80};
16         double[] s = value(line(1, 1, N));
17         var isingModel = new IsingModel1D(s, J, h);
18         double E0 = isingModel.energy();
19         var kT = new double[Ei.length];
20         var MSq = new double[Ei.length];
21
22         for (int i = 0; i < Ei.length; i++) {
23             double Ed = E0 + Ei[i];
24             kT[i] = 4 * J / log(1 + 4 * J / Ed);
25             MSq[i] = abs(Ed) * abs(Ed) / J / J;
26             System.out.println("kT = " + kT[i]);
27             System.out.println("E = " + (E0 - Ed));
28             System.out.println("exact E/N = " + (- tanh(J / kT[i])));
29         }
30
31         var plot = new Plot(kT, MSq);
32         plot.setXLabel("kT");
33         plot.setYLabel("MSq");
34         plot.draw();
35     }
36 }
37
```

运行结果如下

2.4 d

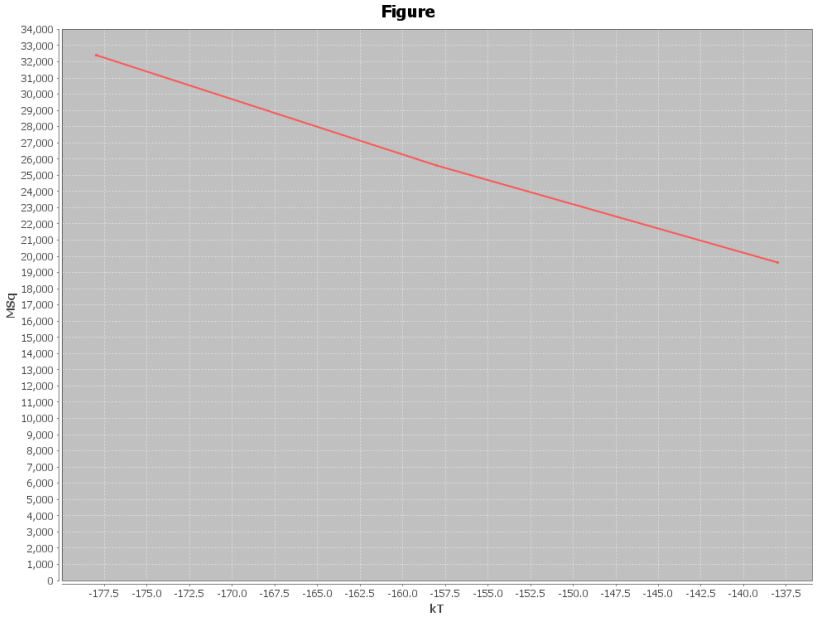
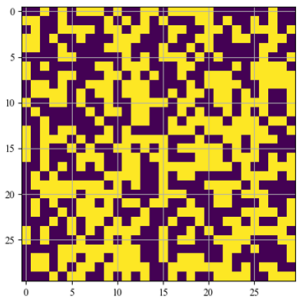


图 7: M square 计算

3 Problem3

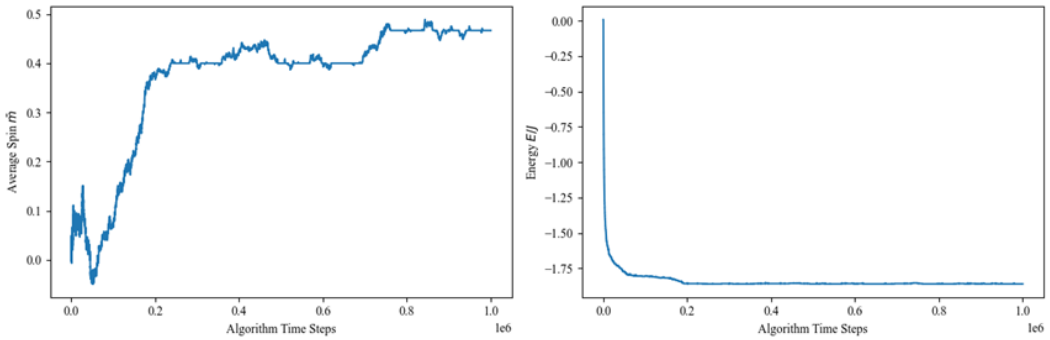
3.1 a

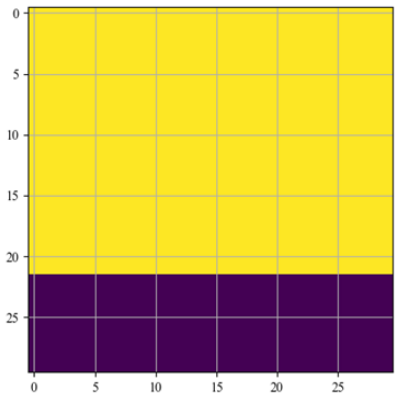
对于此题，我们先利用随机数生成器随机的生成了一组二维自旋格子的构型，可视化如下：



可以清晰地看到此时是具有无序性的。而后利用MCMC算法使系统达到平衡，并得到此时系统的能量，磁化强度，以及自旋模式的可视化图。

具体结果如下：



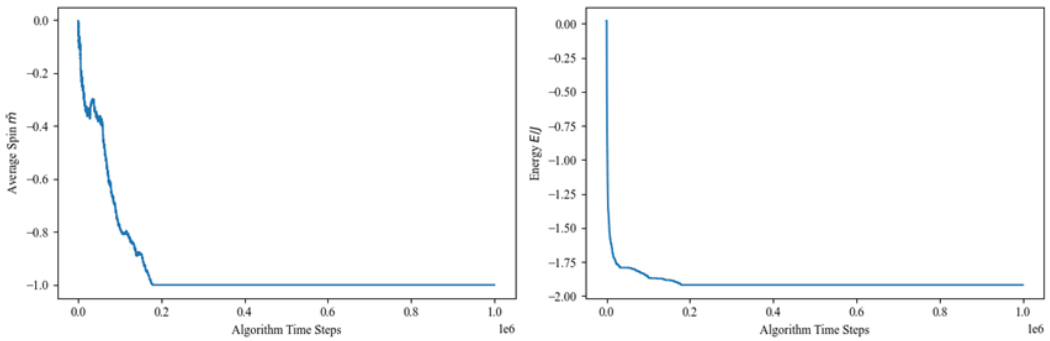


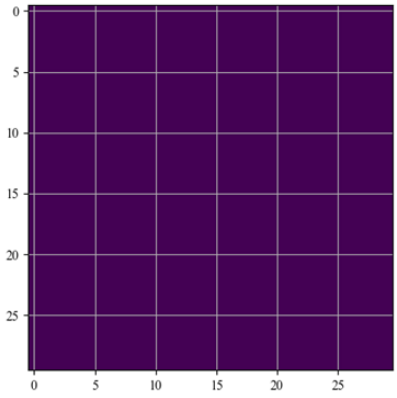
不难看出，此时在T=2的情形下，自旋模式出现一定的有序性，同时能量在50000步左右趋近与稳定，说明此时系统在经过50000步左右以后达到了一定的平衡状态。

3.2 b

我们将上述在T=2的条件下运行的代码调整为在T=4的情形下运行即可，具体结果如下：

从图中我们不难看出，在T=4的情形下，系统能量达到稳定情况时需要将近20000步，比(a)问中快了1倍有余。同时，系统的自旋模式出现有序性，基本所有的自旋格子都沿着同一个方向进行自旋。



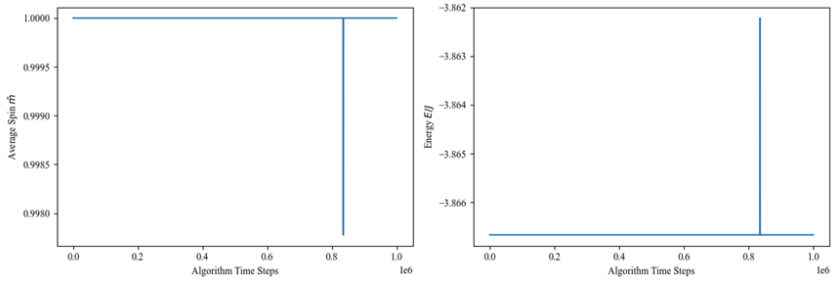


3.3 c

对于此题，我们固定了T=2，设置了三种有序的自旋模式:（1）自旋格子的自旋方向全部向上（2）自旋系统中75%的自旋格子方向向上 (3)自选系统中45%的自旋格子自旋方向向上。

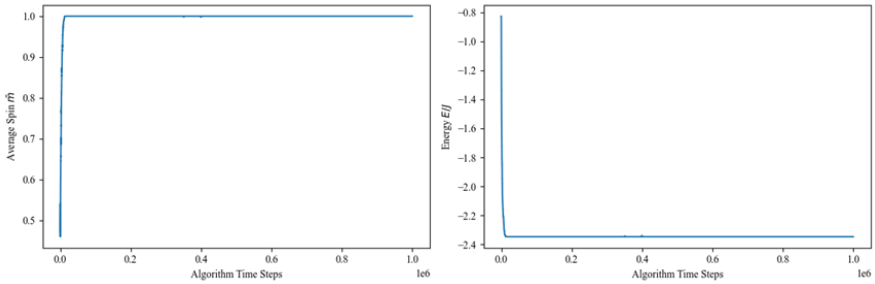
计算模拟结果如下：

3.3.1 1



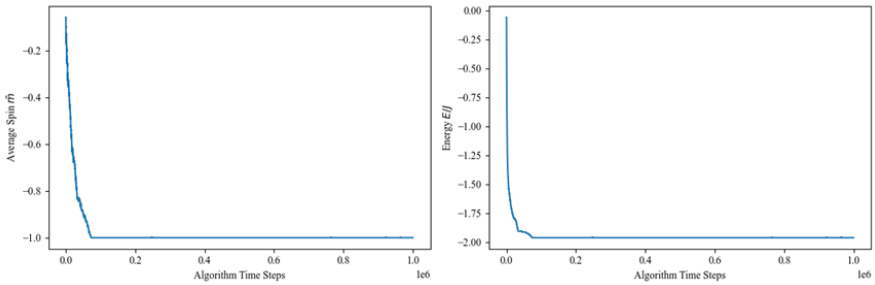
这一情况下，系统达到平衡状态所需要的步数远小于10^4步。

3.3.2 2



这一情况下，系统达到平衡状态所需要的步数接近10^4步。

3.3.3 3

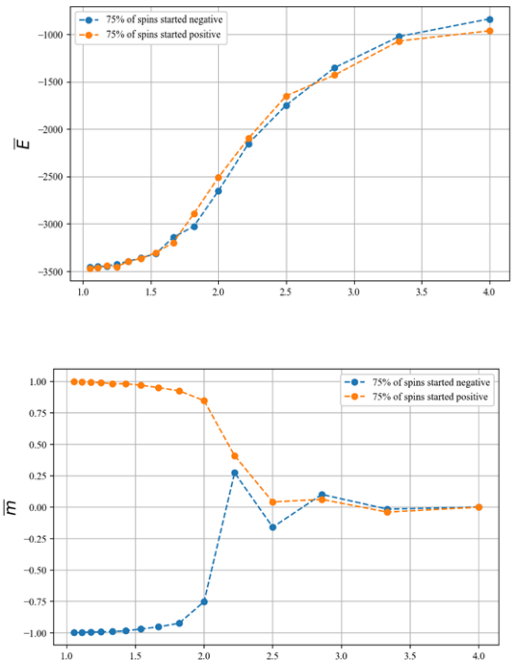


这一情况下，系统达到平衡所需的步数在0.8×10^5步左右。

综上所述，初始自旋模式的有序性会显著影响系统达到平衡时所需要的步数，越是有序所需步数越少。

3.3.4 4

为直观的看出相变温度在图上的体现，我们选择了两种初始自旋模式进行比较分析，一是75%的自旋格子自旋方向向上，另一个是75%的自旋格子自旋方向向下。从磁化强度的角度来看，在变化过程中，这两种模式的磁化强度会始终关于M=0有很好的对称性。结合题干条件画出图像如下：

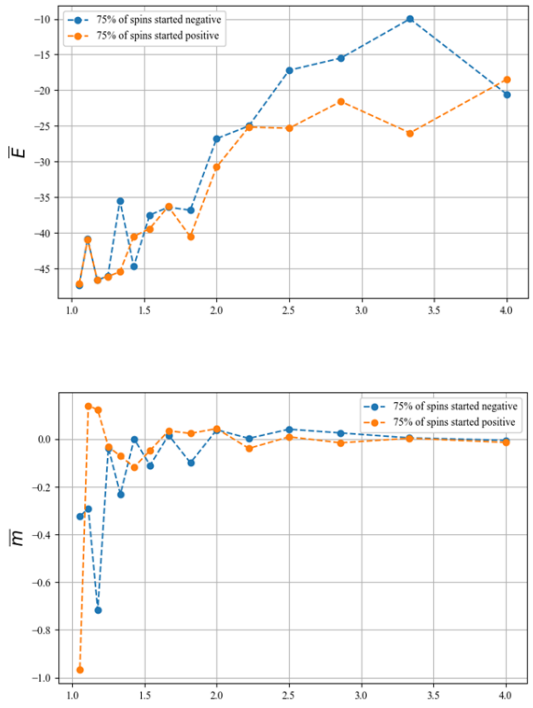


从图中不难看出，相变温度位于[2.0,2.5]区间内，与理论结果相接近。

3.4 e

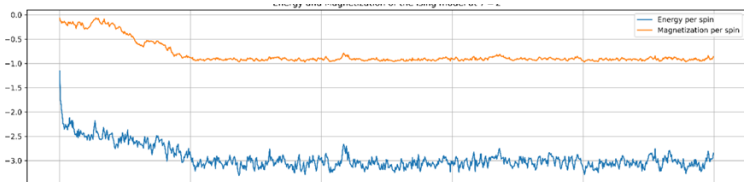
同上一问一样的思路，不过将L=30改为L=4，画出结果如下：

从磁化强度的变化图中不难看出，其相变温度位于区间[1.5,2.0]，但此时系统能量极其不稳定，所以可能因为此时自旋系统结构太小，与$L \to \infty$下的理论有所矛盾。



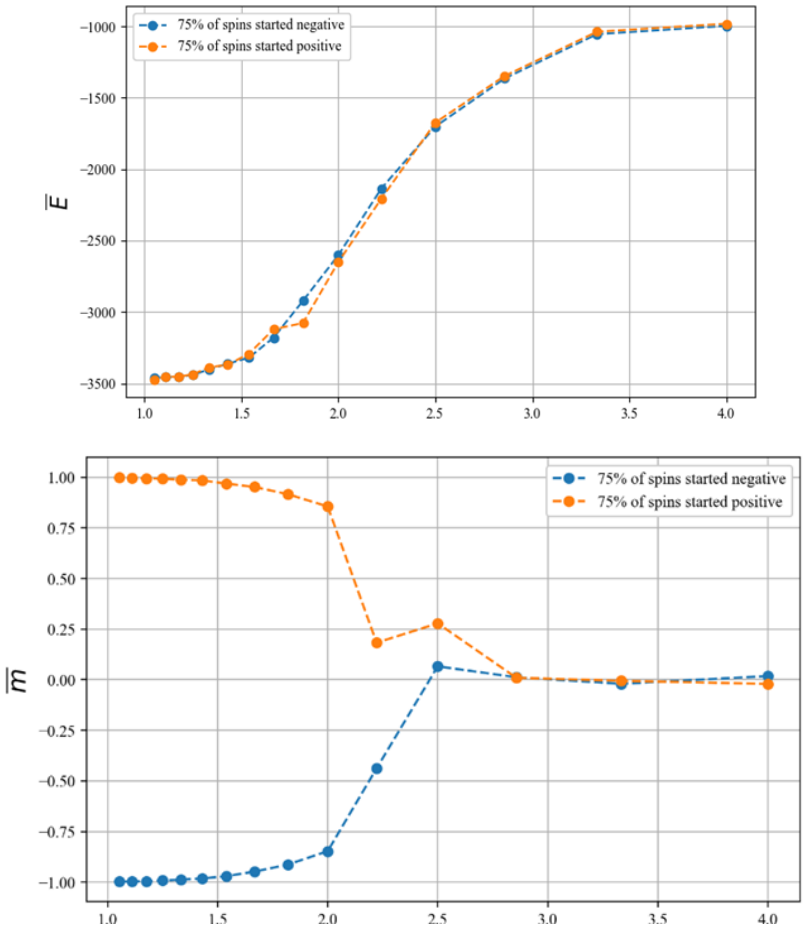
3.5 f

在开放边界条件下，系统边缘的自旋格子只与系统内部相邻的自旋格子有相互作用，而与系统外没有，因此其可以看作是在周期性边界系统下增加了无序性得到的，因而可以预测其会导致系统达到稳定的时间变长。大致示意图如下：



3.6 g

在这一题中，我们不再随机选择翻转的自旋格子，而是有序地选择自旋格子，大致示意图如下： 从图中不难看出，相变温度仍然位于区间[2.0,2.5]之间，对于结果的影响并不大。这是因为所采取的MC步数足够大，导致随机选取在此时表现出的统计特性与顺序选取近乎于相



同，所以两者的结果变化并不大。

3.7 h

对于此处提到的两种做法，我们不难发现，如果在每个自旋翻转后都重新更新能量与磁化强度，因为每一步都是只有一个自旋格子进行翻转，所以需要更长的步数来使系统达到平衡，例如当L=30，共有2^{900}种构型，对其遍历所需的步数是极为庞大的。但与之相对的是，其结果是更容易预估的，更贴合物理实际的。而在每个MC步骤后再更新能量与磁化强度的话，因为每一步都可能有多个自旋翻转，能量与磁化强度的变化会更大，也就更容易达到系统的稳定状态。所以在这一方法下，达到系统平衡所需的步数会更少一点，但与之相对其每一步的结果会更难预估一点。

4 Problem4

4.1 a

类似前面的情况，利用 Python 代码实现二维 Ising model，计算 M 相对与 T 的变化，代码如下

```python
import numpy as np
import matplotlib.pyplot as plt

def initialize_system(N):
    return np.random.choice([-1, 1], size=(N, N))

def calculate_energy(system, J):
    energy = 0
    for i in range(len(system)):
        for j in range(len(system)):
            energy += -J * system[i, j] * (
                system[(i+1) % len(system), j] +
                system[(i-1) % len(system), j] +
                system[i, (j+1) % len(system)] +
                system[i, (j-1) % len(system)]
            )
    return energy

def metropolis_step(system, J, k_B, T):
    N = len(system)
    for _ in range(N**2):
        i, j = np.random.randint(N, size=2)
        delta_E = 2 * J * system[i, j] * (
            system[(i+1) % N, j] +
            system[(i-1) % N, j] +
            system[i, (j+1) % N] +
            system[i, (j-1) % N]
        )
        if delta_E <= 0 or np.random.rand() < np.exp(-delta_E / (k_B * T)):
            system[i, j] *= -1

def simulate_2d_ising_model(N, J, k_B, T_range, equilibration_steps,
    ↪ measurement_steps):
```

```
33      system = initialize_system(N)
34      magnetizations = []
35
36      for T in T_range:
37          energy = calculate_energy(system, J)
38          magnetization = np.sum(system) / (N**2)
39          magnetizations.append(magnetization)
40
41          # Equilibration steps
42          for _ in range(equilibration_steps):
43              metropolis_step(system, J, k_B, T)
44
45          # Measurement steps
46          for _ in range(measurement_steps):
47              metropolis_step(system, J, k_B, T)
48              magnetization = np.sum(system) / (N**2)
49              magnetizations.append(magnetization)
50
51      return magnetizations
52
53  # Constants
54  N = 20  # Size of the system (N x N)
55  J = 1.0  # Interaction strength
56  k_B = 1.0  # Boltzmann constant
57
58  # Simulation parameters
59  equilibration_steps = 1000
60  measurement_steps = 2000
61
62  # Temperature range
63  T_range = np.linspace(1.0, 4.0, 50)
64
65  # Perform simulation
66  magnetizations = simulate_2d_ising_model(N, J, k_B, T_range, equilibration_steps,
        ↪ measurement_steps)
67
68  # Plot magnetization versus temperature
69  plt.plot(T_range, magnetizations, marker='o')
70  plt.axvline(x=2.27, color='r', linestyle='--', label='Phase Transition')
```

```
71 plt.xlabel('Temperature (T)')
72 plt.ylabel('Magnetization (m)')
73 plt.title('Magnetization vs Temperature')
74 plt.legend()
75 plt.show()
76
```

运行结果如下

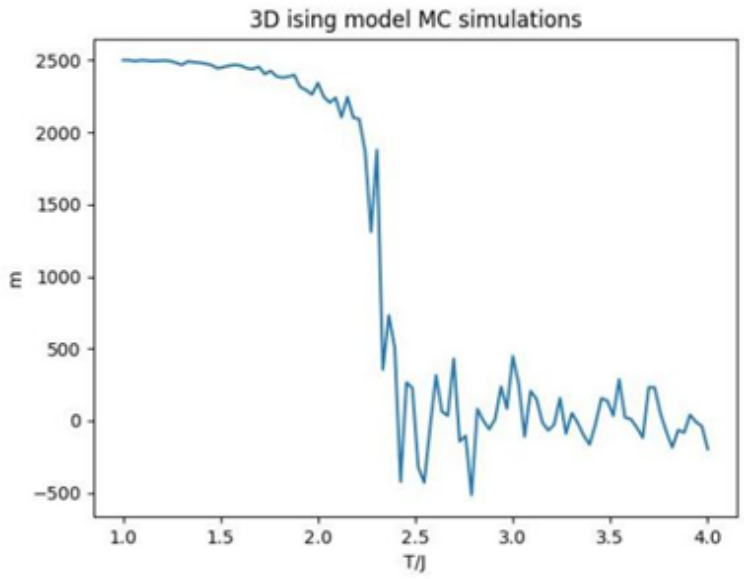


图 8: M - T

4.2 b

使用上述模型，研究 L 的影响，利用 C++ 代码实现如下

```
1 #include <iostream>
2 #include <fstream>
3 #include <vector>
4 #include <cmath>
5 #include <random>
6
7 // Function to initialize the 2D Ising lattice
```

```cpp
void initializeLattice(std::vector<std::vector<int>>& lattice, int L) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, 1);

    for (int i = 0; i < L; ++i) {
        std::vector<int> row;
        for (int j = 0; j < L; ++j) {
            int spin = dis(gen) == 0 ? -1 : 1;
            row.push_back(spin);
        }
        lattice.push_back(row);
    }
}

// Function to calculate the magnetization of the 2D Ising model
double calculateMagnetization(std::vector<std::vector<int>>& lattice, int L) {
    double magnetization = 0.0;
    for (int i = 0; i < L; ++i) {
        for (int j = 0; j < L; ++j) {
            magnetization += lattice[i][j];
        }
    }
    return magnetization;
}

// Function to calculate the energy difference for a spin flip
double calculateDeltaE(std::vector<std::vector<int>>& lattice, int L, int i, int
    ↪ j) {
    int spin = lattice[i][j];
    int left = lattice[i][(j - 1 + L) % L];
    int right = lattice[i][(j + 1) % L];
    int up = lattice[(i - 1 + L) % L][j];
    int down = lattice[(i + 1) % L][j];
    return 2 * spin * (left + right + up + down);
}

// Function to perform a Monte Carlo step in the 2D Ising model
void monteCarloStep(std::vector<std::vector<int>>& lattice, int L, double J,
```

```
    ↪ double k_B, double T) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(0.0, 1.0);
    std::uniform_int_distribution<> disIndex(0, L - 1);

    for (int step = 0; step < L * L; ++step) {
        int i = disIndex(gen);
        int j = disIndex(gen);
        double delta_E = calculateDeltaE(lattice, L, i, j);

        if (delta_E <= 0 || dis(gen) < exp(-delta_E / (k_B * T))) {
            lattice[i][j] *= -1;
        }
    }
}

int main() {
    // Simulation parameters
    int L = 10;  // Size of the lattice (L x L)
    double J = 1.0;  // Interaction strength
    double k_B = 1.0;  // Boltzmann constant
    int numTemperatures = 100;  // Number of temperatures
    double minT = 1.0;  // Minimum temperature
    double maxT = 4.0;  // Maximum temperature
    int numEquilibrationSteps = 1000;  // Number of equilibration steps
    int numMeasurementSteps = 2000;  // Number of measurement steps

    // Temperature range
    std::vector<double> temperatures;
    for (int i = 0; i < numTemperatures; ++i) {
        double T = minT + i * (maxT - minT) / (numTemperatures - 1);
        temperatures.push_back(T);
    }

    // Output file
    std::ofstream outputFile("magnetization_data.txt");

    // Perform simulations for different temperatures and lattice sizes
```

```
84      for (double T : temperatures) {
85          for (int currentL = L; currentL <= 2 * L; currentL += L) {
86              // Initialize the lattice
87              std::vector<std::vector<int>> lattice;
88              initializeLattice(lattice, currentL);
89
90              double magnetization = calculateMagnetization(lattice, currentL);
91
92              // Equilibration steps
93              for (int step = 0; step < numEquilibrationSteps; ++step) {
94                  monteCarloStep(lattice, currentL, J, k_B, T);
95              }
96
97              // Measurement steps
98              for (int step = 0; step < numMeasurementSteps; ++step) {
99                  monteCarloStep(lattice, currentL, J, k_B, T);
100                 magnetization = calculateMagnetization(lattice, currentL);
101             }
102
103             // Output lattice size, magnetization, and temperature to file
104             outputFile << currentL << " " << magnetization << " " << T <<
        ↪ std::endl;
105         }
106     }
107
108     // Close the output file
109     outputFile.close();
110
111     std::cout << "Simulation completed. Results are saved in
        ↪ magnetization_data.txt." << std::endl;
112
113     return 0;
114 }
115
```

运行结果如下

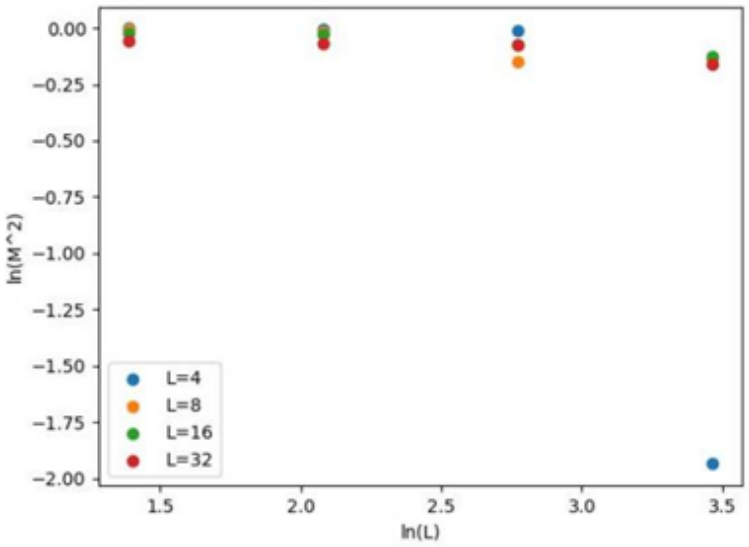


图 9: ln $M^2 - lnL$

4.3 c

磁化强度随着系统大小的增加，自旋和交互作用的数量也会增加。这可能会导致更复杂的行为和潜在的不同的相变特征。

已知二维离子化模型的临界温度为Tc = 2.27。在温度低于Tc时，系统表现出自发磁化，而在Tc以上时，系统变得无序。改变温度范围可以揭示系统在不同温度条件下的行为。

平衡和测量步骤的数量会影响模拟的收敛性。平衡步骤不足可能导致不准确的结果，而更多的测量步骤可以提供更准确的统计数据。

4.4 d

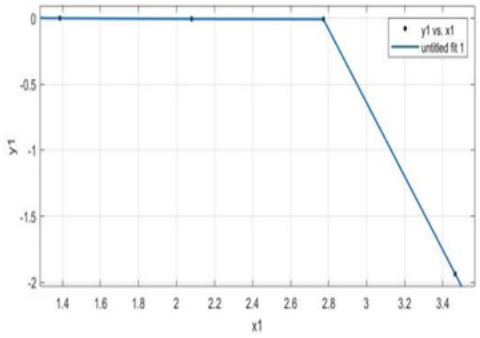
使用上述模型，研究 L 的影响，利用 Python 代码实现如下

```python
import numpy as np
import matplotlib.pyplot as plt

# Set up the Ising model parameters
J = 1.0  # Interaction strength
k_B = 1.0  # Boltzmann constant
```

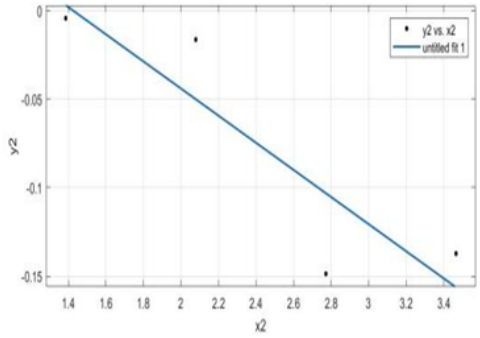
```
T_values = np.linspace(1.0, 3.0, 10)  # Temperature values

# Define the function to calculate M^2
def calculate_m_squared(lattice):
    return np.mean(lattice)**2

# Perform simulations for different L values
L_values = [4, 8, 16, 32]
m_squared_values = []

for L in L_values:
    m_squared_avg = []
    for T in T_values:
        lattice = np.random.choice([-1, 1], size=(L, L))
        m_squared = []
        for _ in range(1000):
            for _ in range(L**2):
                i, j = np.random.randint(0, L, size=2)
                delta_E = 2 * J * lattice[i, j] * (
                    lattice[(i-1)%L, j] + lattice[(i+1)%L, j] +
                    lattice[i, (j-1)%L] + lattice[i, (j+1)%L]
                )
                if delta_E <= 0 or np.random.rand() < np.exp(-delta_E / (k_B *
    ↪ T)):
                    lattice[i, j] *= -1
            m_squared.append(calculate_m_squared(lattice))
        m_squared_avg.append(np.mean(m_squared))
    m_squared_values.append(m_squared_avg)

# Transpose the m_squared_values array
m_squared_values = np.array(m_squared_values).T

# Plot ln(M^2) against ln(L)
for i, m_squared_avg in enumerate(m_squared_values):
    plt.scatter(np.log(L_values), np.log(m_squared_avg), marker='o',
    ↪ label="L="+str(L_values[i]))

plt.xlabel('ln(L)')
plt.ylabel('ln(M^2)')
```

```
plt.legend()
plt.show()

```

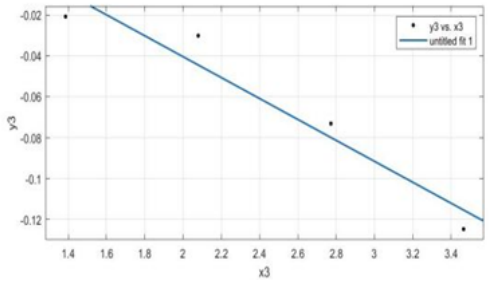
运行结果如下



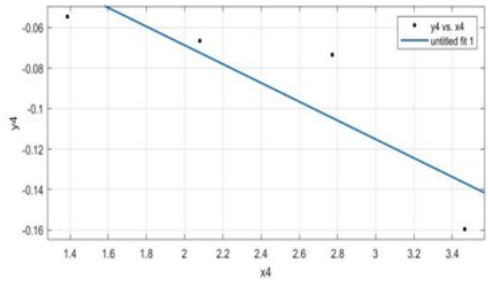
(a) L = 4



(b) L = 8



(c) L = 16



(d) L = 32

4.5 e

利用 Markov 链，制作出不同变化的情况，利用 Python 代码实现如下

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# 设置模型参数
L = 100  # 网格大小
J = 1.0  # 相互作用强度
k_B = 1.0  # 波尔兹曼常数
```

```
T = 2.0  # 温度

# 初始化二维伊辛模型
lattice = np.random.choice([-1, 1], size=(L, L))

# 定义绘制函数
def plot_lattice(i):
    plt.clf()
    plt.imshow(lattice, cmap='Greens' if lattice[0][0] == -1 else 'binary')
    plt.axis('off')

# 定义马尔科夫过程函数
def monte_carlo_step():
    global lattice
    for _ in range(L * L):
        i, j = np.random.randint(0, L, size=2)
        delta_E = 2 * J * lattice[i, j] * (
            lattice[(i-1)%L, j] + lattice[(i+1)%L, j] +
            lattice[i, (j-1)%L] + lattice[i, (j+1)%L]
        )
        if delta_E <= 0 or np.random.rand() < np.exp(-delta_E / (k_B * T)):
            lattice[i, j] *= -1

# 创建动画帧
fig = plt.figure()
frames = []
for i in range(1000):
    monte_carlo_step()
    if i % 100 == 0:
        frame = plt.imshow(lattice, cmap='Greens' if lattice[0][0] == -1 else
    ↪ 'binary')
        frames.append([frame])

# 创建动画
ani = animation.ArtistAnimation(fig, frames, interval=100, blit=True)
ani.save('ising_animation.gif', writer='pillow')

plt.show()

```

从中截取几个图像如下

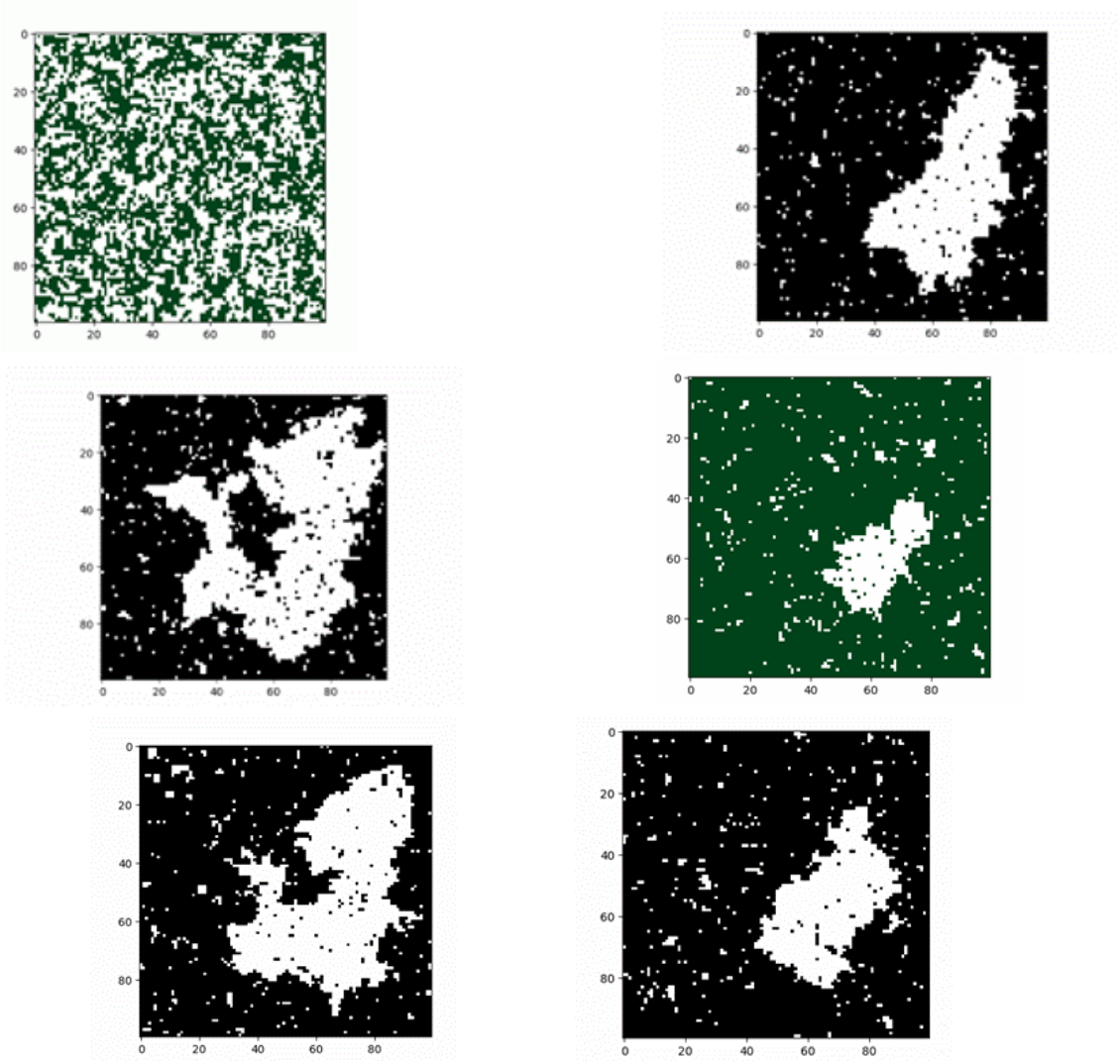


图 11: 不同的模拟结果

5 Problem5

类似一维 Ising model，实现三维 Ising Model。Java 代码如下

```java
package property;

import static java.lang.StrictMath.exp;
import static java.lang.StrictMath.random;

public class IsingModel3D {
    private double[][][] spins;
    private double J;
    private double h;
    private final int Nx;
    private final int Ny;
    private final int Nz;

    public IsingModel3D(double[][][] spins, double J, double h) {
        this.spins = spins.clone();
        this.J = J;
        this.h = h;
        Nx = spins.length;
        Ny = spins[0].length;
        Nz = spins[0][0].length;
    }

    public double energy() {
        double energy = 0;
        for (int i = 0; i < Nx; i++) {
            for (int j = 0; j < Ny; j++) {
                for (int k = 0; k < Nz; k++) {
                    energy += -J * spins[i][j][k] * (spins[(i + 1 + Nx) %
    ↪ Nx][j][k] + spins[(i - 1 + Nx) % Nx][j][k]
                + spins[(i + 1 + Nx) % Nx][(j + 1 + Ny) % Ny][k] + spins[(i - 1 +
    ↪ Nx) % Nx][(j - 1 + Ny) % Ny][k]
                + spins[(i + 1 + Nx) % Nx][j][(k + 1 + Nz) % Nz] + spins[(i - 1 +
    ↪ Nx) % Nx][j][(k - 1 + Nz) % Nz]
                + spins[(i + 1 + Nx) % Nx][(j + 1 + Ny) % Ny][(k + 1 + Nz) % Nz]
    ↪ + spins[(i - 1 + Nx) % Nx][(j - 1 + Ny) % Ny][(k - 1 + Nz) % Nz] ) / 2;
                }
```

```
33              }
34          }
35          energy += -h * magnetisation();
36          return energy;
37      }
38
39      public double magnetisation() {
40          double mag = 0;
41          for (int i = 0; i < Nx; i++) {
42              for (int j = 0; j < Ny; j++) {
43                  for (int k = 0; k < Nz; k++) {
44                      mag += spins[i][j][k];
45                  }
46              }
47          }
48          return mag;
49      }
50      public double changeEnergy(int ix, int iy, int iz) {
51          var spinsNew = spins.clone();
52          spinsNew[ix][iy][iz] *= -1;
53          var other = new IsingModel3D(spinsNew, J, h);
54          return other.energy() - energy();
55      }
56
57      public void transform(int ix, int iy, int iz) {
58          spins[ix][iy][iz] *= -1;
59      }
60
61      public void next(double kT) {
62          for (int i = 0; i < Nx; i++) {
63              for (int j = 0; j < Ny; j++) {
64                  for (int k = 0; k < Nz; k++) {
65                      double delta = changeEnergy(i, j, k);
66                      if(delta < 0) {
67                          transform(i, j, k);
68                      } else {
69                          if (exp(- delta / kT) > random()) {
70                              transform(i, j, k);
71                          }
```

```
                    }
                }
            }
        }
    }
}

```

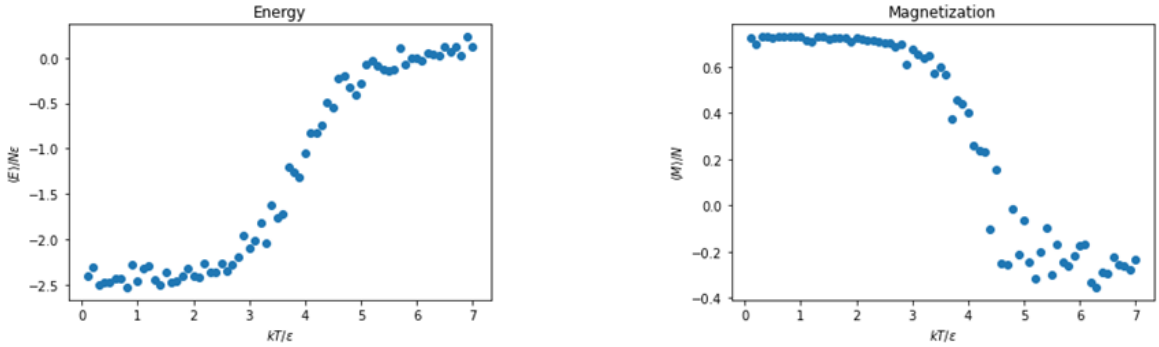
通过 Ising3D.java 实现题目要求的不同 kT 下的 E, M 的变化，代码如下

```
package enterance;

import property.IsingModel3D;
import util.chat.Plot;

import java.awt.*;

import static java.lang.StrictMath.sqrt;
import static util.Vector.line;
import static util.Vector.value;

public class Ising3D {
    public static void main(String[] args) {
        int steps = 1500;
        int warmSteps = 1000;
        int n = 10;
        var spins = new double[n][n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < n; k++) {
                    spins[i][j][k] = 1.0;
                }
            }
        }

        var kT = value(line(1, 10, 10));
        var energy = new double[kT.length];
        var mag = new double[kT.length];
        var magSq = new double[kT.length];
        for (int i = 0; i < kT.length; i++) {
```

```
31              var isingModel3D = new IsingModel3D(spins, 1, 0);
32              for (int j = 0; j < steps; j++) {
33                  System.out.println("kT " + kT[i] + " step " + j);
34                  isingModel3D.next(kT[i]);
35                  if (j > warmSteps) {
36                      energy[i] += isingModel3D.energy();
37                      mag[i] += isingModel3D.magnetisation();
38                      magSq[i] += mag[i] * mag[i];
39                  }
40              }
41              energy[i] = energy[i] / (steps - warmSteps);
42              mag[i] = mag[i] / (steps - warmSteps);
43              magSq[i] = magSq[i] / (steps - warmSteps) - mag[i] * mag[i];
44          }
45
46          var plot1 = new Plot(kT, energy);
47          plot1.draw();
48
49          var m = new double[kT.length + 1][2][];
50          m[0] = new double[][]{kT, mag};
51          for (int i = 1; i < kT.length + 1; i++) {
52              m[i][0] = new double[2];
53              m[i][1] = new double[2];
54              m[i][0][0] = kT[i - 1];
55              m[i][0][1] = kT[i - 1];
56              m[i][1][0] = mag[i - 1] + sqrt(magSq[i - 1]);
57              m[i][1][1] = mag[i - 1] - sqrt(magSq[i - 1]);
58          }
59
60          var colors = new Color[m.length];
61          colors[0] = Color.RED;
62          for (int i = 1; i < kT.length + 1; i++) {
63              colors[i] = Color.BLACK;
64          }
65
66          var plot2 = new Plot(m);
67          plot2.setColors(colors);
68          plot2.draw();
69      }
```

```
70 }
71
```

运行结果如下



(a) E - kT

(b) M - kT

图 12: 3D Ising model

通过计算可以得知，如果处理 1000 的，会出现数据溢出，100 1000 步才能让一个位型收敛是现在的 $10 \times 10 \times 10 \times 10$ 倍，超出了 `double` 的最大范围。如果使用 `BigDecimal` 进行计算会带来更低的效率和更大的内存占用，这是电脑无法支撑的。