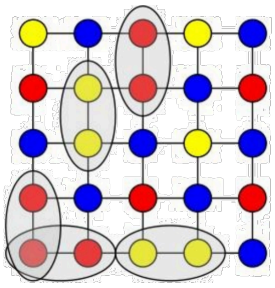
Open Access



# Simulation of 2D 3-State Potts Model

## A Metroplis Monte Carlo Approach

**Xiaoxuan Yu**[1]

[1]College of Chemistry and Molecular Engineering, Peking University

**Abstract**

The Potts model is a generalization of the Ising model, which is a model of ferromagnetism in statistical mechanics. By using the Metropolis algorithm, a 2D 3-state Potts model is simulated under different temperatures. We studied the internal energy, magnetization, specific heat and magnetic with reference to the temperature and found that the system undergoes a phase transition at a critical temperature around 1.05.

**Keywords** Monte Carlo, Metropolis Algorithm, Potts Model, Phase Transition

**Course Project**

**Submitted** Nov 28, 2023
Due Date Nov 30, 2023

**Correspondence to**
Xiaoxuan Yu
xiaoxuan_yu@pku.edu.cn

**Data Availability**
Associated code repository is available on GitHub.

**Honesty Statement**
The author declares that the project was completed independently and without plagiarism.

## 1. Background

### *1.a. Spin models*

The Potts model is one of a group of models called spin models. In a spin model, each site on a lattice is assigned a spin variable. The spin variable can take on a discrete set of values, which are usually integers.

1.a.i. *Ising model:*

The Ising model is the most basic spin model. In the Ising model, the spin variable can take on only two values, +1 or –1. The Ising model is a model of ferromagnetism in statistical mechanics. The Hamiltonian of the Ising model is given by

$$H = -J \sum_{\langle i,j \rangle} S_i S_j \tag{1}$$

where $S_i$ is the spin variable at site $i$, $J$ is the coupling constant, and the sum is over all pairs of nearest-neighbor sites.

1.a.ii. *Potts model:*

The Potts model is a generalization of the Ising model. In the Potts model, the spin variable can take on $q$ different values, where $q$ is an integer greater than or equal to 2. The Hamiltonian of the Potts model is given by

$$H = -J \sum_{\langle i,j \rangle} \delta(S_i, S_j) \tag{2}$$

The Kronecker delta function $\delta(S_i, S_j)$ is defined as

$$\delta(S_i, S_j) = \begin{cases} 1 \text{ if } S_i = S_j \\ 0 \text{ if } S_i \neq S_j \end{cases} \tag{3}$$

If magenetic field is applied, an extra term is added to the Hamiltonian

$$H = -J \sum_{\langle i,j \rangle} \delta(S_i, S_j) - h \sum_i S_i \tag{4}$$

where $h$ is the magnetic field strength.

*1.b. Interested physical quantities in Potts model*

Pratically, physicists are interested in the following physical quantities in Potts model:

1.b.i. *Internal energy:*

$$u = \frac{\langle H \rangle}{N^2} \tag{5}$$

1.b.ii. *Specific heat:*

$$c = \frac{k_B \beta^2 \left( \langle H^2 \rangle - \langle H \rangle^2 \right)}{N^2} \tag{6}$$

where $k_B$ is the Boltzmann constant and $\beta = 1/(k_B T)$ is the inverse temperature.

1.b.iii. *Magnetization:*

$$m = \frac{\left\langle \sum_i S_i \right\rangle}{N^2} \tag{7}$$

1.b.iv. *Characteristic length:* Define the spatial correlation function

$$C(i,j) = \langle S_i S_j \rangle - \langle S_i \rangle \langle S_j \rangle \tag{8}$$

Then we can define

$$\Gamma(k) = C(i,j)|_{|i-j|=k} \approx \frac{1}{4N^2} \sum_i \sum_{j \in S_i} C(i,j) \tag{9}$$

where

$$S_i = \{ i | i - j = \pm(k,0) \text{ or } (0,k) \} \tag{10}$$

And the characteristic length $\xi$ is the length that $\Gamma(k)$ decays to 0. Thus the correlation length is given by

$$\Gamma(k) \propto \Gamma_0 \exp(-k/\xi), \quad k \gg 1 \tag{11}$$

*1.c. Phase transition in Potts model*

The Potts model undergoes a phase transition at a critical temperature $T_*$. Below $T_*$, the system is in a ferromagnetic phase, where the spins are aligned. Above $T_*$, the system is in a paramagnetic

phase, where the spins are randomly oriented. The critical temperature $T_*$ depends on the coupling constant $J$ and the dimensionality of the system. For a 2D square lattice, the critical temperature is given by

$$T_* = \frac{J}{k_B \ln(1+\sqrt{q})} \tag{12}$$

where $k_B$ is the Boltzmann constant and $q$ is the number of spin states. In the simulation approach, the critical temperature can be determined by finding a peak in the specific heat.

## 2. Problem setup

In the simulation, a 2D square lattice with periodic boundary is used as the target system. The lattice has size $N = 32$. For simplicity, we set the coupling constant $J = 1$ and the magnetic field strength $h = 0$. Besides we use the reduced temperature s.t. $k_B = 1$. $q$ is set to 3, which means the spin variable can take on 3 different values, 1, 2, and 3.

The initial state of the system is randomly generated. The simulation is performed under different reduced temperatures $T$ from 0.5 to 2.5. For each temperature, the system is evolved for 10,000,000 steps. The trajectory of the system is recorded every 1,000 steps and thus the physical quantities are calculated with the same interval.

We selected the last half of the trajectory for analysis. The first half of the trajectory is used to ensure the system has reached equilibrium. The physical quantities are calculated by averaging over the last half of the trajectory.

## 3. Implementation of the simulation

### *3.a. Setup the dependencies*

In the simulation, `numpy` is used for the main computation. For better performance, `numba` is used to compile the functions just in time. `multiprocessing` is used to parallelize the computation, and `functools.partial` is used to create partial functions for the parallelization.

```
import numpy as np
from functools import partial
from numba import njit, prange, objmode, jit
import multiprocessing as mp
```

### *3.b. The Potts Hamiltonian*

The Hamiltonian of Potts model is implemented as the following `potts_energy` function

```
@jit(forceobj=True)
def potts_energy(spins, J, h):
    """计算 Potts 模型的能量

    Args:
        spins: 自旋状态,shape=(N, N)的数组,其中 N 是格子数目
        J: 相邻自旋互相作用能量强度

    Returns:
        系统总能量,标量
    """
```

```
    # 获取形状
    N = spins.shape[0]

    # 计算相邻自旋互作用能
    E_J = J * np.sum(spins == np.roll(spins, 1, axis=0))
    E_J += J * np.sum(spins == np.roll(spins, -1, axis=0))
    E_J += J * np.sum(spins == np.roll(spins, 1, axis=1))
    E_J += J * np.sum(spins == np.roll(spins, -1, axis=1))
    E_J += h * np.sum(spins)

    # 求和得到能量
    E = -E_J
    return E
```

The implementation is straightforward. Using the `np.roll` operation, the spin matrix is shifted periodically by one unit horizontally and vertically. Thus we can effectively calculate the energy of the system by comparing the neighboring spins, and then summing up the results. The influence of the magnetic field is calculated by naively summing up all the spins. An decorator `@jit` is added to the function to compile it just in time. This will significantly improve the performance of the function.

### *3.c. The Metropolis algorithm*

By splitting the Metropolis algorithm into the main loop and the operation of a single step, it is much more easier for `numba` to compile the code. The single step operation is implemented as the following `potts_mc_step` function

```
# Finding neighbors in PBC
@jit
def finding_neighbor(N, i, j):
    return [((i + 1) % N, j), ((1 - i) % N, j), (i, (j + 1) % N), (i, (j - 1) % N)]

@njit(fastmath=True)
def potts_mc_step(spins, J, h, beta, q):
  # 获取形状
  N = spins.shape[0]
  # 选取随机格点
  i = np.random.randint(0, N)
  j = np.random.randint(0, N)
  # 建议新的自旋状态
  spins_new = spins.copy()
  spins_new[i, j] = np.random.randint(1, q + 1)
  # 直接计算能量差，由于只改变了一个格点，只需要计算该格点的能量差
  Eij_old = 0
  Eij_new = 0
  # 计算与相邻格点的能量差
    neighbors = finding_neighbor(N, i, j)
    for neighbor in neighbors:
        Eij_old -= J * (spins[i, j] == spins[neighbor[0], neighbor[1]])
        Eij_new -= J * (spins_new[i, j] == spins[neighbor[0], neighbor[1]])
  dE = Eij_new - Eij_old
  # 判断是否接受
  if dE <= 0 or np.random.rand() < np.exp(-beta * dE):
  spins = spins_new
  return spins
```

It is a standard implementation of the Metropolis algorithm. The only thing to note is that the energy difference `dE` is not calculated by comparing the energy of the new state and the old state. Instead, we only calculate the energy difference of the changed spin. During the calculation, the periodic boundary condition is naturally satisfied by using `finding_neighbor` function. The time complexity is thus reduced to $O(1)$, which is much better than the naive implementation with time complexity $O(N^2)$. The main loop of the Metropolis algorithm is implemented naively and is not shown here. The sampled trajectories are recorded in the format of a `numpy` array as `.npy` files in the disk for further analysis.

### *3.d. Parallel simulation of different temperatures*

The simulation of different temperatures is parallelized by using the `multiprocessing` module. The main idea is to create a partial function of the Metropolis algorithm with a fixed temperature. Then we can use the `map` function of the `multiprocessing.Pool` class to run the simulation of different temperatures in parallel. The implementation is shown as the following `potts_simulate_parallel` function

```
def potts_simulate_parallel_temp(
    init_spin, J, h, q, k, T_series, n_steps, n_step_save=1000, path="./Potts_Data"
):
    pool = mt.Pool(8)

    # partial function
    potts_mc_partial = partial(
        potts_mc_mt,
        spins=init_spin,
        J=J,
        h=h,
        q=q,
        n_steps=n_steps,
        n_step_save=n_step_save,
        path=path,
    )
    # run simulation
    pool.map(potts_mc_partial, T_series)
    return 0
```

The `potts_mc_partial` function is a partial function of the `potts_mc_mt` function, which is the main loop of the Metropolis algorithm, in which parameters other than temperature are all fixed. It is then mapped to the `T_series` array, which contains the temperatures of the simulation. The `potts_mc_partial` function is run in parallel with 8 processes by using the `multiprocessing.Pool` class. The parallelization w.r.t. $h$ can be implemented in a similar way.

### *3.e. Data analysis*

The interested physical quantities are calculated by averaging over the last half of the trajectory. The implementation is shown is the attached notebook `Post-simulation.ipynb`. Simply speaking, the data is loaded from the disk and then averaged over the last half of the trajectory. For internal energy, specific heat, and magnetization, it is done naively.

For the characteristic length, the spatial correlation function is calculated first by convoluting the spin matrix $\Sigma$

$$\Gamma(k) \equiv \langle \Sigma * \mathrm{roll}(\Sigma, \pm k, \mathrm{axix}{=}0 \text{ or } 1) \rangle + \langle \Sigma \rangle * \langle \mathrm{roll}(\Sigma, \pm k, \mathrm{axix}{=}0 \text{ or } 1) \rangle \tag{13}$$

where $A * B \overset{\text{def}}{=} \sum_{i,j} A_{ij} B_{ij}$. Then the characteristic length is calculated by linear fitting the correlation function using `np.optimize.curve_fit`.

The limiting behavior around the critical temperature is also studied by linear fitting the data around the critical temperature using log-log scale.

## 4. Numerical results

### *4.a. Internal energy and specific heat*

The internal energy and specific heat with respect to the temperature are shown in the following figure.

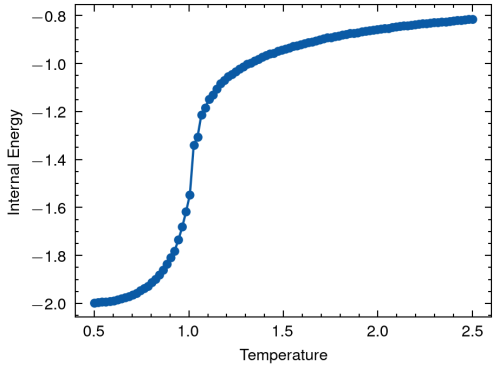



Figure 1: $u$ w.r.t $T$

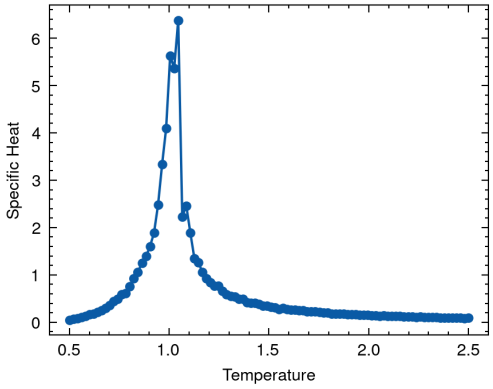



Figure 2: $c$ w.r.t $T$

A peak is observed which is a obvious sign of phase transition. Before the critical temperature, the specific heat ismonotonically increasing while after the critical temperature, it is monotonically decreasing. The **critical temperature** is around

$$T_* = 1.05 \tag{14}$$

while the theoretical value of the critical temperature is

$$T_*^t = \frac{1}{\ln\left(1+\sqrt{3}\right)} = 0.995 \tag{15}$$

which is very close to the simulated value.

By using renormalisation techniques,the theory predicts that for $q \leq 4$, the ferromagnetic to paramagnetic phase transition is second order and otherwise first order. In the simulation, we found that the internal energy is a monotonically increasing function of the temperature. Besides, $u$ is almost continuous. Discontinuity is observed in the specific heat around the peak. Together with the continuity of the internal energy, we can conclude that the phase transition is **second order**, which is consistent with the theory.

### *4.b. Magnetization*

The magnetization with respect to outer magnetic field under different temperatures is shown in the following figure

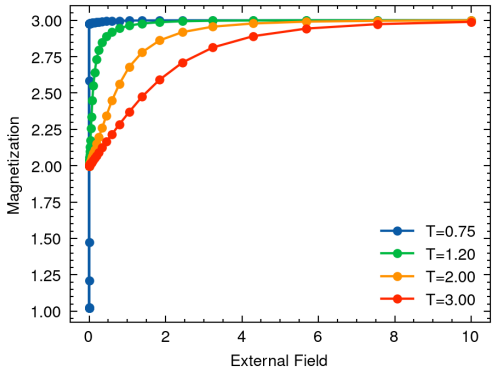



Figure 3: $m$ w.r.t $h$

The dependence of the magnetization on the outer magnetic field clearly shows the magnetic property of 2D lattice before and after phase transition. When temperature is lower than $T_*$, the lattice is in a ferromagnetic phase, which means that once the outer magnetic field is applied, the spins will be aligned along the direction of the outer magnetic field, even the outer magnetic field is very weak. The $T = 0.75$ line in Figure 3 shows this property. We can easily observe that the magnetization has a huge jump near $h = 0$, and the lattice is almost fully magnetized, symbolized by the magnetization close to 3.

When temperature is higher than $T_*$, the lattice is in a paramagnetic phase, which means that the spins will not be aligned along the direction of the outer magnetic field unless the outer magnetic field is strong enough. The other lines in Figure 3 shows this property. In fact, we found that with the increase of temperature, the magnetization curve becomes more and more smooth, indicating the influence of the outer field becomes more obvious.

Another thing to note is the magnetization without outer magnetic field. For ferromagnetic phase, the magnetization is close to 1, which means that the spins are almost **fully aligned**. On the other hand, for paramagnetic phase, the magnetization is close to 2, leading to the **randomlly oriented** nature of the spins. The below figure shows the states of the lattice under different temperatures.

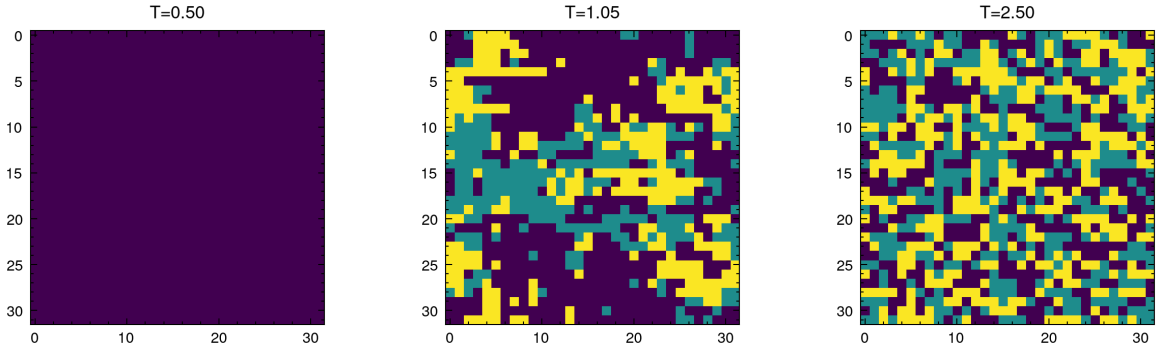



Figure 4: States of the lattice under different temperatures

For $T = 0.5 < T_*$, the spins are almost fully aligned. For $T = 1.05 \approx T_*$, the spins are mixture of oriented and random states, resulting in the jump of specific heat at the critical temperature. For $T = 2.5 \gg T_*$, the orientation of spins are almost randomly distributed. This is consistent with the magnetization curve and theoretical prediction.

### *4.c. Characteristic length*

The characteristic length with respect to the temperature is shown in the following figure

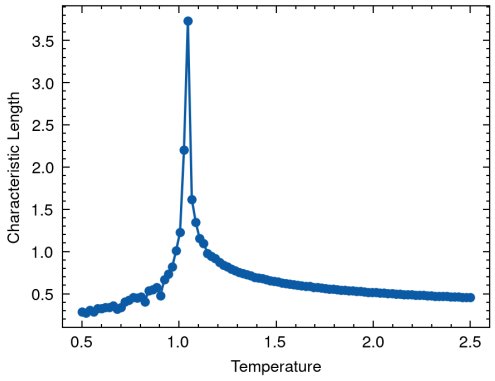



Figure 8: $\xi$ w.r.t $T$

The relatively large fluctuation of the characteristic length at lower temperature indicates that the system is not well equilibrated. However, the rough trend of the characteristic length is still observable. Similar to the specific heat, the characteristic length shows a peak at the critical temperature. We found the peak and identified the corresponding temperature as the critical temperature. The critical temperature is around $T_*{}' = 1.05$, which is the same as the critical temperature identified by the specific heat.

When temperature is very low, the characteristic length tends to increase, which means that the spins are more correlated. When temperature is extrapolated to 0, the characteristic length should be infinite, which means that the spins are fully aligned.

The correlation length at the critical temperature should goes to infinity theoretically. However, in the simulation, the fact is hard to observe. Since we are not able to simulate the system at the accurate critical temperature *(it is even not a rational number!)*, the infinite correlation length is not observed.

### *4.d. Limiting behavior around the critical temperature*

The limiting behaviour of the specific heat and characteristic length around the critical temperature is shown in the following figure

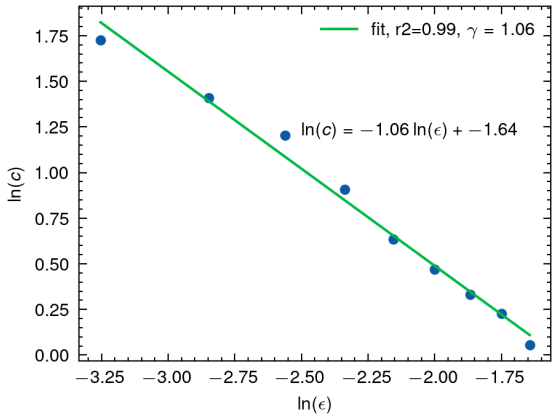



Figure 9: Limiting behaviour of $c$, left side

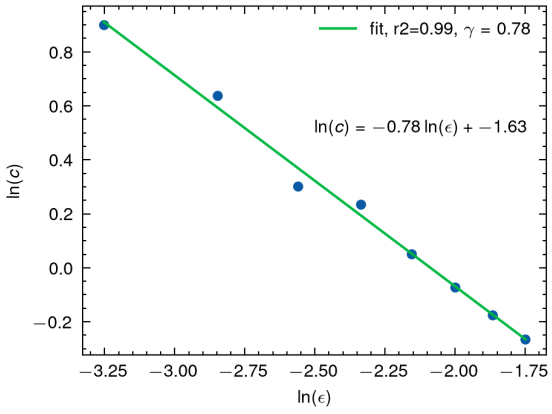



Figure 10: Limiting behaviour of $c$, right side

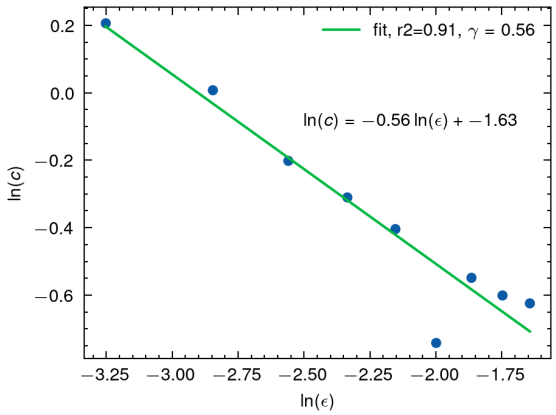



Figure 11: Limiting behaviour of $\xi$, left side

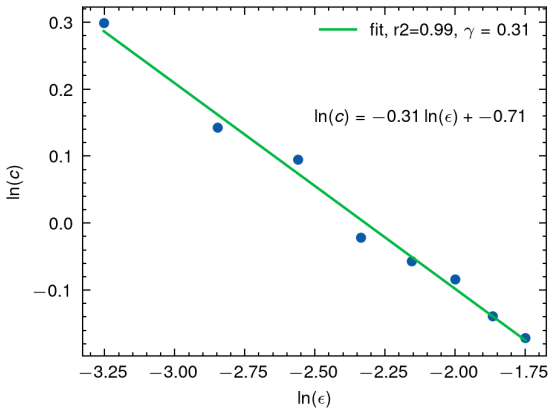



Figure 12: Limiting behaviour of $\xi$, right side

As a conclusion, the scaling exponents $\gamma$ and $\delta$ are listed in the table below

Table 1: Scaling exponents

| | Left side | Right side |
|---|---|---|
| $\gamma$ | 1.06 | 0.78 |
| $\delta$ | 0.56 | 0.31 |

It is necessary to emphasize the fact that the system is not well converged at lower temperature, which means that the scaling exponent may not be accurate.

## 5. Issues and solutions

The most serious issue we encountered is the convergence of the simulation. Since the state space of the system is very large, it is very hard to ensure the system is well equilibrated. A naive solution is to increase the simulation steps. However, this will significantly increase the time cost. The interpreted nature of `Python` makes the situation even worse. Several tricks are used to improve the performance of the simulation and in the following section we will discuss them.

### *5.a. Vectorization and correct usage of* `numpy`

The first thing to note is that `numpy` is a very powerful tool for scientific computation. Although `numpy` is written in `C` which is much faster than `Python`, the performance of `numpy` is still highly dependent on the correct usage. Simply change all the math operations to `numpy` operations will not improve the performance as much as expected. Without vectorization, the performance of `numpy` is sometimes even worse than the naive implementation due to the overhead of `C/Python` interface. But if the vectorization is used correctly, the usage of `numpy` can significantly improve the performance of the simulation.

For example, the naive implementation of the Metropolis algorithm has a time complexity of $O(N^2)$, which is very slow. By using `np.roll` for periodic boundary condition and `np.sum` for summing up the energy, the time complexity is reduced to $O(1)$, which is much faster.

Another improvement brought by the usage of `numpy` is to bypass the global interpreter lock (GIL) and make full use of the CPU resources.

### *5.b. Reduction of unnecessary computation*

The second thing to note is that unnecessary computation should be avoided. For example, in the Metropolis algorithm, the energy difference is calculated by comparing the energy of the new state and the old state. However, this is not necessary to compute the energy of the whole system. Since we only change one spin, we only need to calculate the energy difference of the changed spin. This will reduce the time complexity from $O(N^2)$ to $O(1)$. Even if the energy calculation is vectorized, it is still much slower than the current implementation.

### *5.c. Just-in-time compilation*

The third thing to note is that `numba` is a very powerful tool for accelerating the computation by just-in-time compilation. Although we optimize the code a lot, for Metropolis simulation tasks, the frequency of computing operation itself is still quite low due to ubiquitous control flow, including `if` and `for` statements. Thus, the bottleneck of the simulation migrated from the computation to the control statements.

By using the `@jit` decorator provided `numba`, the whole functions will be compiled just in time. This will significantly improve the performance of the simulation. However, many details should be treated carefully and the performance of `numba` is highly dependent on the correct usage. Many functions are not available under the most efficient `nopython` mode. By mixing the usage of `nopython` and `object` modes, the performance of the simulation can still be largely improved.

### *5.d. Multiprocessing and parallelization*

The fourth thing to note is that `multiprocessing` is a very powerful tool for parallelization. In this task, we are insterested in the physical quantities w.r.t. temperature and outer magnetic field. Additionally, the MC simulation is independent for different temperatures and outer magnetic fields. Thus, we can parallelize the simulation w.r.t. temperature and outer magnetic field. By using the `multiprocessing.Pool` class, we can easily parallelize the simulation to take full advantage of the multi-core CPU resources. When similar strategy is used in much larger cases, it is called ‘high throughput computing’ and is widely used in scientific computation. Even if for each single task, the performance is not improved, sometime even worse, the total performance can be largely improved by parallelization.