# MA 416 Final Project: The Ising Model of Ferromagnets

## Modeling Ferromagnets using Monte Carlo Simulations

Can we use a computer simulation to predict the critical temperature of an Ising ferromagnet?

---

## Background, Overview, Asssumptions

**What is a magnetic dipole?** In electrostatics, electric fields are created by stationary charges. In magnetostatics, magnetic fields are created by steady currents. Current is just the movement of charge. On the atomic scale, moving charges come in the form of electrons (negative charge) orbiting the nucleus and spinning around their axes. When we zoom out to the macroscopic scale, we can treat these tiny current loops as magnetic dipoles. The direction of such a dipole is orthoganol to the direction of the current according to the right hand rule. For our purposes, we only care about the *direction* of our magnetic dipoles.

**What is a ferromagnet?** A feromagnet is a material whose magnetic dipoles align either parallel or antiparallel to each other in the presence of an external magnetic field. However, these dipoles keep their orientation in the *absence* of the external field. Ferromagnets will have a net non-zero magnetization, which is dependent on temperature. Every ferromagnet has a critical temperature called the ***Curie Temperature*** at which the net magnetization is 0. A Ferromagnet's dipoles will divide themselves into chunks called ***domains***. When one domain is magnetized, the magentic field created by this domain has the tendency to magnetize its neighbors in the opposite direction.

**The Ising Model:** The Ising Model is a framework which considers a single domain of a ferromagnet (A typical domain usualy consists of billions of dipoles). Assumptions made by the Ising Model:

- All long range magnetic interatctions between dipoles are neglected
- The tendency of neighboring dipoles to align parallel or antiparallel are accounted for
- The material has a prefered axis of magnetization
- Each dipole can only point parallel or antiparallel to this axis

---

## Mean Field Approximation

Using Mean Field approximation, an estimate for the critical temperature can be found. I will give a brief overview of how this approximation works (which relies on stastical mechanics, so I will try to keep it general), so it's clear where our result comes from. It is worth noting that Mean Field approximations are estimations that come up often in physics, when dealing with complex systems.

First, we "freeze" our dipole's neighbors and let our dipole orient itself based on its neighbors. Based on whether the dipole aligns up or down, there will be some energy difference, which is used to calculate the partition function. The partition function is a value in stastical mechanics that is used to calculate probabilities. With the partition function, we are able to calculate the average value of our dipole's alignment, using the convention that up = 1 and down = -1. The **mean field approximation** assumes that the thermal average value of alignment is equal to the average value of the instantenous alignment of the dipole's neighbors. Graphing this approximation of average values produces a transcendental equation, which upon inspection, tells us that at low temperatures ($kT < n\epsilon$), the system will accquire a net nonzero magnetization which is equally likely to be positive or negative.

Thus we can write the following expression: $kT_c = n\epsilon$

Where k is the bolttzman constant, Tc is the critical temperature, n is the number of neighbors a dipole has, and $\epsilon$ is the neighbor to neighbor interaction energy.

Using this approximation, we discover that the critical temperature is proportional to the number of neighbors each dipole has. Thus, the more neighbors a dipole has, the more likely it is to magnetize. Below the critical temperature, the system will become magnetized.

---

**The Metroplis Algorithm:** If we want to simulate a simple 20x20 2D ferromagnet, we cannot possibly consider all of its possible states. This lends itself to random sampling of states using a Monte Carlo simmulation; however this method is not efficient since this will miss the important states around the critical temperature. So we have to use a different method.

The Metropolic Algorithm Procedure:

- Start in any state
- Chose a dipole at random and consider the possbility of flipping it
- Compute the energy difference (between the two possible states) $\Delta U$ that would result from the flip
  - If $\Delta U \leq 0$ the system's energy would not increase so the dipole can be flipped and generate the next state
  - If $\Delta U \geq 0$ the system's energy would increase, and the decision to flip the dipole can be decided at random with the probability to flip being $e^{\frac{-\Delta U}{kT}}$
    - If the dipole does not get flipped, the new state will be the same as before and a new dipole will be evaluated
- This process is repeated until every dipole has had multiple chances to be flipped.

This algorithm generates states with the correct probabilities according to Bolttzman Stastics.

At low temperatures, our algorithm will push the system into a "metastable" state in which nearly all of the dipoles are parallel to their neighbors.

Our inputs into this simmulation will be the size of our grid and the temperature of the system (in units of $\frac{\epsilon}{k}$).

Our program can be broken down into 3 main components:

- deltaU : compute the $\Delta U$ of flipping a dipole
- randomGrid: Generate a random array of dipoles
- markDipole : fill (or add a point in a scatter plot) according to the orientation of the dipole

Simmulation Outline:

- randomGrid()
- for loop to run through all iterations
  - pick a random dipole
  - deltaU()
    - if deltaU is less than 0:
      - markDipole()
    - else, compute probaility of flipping by generating random number
      - markDipole()
- next interation
- end simmulation

---

## The Ising Program

```python
import matplotlib.pyplot as plt
import numpy as np
plt.rcParams['figure.figsize']=[10,10]

def randomGrid(size):
    a=np.random.rand(size,size)
    for i in range(size):
        for j in range(size):
            a[i,j]=round(a[i,j],0)
            if a[i,j]==0:
                a[i,j]=-1
    return a
```

Our first program component is to generate a random starting state for our simulation. First, we generate a NxN array filled with random values between 0 and 1 from Numpy. Next, we iterate through the array using a nested loop to round each entry to a 0 or 1. Finally, we change all of the 0's to -1.

"Up" dipoles are denoted by a 1, and "Down" dipoles are denoted by a -1. Using this number system, it is easy to see how when an up and down dipole are next to each other, their magnetization will be 0 since they cancel each other out.

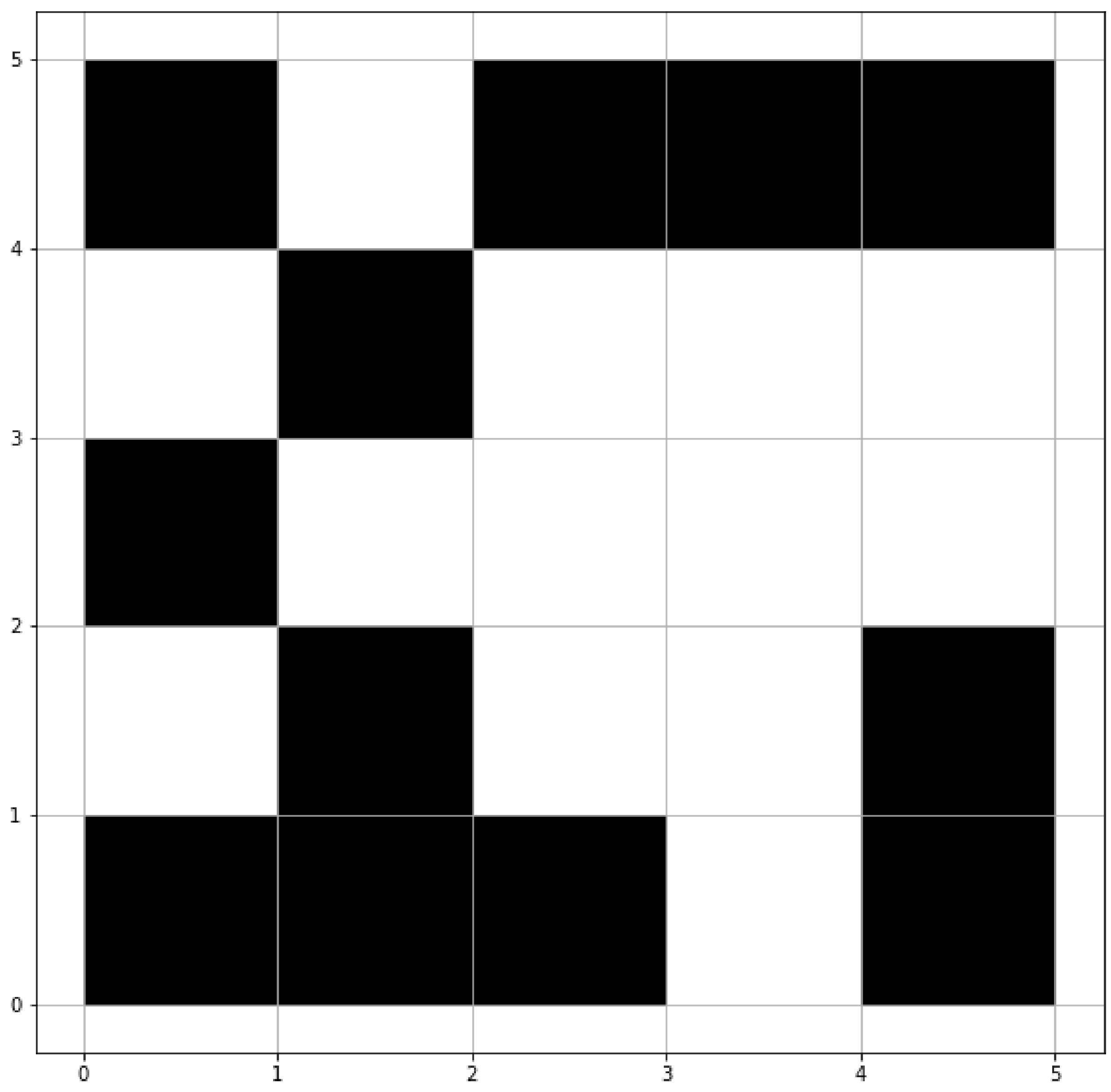
```
d=randomGrid(5)

def markDipole(data,colorCode):
    size=len(data)
    for i in range(0,size):
        for j in range(0,size):
            if data[i,j]==1:
                plt.fill_between((i,i+1),j,j+1, color=colorCode)
```

Our second program component takes an array and coverts it into a graphical output. Each "pixel" in our graph represents a dipole, whose orientation is marked by whether or not it is colored. The colored dipoles are "up", and the white dipoles are "down".

Additionally, I added the functionality for the color to be changed, which will be useful later on during the analysis section.

This component took the most time to work out and was the first task I tackled. It was important to make sure my graphical output would be legible and easy to understand.

```
markDipole(d, "black")
plt.grid()
```



This is an example of what the output of my first two program components will look like. With the expanded graphic size and small array size, each dipole is easily identifiable and distinct.

```
def deltaU(i,j,data):
    size=len(data)-1
    top=0
    bottom=0
    left=0
    right=0
    Ediff=0

    if i==0:
        top=data[size,j]
    else:
        top=data[i-1,j]
    if i==size:
        bottom=data[0,j]
    else:
        bottom=data[i+1,j]
    if j==0:
        left=data[i,size]
    else:
        left=data[i,j-1]
    if j==size:
        right=data[i,0]
    else:
        right=data[i,j+1]

    Ediff=2*data[i,j]*(top+bottom+left+right)
    return Ediff
```

This component computes the energy difference that would occur if the dipole flips or not. If the energy difference is negative, the system will move towards its lowest energy configuration and flip the dipole. If the energy is positive, the probability of a flip due to thermal energy (set by the external temperature of the system) is calculated. If the dipole is not flipped, nothing changes and the main loop will select a new dipole to evaluate.

In order to calculate the energy, deltaU sums up the magnetization of the dipoles and uses the sign in its determination.

At this section, we apply periodic boundary conditions. We are evaluating the dipole of a single domain within a ferromagnet, which we can assume is surrounded by other domains. However, our simulation only focusses on one domain and does not account for the dipoles on the edges of the neighboring domains. These edge dipoles will have an effect on the flipping of our simulation's edge dipoles. To account for this, we assume that our domain is surrounded by domains *exactly* equal to it. One way to think about this is that our domain is perfectly periodic, so moving within the ferromagnet by the length of the domain in any direction results in an equivalent spot where all the dipoles have the same alignment. Thus, we are modelling our domain with peridoic boundary conditions to minimize edge effects.

Because of these boundary conditions, when selecting a dipole at one of the edges, its "neighbors" will actualy be the dipole on the opposite edge, in line with it. For example, if we have a row of dipoles, the neighbor of the first dipole will be the last dipole in the row.

I consulted the textbook, which suggested using periodic boundary conditions. If you look at my Test notebook, you will see that I experimented with evaluating the edges with the physical number of neighbors each dipole had.

Additionally, this method fits within the larger theory that the critical temperature will be proportional to the number of neighbors, since now we have normalized our simulation such that each dipole has an equal number of neighbors.

```python
def ising(sizeGrid,T,startImageONOFF,heatMapONOFF):
    #Itialization
    d=randomGrid(sizeGrid)
    h=np.zeros([sizeGrid,sizeGrid])
    xlab="T=", T, "[ε/k]"
    st=startImageONOFF
    hm=heatMapONOFF

    #Determine Graphic output
    if st=="ON":
        plt.subplot(2,2,1)
        plt.title("Initial State")
        plt.xlim(0,len(d))
        plt.ylim(0,len(d))
        plt.xlabel(xlab)
        plt.grid()
        markDipole(d, "Black")
        row=2
        col=2
        indexx=2
        indexh=3
    elif hm=="ON":
        row=2
        col=2
        indexx=1
        indexh=2
    else:
        row=1
        col=1
        indexx=1

    #mainloop
    for i in range(0,100*sizeGrid**2):
        x=np.random.randint(0,sizeGrid)
        y=np.random.randint(0,sizeGrid)
        Ediff=deltaU(x,y,d)
        if Ediff <= 0:
                d[x,y]=-1*d[x,y]
                h[x,y]+=1
        else:
            prob=round(100*np.exp(-Ediff/T))
            rand=np.random.randint(1,101)
            if rand <= prob:
                d[x,y]=-1*d[x,y]
                h[x,y]+=1

    #Display Heat map
    if hm=="ON":
        plt.subplot(row,col,indexh)
        plt.title("Heat Map")
        plt.xlim(0,len(d))
        plt.ylim(0,len(d))
        markIndex(h)

    #Calculate Magnetization
    magnetization(d)

    #Display final state of simulation
    plt.subplot(row,col,indexx)
    plt.title("Final State")
    plt.suptitle("Ising Simulation", fontsize=30)
    plt.xlabel(xlab)
    plt.xlim(0,len(d))
    plt.ylim(0,len(d))
    plt.grid()
    markDipole(d, "Black")
```

Okay, so there is a lot of stuff to break down within the main program. After getting the primary code to work, I went back in and added quality of life features to aid my analysis.

First let's begin with the inputs.

- sizeGrid = input the size of the N x N model we want to produce
- T = Temperature in units of $\frac{\epsilon}{k}$
- startImageONOFF = Toggle whether you want the program to show you your starting state
- heatMapONOFF = Toggle whether you want the program to show you a heat map of how many times each dipole has flipped

The initialization section generates our starting array, generates an empty array for the heat map to tally each dipole's number of flips, and defines some variables.

The next section of code determines how it should arrange the graphical output based on which features you chose to display, working out the subplots and positions of each graph.

The main loop is the heart of my simulation, so let's break it down. First, the simulation runs for $(100 * sizeGrid)^2$ iterations. This is quite a lot, but it ensures that every dipole has 100 chances, when picked randomly, to flip. The structure follows what was written in the introduction.

We use Numpy to pick a random dipole by generating random intergers for the dipole's x and y position within the data array. Next, it computes the energy difference sign using the deltaU function. In the case of a positive energy difference, the probability is first calculated by rounding the probability to an interger. Next, a random number between 1 and 100 is generated by Numpy, and compared to the probability. If the number is less than or equal to the probability, we count that as a flip.

In order to flip a dipole, we multiply it by -1. We count each flip by adding +1 to that dipole's counterpart in the heatmap array.

```
def markIndex(data):
    size=len(data)
    for i in range(0,size):
        for j in range(0,size):
            if data[i,j]<=10:
                plt.fill_between((i,i+1),j,j+1, color="white")
            elif data[i,j]>10 and data[i,j]<=20:
                plt.fill_between((i,i+1),j,j+1, color="papayawhip")
            elif data[i,j]>20 and data[i,j]<=30:
                plt.fill_between((i,i+1),j,j+1, color="navajowhite")
            elif data[i,j]>30 and data[i,j]<=40:
                plt.fill_between((i,i+1),j,j+1, color="lightsalmon")
            elif data[i,j]>40 and data[i,j]<=50:
                plt.fill_between((i,i+1),j,j+1, color="coral")
            elif data[i,j]>50 and data[i,j]<60:
                plt.fill_between((i,i+1),j,j+1, color="orangered")
            elif data[i,j]>60:
                plt.fill_between((i,i+1),j,j+1, color="red")
```

After the main loop is complete, this section of code generates the heat map. The heat map only counts the number of times each dipole flips, not the number of times it is evaulated. In testing, I played around with the spacing and colors of the heat map. At low temperatures, when the dipoles have a lower probability of flipping, the heatmap is relatively diverse. As temperature increases, we have more dipoles that flip more often.

The color scale works like you would expect, with each bracket getting closer to red as the count increases. I tried to make use of the named colors within Python and arrange them in a gradient scale.

```
def magnetization(data):
    m=0
    for i in range(0, len(data)):
        for j in range(0, len(data)):
            m+=data[i,j]
    mag=m/(len(data)**2)
    print("Magnetization:",mag*100, "%")
```

Lastly, we compute the magnetization of our domain by adding up all of the values of the dipoles. It adds up all of the dipole values from the data array using a nested for loop and divides by the total number of dipoles to calculate a magnetization percentage. We expect a higher percentage at low temperatures (below critical temperature) and a low percentage at high temperatures.

In the last sections of code, we generate our output grid of dipoles. I chose to have the graphics generated after the main loop is done iterating. It made more sense to have all the calculations and data stored in an array and then generate an image based on the final array.

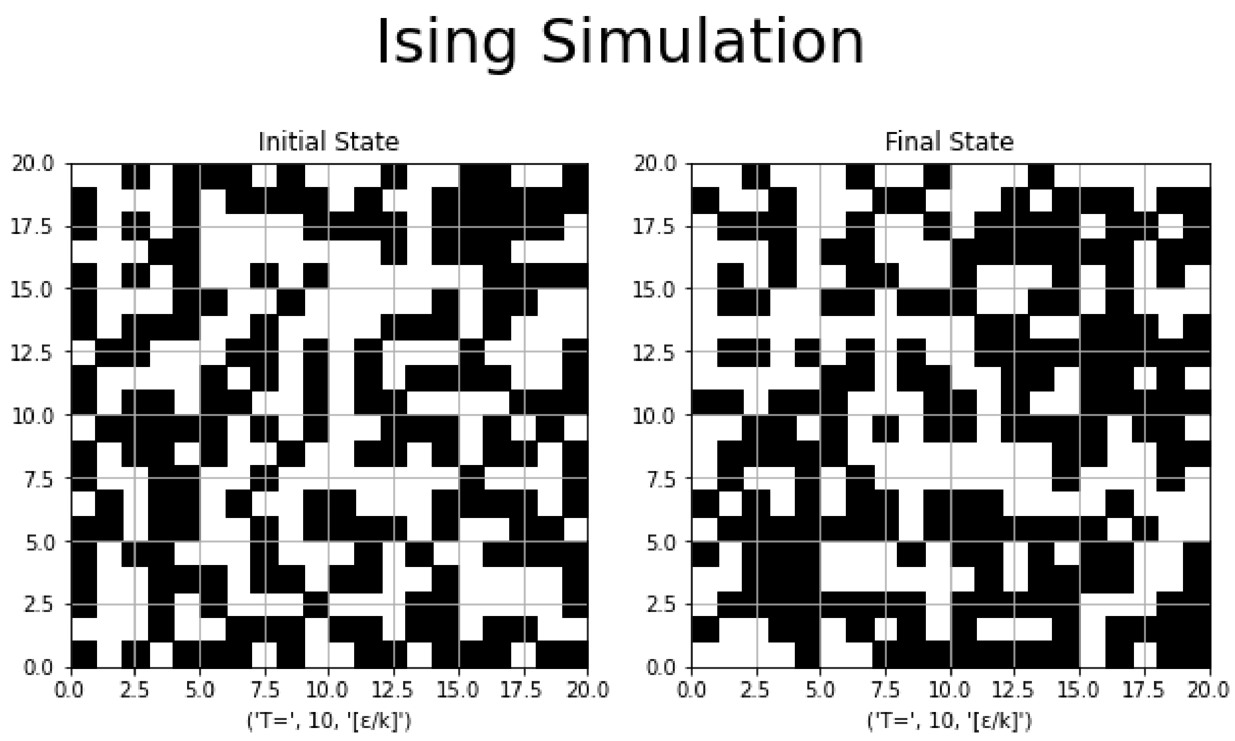
Although I plan to do my analysis of the program with a 20x20 grid, the ising program is capable of handinging any reasonable size. I had some fun testing with 100x100 array.

## Analysis

Let's begin by estimating the critical temperature of our system, by starting with a relatively high temperature, and working our way down until we reach a system with shows critical temperature behavior.

```
ising(20,10,"ON","OFF")
```

```
Magnetization: 3.0 %
```

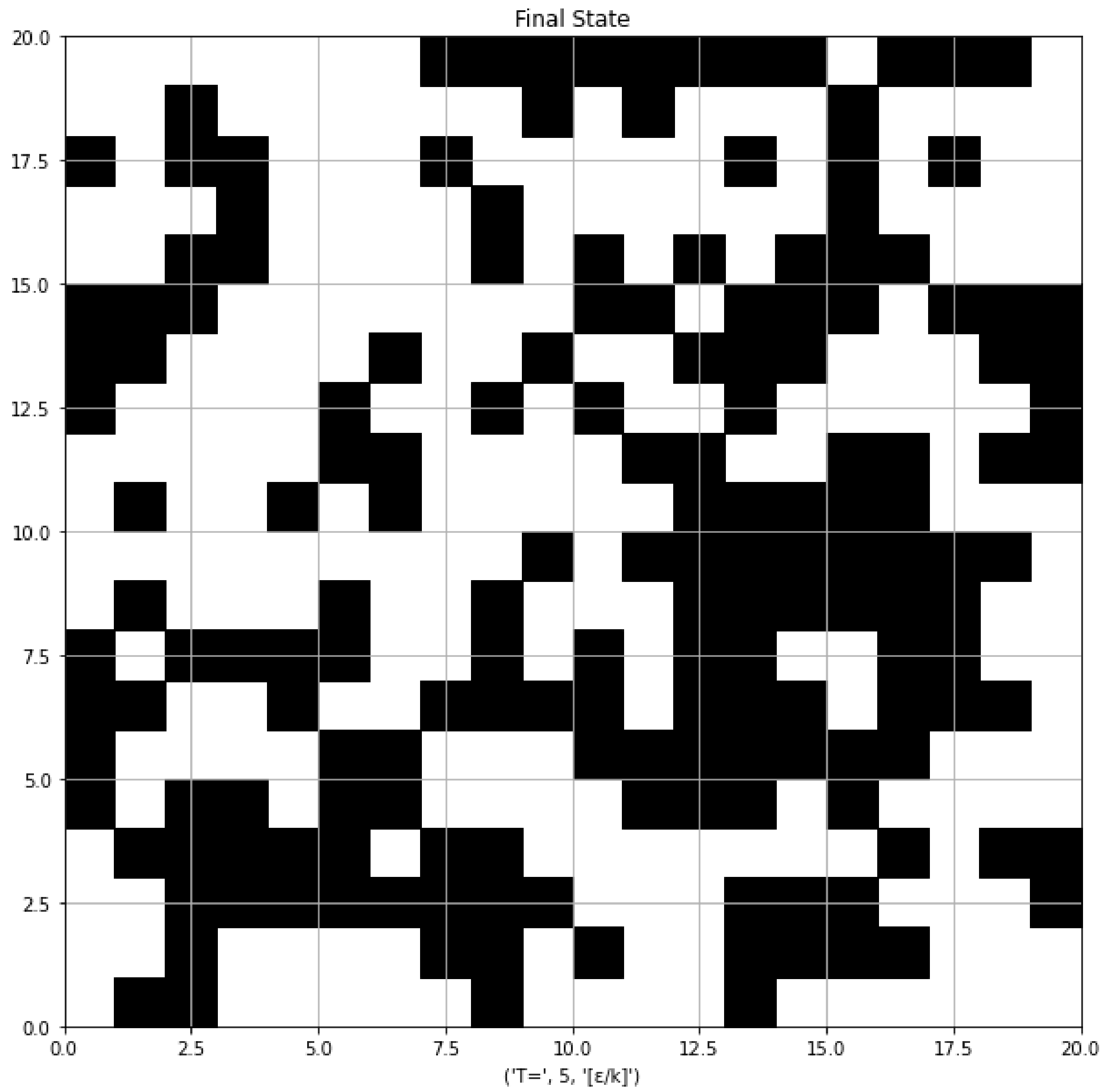


At T=10, the final state is almost random. The dipoles are only showing a slight tendency to align with their neighbors, but the thermal interactions are winning. In fact, it is impossible to distinguish our final state from our initial.

```
ising(20,5,"OFF","OFF")
```

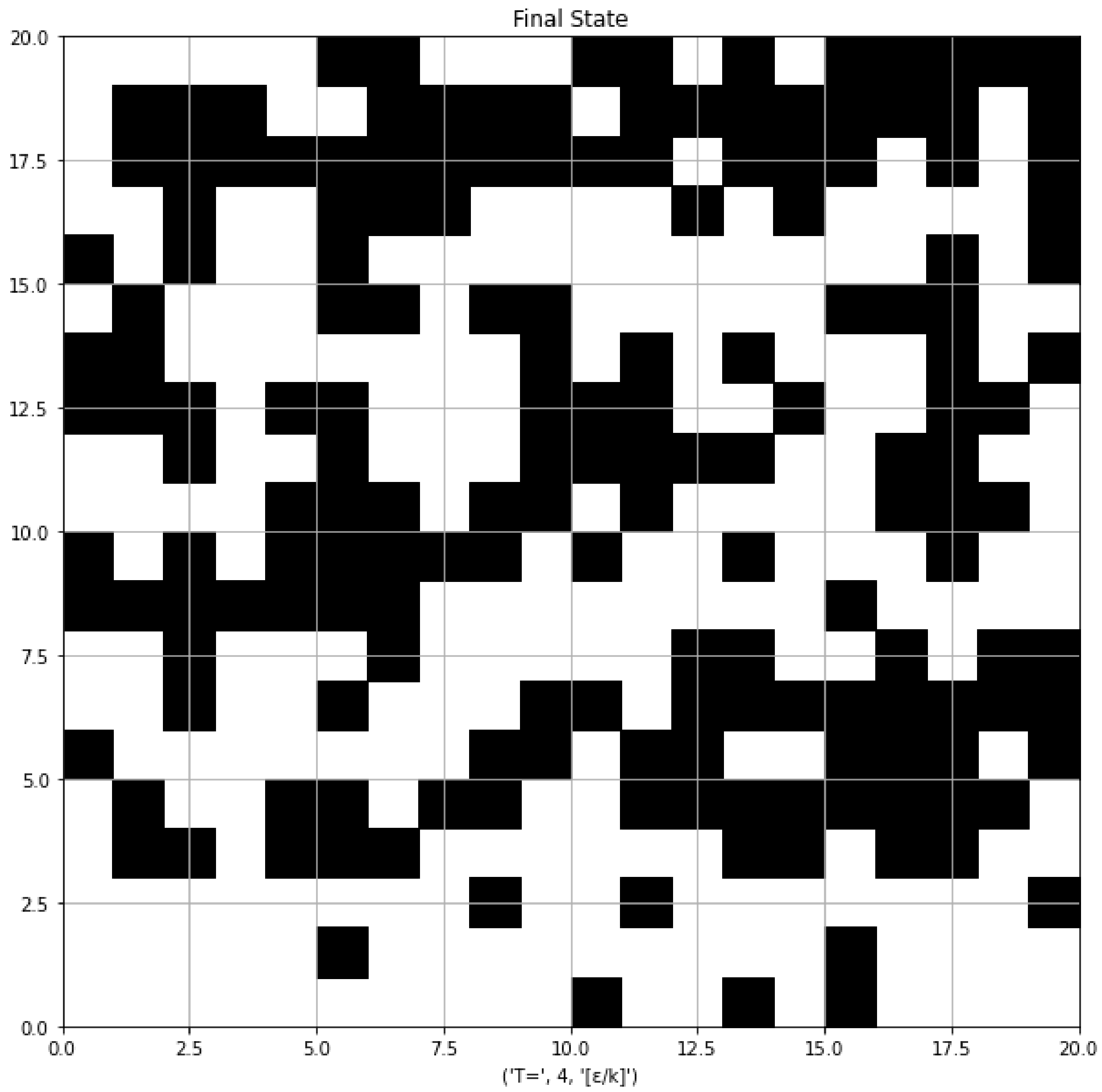
```
Magnetization: -15.0 %
```

# Ising Simulation



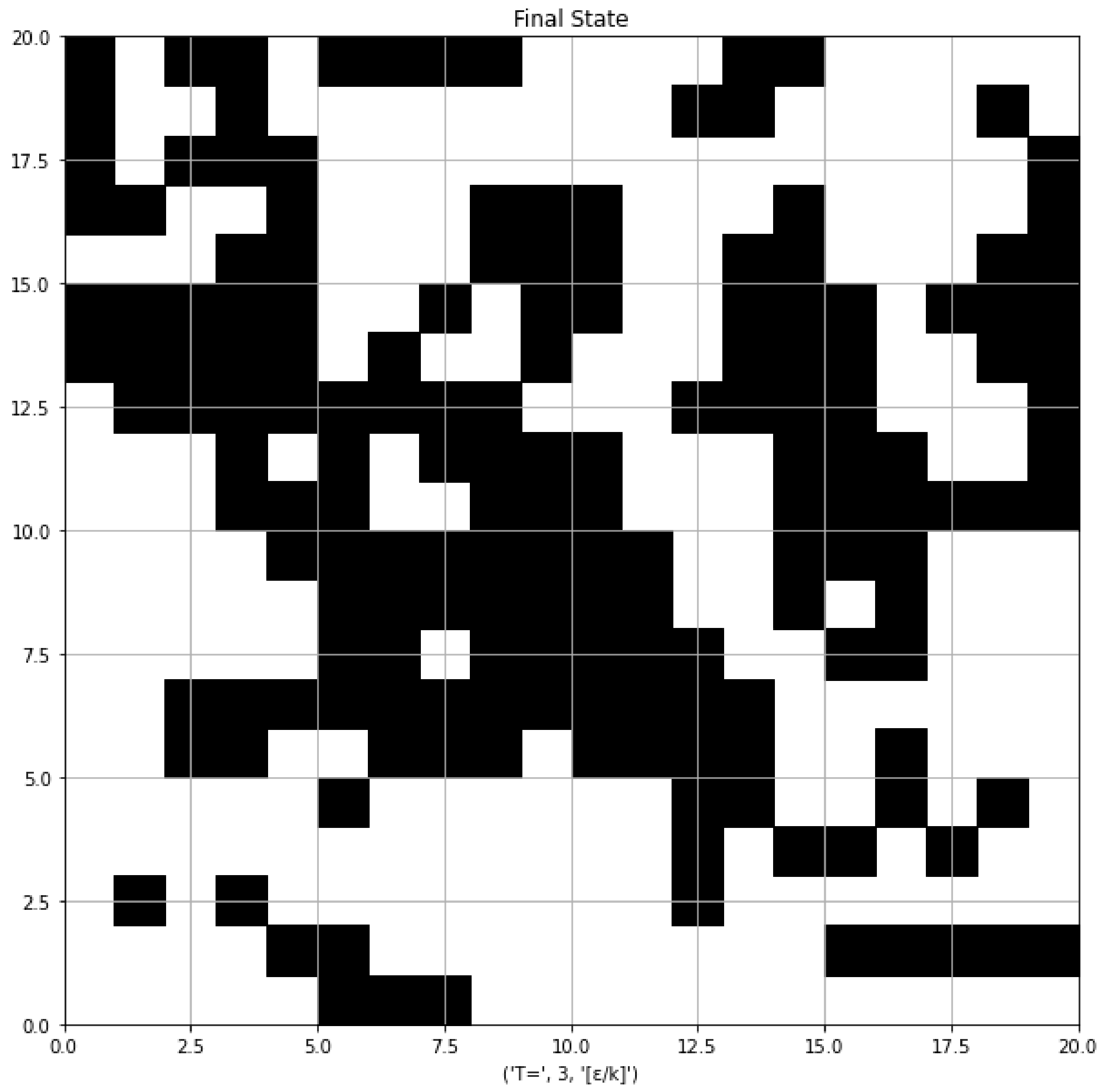
```
ising(20,4,"OFF","OFF")

Magnetization: -13.5 %
```

# Ising Simulation



```
ising(20,3,"OFF","OFF")

Magnetization: -15.0 %
```

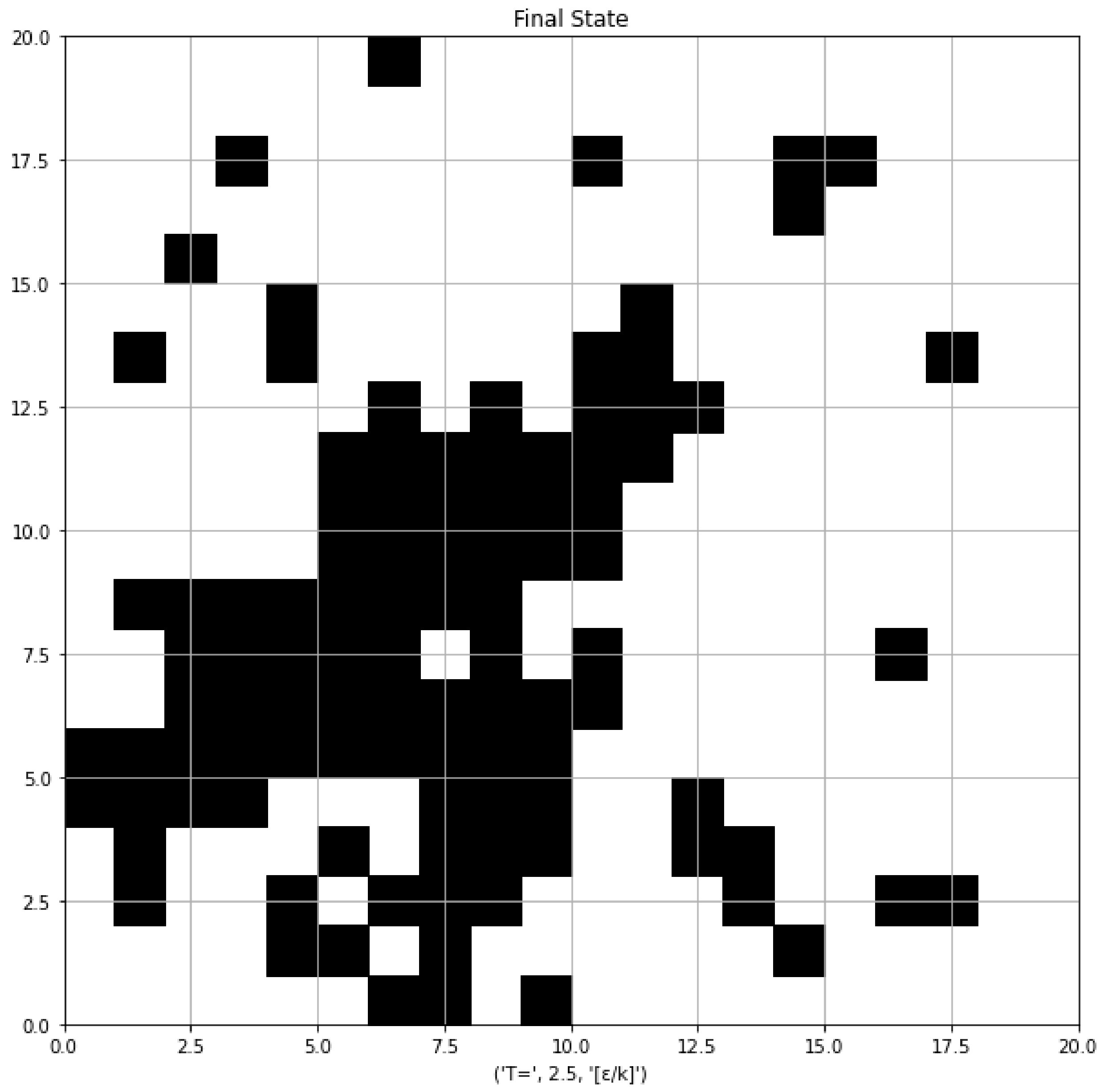
Ising Simulation



As the temperature decreases, we are starting to see clusters of dipoles forming. This tells us that the dipoles are starting to have a greater tendency to align with their neighbors.

```
ising(20,2.5,"OFF","OFF")

Magnetization: -48.5 %
```

# Ising Simulation

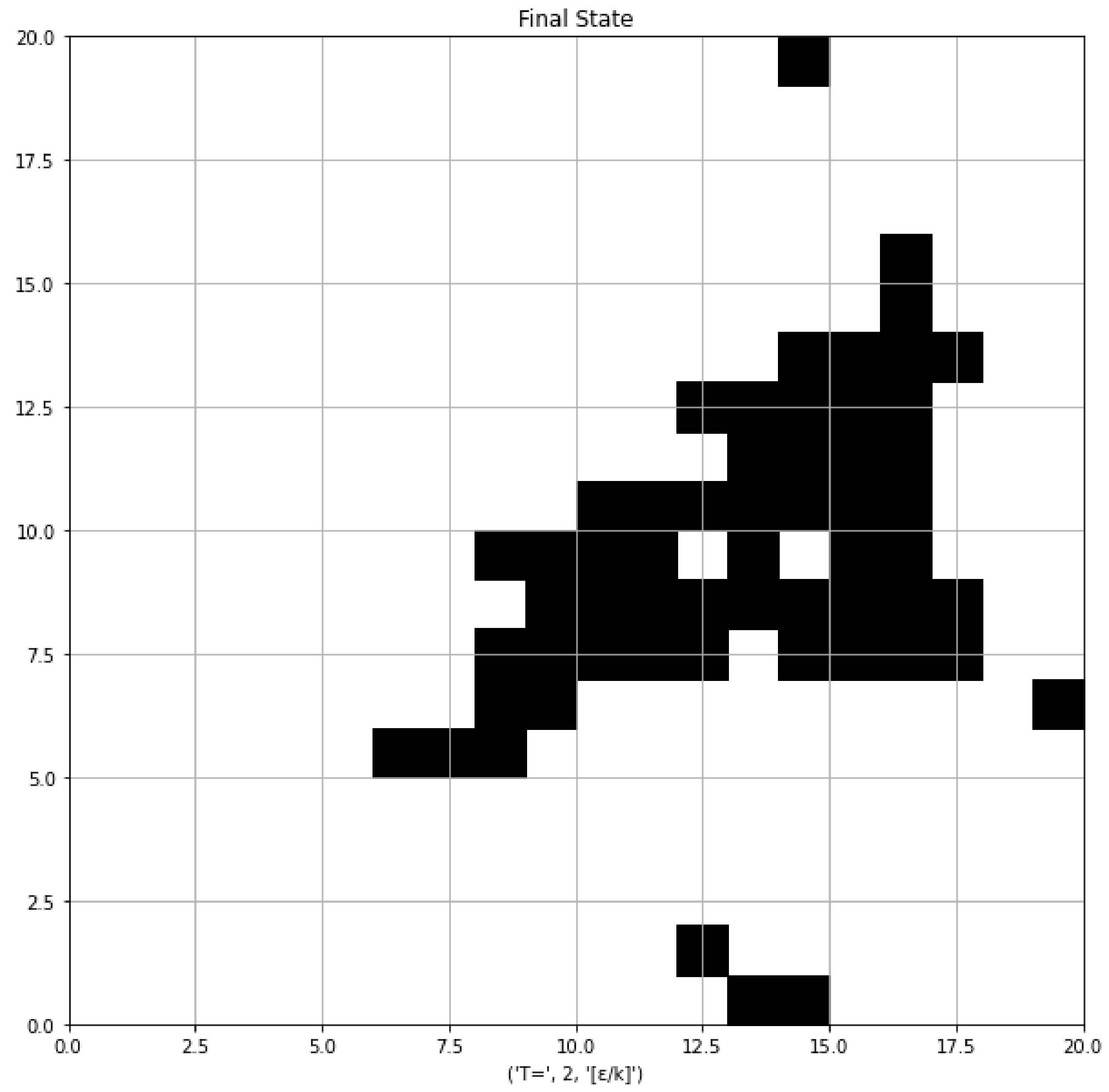


Here we go! We are seeing a spike in magnetization. We can see well-defined clusters, although there still are a few stray dipoles.

```
ising(20,2,"OFF","OFF")
```

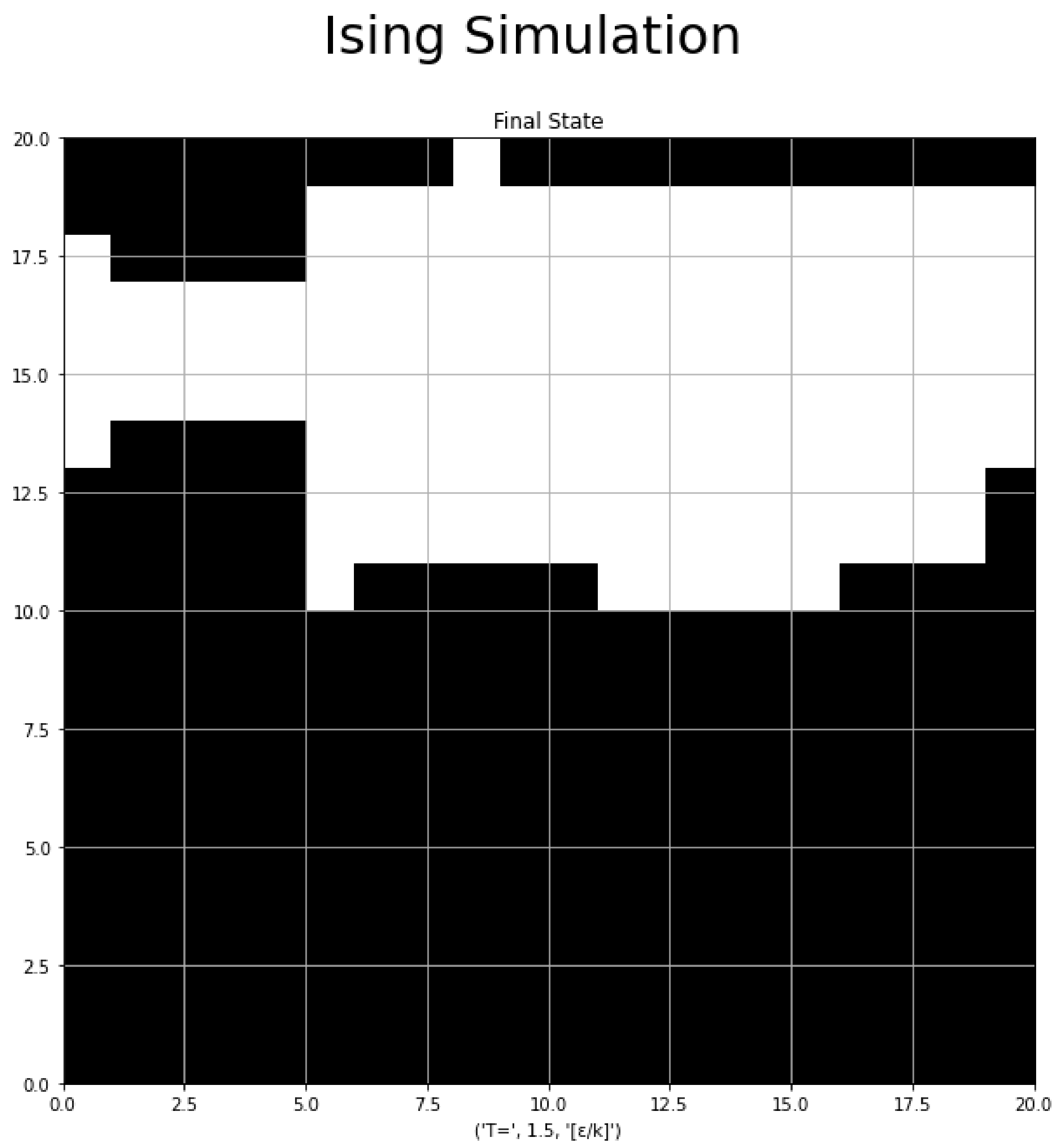
```
Magnetization: -71.5 %
```

Ising Simulation



We can see two distinct clusters of dipoles, which favors a negative magnetization.

```
ising(20,1.5,"OFF","OFF")

Magnetization: 28.999999999999996 %
```

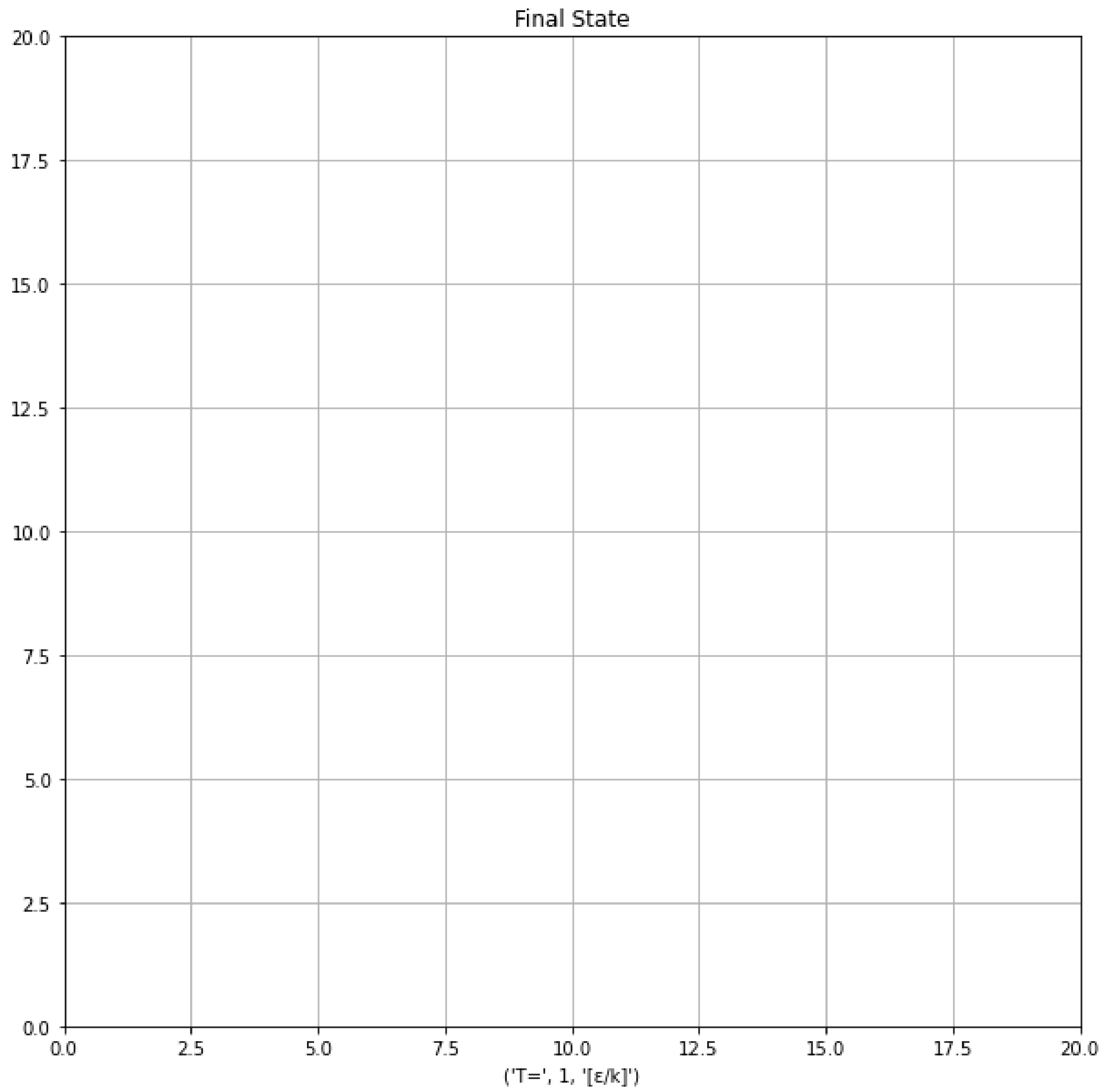


We can see that our system has settled into a meta-stable state with two clusters, one with positive and one negative.

```
ising(20,1,"OFF","OFF")
```

```
Magnetization: -100.0 %
```

Ising Simulation

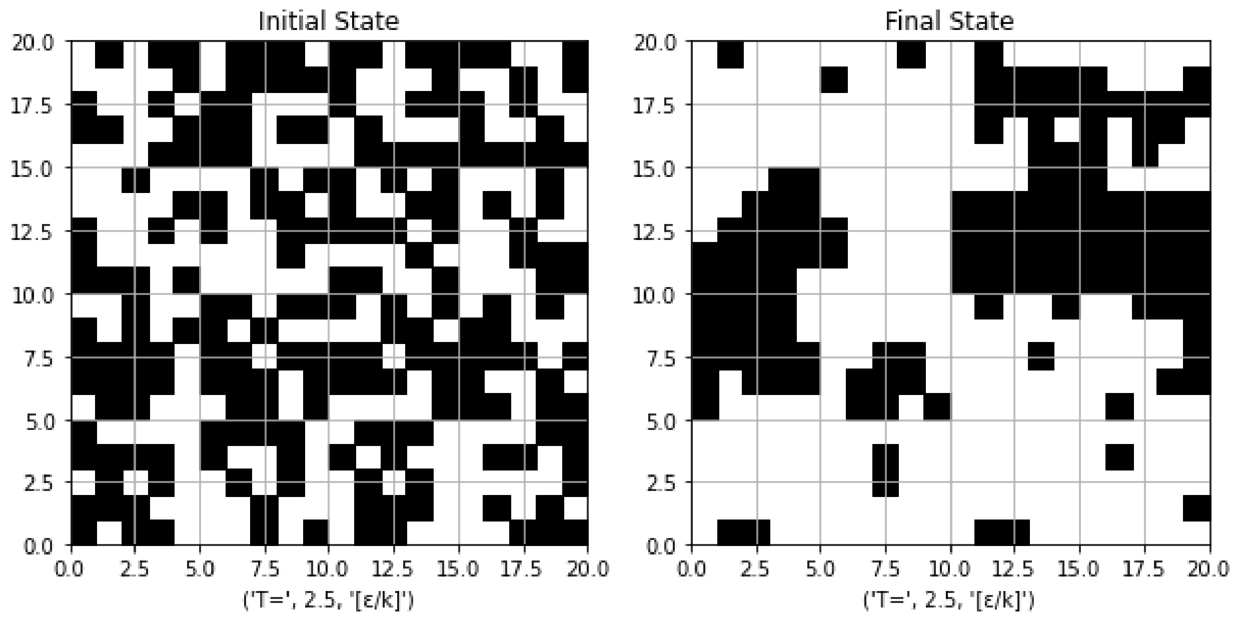


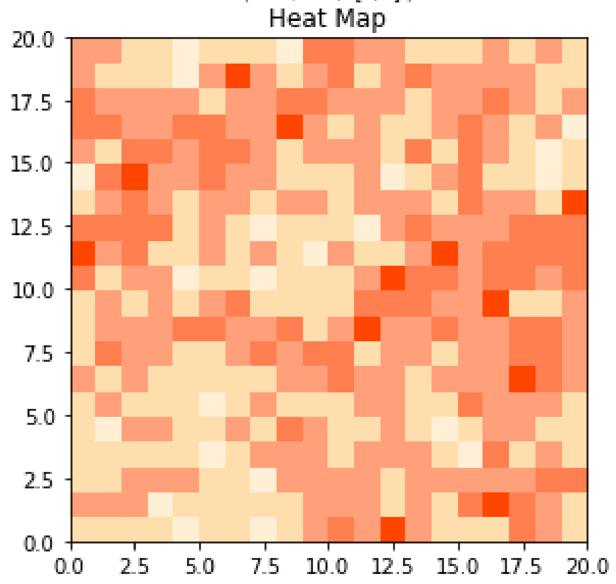
At T=1, we can see that our system has completely magnetized.

Based on these results, we see that our critical temperature is some where between 2 and 2.5 due to the spike in magnetization. Let's take a closer look at these cases.

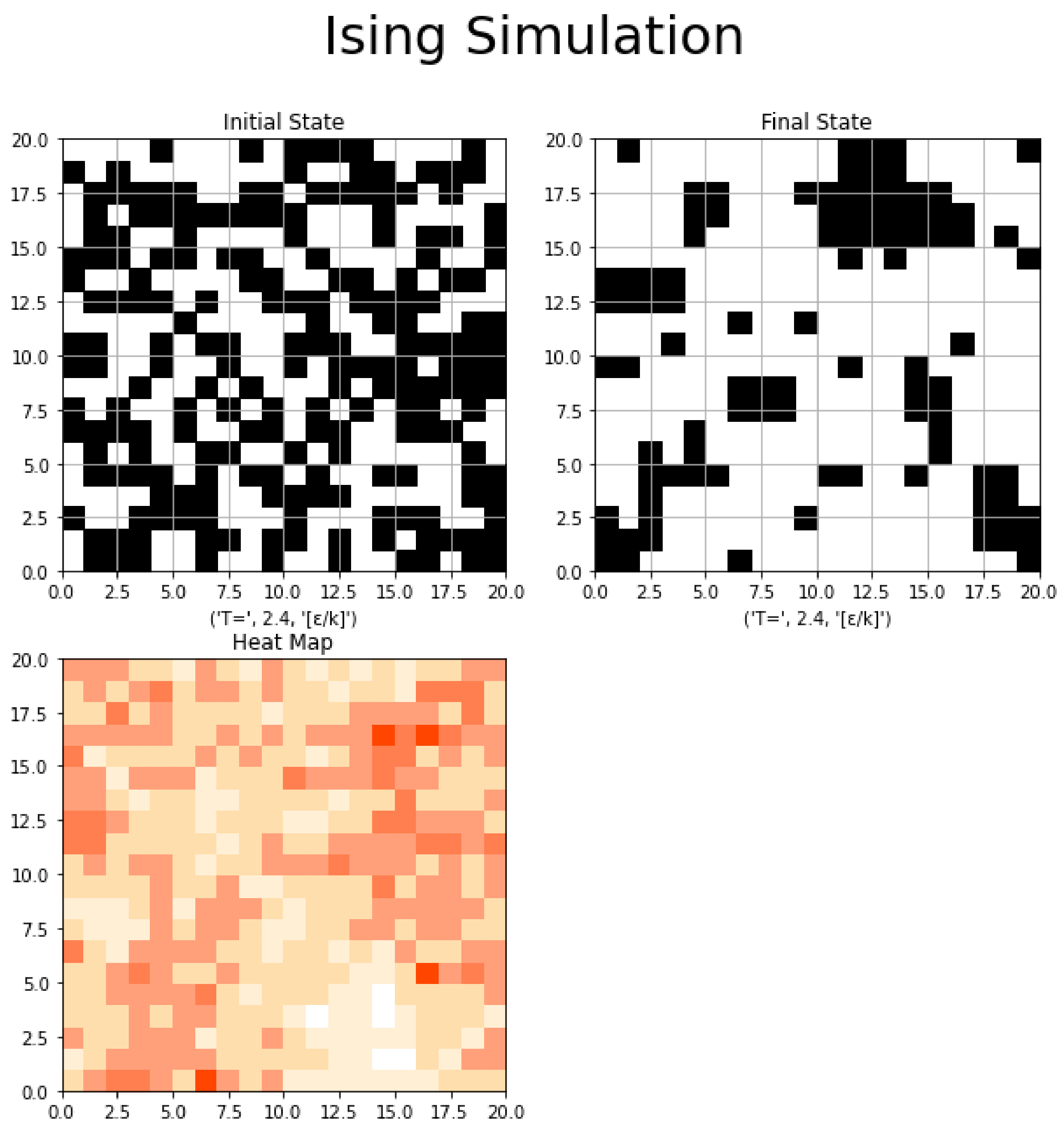
```
ising(20,2.5,"ON","ON")

Magnetization: -32.0 %
```

# Ising Simulation



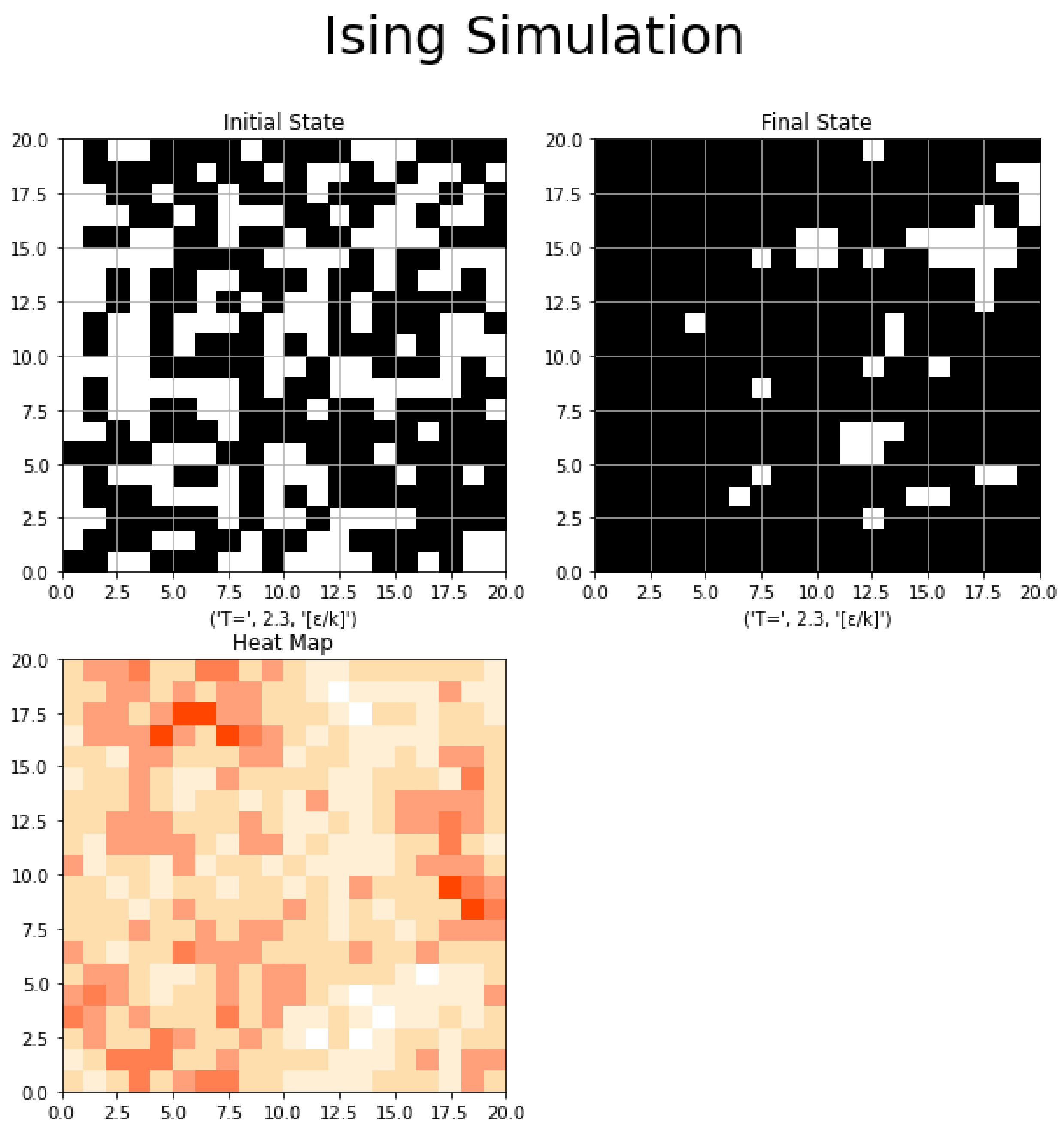


```
ising(20,2.4,"ON","ON")

Magnetization: -51.5 %
```

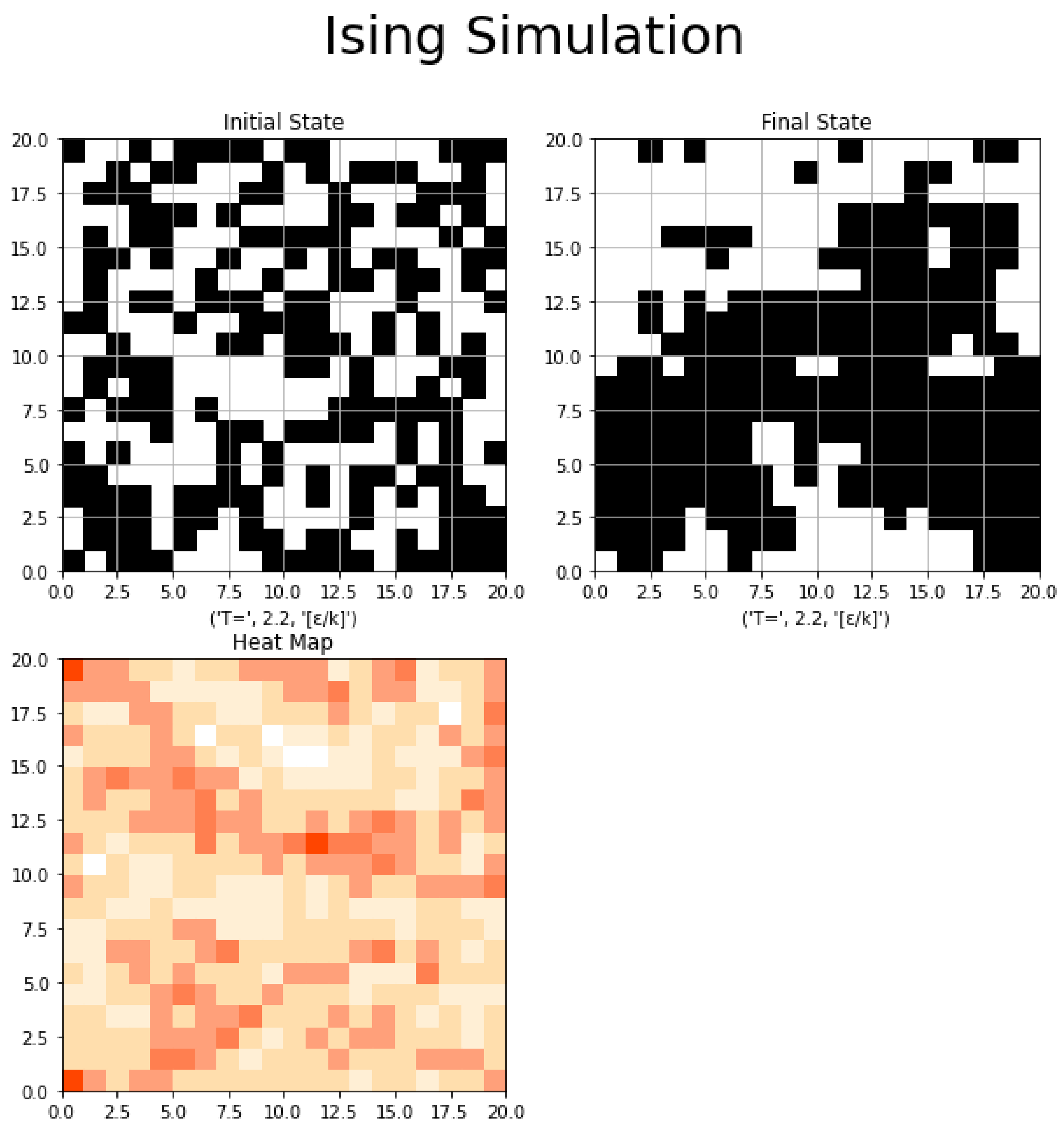


```
ising(20,2.3,"ON","ON")
```

```
Magnetization: 79.5 %
```

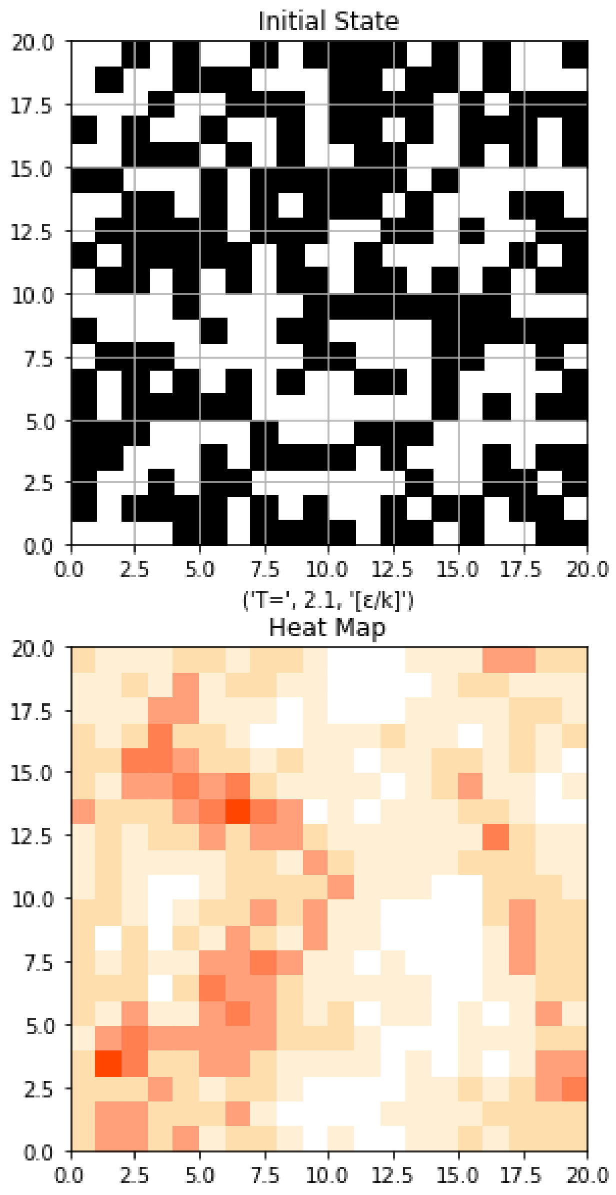


```
ising(20,2.2,"ON","ON")

Magnetization: 19.5 %
```

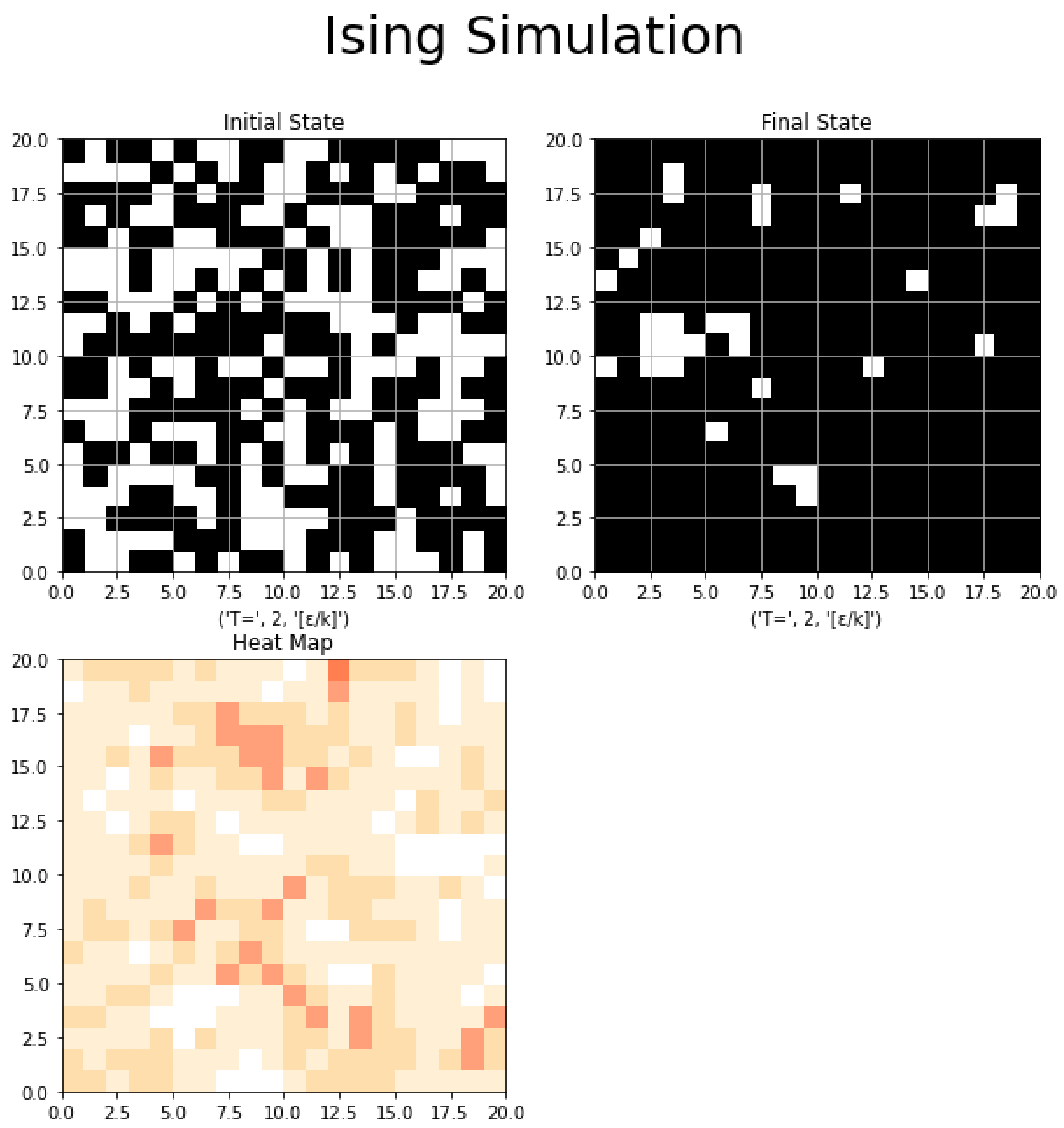


```
ising(20,2.1,"ON","ON")

Magnetization: 55.00000000000001 %
```

# Ising Simulation



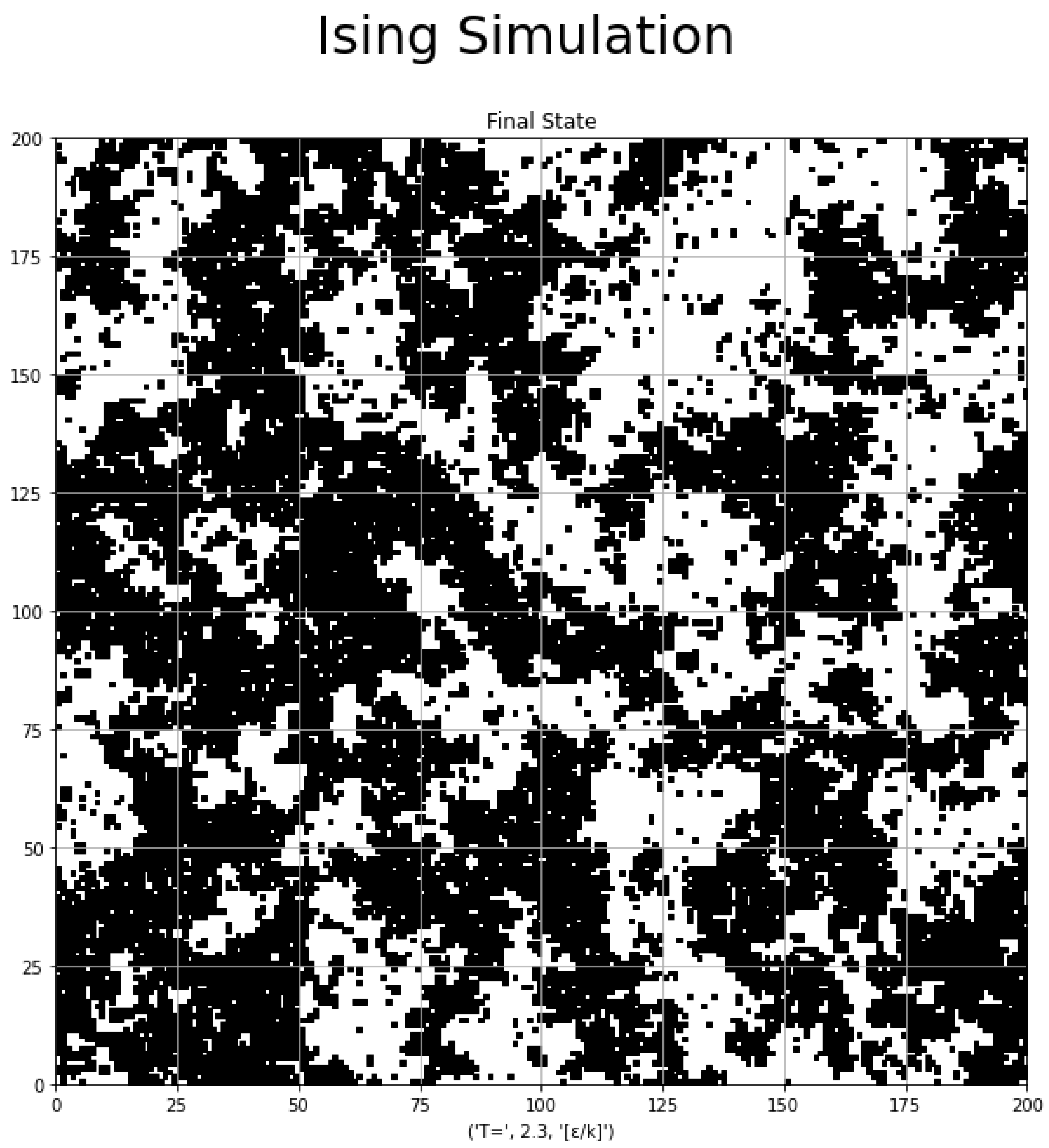
```
ising(20,2,"ON","ON")

Magnetization: 85.0 %
```



From this closer inspection, I estimate that our critical temperature is around T=2.3. Let's see what happens at this temperature with a much larger lattice.

```
ising(200,2.3,"OFF","OFF")
```

```
Magnetization: 6.515 %
```

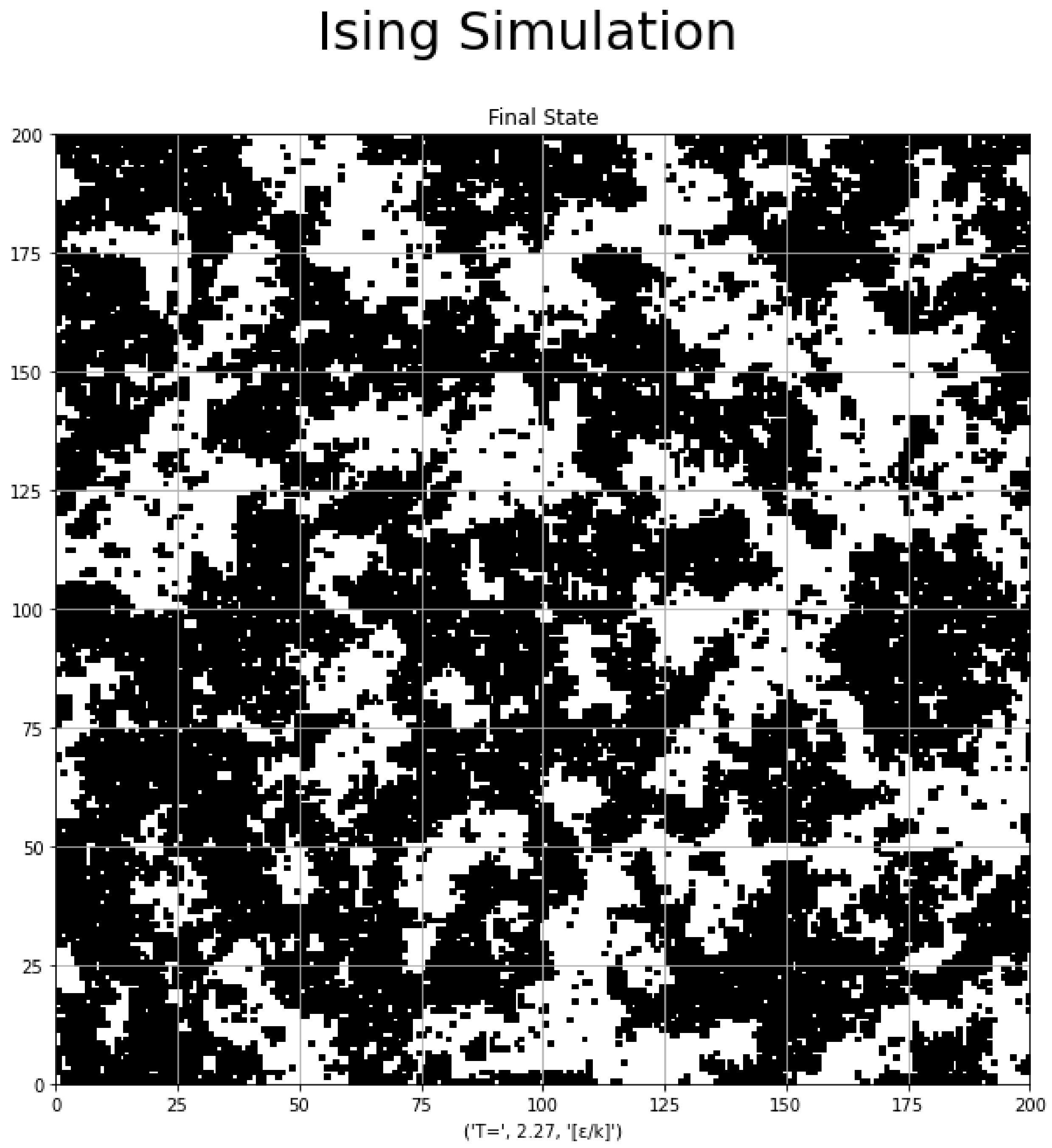


In this lattice, we see lots of clusters or various sizes that range from individual dipoles to clusters that are comparable to the size of the lattice.

Our simulation gets us very close to predicting the exact critical temperature, which was determined mathematically by Lars Onsager in the 1940s. Onsager's solution for an Ising ferromagnet is beyond my level of comprehension, but he found the critical temperature to be 2.27, which our simulation gets us very close to. For the sake of it, let's generate a model at the critical temperature.

```
ising(200,2.27,"OFF", "OFF")
```

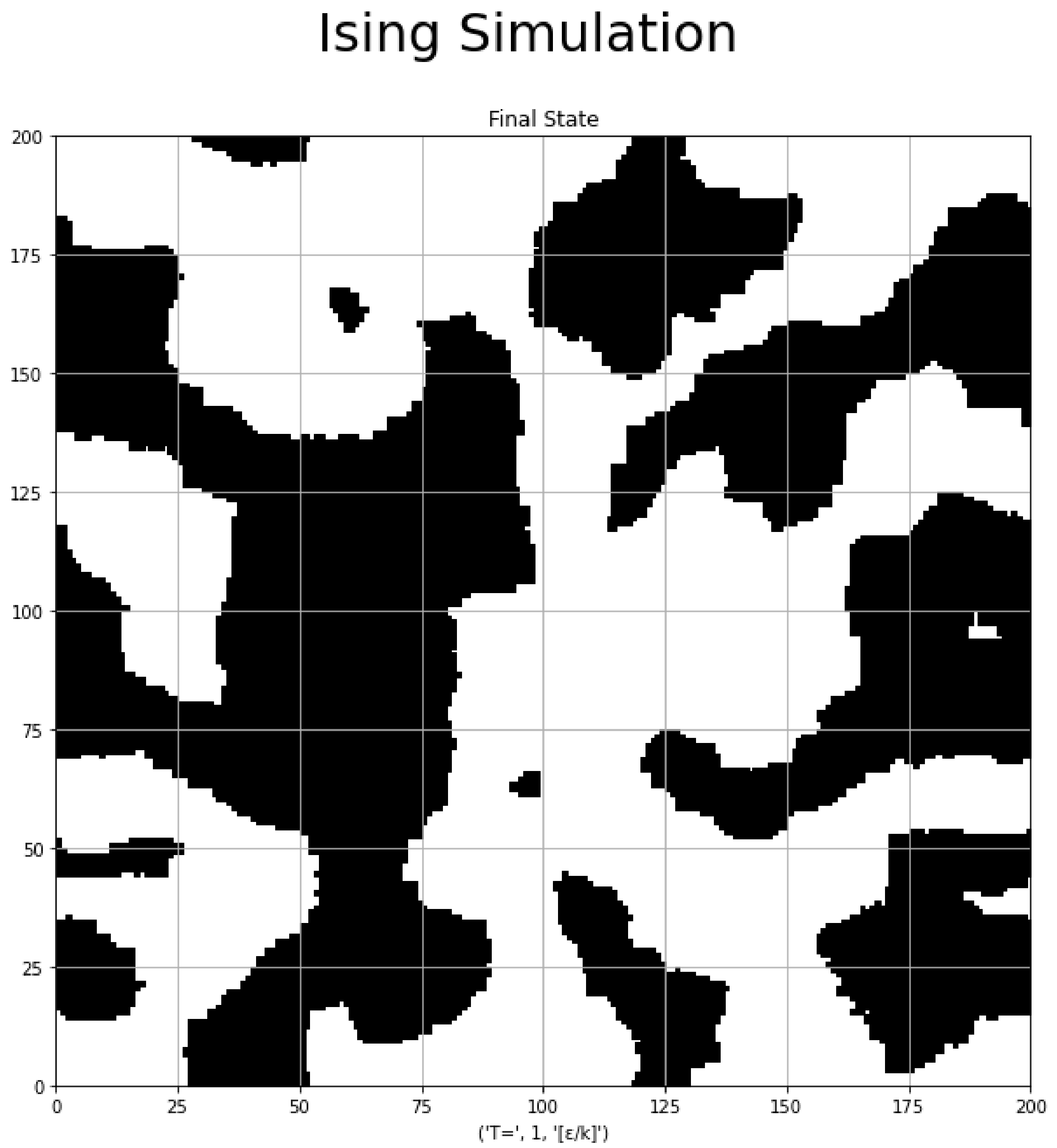
```
Magnetization: 12.49 %
```



In all honesty, this looks really similar to the T=2.3 model, which shows that our simulation was able to get us really close to the exact solution.

```
ising(200,1,"OFF","OFF")
```

```
Magnetization: -9.84 %
```



A large scale model at T=1 shows clear, large clusters with no individual dipoles. I have had a lot of fun generating these images because they resemble the patterns of cows.

## Conclusion

We were able to estimate the critical temperature of a 2D Ising ferromagnet using a Monte Carlo simmulation, within close proximity to the exact solution. Ther are some limitations with our simmulation however. We rely on the random intergers generated by Numpy to pick our dipole for evaluation and to determine the probability of that dipole flipping. Since these are not truly random numbers, after many iterations, which could be the case for an exceedingly large lattice, divergent patterns would form and the process would not be random.

Additionally, this simmulation was made within the context of an Ising model which employed periodic boundary conditions. These assumptions made by the Ising model give us a qualitative insight into the behavior of ferromagnets, but real ferromagnets are much more complex than this model. Real ferromagnets are in 3D crystal lattices, where atomic properties and applied external magnetic fields are key factors.

---

## Refrence Textbook

Schroeder, D. V. (2008). Systems of Interacting Particles. *In An Introduction to Thermal Physics* (pp. 327-357). San Francisco, CA: Addison Wesley Longman.