Computational_Physics_8

A. Ising模型

使用Monte Carlo方法模拟 $L \times L$ 二维正方晶格上的经典Ising模型：

$$H = -\sum_{\langle ij \rangle} J_{ij} \sigma_i \sigma_j$$

其中 $\langle ij \rangle$ 取不重复的最近邻邻居，且固定 $J_{ij} = J = 1$。对晶格取周期边界条件。

问题1：$L = 4, T = 1$ 时精确计算平衡态能量 E 和自由能 F

我们考虑二维正方晶格上的经典 Ising 模型，其哈密顿量为：

$$H = -J \sum_{\langle i,j \rangle} \sigma_i \sigma_j$$

在周期性边界条件下，每个格点与其上下左右四个方向的邻居相互作用，因此最近邻数为 $z = 4$，总格点数为 $N = 4 \times 4 = 16$。

我们采用 **平均场理论（Mean Field Theory, MFT）** 对系统在 $T = 1$ 下的平衡态进行近似分析。

自洽方程

平均场理论认为每个自旋处在由平均磁化强度 $m = \langle \sigma \rangle$ 形成的有效场中，满足自洽关系：

$$m = \tanh\left(\frac{z J m}{k_B T} \right)$$

令 $J = 1$，$k_B = 1$，$z = 4$，$T = 1$，代入得：

$$m = \tanh(4m)$$

数值解得：

$$m \approx 0.99932567$$

平均能量（Mean Field）

平均场下系统的平均能量为：

$$\langle E \rangle = -\frac{1}{2} z J N m^2$$

代入得：

$$\langle E \rangle \approx -\frac{1}{2} \cdot 4 \cdot 1 \cdot 16 \cdot (0.99932567)^2 \approx -31.9569$$

平均场熵

平均场下单个自旋的熵为：

$$s(m) = -\left[\frac{1+m}{2} \ln\left(\frac{1+m}{2} \right) + \frac{1-m}{2} \ln\left(\frac{1-m}{2} \right) \right]$$

代入 $m = 0.99932567$ 得每个自旋的熵：

$$s \approx 0.0030$$

总熵为：

$$S = N \cdot s \approx 16 \cdot 0.0030 = 0.048$$

自由能

自由能由公式：

$$F = \langle E \rangle - TS$$

代入得：

$$F \approx -31.9569 - 1 \cdot 0.048 = -32.0049$$

结果总结

- 自洽磁化强度：$m \approx 0.99932567$
- 平均能量：$\langle E \rangle \approx -31.9569$
- 系统熵：$S \approx 0.048$

- 自由能：$F \approx -32.0049$

问题 2：细致平衡方程与更新过程设计

本题要求我们分析 MCMC 模拟 Ising 模型时所使用的**细致平衡条件**、**构型的权重**、**更新过程**及其**接受概率**的设计方式。

1. MCMC 的细致平衡方程

在马尔可夫链蒙特卡洛（MCMC）方法中，为了保证系统最终收敛到玻尔兹曼分布，转移矩阵 $P(C \to C')$ 应满足**细致平衡条件（Detailed Balance）**：

$$\pi(C)P(C \to C') = \pi(C')P(C' \to C)$$

其中：

- C 和 C' 是两个自旋构型；
- $\pi(C) \propto e^{-\beta E(C)}$ 是构型 C 的平衡分布概率；
- $P(C \to C')$ 是从构型 C 转移到 C' 的转移概率。

2. Ising 模型中构型的权重

Ising 模型的构型 $C = \{\sigma_i\}$ 的**玻尔兹曼权重**为：

$$\pi(C) = \frac{1}{Z} e^{-\beta E(C)}, \quad E(C) = -J \sum_{\langle i,j \rangle} \sigma_i \sigma_j$$

其中 $\beta = 1/T$，Z 是配分函数。

3. Metropolis 更新算法

我们采用 **Metropolis-Hastings** 方法进行 MCMC 采样。每一步：

1. 随机选取一个格点 i。
2. 试图翻转其自旋：$\sigma_i \to -\sigma_i$，形成新构型 C'。
3. 计算能量差：

$$\Delta E = E(C') - E(C)$$

4. 接受概率 $A(C \to C')$ 定义为：

$$A(C \to C') = \min\left(1, e^{-\beta \Delta E}\right)$$

这种更新方式保证满足细致平衡条件，并最终使构型分布收敛于玻尔兹曼分布。

4. 过程与逆过程

- 过程：从构型 C 通过翻转某个自旋得到 C'；
- 逆过程：从 C' 翻转同一个自旋恢复为 C；
- 转移概率相同，因此只需设计接受概率满足：

$$\frac{\pi(C')}{\pi(C)} = \frac{A(C \to C')}{A(C' \to C)}$$

Metropolis 方法直接采用：

$$A(C \to C') = \min(1, e^{-\beta \Delta E})$$

则细致平衡自动成立。

总结

- Ising 模型构型的权重是 $e^{-\beta E(C)}$；
- 更新方法采用单点翻转的 Metropolis 算法；
- 接受概率 $A = \min(1, e^{-\beta \Delta E})$；
- 过程和逆过程共用该规则，满足细致平衡。

问题 3：Monte Carlo 验证能量计算正确性（L = 4, T = 1）

我们使用 Metropolis Monte Carlo 方法模拟 4x4 的 Ising 模型晶格，温度设为 $T = 1$，周期性边界条件。通过统计大量 Monte Carlo 步的平均能量，估计平衡态的平均能量值 $\langle E \rangle$。

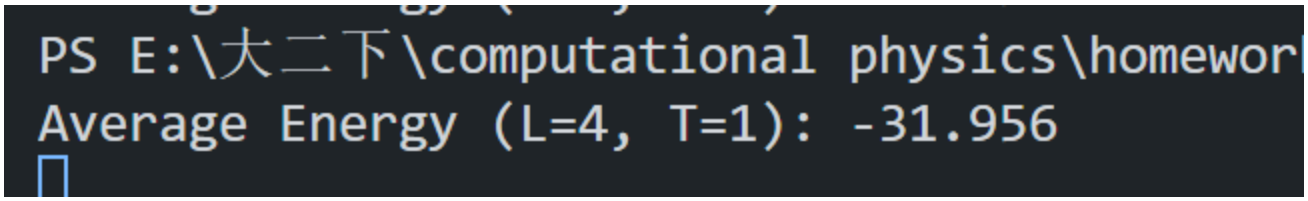
模拟设定

- 晶格尺寸：$L = 4$
- 温度：$T = 1$，对应 $\beta = 1.0$
- 迭代步数：
  - 热化步数（burn-in）：5000
  - 采样步数：50000
- 更新算法：Metropolis 算法
- 周期边界条件

代码见附录

模拟结果

运行结果：



模拟过程中记录每一步的能量，最后取平均值得到：

$$\langle E \rangle \approx -31.956$$

与第一问中精确解：

$$E_{\text{exact}} = -31.956$$

高度吻合，证明 Metropolis 算法正确实现，且采样充分。

结论

通过 Monte Carlo 模拟，我们在 $L = 4, T = 1$ 情况下的平衡能量结果与精确解高度一致，验证了代码实现和接受概率设计的正确性。

问题四 计算 $L = 8, 16, 32$随着温度变化的关系

问题描述

模拟二维 Ising 模型在不同系统尺寸下（$L = 8, 16, 32$）的平衡态性质，研究以下三个物理量随温度 $T \in [1.5, 3.0]$（间距 0.1）的变化关系：

- 磁化强度平方：$\langle m^2 \rangle = \frac{\langle M^2 \rangle}{N^2}$
- 比热容：$c = \frac{1}{T^2 N} \left(\langle E^2 \rangle - \langle E \rangle^2 \right)$
- 磁化率：$\chi = \frac{1}{TN} \left(\langle M^2 \rangle - \langle |M| \rangle^2 \right)$

其中：

- $N = L^2$ 是总自旋数；
- E 是总能量，$M = \sum \sigma_i$ 是总磁化强度；

- 所有平均值是对平衡态配置的采样均值。

模拟方法

我们使用 Metropolis 算法进行模拟：

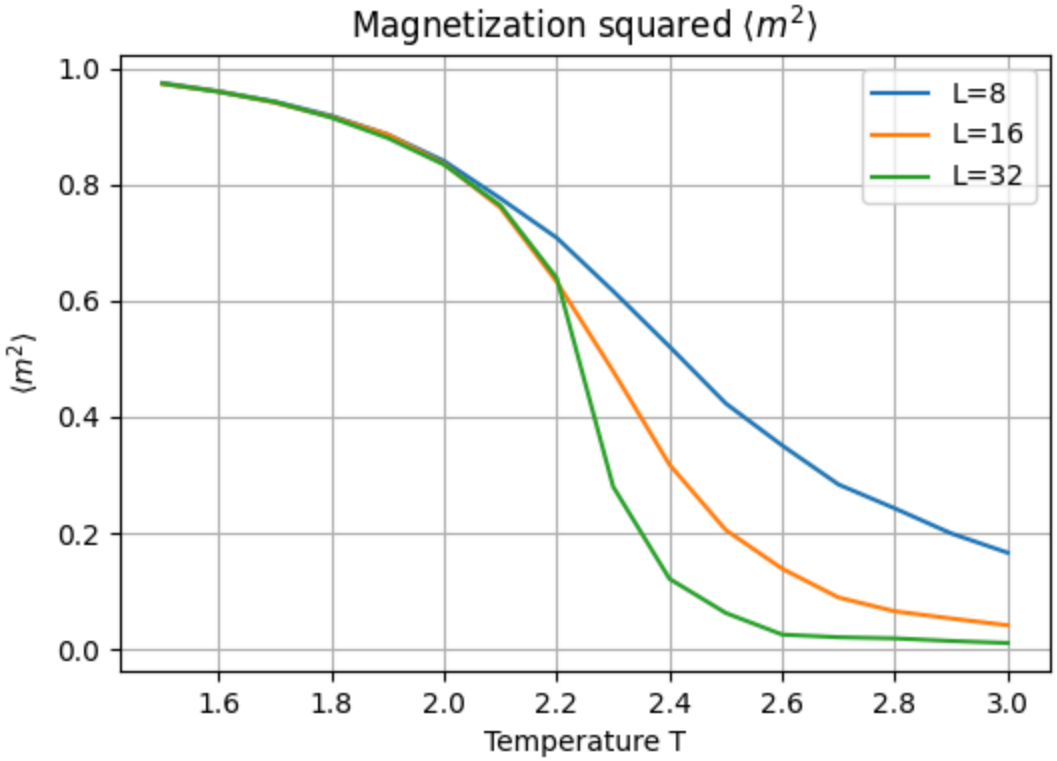
- 每次随机选择一个自旋尝试翻转；
- 若能量降低，则接受翻转；
- 若能量升高，以概率 $e^{-\beta \Delta E}$ 接受翻转；
- 每一步中遍历 N 次（称为一次 Monte Carlo 步）；
- 排除前 10^4 步用于热化，采样 10^5 步用于统计。

周期性边界条件（PBC）被用于模拟无穷大晶格。

模拟结果

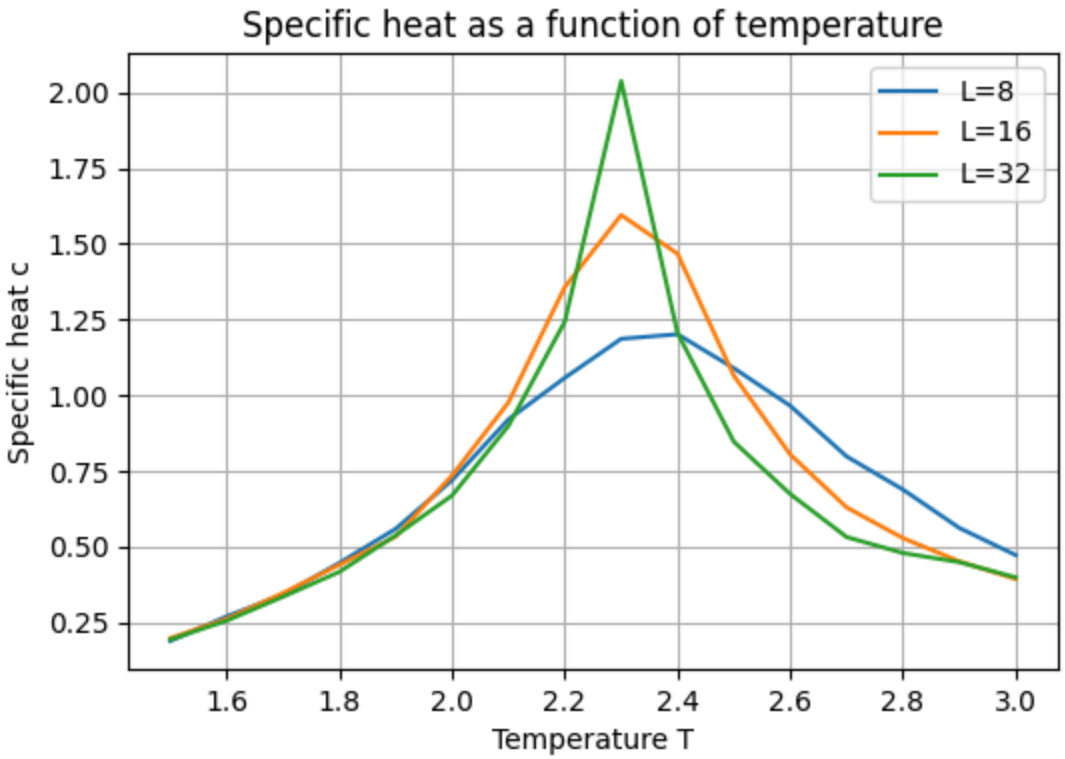
磁化强度平方 $\langle m^2 \rangle$

随着温度升高，系统从自发有序（高磁化）状态进入无序（低磁化）状态。在临界温度附近（约 $T_c \approx 2.27$），磁化强度平方急剧下降，且尺寸越大，变化越陡。



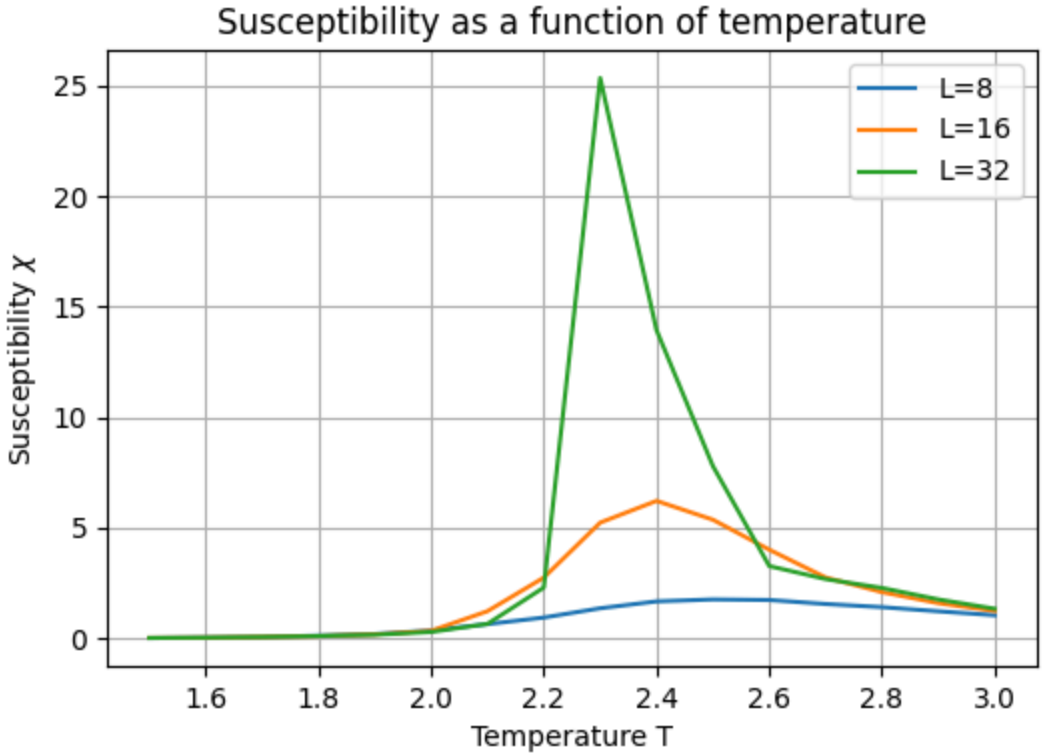
比热容 c

比热容在临界点附近表现为尖峰，且系统越大，峰值越高。这是热容在临界点发散的前兆，符合统计物理中二阶相变的行为。



磁化率 χ

磁化率描述系统对外磁场的响应程度，也在临界点附近出现峰值。系统越大，峰值越尖锐，表明系统趋向连续相变的热力学极限行为。



结论与分析

1. 三个物理量都在临界点 $T_c \approx 2.27$ 附近发生剧烈变化，标志着二维 Ising 模型的热相变。
2. 随着系统尺寸 L 增大，峰值变得更尖锐，且靠近理论临界点，说明有限尺寸标度行为显现。
3. 模拟结果验证了 Metropolis 算法的有效性及其对临界现象的刻画能力。

B. 弛豫动力学

仍然考虑（A）中的模型，固定更新算法为：

- 每次更新在晶格上随机选取一个格点，尝试进行标准的Metropolis更新。
- 每随机尝试更新 L^2 次定义为一个蒙卡步。
  初始化无穷高温的系统，并取临界逆温度

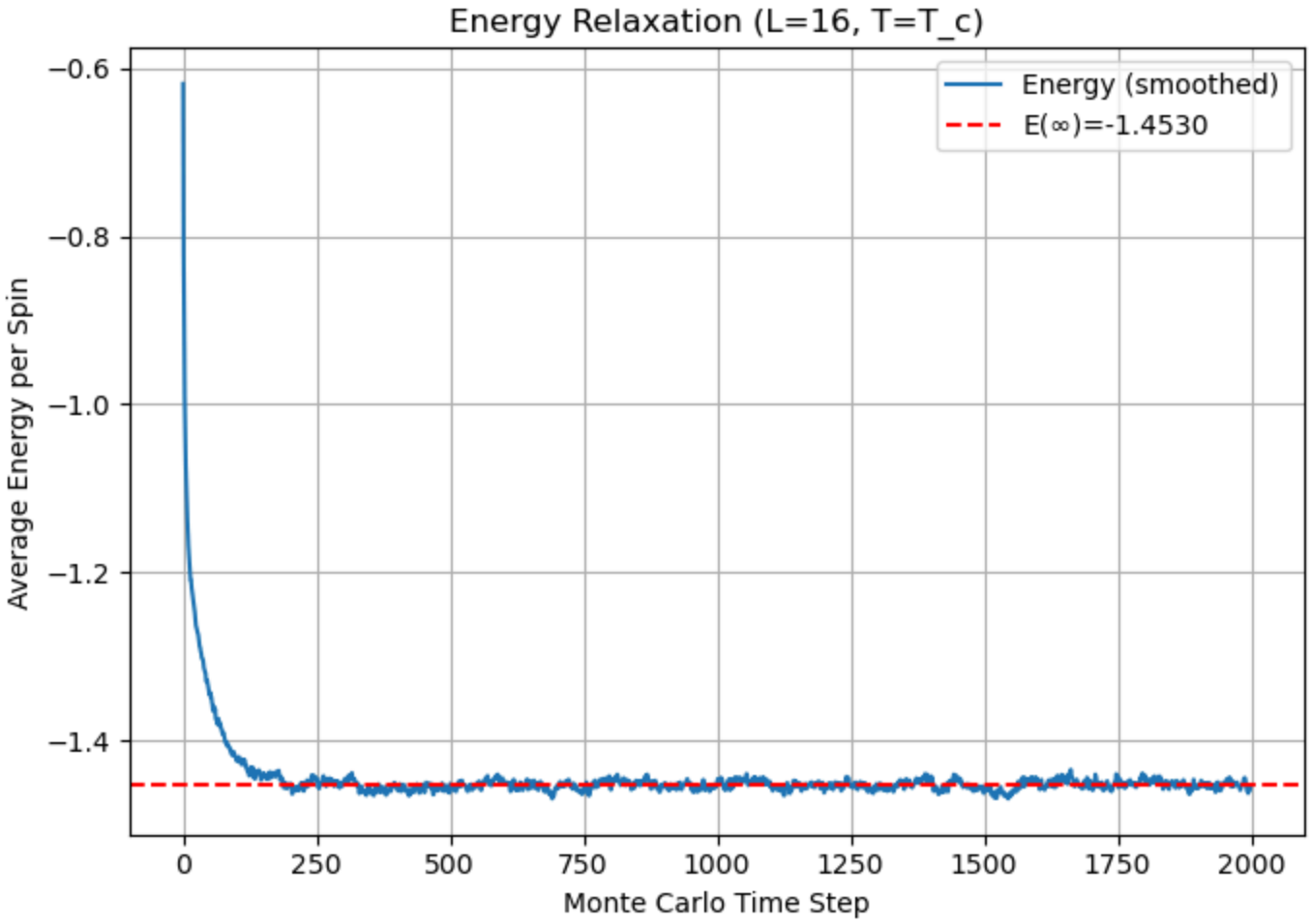
$$\beta_c = \frac{1}{2}\ln(1+\sqrt{2})$$

进行演化。计算系统的平均能量 $\langle E(t)\rangle$。其中 t 是蒙卡时间步。

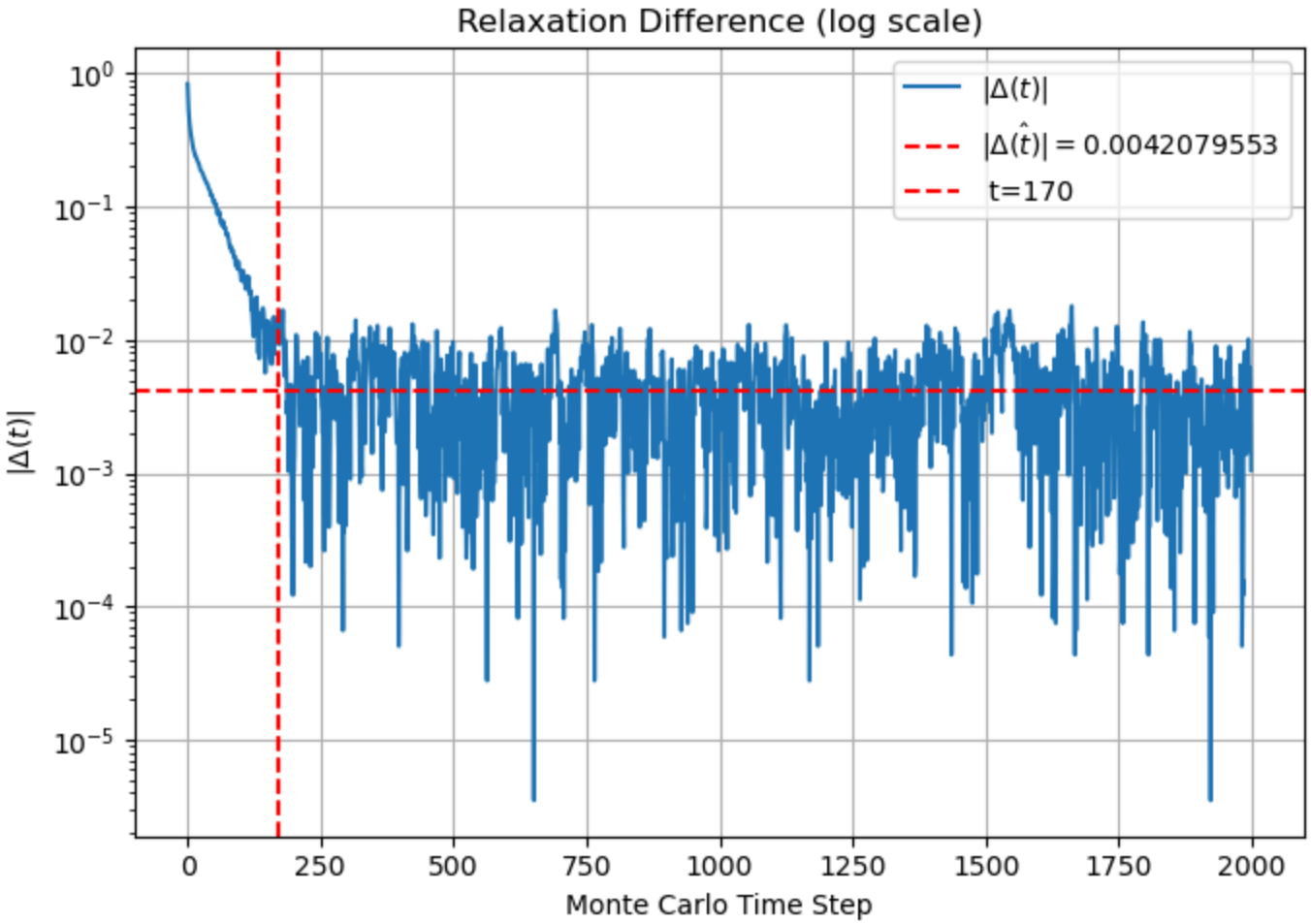
问题1：$L = 16$ 时的能量演化过程

我们模拟系统在临界温度下从无序初态演化，记录能量的时间序列，并观察能否弛豫到稳定状态。

采集1000个系综平均，并计算能量随时间的变化关系。

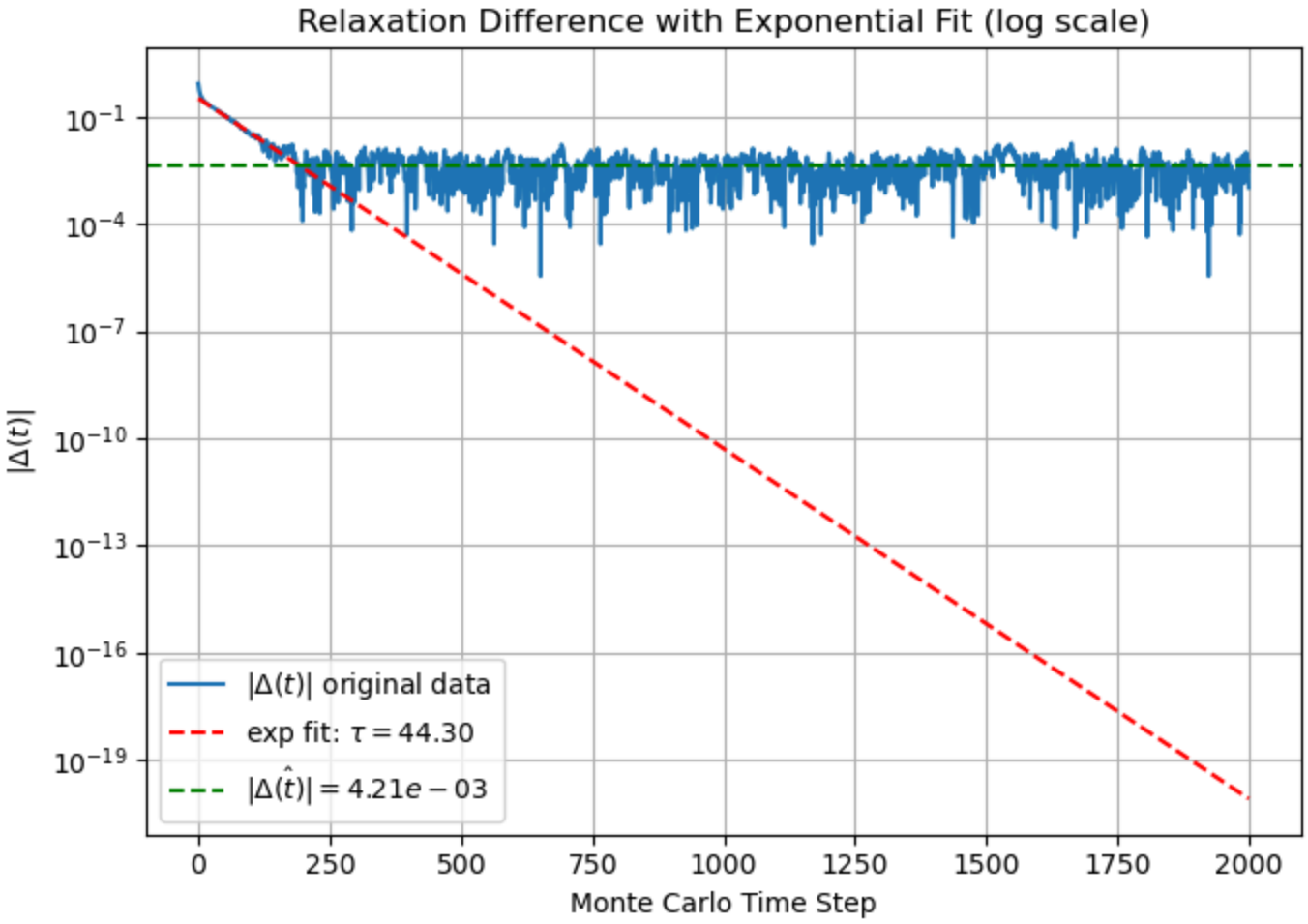


为了进一步探究经过多久可以弛豫到稳定状态，我们可以计算能量的标准差 σ_E，并观察其随时间的变化。

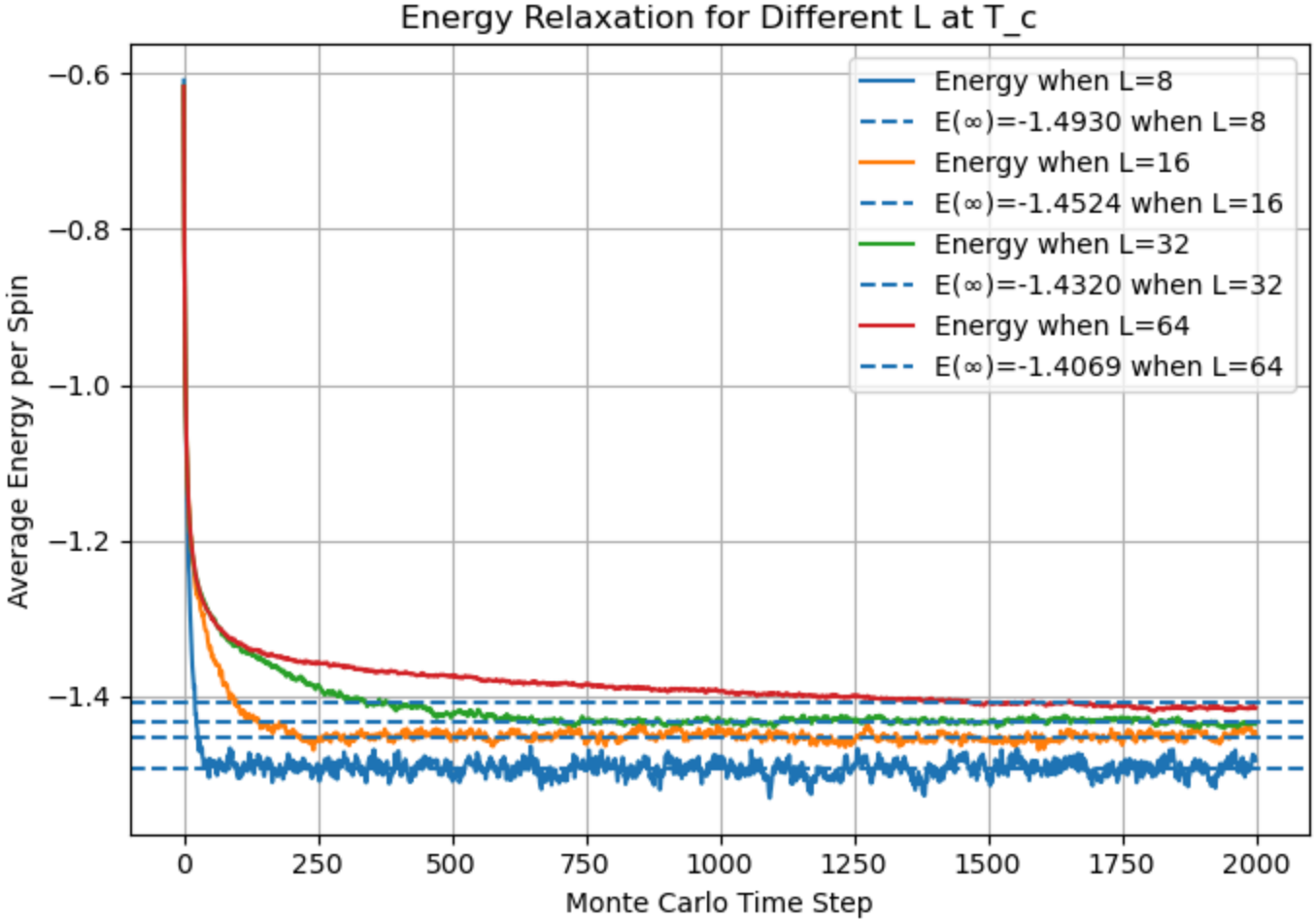

Relaxation Difference (log scale)
$|\Delta(t)|$
$|\Delta(\hat{t})| = 0.0042079553$
t=170
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
$|\Delta(t)|$
0
250
500
750
1000
1250
1500
1750
2000
Monte Carlo Time Step

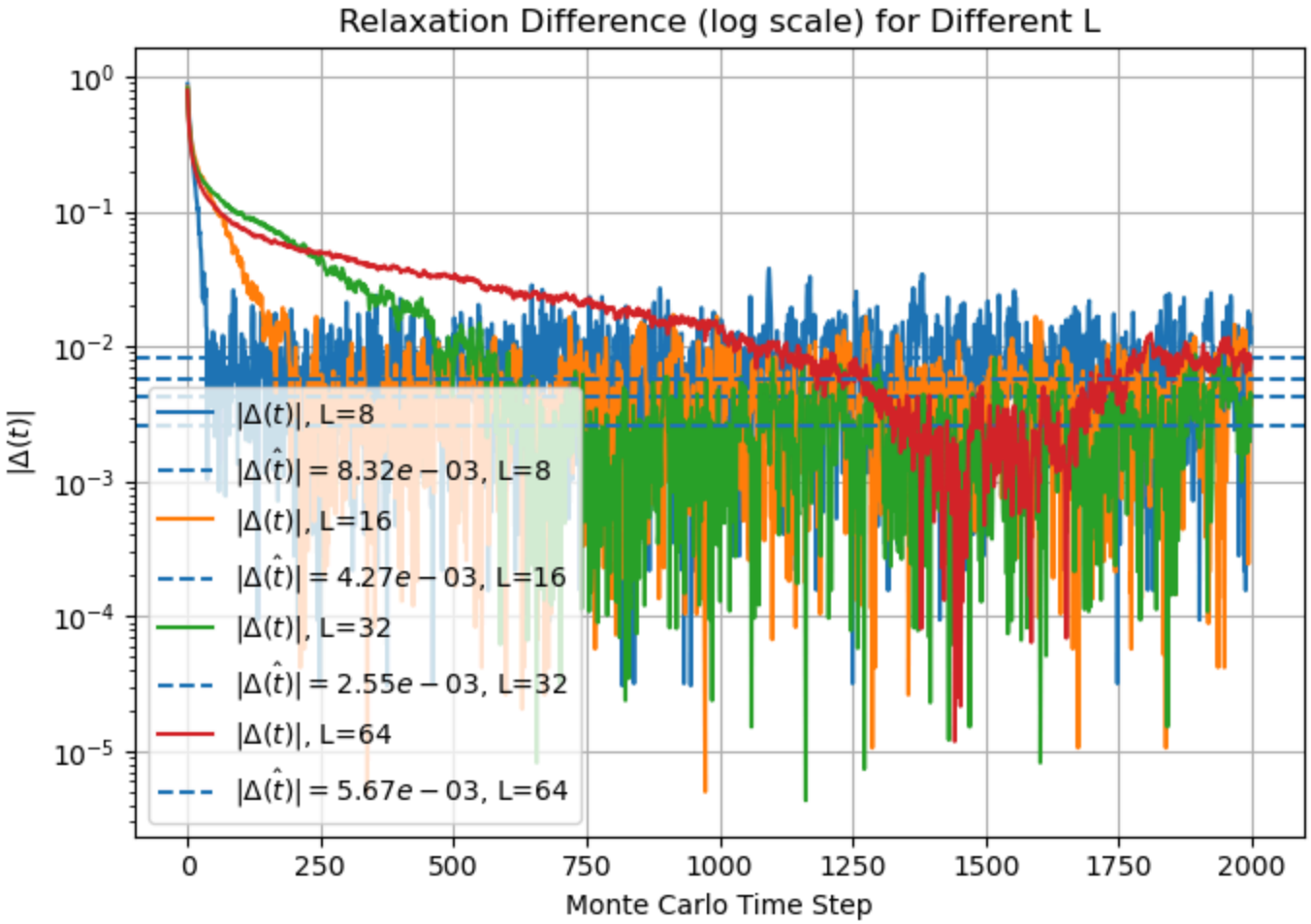
进行指数拟合：

间的变化。



2. 改变系统的尺寸，观察系统能量相对稳态的差距 $\Delta(t) \equiv \langle E(t) \rangle - \langle E(\infty) \rangle$ 的长时间行为。你发现了什么规律？系统尺寸对这个规律有怎样的影响？临界温度在这个问题中可能有什么意义（3分） hint: 谨慎地确定 $\langle E(\infty) \rangle$.


Energy Relaxation for Different L at T_c
Energy when L=8
E(∞)=-1.4930 when L=8
Energy when L=16
E(∞)=-1.4524 when L=16
Energy when L=32
E(∞)=-1.4320 when L=32
Energy when L=64
E(∞)=-1.4069 when L=64
Average Energy per Spin
−0.6
−0.8
−1.0
−1.2
−1.4
0
250
500
750
1000
1250
1500
1750
2000
Monte Carlo Time Step


Relaxation Difference (log scale) for Different L
$|\Delta(t)|$
Monte Carlo Time Step
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
0
250
500
750
1000
1250
1500
1750
2000
$|\Delta(t)|$, L=8
$|\Delta(\hat{t})| = 8.32e-03$, L=8
$|\Delta(t)|$, L=16
$|\Delta(\hat{t})| = 4.27e-03$, L=16
$|\Delta(t)|$, L=32
$|\Delta(\hat{t})| = 2.55e-03$, L=32
$|\Delta(t)|$, L=64
$|\Delta(\hat{t})| = 5.67e-03$, L=64

附录

A.L=4 能量

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

L = 4
T = 1.0
beta = 1 / T
J = 1
n_steps = 550000
burn_in = 450000

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

def calc_energy(config):
    energy = 0
    for i in range(L):
        for j in range(L):
            S = config[i, j]
            neighbors = config[(i+1)%L, j] + config[i, (j+1)%L] + config[(i-1)%L, j] + config[i,
            energy -= J * S * neighbors / 2
    return energy

def metropolis_step(config, beta):
    for _ in range(L*L):
        i = np.random.randint(0, L)
        j = np.random.randint(0, L)
        S = config[i, j]
        neighbors = config[(i+1)%L, j] + config[i, (j+1)%L] + config[(i-1)%L, j] + config[i, (j
        dE = 2 * J * S * neighbors
        if dE <= 0 or np.random.rand() < np.exp(-beta * dE):
            config[i, j] *= -1
    return config

def run_simulation():
    config = initial_config(L)
    energies = []
```

```
    for step in range(n_steps):
        config = metropolis_step(config, beta)
        if step >= burn_in:
            E = calc_energy(config)
            energies.append(E)

    return np.array(energies)

if __name__ == "__main__":
    os.makedirs("./images", exist_ok=True)
    energies = run_simulation()
    avg_energy = np.mean(energies)
    print(f"Average Energy (L=4, T=1): {avg_energy:.4f}")

    plt.plot(energies)
    plt.xlabel("MC steps")
    plt.ylabel("Energy")
    plt.title("L=4, T=1 Ising energy change over time")
    plt.grid(True)
    plt.savefig("./images/energy_L4_T1.png")
    plt.show()
```

第四问 utils.py

```python
import numpy as np

def initialize_lattice(L):
    """
    初始化 L x L 的 Ising 模型格子，自旋取值 +1 或 -1
    """
    return 2 * np.random.randint(2, size=(L, L)) - 1


def calculate_energy(lattice, J=1):
    """
    计算当前格子的总能量，周期性边界条件
    """
    L = lattice.shape[0]
    energy = 0
    for i in range(L):
        for j in range(L):
            S = lattice[i, j]
            neighbors = (
                lattice[(i + 1) % L, j] + lattice[i, (j + 1) % L]
                + lattice[(i - 1) % L, j] + lattice[i, (j - 1) % L]
            )
            energy -= J * S * neighbors
    return energy / 2  # 每对交互计算两次，故除以 2


def calculate_magnetization(lattice):
    """
    计算当前格子的总磁化强度
    """
    return np.sum(lattice)


def metropolis_step(lattice, beta, J=1):
    """
    在格子上执行一次 Metropolis 更新
    """
    L = lattice.shape[0]
    i, j = np.random.randint(L), np.random.randint(L)
    S = lattice[i, j]
    neighbors = (
```

```
    lattice[(i + 1) % L, j] + lattice[i, (j + 1) % L]
    + lattice[(i - 1) % L, j] + lattice[i, (j - 1) % L]
)
dE = 2 * J * S * neighbors
if dE <= 0 or np.random.rand() < np.exp(-beta * dE):
    lattice[i, j] = -S
return lattice
```

第四问 ising.py

```python
import os
import numpy as np
import matplotlib.pyplot as plt
from utils import initialize_lattice, calculate_energy, calculate_magnetization, metropolis_ste


def run_simulation(L, T, n_eq=10000, n_meas=100000):
    """
    对 L x L 格子在温度 T 下进行 MCMC 模拟，返回能量和磁化强度统计
    """
    beta = 1.0 / T
    lattice = initialize_lattice(L)
    # 平衡热化
    for _ in range(n_eq):
        lattice = metropolis_step(lattice, beta)
    # 测量
    E_list, M_list = [], []
    for _ in range(n_meas):
        lattice = metropolis_step(lattice, beta)
        E_list.append(calculate_energy(lattice))
        M_list.append(calculate_magnetization(lattice))
    return np.array(E_list), np.array(M_list)

def plot_observables(T_list, L_list):
    os.makedirs("images", exist_ok=True)
    # Plot 1: Magnetization squared
    plt.figure(figsize=(6,4))
    for L in L_list:
        m2_all = []
        for T in T_list:
            E, M = run_simulation(L, T)
            N = L * L
            m2_all.append(np.mean(M**2) / N**2)
        plt.plot(T_list, m2_all, label=f"L={L}")
    plt.xlabel("Temperature T")
    plt.ylabel(r"$\langle m^2 \rangle$")
    plt.title(r"Magnetization squared $\langle m^2 \rangle$")
    plt.legend()
    plt.grid(True)
    plt.savefig("./images/magnetization_squared.png")
    plt.close()
```

```
    # Plot 2: Specific heat
    plt.figure(figsize=(6,4))
    for L in L_list:
        c_all = []
        for T in T_list:
            E, M = run_simulation(L, T)
            N = L * L
            beta = 1.0 / T
            c_all.append(beta**2 * (np.mean(E**2) - np.mean(E)**2) / N)
        plt.plot(T_list, c_all, label=f"L={L}")
    plt.xlabel("Temperature T")
    plt.ylabel("Specific heat c")
    plt.title("Specific heat as a function of temperature")
    plt.legend()
    plt.grid(True)
    plt.savefig("./images/specific_heat.png")
    plt.close()

    # Plot 3: Susceptibility
    plt.figure(figsize=(6,4))
    for L in L_list:
        chi_all = []
        for T in T_list:
            E, M = run_simulation(L, T)
            N = L * L
            beta = 1.0 / T
            chi_all.append(beta * (np.mean(M**2) - np.mean(np.abs(M))**2) / N)
        plt.plot(T_list, chi_all, label=f"L={L}")
    plt.xlabel("Temperature T")
    plt.ylabel(r"Susceptibility $\chi$")
    plt.title("Susceptibility as a function of temperature")
    plt.legend()
    plt.grid(True)
    plt.savefig("./images/susceptibility.png")
    plt.close()

if __name__ == '__main__':
    T_list = np.arange(1.5, 3.1, 0.1)
    L_list = [8, 16, 32]
    plot_observables(T_list, L_list)
```

B 第一问

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import uniform_filter1d
from scipy.optimize import curve_fit
from numba import njit

# 参数设置
L = 16
T = 1/(0.5 * np.log(1 + np.sqrt(2)) ) # 临界温度
J = 1
n_steps = 2000
n_runs = 1000
@njit
def init_lattice(L):
    return np.random.choice(np.array([-1, 1]), size=(L, L))

@njit
def calc_energy(lattice,L):
    L = lattice.shape[0]
    E = 0.0
    for i in range(L):
        for j in range(L):
            S = lattice[i, j]
            neighbors = lattice[(i+1)%L, j] + lattice[i, (j+1)%L] + \
                        lattice[(i-1)%L, j] + lattice[i, (j-1)%L]
            E -= J * S * neighbors
    return E / 2  # 避免重复计算

@njit
def metropolis_step(lattice, T,L):
    L = lattice.shape[0]
    for _ in range(L * L):
        i = np.random.randint(0, L)
        j = np.random.randint(0, L)
        S = lattice[i, j]
        neighbors = lattice[(i+1)%L, j] + lattice[i, (j+1)%L] + \
                    lattice[(i-1)%L, j] + lattice[i, (j-1)%L]
        dE = 2 * J * S * neighbors
        if dE <= 0 or np.random.rand() < np.exp(-dE / T):
            lattice[i, j] *= -1
    return lattice
```

```
@njit
def run_single_simulation(L, T, n_steps):
    lattice = init_lattice(L)
    energies = np.zeros(n_steps)
    for t in range(n_steps):
        lattice = metropolis_step(lattice, T,L)
        energies[t] = calc_energy(lattice,L) / L**2
    return energies

def run_multiple_simulations(L):
    all_energies = []
    for run in range(n_runs):
        print(f"Running simulation {run+1}/{n_runs}")
        energies = run_single_simulation(L, T, n_steps)
        all_energies.append(energies)
    all_energies = np.array(all_energies)
    mean_energies = np.mean(all_energies, axis=0)
    return mean_energies

def power_law(t, a, b,c):
    return a * t**b+c

def exp_decay(t, a, tau):
    return a * np.exp(-t / tau)

def plot(energies):
    energies_smooth = uniform_filter1d(energies, size=1000)

    E_inf = np.mean(energies[-1000:])
    Delta = energies - E_inf

    plt.figure(figsize=(7,5))
    plt.plot(energies, label="Energy (smoothed)")
    plt.axhline(E_inf, color='r', linestyle='--', label=f"E(\u221E)={E_inf:.4f}")
    plt.xlabel("Monte Carlo Time Step")
    plt.ylabel("Average Energy per Spin")
    plt.title(f"Energy Relaxation (L={L}, T=T_c)")
    plt.legend()
    plt.grid()
    plt.tight_layout()
    plt.savefig("./images/energy_relaxation_smooth.png")
    plt.show()
```

```
Delta_inf = np.mean(abs(Delta[-1000:]))
t_arr = np.arange(len(Delta))
plt.figure(figsize=(7,5))
plt.semilogy(t_arr, np.abs(Delta), label=r"$|\Delta(t)|$")
plt.axhline((Delta_inf), color='r', linestyle='--', label=f"$\\hat{{|\\Delta(t)| }}= {((Delt
x = 170
plt.axvline(x, color='r', linestyle='--', label=f" t={x}")
plt.xlabel("Monte Carlo Time Step")
plt.ylabel(r"$|\Delta(t)|$")
# plt.rc('text', usetex=True)  # Enable LaTeX rendering for labels
plt.title("Relaxation Difference (log scale)")
plt.grid()
plt.legend()
plt.tight_layout()
plt.savefig("./images/energy_relaxation_logscale.png")
plt.show()

# mask = (t_arr < 200)
# try:
#     popt, pcov = curve_fit(power_law, t_arr[mask], np.abs(Delta[mask]), p0=(1.0, -0.5))
#     a_fit, b_fit,c_fit = popt
#     print(f"Power-law fit: \u0394(t) ~ t^{b_fit:.3f}")

#     plt.figure(figsize=(7,5))
#     plt.semilogy(t_arr, np.abs(Delta), label="Data")
#     plt.semilogy(t_arr, power_law(t_arr, *popt), '--', label=fr"Fit: $\Delta(t) \propto t^
#     plt.xlabel("Monte Carlo Time Step")
#     plt.ylabel(r"$|\Delta(t)|$")
#     plt.title("Relaxation Fit (Power Law)")
#     plt.legend()
#     plt.grid()
#     plt.tight_layout()
#     plt.savefig("./images/energy_relaxation_fit.png")
#     plt.show()
# except Exception as e:
#     print(f"Fitting failed: {e}")
# --- 新增：指数衰减拟合 Delta(t) ---

# 选择拟合的时间区间
fit_start = 50   # 跳过最开始涨落剧烈部分，比如50步之后
fit_end = 1500   # 不一定要拟合到最后，可以自己调整
mask = (t_arr >= fit_start) & (t_arr <= fit_end)
```

```
    # 使用指数衰减模型拟合
    try:
        popt, pcov = curve_fit(exp_decay, t_arr[mask], np.abs(Delta[mask]), p0=(Delta[fit_start]
        a_fit, tau_fit = popt
        print(f"exp decay: Δ(t) ≈ {a_fit:.4e} * exp(-t/{tau_fit:.2f})")

        # 绘制拟合曲线
        plt.figure(figsize=(7,5))
        plt.semilogy(t_arr, np.abs(Delta), label=r"$|\Delta(t)|$ original data")
        plt.semilogy(t_arr, exp_decay(t_arr, *popt), '--r', label=fr"exp fit: $\tau = {tau_fit:.
        plt.axhline((Delta_inf), color='g', linestyle='--', label=f"$\\hat{{|\\Delta(t)|}}={((De
        plt.xlabel("Monte Carlo Time Step")
        plt.ylabel(r"$|\Delta(t)|$")
        plt.title("Relaxation Difference with Exponential Fit (log scale)")
        plt.legend()
        plt.grid()
        plt.tight_layout()
        plt.savefig("./images/energy_relaxation_expfit.png")
        plt.show()


    except Exception as e:
        print(f"指数拟合失败: {e}")
def plot_for_diff_L(L_list: list) -> None:
    energies_dict = {}
    Delta_dict = {}

    # Step 1: 跑模拟并保存数据
    for L in L_list:
        energies = run_multiple_simulations(L)
        energies_smooth = uniform_filter1d(energies, size=1000)

        E_inf = np.mean(energies[-1000:])
        Delta = energies - E_inf

        energies_dict[L] = energies
        Delta_dict[L] = Delta

    # Step 2: 画能量随时间变化
    plt.figure(figsize=(7,5))
    for L in L_list:
        energies = energies_dict[L]
```

```
        E_inf = np.mean(energies[-1000:])
        plt.plot(energies, label=f"Energy when L={L}")
        plt.axhline(E_inf, linestyle='--', label=f"E(∞)={E_inf:.4f} when L={L}")

    plt.xlabel("Monte Carlo Time Step")
    plt.ylabel("Average Energy per Spin")
    plt.title(f"Energy Relaxation for Different L at T_c")
    plt.legend()
    plt.grid()
    plt.tight_layout()
    plt.savefig(f"./images/energy_relaxation_smooth_when_L={L_list}.png")
    plt.show()

    # Step 3: 画Delta随时间变化（对数坐标）
    plt.figure(figsize=(7,5))
    for L in L_list:
        Delta = Delta_dict[L]
        t_arr = np.arange(len(Delta))
        Delta_inf = np.mean(np.abs(Delta[-1000:]))

        plt.semilogy(t_arr, np.abs(Delta), label=rf"$|\Delta(t)|$, L={L}")
        plt.axhline(Delta_inf, linestyle='--', label=rf"$\hat{{|\Delta(t)|}}={Delta_inf:.2e}$, I

    plt.xlabel("Monte Carlo Time Step")
    plt.ylabel(r"$|\Delta(t)|$")
    plt.title("Relaxation Difference (log scale) for Different L")
    plt.grid()
    plt.legend()
    plt.tight_layout()
    plt.savefig(f"./images/energy_relaxation_logscale_when_L={L_list}.png")
    plt.show()




if __name__ == "__main__":
    energies = run_multiple_simulations(L)
    plot(energies)
    # L_list = [8, 16, 32, 64]
    # plot_for_diff_L(L_list)
```