目录

1 问题背景与处理思路

体系的相变是凝聚态物理的一个重要研究方向，伊辛模型是其中一个经典且较为成熟的模型，本项目采用Metropolis算法和Wolff算法研究无外场下二维Ising模型在临界温度发生的由铁磁向顺磁的相变过程.

1.1 二维Ising模型

二维Ising模型描述的是一系列固定排列在二维晶格中的自旋，每个自旋仅可取$S_i = \pm 1$两种状态. 外加磁场B下，体系的哈密顿量分为两部分，一部分是晶格中所有相邻自旋之间的交换相互作用能之和，另一部分是各个自旋本身在磁场中的势能：

$$H(\{S_i\}) = -J\sum_{\langle i,j\rangle} S_iS_j + B\sum_{i=1}^{N} S_i \tag{1}$$

其中J是交换相互作用参数，在铁磁性材料中，$J > 0$，这意味着相邻自旋同向排列相较于反向排列能量更低，自旋更倾向于同向排列，$\sum_{\langle i,j\rangle}$表示对相邻的自旋对求和，$i$和$j$的顺序不重要，即$\langle i,j\rangle$项和$\langle j,i\rangle$项算作同一项，在求和中仅累加一次，不做重复计算，$N$为体系中总自旋数.

二维Ising模型的配分函数可表为

$$Z = \sum_{S_1=\pm 1}\sum_{S_2=\pm 1}\cdots\sum_{S_N=\pm 1} \exp[\beta(J\sum_{\langle i,j\rangle} S_iS_j + B\sum_{i=1}^{N} S_i)]. \tag{2}$$

理论上，从这一配分函数出发可计算出体系的各物理量，例如，可先用

$$\langle M\rangle = \langle\sum_i S_i\rangle = \frac{1}{Z}\sum_{\{S_i\}}\sum_{i=1}^{N} S_i\exp[-\beta H(\{S_i\})] = \frac{1}{\beta}\frac{\partial \ln Z}{\partial B} \tag{3}$$

计算体系的总磁矩，再用

$$\chi = \frac{\partial\langle M\rangle}{\partial B} = \frac{\beta}{N}(\langle M^2\rangle - \langle M\rangle^2) \tag{4}$$

求解体系的磁化率，类似地，也可用

$$C_v = \frac{k_B\beta^2}{N}(\langle E^2\rangle - \langle E\rangle^2) \tag{5}$$

求得体系的热容.

然而在实际计算中，由于配分函数式中共有2^N个求和项，其计算复杂度随体系中原子数的增加而指数增长，因此直接用配分函数精确求解较大体系的物理量是不现实的. 这时，利用单自旋翻转的Metropolis算法来随机模拟体系的演化过程，从而得到各物理量，成为一个相对可行的选择.

1.2 Metropolis算法与其面临的问题——临界变慢

Metropolis算法的大致思路是，从一个初始状态开始，每次随机取晶格中的一个自旋，按照

$$A_{mn} = \begin{cases} \exp[-\beta(E_n - E_m)], & \text{if } E_n > E_m, \\ 1, & \text{otherwise,} \end{cases} \tag{6}$$

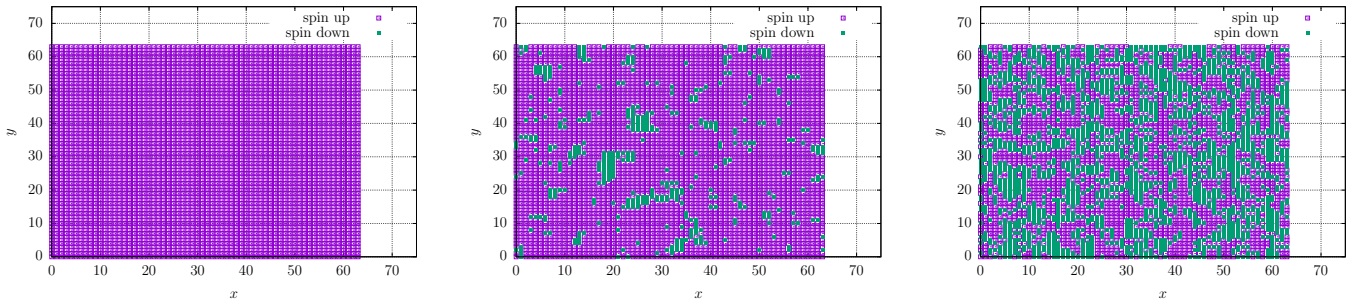
的概率翻转这一自旋，其中E_m和E_n分别为翻转前状态m和翻转后状态n的能量. 重复执行此算法，就可以模拟体系在平衡状态下演化（或者向平衡状态演化）的过程，这是因为：首先，每个自旋在体系的每步演化中都有$\frac{1}{N}$的概率（即体系从状态m演化至n的提出概率g_{mn}）被取到，从而满足了各态遍历性，其次，式(6)规定的所取自旋的翻转概率A_{mn}（即体系从状态m演化至n的接受概率）则使得

$$\frac{g_{mn}A_{mn}}{g_{nm}A_{nm}} = \left\{\begin{array}{ll} \frac{\frac{1}{4}\times\exp[-\beta(E_n-E_m)]}{\frac{1}{4}\times 1}, & \text{if } E_n > E_m \\ \frac{\frac{1}{4}\times 1}{\frac{1}{4}\times\exp[-\beta(E_m-E_n)]}, & \text{otherwise} \end{array}\right\} = \exp[-\beta(E_n - E_m)] \tag{7}$$

满足了细致平衡条件.

单自旋翻转的Metropolis算法在远离临界点的范围内表现出较高的效率和精度（见计算结果部分），但其一个严重的问题在于临界变慢，所谓临界变慢，指的是自关联时间（即一个状态演化为与自己完全无关的状态所需要的步数，更严谨地，即自相关函数$\chi(t) = \int dt'[m(t') - \langle m\rangle][m(t'+t) - \langle m\rangle]$关于$t$指数衰减的特征时间）在体系温度接$T$近于临界温度$T_c$时显著变长. 这导致在临界区域附近需要演化更多步数才能得到与非临界区域相同的计算精度，其在计算结果上表现为体系的各物理量在临界区域附近出现较为明显的涨落（见计算结果部分）.

临界变慢的根源并不完全是算法的问题，它的出现是Ising的物理规律导致的必然现象：在温度T远高于临界温度T_c的区域，自旋呈现杂乱的排布，各个相邻的自旋多反向，如图1(c)所示，因此翻转自旋有较高的概率造成相邻自旋同向，导致能量降低，因此成功翻转自旋的概率较高，此时体系可以较为高效地演化；在体系温度T远低于临界温度T_c的区域，虽然有大量自旋同向，如图1(a)所示，导致翻转自旋困难，但是低温下，体系本身就应该呈现出铁磁性（大量自旋同向），铁磁性的可能状态总数并不多，因此计算结果也并不会有太大误差；而在临界区域，体系呈现出很多同向自旋团簇状聚集的现象，如图1(b)所示，只有当随机选取的自旋在这些团簇的边界上时，自旋翻转的成功率较高，此外绝大多数处于团簇中的自旋都很难翻转，这导致体系的演化缓慢，无法遍历当前温度对应的各个可能状态，从而引起较大误差，特别是当模拟的晶格较小，团簇的尺寸可以与整个晶格相当或者覆盖整个晶格时，这一问题尤为显著. 虽然临界变慢无法避免，但是我们可以改进算法，降低临界区域附近的自关联时间.



(a) $T = 1.00$ K (b) $T = 2.30$ K (c) $T = 4.00$ K

图 1: 晶格中自旋随着温度升高的变化情况. 晶格尺寸：64×64. 算法：Metropolis.

1.3 临界变慢现象的对策——Wolff算法

克服临界变慢问题的一个方法是采用Wolff算法. 与单自旋翻转算法不同，Wolff算法属于团簇算法(cluster algorithm)，通过每次翻转一个团簇来加快体系的演化速度. Wolff算法的大致思路是，对于每一步演化，随机选择晶格中的一个自旋首先加入团簇，并将其作为种子遍历与其相邻的所有自旋，将这些自旋中与种子处于同一自旋态且尚未被考虑的以$P_{\text{add}} = 1-\exp(-2\beta J)$的概率也加入到团簇中，并且以这些加入团簇的自旋作为新的种子继续重复上述操作，直至无种子待考虑，最后同时翻转团簇中的所有原子.

Wolff算法同样满足各态遍历性和细致平衡条件：首先，随机取种子，并随机将种子周围的自旋纳入团簇，显然可以遍历系统的所有可能状态；细致平衡条件的证明可能稍显复杂：假设翻转的团簇由f个自旋构成，其周围有h个与之方向相同的自旋（即翻转需要破坏h个键），则提出概率

$$g_{mn} = (1 - P_{\text{add}})^h P_{\text{add}}^f, \tag{8}$$

类似地，将同样这一团簇的自旋翻转回来的提出概率

$$g_{nm} = (1 - P_{\text{add}})^k P_{\text{add}}^f, \tag{9}$$

其中k为反向翻转时需要破坏的键数. 选定（提出）团簇后必须翻转，因此接受几率$A_{mn} = A_{nm} = 1$，故

$$\frac{g_{mn}A_{mn}}{g_{nm}A_{nm}} = (1 - P_{\text{add}})^{h-k} = \exp[-2\beta J(h-k)]. \tag{10}$$

易见$2J(h-k)$恰好是团簇翻转后和团簇翻转前体系的能量之差，细致平衡条件就此满足.

1.4 临界指数

除了直接用肉眼识别体系各物理量随着温度的变化曲线的趋势来判断体系的相变，临界指数是一种更好的表征体系相变的指标. 在$T \to T_c$的极限下，我们有

$$M(T) \sim (T_c - T)^{\beta}, \qquad \text{对于} T < T_c, \tag{11}$$

$$\chi \sim |T_c - T|^{-\gamma}, \tag{12}$$

$$C_v \sim |T_c - T|^{-\alpha}. \tag{13}$$

此处的β,γ和α即为所谓的临界指数. 对于无限大二维Ising模型，$\beta = 1/8$，$\gamma = 7/4$，$\alpha = 0$.

2 Metropolis算法和Wolff算法比较

2.1 算法详述

2.1.1 Metropolis算法

Metropolis算法步骤主要分为如下6步（代码见附录）：

1. **初始化**：设晶格尺寸$L_x \times L_y$，初始温度$T = 0$，所有自旋均向上，$S_i = +1 \, \forall i$；

2. **Warming up**：随机选取晶格中的一个自旋i，计算所取自旋在翻转前的能量：

$$H_i = -JS_i \times \sum_{j \in \{\text{neighbors of } i\}} S_j, \tag{14}$$

（注意此处求和是针对与所取自旋i相邻的自旋）
和系统在翻转前后的总能量差（即所取自旋在翻转前后的能量差，而该自旋在翻转前后的能量符号相反，$H_i' = -H_i$）：

$$\Delta E = H_i' - H_i = -2H_i. \tag{15}$$

生成一在$[0, 1)$范围内均匀分布的随机数r，若$r < e^{-\beta \Delta E}$，则翻转所取自旋，重复$n_{\mathbf{warmup}}$次；

3. **演化和测量**：用与上一步相同的方法，尝试翻转自旋，然后计算体系总磁矩大小：

$$M = \left| \sum_i S_i \right|, \tag{16}$$

体系总能量：

$$E = \sum_{\langle i,j \rangle} S_i S_j \tag{17}$$

以及这两个物理量的平方：M^2和E^2；

4. 重复上一步$n_{\mathbf{evol}}$次，然后计算体系总磁矩大小的平均值$\langle M \rangle$和体系总磁矩平方的平均值$\langle M^2 \rangle$，进而计算体系的磁化率：

$$\chi = \frac{\beta}{N}(\langle M^2 \rangle - \langle M \rangle^2). \tag{18}$$

计算体系总能量的平均值$\langle E \rangle$和体系总能量平方的平均值$\langle E^2 \rangle$，进而计算体系的热容：

$$C_v = \frac{k_B \beta^2}{N}(\langle E^2 \rangle - \langle E \rangle^2); \tag{19}$$

5. 以dT为步长，逐渐增加T，重复第2,3,4步，直至T达到T_{final}；

6. 绘制单自旋平均磁矩大小$|m| = \frac{|M|}{N} = \frac{|M|}{L_x \times L_y}$，磁化率$\chi$，热容$C_v$这三个物理量随温度$T$变化的曲线.

2.1.2 Wolff算法

相比Metropolis算法，Wolff算法仅在翻转自旋的操作上有所差异，其他步骤基本类似（代码见附录）：

1. **初始化**：设晶格尺寸$L_x \times L_y$，初始温度$T = 0$，所有自旋均向上，$S_i = +1 \, \forall i$；

2. **Warming up**：重复如下步骤n_{warmup}次：

(a) 随机选取晶格中的一个自旋i，加入团簇，并翻转该自旋，团簇大小设为$N_{\text{cluster}}=1$，种子序号设为$n_{\text{cluster}}=1$（团簇中第n_{seed}个到第n_{cluster}自旋为待检查的格点），团簇自旋方向cluster_spin设为与所取自旋方向相同；

(b) 选出团簇中第n_{seed}个自旋作为种子，并重新赋值$n_{\text{seed}}=n_{\text{seed}}+1$；

(c) 检查与种子相邻的四个自旋，对于满足以下条件的自旋：

- 方向与团簇自旋方向cluster_spin相同；
- 尚未被纳入团簇中，

以$P_{\text{add}}=1-\exp[-2\beta J]$的概率将它加入团簇并翻转它，每将一个自旋加入团簇就重新赋值团簇大小$n_{\text{cluster}}=n_{\text{cluster}}$一次；

(d) 重复(b)(c)两步，直至$n_{\text{seed}}>n_{\text{cluster}}$；

3. **演化和测量**：用与上一步相同的方法，翻转团簇，然后计算体系总磁化强度大小：

$$M=\left|\sum_i S_i\right|, \tag{20}$$

体系总能量：

$$E=\sum_{\langle i,j\rangle} S_iS_j \tag{21}$$

以及这两个物理量的平方：M^2和E^2；

4. 重复上一步$n_{\mathbf{evol}}$次，然后计算体系总磁矩大小的平均值$\langle M\rangle$和体系总磁矩平方的平均值$\langle M^2\rangle$，进而计算体系的磁化率：

$$\chi=\frac{\beta}{N}(\langle M^2\rangle-\langle M\rangle^2). \tag{22}$$

计算体系总能量的平均值$\langle E\rangle$和体系总能量平方的平均值$\langle E^2\rangle$，进而计算体系的热容：

$$C_v=\frac{k_B\beta^2}{N}(\langle E^2\rangle-\langle E\rangle^2); \tag{23}$$

5. 以dT为步长，逐渐增加T，重复第2,3,4步，直至T达到T_{final}；

6. 绘制单自旋平均磁矩大小$|m|=\frac{|M|}{N}=\frac{|M|}{L_x\times L_y}$，磁化率$\chi$，热容$C_v$这三个物理量随温度$T$变化的曲线.

2.2 计算结果与讨论

简单起见，在本项目的计算中，统一采用正方形晶格，取$k_B=1$，$J=1$. 为了提高效率，本项目的代码统一适配MPI加速，程序使用16个核并行，最终的计算结果是各个核计算结果的平均.

2.2.1 Metropolis算法计算结果

取初始温度$T=0.01$ K，温度步长$dT=0.01$ K，最终温度$T_{\text{final}}=5.00$ K，每个温度下warming up的步数$n_{\text{warmup}}=10000$，warming up后正式模拟演化并计算相关物理量的步数$n_{\text{evol}}=100000$，分别计算晶格尺寸为32×32,64×64,128×128的情况，计算结果分别如图2,3,4所示.

以单自旋平均磁矩大小最接近0.5对应的温度为临界温度，三种尺寸的晶格的临界温度如表1所示. 之所以选用单自旋平均磁矩大小最接近0.5对应的温度作为，而不选取磁极化率或热容的最大值对应的温度作为临界温度，是因为前者的计算结果涨落最不明显，且在后面计算临界指数时我们也发现选用单自旋平均磁矩作为临界温度的判据所得的结果是最好的. 随着晶格尺寸的增大，Metropolis算法计算得到的临界温度升高.

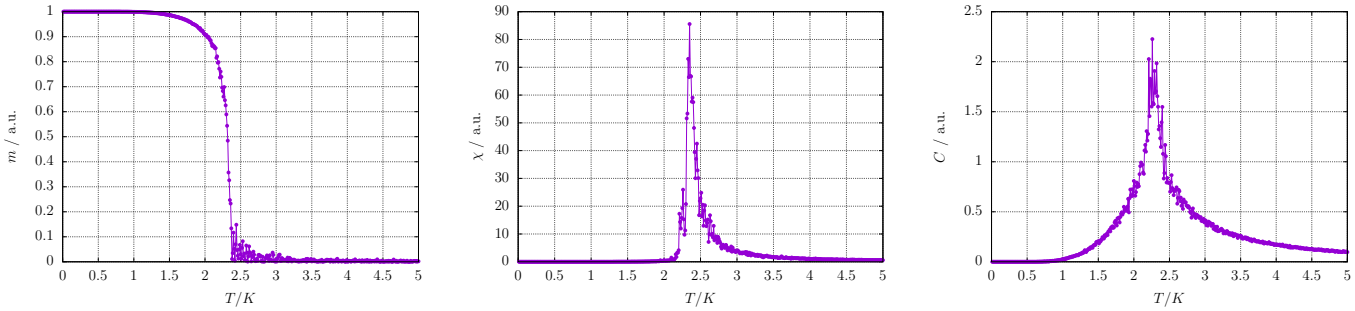
我们注意到，Metropolis算法计算速度较快，但在临界点附近各物理量都出现了较为明显的涨落，曲线走势不平滑，也就是前一节中所介绍的临界变慢现象. 对于平均磁极化强度大小，晶格尺寸越小，这一问题越明显；对于磁极化率和热容，晶格尺寸越大，这一问题越明显.

表 1: Metropolis算法计算得各尺寸晶格临界温度

晶格尺寸	32×32	64×64	128×128
临界温度 T_c / K	2.32	2.36	2.43

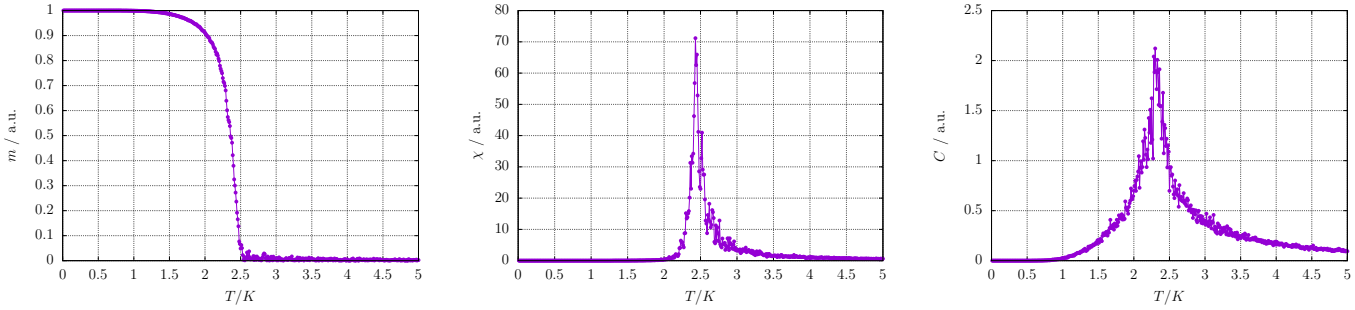
图1-3：Metropolis算法计算结果

(a)：单自旋平均磁矩大小随温度的变化曲线 (b)：磁化率随温度的变化曲线 (c)：热容随温度的变化曲线



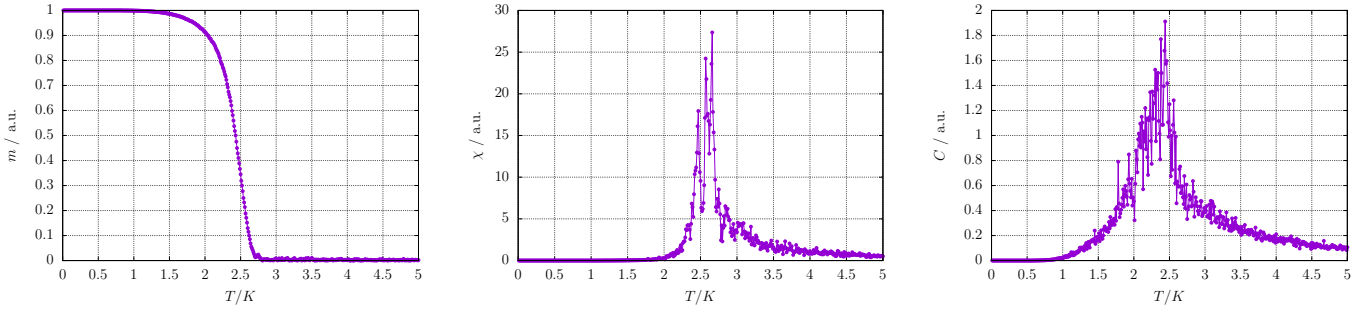
(a) (b) (c)

图 2: 晶格尺寸：32×32.



(a) (b) (c)

图 3: 晶格尺寸：64×64.



(a) (b) (c)

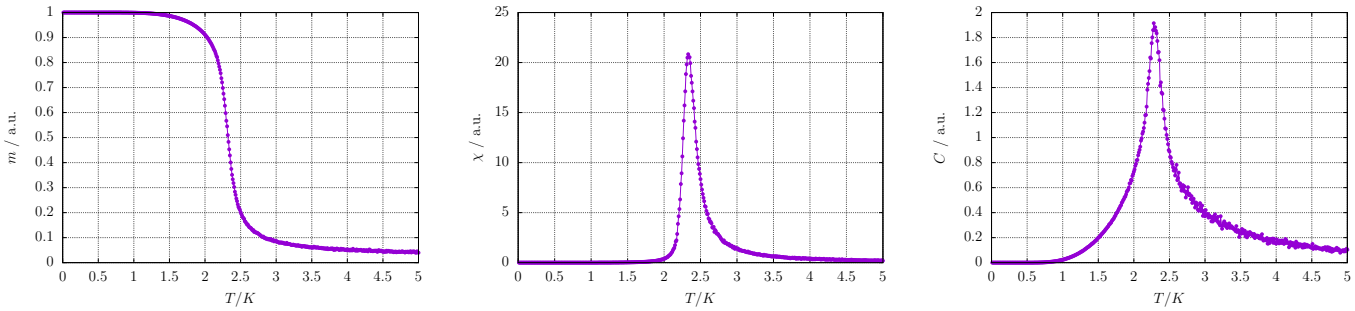
图 4: 晶格尺寸：128×128.

2.2.2 Wolff算法计算结果

取初始温度$T = 0.01$ K，温度步长$dT = 0.01$ K，最终温度$T_{\text{final}} = 5.00$ K，每个温度下warming up的步数$n_{\text{warmup}} = 200$，warming up后正式模拟演化并计算相关物理量的步数$n_{\text{evol}} = 2000$，分别计算晶格尺寸为32×32,64×64,128×128的情况，计算结果分别如图5,6,7所示.

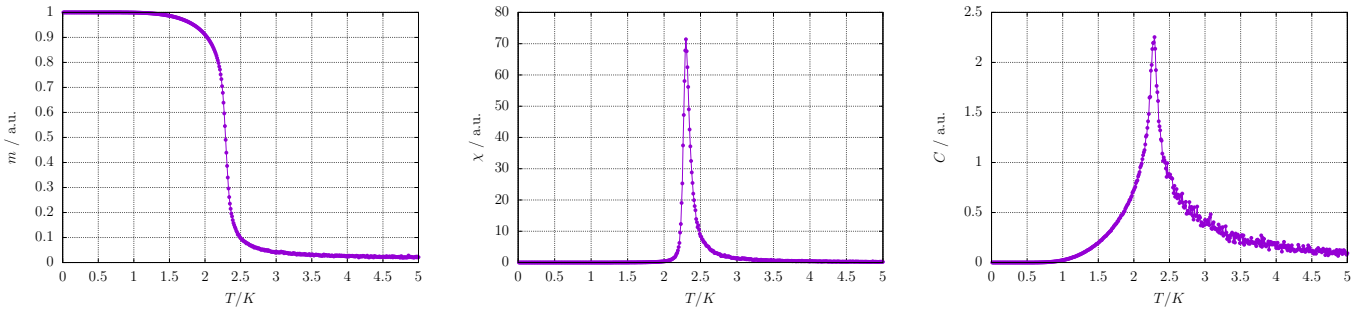
图4-6：Metropolis算法计算结果

(a)：单自旋平均磁矩大小随温度的变化曲线 (b)：磁化率随温度的变化曲线 (c)：热容随温度的变化曲线



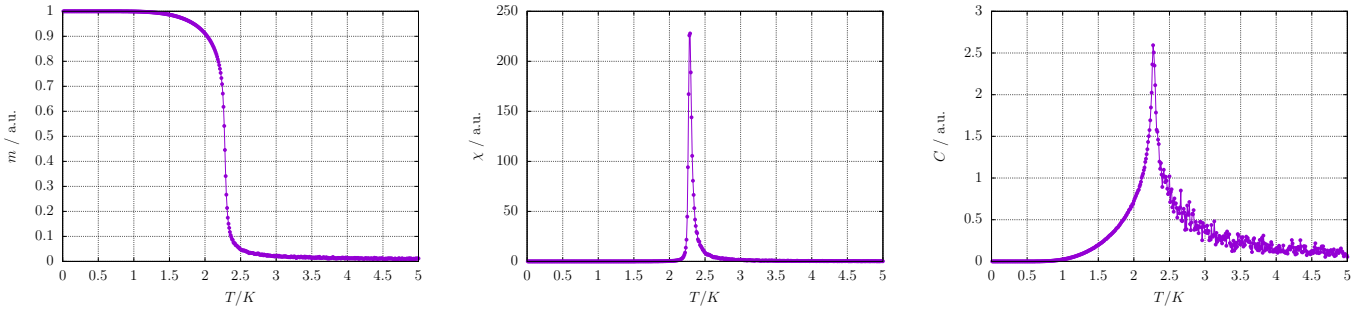
(a) (b) (c)

图 5: 晶格尺寸：32×32.



(a) (b) (c)

图 6: 晶格尺寸：64×64.



(a) (b) (c)

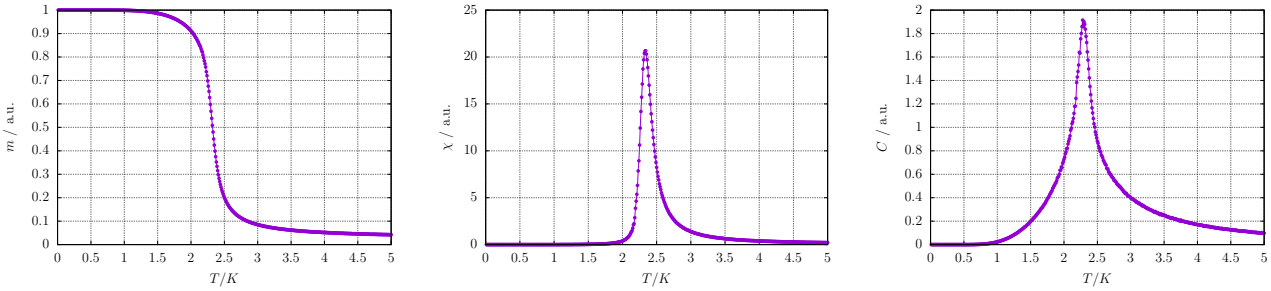
图 7: 晶格尺寸：128×128.

由于Wolff算法计算效率较低，特别是对于晶格尺寸较大的情况，计算速度尤为低下，对于128×128的晶格，计算耗时将近3天，且考虑到不同于Metropolis算法每次演化只能翻转一个自旋，Wolff算法每次演化一般可以翻转多个自旋，故我们设定的warming up和正式演化和测量的次数都较Metropolis算法的少. 我们注意到，Wolff算法的耗时大部分集中在计算低于临界温度范围，而对于高于临界温度的范围，可以较快地得到计算结果，这应该是因为，随着温度的上升，将自旋加入团簇的概率降低，所以计算时所需遍历的自旋较少，团簇的尺寸较小，因而耗时更少；而在计算精度上，Wolff算法得到的平均磁极化强度大小和磁化率随温度的变化曲线都较为平滑，而热容随温度变化的曲线虽然在低于临界温度范围较为平滑，但在高于临界温度范围出现少量的涨落，也就是说在高于临界温度的范围内，应当需要更大的warming up步数n_{warmup}和正式演化和测量的步数n_{evol}.

因此我们在原有计算结果的基础上进行改进：我们现已知临界温度大约为2.30K，对于温度T低于2.30 K的范围内我们采用与先前相同的参数，即warming up步数$n_{\text{warmup}} = 200$，正式演化和测量的步数$n_{\text{evol}} = 2000$；对于温度高于2.30 K的范围内我们设定warming up步数$n_{\text{warmup}} = 10000$，正式演化和测量的步数$n_{\text{evol}} = 100000$，分别重新计算晶格尺寸为$32 \times 32$,$64 \times 64$,$128 \times 128$的情况，计算结果分别如图5,6,7所示.

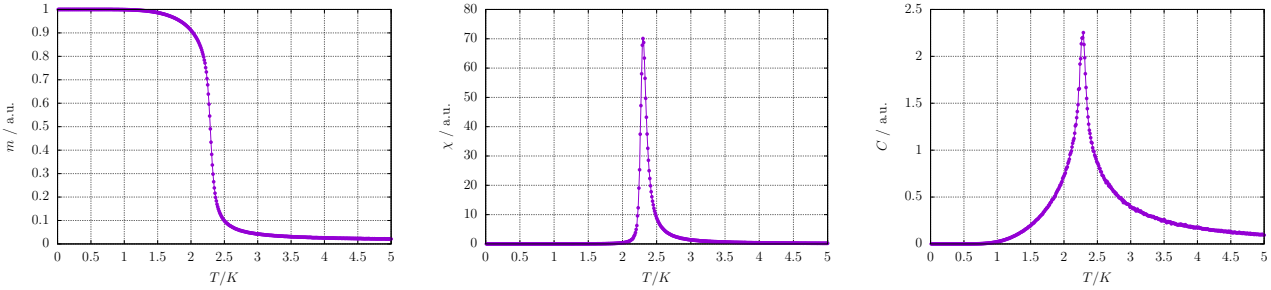
图4-6：Metropolis算法（改进后）计算结果

(a)：平均磁极化强度大小随温度的变化曲线 (b)：磁化率随温度的变化曲线 (c)：热容随温度的变化曲线



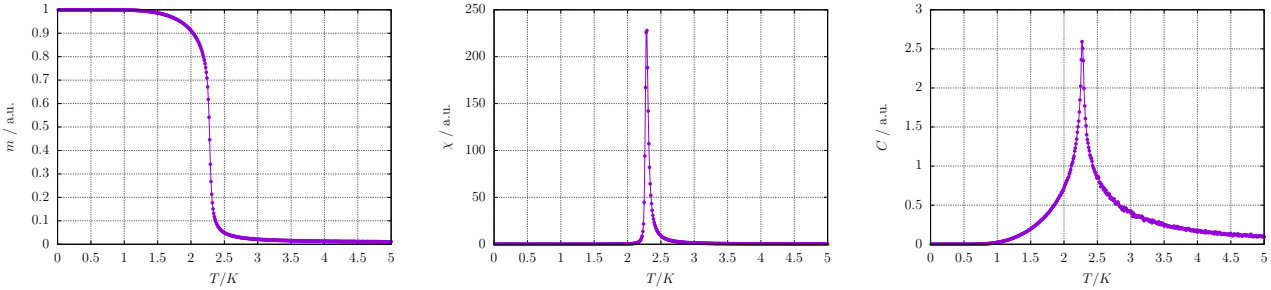
(a) (b) (c)

图 8: 晶格尺寸：32×32.



(a) (b) (c)

图 9: 晶格尺寸：64×64.



(a) (b) (c)

图 10: 晶格尺寸：128×128.

改进后的计算结果相对于改进前有明显改善，曲线均变得非常平滑，几乎完全看不出涨落.

以单自旋平均磁矩大小最接近0.5对应的温度为临界温度，三种尺寸的晶格的临界温度如表2所示. 随着晶格尺寸的增大，Wolff算法计算得到的临界温度下降.

表 2: Wolff算法（改进后）计算得各尺寸晶格临界温度

晶格尺寸	32×32	64×64	128×128
临界温度 T_c / K	2.32	2.29	2.27

2.2.3 两种算法比较

由上述的计算结果，Metropolis算法速度更快，但是存在较为严重的临界变慢现象，计算结果不精确；而Wolff算法可以较好地克服临界变慢问题，在临界点附近得到更为精确的结果，但是计算效率较低，特别是针对晶格尺寸较大的问题，在低于临界温度和临界温度附近的范围内，其计算速度严重落后于Metropolis算法.

根据这两种算法的特点，如果要对大晶格进行高精度的计算，一个较为理想的方法是：先用Metropolis算法以较大的温度步长dT“预计算”，估计出临界区域的范围；然后对于远低于临界温度的范围，采用Metropolis算法计算，对于临界温度附近及高于临界温度的范围，采用Wolff算法进行计算.

对于计算得到的临界温度随晶格尺寸变化的情况，两种算法表现出相反的趋势. 根据文献[1]，二维伊辛模型（无限大晶格）的临界温度的理论值满足

$$\sinh^2\left(\frac{2J}{k_B T_c}\right) = 1 \tag{24}$$

$$\Longrightarrow T_c = \frac{2J}{k_B \ln(1+\sqrt{2})} \approx 2.2692 \text{ K}. \tag{25}$$

因此Wolff算法计算得到的临界温度更加准确.

[1] Onsager, Lars. "Crystal statistics. I. A two-dimensional model with an order-disorder transition." *Physical Review* 65.3-4 (1944): 117.

3 临界指数计算

本节中我们用两种方法——线性拟合法和有限尺度标度分析法分别计算体系的临界指数.

3.1 线性拟合法计算临界指数

式(11)和(12)可化为

$$\ln m = \beta \ln(T_c - T) + C_1, \qquad 对于 T < T_c, \tag{26}$$

$$\ln \chi = -\gamma \ln(T_c - T) + C_2, \qquad 对于 T < T_c, \tag{27}$$

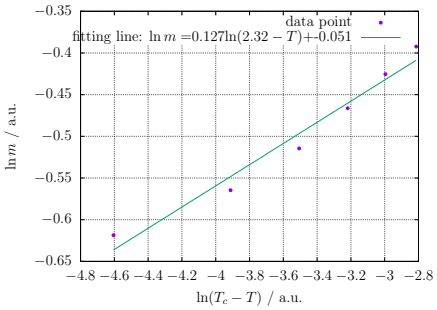
$$\ln C_v = -\alpha \ln(T_c - T) + C_3, \qquad 对于 T < T_c. \tag{28}$$

故用直线在略小于临界温度的范围内拟合$\ln m$与$\ln(T_c - T)$的关系,$\ln \chi$与$\ln(T_c - T)$的关系和$\ln C_v$与$\ln(T_c - T)$的关系，所得斜率即分别为β,$-\gamma$和$-\alpha$.

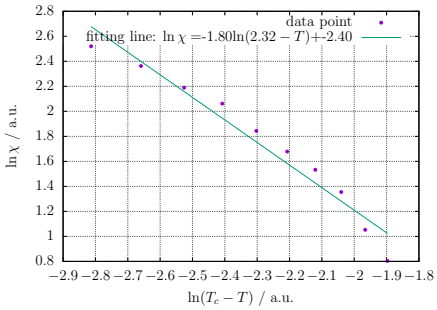
用上一节Wolff算法计算得到的数据，拟合得到32×32,64×64,128×128这三个尺寸的晶格对应的临界指数，拟合图线如图11,12,13所示，拟合所得临界指数如表3所示，β和γ与其理论值均符合得较好，而对于小晶格α的计算值与理论值符合得较好，对于大晶格，由于临界点附近热容变化极为剧烈，而我们所取的温度步长不够小，故误差较大.

图4-6：线性拟合图线

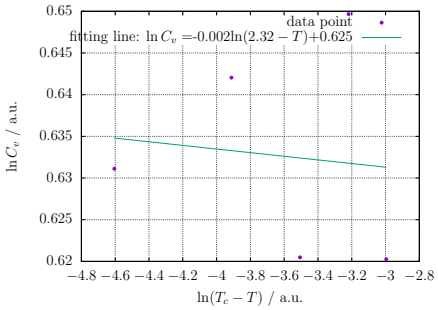
(a)：$\ln m - \ln(T_c - T)$ (b)：$\ln \chi - \ln(T_c - T)$ (c)：$\ln C_v - \ln(T_c - T)$



(a)

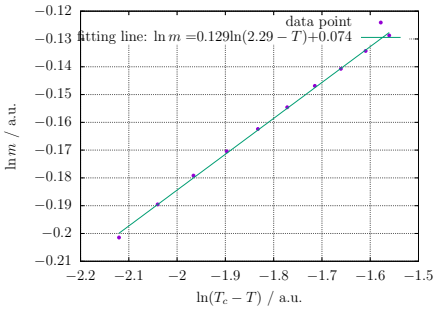


(b)

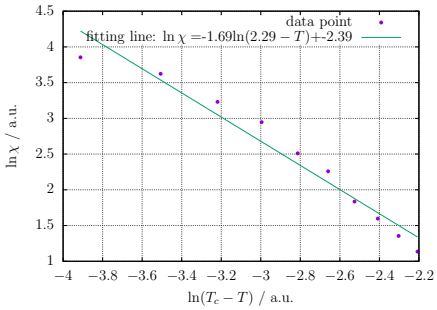


(c)

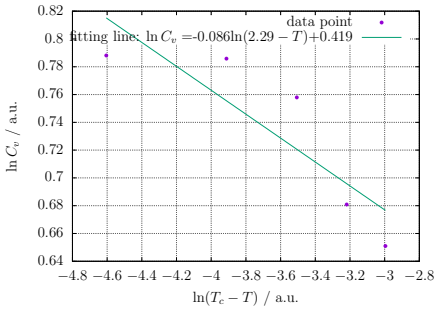
图 11: 晶格尺寸：32×32.



(a)

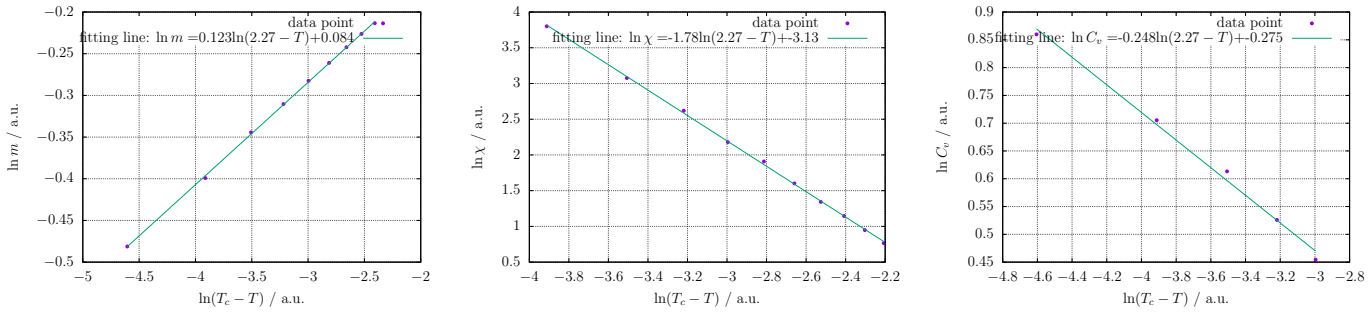


(b)



(c)

图 12: 晶格尺寸：64×64.



(a) (b) (c)

图 13: 晶格尺寸：128×128.

表 3: 临界指数计算结果

晶格尺寸	32×32	64×64	128×128
β	0.127	0.129	0.123
β的相对误差	1.6%	3.2%	−1.6%
γ	1.80	1.69	1.78
γ的相对误差	2.9%	−3.4%	1.7%
α	0.002	0.086	0.248

3.2 有限尺度标度分析计算临界指数

除了从各物理量与温度之间的关系中得到临界指数外，我们还可以用有限尺度分析的方法得到他们. 在无限大的系统中，相关长度在临界点附近发散：

$$\xi \sim |T - T_c|^{-\nu}, \qquad (\nu \geq 1). \tag{29}$$

而在尺度有限的系统中，当关联长度大于系统（晶格）的尺寸时，整个系统就可视为长程有序，系统达到临界点. 因此，对于尺度有限的系统，其临界温度$T_{\max}$与无限大系统的临界温度T_c存在如下关系：

$$|T_{\max} - T_c|^{-\nu} \sim L, \tag{30}$$

或

$$-\ln L = \nu \ln |T_{\max} - T_c| + C \tag{31}$$

从而

$$|M(T)| \sim (T_c - T_{\max})^{\beta} \sim L^{-\beta/\nu}, \tag{32}$$

$$\chi(T) \sim |T_c - T_{\max}|^{-\gamma} \sim L^{\gamma/\nu}, \tag{33}$$

$$\tag{34}$$

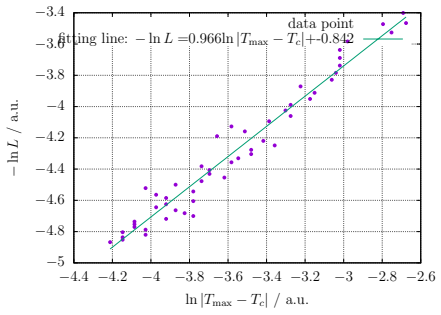
或

$$\ln |M| = -\frac{\beta}{\nu} \ln |L| + C_4, \tag{35}$$

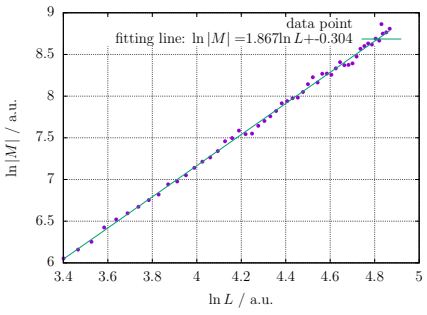
$$\ln \chi = \frac{\gamma}{\nu} \ln L + C_5. \tag{36}$$

$$\tag{37}$$

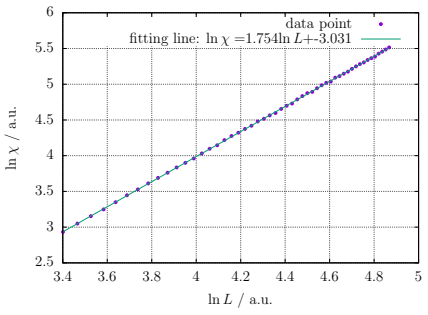
我们以$dL = 2$为步长，将晶格边长从30遍历至130，对每种尺寸的晶格分别用Wolff算法在$[2.26, 2.34]$ K范围内计算，设定温度步长$dT = 0.001$ K，对每个温度设定warming up步数$n_{\text{warmup}} = 200$，正式演化和测量的步数$n_{\text{evol}} = 2000$，最终分别得出各个晶格尺寸对应的临界温度、体系磁极化率极大值和体系磁极化率取极大值时的总磁矩大小（代码见附录）. 根据文献[2]，我们已知临界温度的理论值为$T_c = \frac{2}{\ln(1+\sqrt{2})} = 2.692$ K，如图14(a)所示，做$\ln 1/L - \ln(T_{\max} - T_c)$的线性拟合，得到$\nu = 0.966$. 如图14(b)所示，做$\ln |M| - \ln L$的线性拟合，得到斜率$\frac{\beta}{\nu} = 1.867$，故$\beta = 1.80$，似乎与参考值有较大偏差. 如图14(c)所示，做$\ln \chi - \ln L$的线性拟合，得到斜率$\frac{\gamma}{\nu} = 1.754$，故$\gamma = 1.69$，与参考值符合得较好.



(a) $\ln 1/L - \ln(T_{\max} - T_c)$的线性拟合



(b) $\ln |M| - \ln L$的线性拟合



(c) $\ln \chi - \ln L$的线性拟合

[2]Onsager, Lars. "Crystal statistics. I. A two-dimensional model with an order-disorder transition." *Physical Review* 65.3-4 (1944): 117.

4 附录

4.1 Metropolis算法代码

```
program main
    use mpi
    implicit none
    real(8), parameter :: pi = acos(-1.d0), kB = 1.d0
    integer :: ntasks, id, rc
    integer, allocatable :: status(:)
    integer :: i, n, clock
    integer, allocatable :: seed(:)
    real(8) :: r

    ! temperature initial value, step and final value
    real(8) :: T = .01d0, dT = .01d0, T_final = 5.d0, beta
    ! lattice size
    integer, parameter :: L_x = 32, L_y = 32
    ! spins
    integer :: lattice(0:L_x - 1,0:L_y - 1) = 1
    ! coordinate of spin
    integer :: x, y
    ! warming up and evolution and measurement steps
    integer, parameter :: n_warmup = 10000, n_evol = 100000
    ! exchange interaction coefficient, magnetic field
    real(8) :: J = 1.d0, B = 0.d0
    ! magnetization, square of magnetization, , system energy, square of system energy
    ! average magnetization, average of square of magnetization, susceptibility
    ! average system energy, average of square of system energy, specific heat
    real(8) :: M, M_sqr, E_tmp, E, E_sqr, M_ave, M_sqr_ave, chi, E_ave, E_sqr_ave, C

    ! initialize MPI environment
    call MPI_INIT(rc)
    call MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, rc)
    call MPI_COMM_RANK(MPI_COMM_WORLD, id, rc)
    allocate(status(MPI_STATUS_SIZE))

    ! initialize seeds for different processes
    if (id == 0) then
        call SYSTEM_CLOCK(clock)
        call RANDOM_SEED(size = n)
        allocate(seed(n))
        do i = 1, n
            seed(i) = clock + 37 * i
        end do
```

```
42          call RANDOM_SEED(PUT = seed)
43          deallocate(seed)
44          do i = 1, ntasks - 1
45              call RANDOM_NUMBER(r)
46              clock = clock + Int(r * 1000000)
47              call MPI_SEND(clock, 1, MPI_INTEGER, i, i, MPI_COMM_WORLD, rc)
48          end do
49      else
50          call MPI_RECV(clock, 1, MPI_INTEGER, 0, id, MPI_COMM_WORLD, status, rc)
51          call RANDOM_SEED(size = n)
52          allocate(seed(n))
53          do i = 1, n
54              seed(i) = clock + 37 * i
55          end do
56          call RANDOM_SEED(PUT = seed)
57          deallocate(seed)
58      end if
59
60      ! open file for data storage
61      if (id == 0) then
62          open(unit = 1, file = 'data.txt', status = 'unknown')
63          write(*,'(4a20)') 'T', 'm', 'chi', 'C'
64      end if
65
66      do while (T < T_final)
67          beta = 1.d0 / kB / T
68          M = 0.d0
69          M_sqr = 0.d0
70          E = 0.d0
71          E_sqr = 0.d0
72
73          ! warming up
74          do i = 1, n_warmup
75              call EVOLUTION(lattice, L_x, L_y, beta, B, J)
76          end do
77
78          ! evolution and measurement
79          do i = 1, n_evol
80              ! try to flip a single spin
81              call EVOLUTION(lattice, L_x, L_y, beta, B, J)
82              ! magnetization
83              M = M + sum(lattice)
84              ! square of magnetization
85              M_sqr = M_sqr + sum(lattice)**2
86              ! system energy
```

```
87              E_tmp = 0.d0
88              do x = 0, L_x - 2
89                  do y = 0, L_y - 2
90                      E_tmp = E_tmp + lattice(x, y) * (lattice(x + 1, y) + lattice(x, y +
                            1))
91                  end do
92              end do
93              do x = 0, L_x - 2
94                  E_tmp = E_tmp + lattice(x, L_y - 1) * (lattice(x + 1, L_y - 1) +
                        lattice(x, 0))
95              end do
96              do y = 0, L_y - 2
97                  E_tmp = E_tmp + lattice(L_x - 1, y) * (lattice(L_x - 1, y + 1) +
                        lattice(0, y))
98              end do
99              E_tmp = E_tmp + lattice(L_x - 1, L_y - 1) * (lattice(0, L_y - 1) + Lattice(
                    L_x - 1, 0))
100             E_tmp = - E_tmp * J - B * sum(lattice)
101             E = E + E_tmp
102             ! square of system energy
103             E_sqr = E_sqr + E_tmp**2
104         end do
105         ! gather results
106         call MPI_REDUCE(M, M_ave, 1, MPI_REAL8, MPI_SUM, 0, MPI_COMM_WORLD, rc)
107         call MPI_REDUCE(M_sqr, M_sqr_ave, 1, MPI_REAL8, MPI_SUM, 0, MPI_COMM_WORLD, rc)
108         call MPI_REDUCE(E, E_ave, 1, MPI_REAL8, MPI_SUM, 0, MPI_COMM_WORLD, rc)
109         call MPI_REDUCE(E_sqr, E_sqr_ave, 1, MPI_REAL8, MPI_SUM, 0, MPI_COMM_WORLD, rc)
110         if (id == 0) then
111             ! average magnetization
112             M_ave = M_ave / dble(ntasks * n_evol)
113             ! average of square of magnetization
114             M_sqr_ave = M_sqr_ave / dble(ntasks * n_evol)
115             ! susceptibility
116             chi = beta * (M_sqr_ave - M_ave**2)
117             ! average system energy
118             E_ave = E_ave / dble(ntasks * n_evol)
119             ! average of square of system energy
120             E_sqr_ave = E_sqr_ave / dble(ntasks * n_evol)
121             ! specific heat
122             C = kB * beta**2 / dble((L_x) * (L_y)) * (E_sqr_ave - E_ave**2)
123             ! print and write results
124             write(*,'(4f20.10)') T, M_ave / dble((L_x) * (L_y)), chi, C
125             write(1,'(4f20.10)') T, M_ave / dble((L_x) * (L_y)), chi, C
126         end if
127         ! next temperature
```

```
128             T = T + dT
129         end do
130
131         ! close file for data storage
132         if (id == 0) then
133             close(1)
134         end if
135
136         ! done with MPI
137         call MPI_FINALIZE(rc)
138     end program main
139
140     subroutine EVOLUTION(lattice, L_x, L_y, beta, B, J)
141         ! try to flip a single spin
142         implicit none
143         ! lattice size
144         integer, intent(in) :: L_x, L_y
145         ! spins
146         integer, intent(inout) :: lattice(0:L_x - 1, 0:L_y - 1)
147         ! , magnet field, exchange interaction coefficient
148         real(8), intent(in) :: beta, B, J
149         real(8) :: r
150         ! coordinate of spin
151         integer :: x, y
152         ! energy change of flip spin
153         real(8) :: dE
154
155         ! choose a spin randomly
156         call RANDOM_NUMBER(r)
157         x = floor(r * dble(L_x))
158         call RANDOM_NUMBER(r)
159         y = floor(r * dble(L_y))
160
161         ! energy change to flip the spin
162         dE = 0.d0
163         dE = dE + 2.d0 * J * lattice(x,y) * (lattice(modulo(x - 1, L_x), y)&
164             + lattice(modulo(x + 1, L_x), y)&
165             + lattice(x, modulo(y - 1, L_y))&
166             + lattice(x, modulo(y + 1, L_y)))&
167             + 2.d0 * B * dble(lattice(x,y))
168
169         ! try to flip the spin
170         call RANDOM_NUMBER(r)
171         if (r < exp(-beta * dE)) then
172             lattice(x,y) = -lattice(x,y)
```

```
173     end if
174 end subroutine EVOLUTION
```

4.2 Wolff算法代码

```
1  program main
2      use mpi
3      implicit none
4      real(8), parameter :: pi = acos(-1.d0), kB = 1.d0
5      integer :: ntasks, id, rc
6      integer, allocatable :: status(:)
7      integer :: i, n, clock
8      integer, allocatable :: seed(:)
9      real(8) :: r
10
11     ! temperature initial value, step and final value
12     real(8) :: T = .01d0, dT = .01d0, T_final = 5.d0, beta
13     ! lattice size
14     integer, parameter :: L_x = 32, L_y = 32
15     ! spins
16     integer :: lattice(0:L_x - 1,0:L_y - 1) = 1
17     ! coordinate of spin
18     integer :: x, y
19     ! warming up and evolution and measurement steps
20     integer, parameter :: n_warmup = 200, n_evol = 2000
21     ! exchange interaction coefficient, probability of adding a spin to cluster
22     real(8) :: J = 1.d0, Padd
23     ! magnetization, square of magnetization, , system energy, square of system energy
24     ! average magnetization, average of square of magnetization, susceptibility
25     ! average system energy, average of square of system energy, specific heat
26     real(8) :: M, M_sqr, E_tmp, E, E_sqr, M_ave, M_sqr_ave, chi, E_ave, E_sqr_ave, C
27
28     ! initialize MPI environment
29     call MPI_INIT(rc)
30     call MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, rc)
31     call MPI_COMM_RANK(MPI_COMM_WORLD, id, rc)
32     allocate(status(MPI_STATUS_SIZE))
33
34     ! initialize seeds for different processes
35     if (id == 0) then
36         call SYSTEM_CLOCK(clock)
37         call RANDOM_SEED(size = n)
38         allocate(seed(n))
39         do i = 1, n
```

```
40              seed(i) = clock + 37 * i
41          end do
42          call RANDOM_SEED(PUT = seed)
43          deallocate(seed)
44          do i = 1, ntasks - 1
45              call RANDOM_NUMBER(r)
46              clock = clock + Int(r * 1000000)
47              call MPI_SEND(clock, 1, MPI_INTEGER, i, i, MPI_COMM_WORLD, rc)
48          end do
49      else
50          call MPI_RECV(clock, 1, MPI_INTEGER, 0, id, MPI_COMM_WORLD, status, rc)
51          call RANDOM_SEED(size = n)
52          allocate(seed(n))
53          do i = 1, n
54              seed(i) = clock + 37 * i
55          end do
56          call RANDOM_SEED(PUT = seed)
57          deallocate(seed)
58      end if
59
60      ! open file for data storage
61      if (id == 0) then
62          open(unit = 1, file = 'data.txt', status = 'unknown')
63          write(*,'(4a20)') 'T', 'm', 'chi', 'C'
64      end if
65
66      do while (T < T_final)
67          beta = 1.d0 / kB / T
68          Padd = 1 - exp(-2 * beta * J)
69          M = 0.d0
70          M_sqr = 0.d0
71          E = 0.d0
72          E_sqr = 0.d0
73
74          ! warming up
75          do i =1, n_warmup
76              call EVOLUTION(lattice, L_x, L_y, Padd)
77          end do
78
79          ! evolution and measurement
80          do i = 1, n_evol
81              ! flip a cluster
82              call EVOLUTION(lattice, L_x, L_y, Padd)
83              ! magnetization
84              M = M + abs(sum(lattice))
```

```
85              ! square of magnetization
86              M_sqr = M_sqr + sum(lattice)**2
87              ! system energy
88              E_tmp = 0.d0
89              do x = 0, L_x - 2
90                  do y = 0, L_y - 2
91                      E_tmp = E_tmp + lattice(x, y) * (lattice(x + 1, y) + lattice(x, y +
                            1))
92                  end do
93              end do
94              do x = 0, L_x - 2
95                  E_tmp = E_tmp + lattice(x, L_y - 1) * (lattice(x + 1, L_y - 1) +
                        lattice(x, 0))
96              end do
97              do y = 0, L_y - 2
98                  E_tmp = E_tmp + lattice(L_x - 1, y) * (lattice(L_x - 1, y + 1) +
                        lattice(0, y))
99              end do
100             E_tmp = E_tmp + lattice(L_x - 1, L_y - 1) * (lattice(0, L_y - 1) + Lattice(
                    L_x - 1, 0))
101             E_tmp = - E_tmp * J
102             E = E + E_tmp
103             ! square of system energy
104             E_sqr = E_sqr + E_tmp**2
105         end do
106         ! gather results
107         call MPI_REDUCE(M, M_ave, 1, MPI_REAL8, MPI_SUM, 0, MPI_COMM_WORLD, rc)
108         call MPI_REDUCE(M_sqr, M_sqr_ave, 1, MPI_REAL8, MPI_SUM, 0, MPI_COMM_WORLD, rc)
109         call MPI_REDUCE(E, E_ave, 1, MPI_REAL8, MPI_SUM, 0, MPI_COMM_WORLD, rc)
110         call MPI_REDUCE(E_sqr, E_sqr_ave, 1, MPI_REAL8, MPI_SUM, 0, MPI_COMM_WORLD, rc)
111         if (id == 0) then
112             ! average magnetization
113             M_ave = M_ave / dble(ntasks * n_evol)
114             ! average of square of magnetization
115             M_sqr_ave = M_sqr_ave / dble(ntasks * n_evol)
116             ! susceptibility
117             chi = beta * (M_sqr_ave - M_ave**2)
118             ! average system energy
119             E_ave = E_ave / dble(ntasks * n_evol)
120             ! average of square of system energy
121             E_sqr_ave = E_sqr_ave / dble(ntasks * n_evol)
122             ! specific heat
123             C = kB * beta**2 / dble(L_x * L_y) * (E_sqr_ave - E_ave**2)
124             ! print and write results
125             write(*,'(4f20.10)') T, M_ave / dble(L_x * L_y), chi, C
```

```
126                 write(1,'(4f20.10)') T, M_ave / dble(L_x * L_y), chi, C
127             end if
128             ! next temperature
129             T = T + dT
130         end do
131
132         ! close file for data storage
133         if (id == 0) then
134             close(1)
135         end if
136
137         ! done with MPI
138         call MPI_FINALIZE(rc)
139 end program main
140
141 subroutine EVOLUTION(lattice, L_x, L_y, Padd)
142         ! flip a cluster
143         implicit none
144         ! lattice size
145         integer, intent(in) :: L_x, L_y
146         ! spins
147         integer, intent(inout) :: lattice(0:L_x - 1, 0:L_y - 1)
148         ! probability of adding a spin to cluster
149         real(8), intent(in) :: Padd
150         real(8) :: r
151         ! coordinate of seed spin and neighboring spin
152         integer :: x, y, x_neighbor, y_neighbor, i
153         ! spin in cluster
154         integer :: cluster(L_x * L_y, 2)
155         ! # of seed and cluster spin in cluster, spin direction in cluster
156         integer :: n_seed, n_cluster, cluster_spin
157         ! whether a spin is in cluster
158         logical :: notincluster
159
160         ! choose a seed spin randomly, add it to cluster and spin it
161         call RANDOM_NUMBER(r)
162         x = floor(r * dble(L_x))
163         call RANDOM_NUMBER(r)
164         y = floor(r * dble(L_y))
165         n_cluster = 1
166         n_seed = 1
167         cluster(1,1) = x
168         cluster(1,2) = y
169         cluster_spin = lattice(x, y)
170         lattice(x, y) = -lattice(x, y)
```

```
171
172     ! add neighboring spin to cluster and flip them
173     do while (n_seed <= n_cluster)
174         x = cluster(n_seed, 1)
175         y = cluster(n_seed, 2)
176         n_seed = n_seed + 1
177
178         ! choose a neighboring spin
179         x_neighbor = modulo(x - 1, L_x)
180         y_neighbor = y
181         ! exam if this neighboring spin has the same spin as seed spin
182         if (lattice(x_neighbor, y_neighbor) == cluster_spin) then
183             call RANDOM_NUMBER(r)
184             if (r < Padd) then
185                 ! exam if this neighboring spin has already been in cluster
186                 notincluster = .true.
187                 do i = n_cluster, 1, -1
188                     if ((cluster(i, 1) == x_neighbor) .and. (cluster(i, 2) ==
                            y_neighbor)) then
189                         notincluster = .false.
190                         exit
191                     end if
192                 end do
193                 if (notincluster .eqv. .true.) then
194                     ! add this neighboring spin to cluster
195                     n_cluster = n_cluster + 1
196                     cluster(n_cluster, 1) = x_neighbor
197                     cluster(n_cluster, 2) = y_neighbor
198                     ! flip this neighboring spin
199                     lattice(x_neighbor, y_neighbor) = -lattice(x_neighbor, y_neighbor)
200                 end if
201             end if
202         end if
203
204         x_neighbor = modulo(x + 1, L_x)
205         ! y_neighbor = y
206         if (lattice(x_neighbor, y_neighbor) == cluster_spin) then
207             call RANDOM_NUMBER(r)
208             if (r < Padd) then
209                 notincluster = .true.
210                 do i = n_cluster, 1, -1
211                     if ((cluster(i, 1) == x_neighbor) .and. (cluster(i, 2) ==
                            y_neighbor)) then
212                         notincluster = .false.
213                         exit
```

```
214                     end if
215                 end do
216                 if (notincluster .eqv. .true.) then
217                     n_cluster = n_cluster + 1
218                     cluster(n_cluster, 1) = x_neighbor
219                     cluster(n_cluster, 2) = y_neighbor
220                     lattice(x_neighbor, y_neighbor) = -lattice(x_neighbor, y_neighbor)
221                 end if
222             end if
223         end if
224 
225         x_neighbor = x
226         y_neighbor = modulo(y - 1, L_y)
227         if (lattice(x, y_neighbor) == cluster_spin) then
228             call RANDOM_NUMBER(r)
229             if (r < Padd) then
230                 notincluster = .true.
231                 do i = n_cluster, 1, -1
232                     if ((cluster(i, 1) == x_neighbor) .and. (cluster(i, 2) ==
                            y_neighbor)) then
233                         notincluster = .false.
234                         exit
235                     end if
236                 end do
237                 if (notincluster .eqv. .true.) then
238                     n_cluster = n_cluster + 1
239                     cluster(n_cluster, 1) = x_neighbor
240                     cluster(n_cluster, 2) = y_neighbor
241                     lattice(x_neighbor, y_neighbor) = -lattice(x_neighbor, y_neighbor)
242                 end if
243             end if
244         end if
245 
246         ! x_neighbor = x
247         y_neighbor = modulo(y + 1, L_y)
248         if (lattice(x_neighbor, y_neighbor) == cluster_spin) then
249             call RANDOM_NUMBER(r)
250             if (r < Padd) then
251                 notincluster = .true.
252                 do i = n_cluster, 1, -1
253                     if ((cluster(i, 1) == x_neighbor) .and. (cluster(i, 2) ==
                            y_neighbor)) then
254                         notincluster = .false.
255                         exit
256                     end if
```

```
                end do
                if (notincluster .eqv. .true.) then
                    n_cluster = n_cluster + 1
                    cluster(n_cluster, 1) = x_neighbor
                    cluster(n_cluster, 2) = y_neighbor
                    lattice(x_neighbor, y_neighbor) = -lattice(x_neighbor, y_neighbor)
                end if
            end if
        end if
    end do
end subroutine EVOLUTION
```

4.3 用有限尺度标度分析临界指数的Wolff算法代码

```
program main
    use mpi
    implicit none
    real(8), parameter :: pi = acos(-1.d0), kB = 1.d0
    integer :: ntasks, id, rc
    integer, allocatable :: status(:)
    integer :: i, n, clock
    integer, allocatable :: seed(:)
    real(8) :: r

    ! temperature initial value, step and final value
    real(8) :: T, dT = .002d0, T_final = 2.35d0, beta
    ! lattice size intial value, step and final value
    integer :: L = 30, dL = 2, L_final = 130
    ! spins
    integer, allocatable :: lattice(:,:)
    ! coordinate of spin
    integer :: x, y
    ! warming up and evolution and measurement steps
    integer, parameter :: n_warmup = 200, n_evol = 2000
    ! exchange interaction coefficient, probability of adding a spin to cluster
    real(8) :: J = 1.d0, Padd
    ! magnetization, square of magnetization, , system energy, square of system energy
    ! average magnetization, average of square of magnetization, susceptibility
    ! average system energy, average of square of system energy, specific heat
    real(8) :: M, M_sqr, E_tmp, E, E_sqr, M_ave, M_sqr_ave, chi, E_ave, E_sqr_ave, C
    ! temperature, magnetization, susceptibility and specific heat at critical point
    real(8) :: T_c, m_c, chi_max, C_max

    ! initialize MPI environment
```

```
    call MPI_INIT(rc)
    call MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, rc)
    call MPI_COMM_RANK(MPI_COMM_WORLD, id, rc)
    allocate(status(MPI_STATUS_SIZE))

    ! initialize seeds for different processes
    if (id == 0) then
        call SYSTEM_CLOCK(clock)
        call RANDOM_SEED(size = n)
        allocate(seed(n))
        do i = 1, n
            seed(i) = clock + 37 * i
        end do
        call RANDOM_SEED(PUT = seed)
        deallocate(seed)
        do i = 1, ntasks - 1
            call RANDOM_NUMBER(r)
            clock = clock + Int(r * 1000000)
            call MPI_SEND(clock, 1, MPI_INTEGER, i, i, MPI_COMM_WORLD, rc)
        end do
    else
        call MPI_RECV(clock, 1, MPI_INTEGER, 0, id, MPI_COMM_WORLD, status, rc)
        call RANDOM_SEED(size = n)
        allocate(seed(n))
        do i = 1, n
            seed(i) = clock + 37 * i
        end do
        call RANDOM_SEED(PUT = seed)
        deallocate(seed)
    end if

    do while (L <= L_final)
        if (id == 0) then
            ! open file for storing date in processes
            open(unit = 1, file = 'data.txt', status = 'unknown', position = 'append')
            ! open file for storing summary data
            open(unit = 2, file = 'summary.txt', status = 'unknown', position = 'append
                ')
            write(*,*) '
                !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
                '
            write(*,'("␣!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!␣␣L␣␣=␣␣",i10,"␣␣
                !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!")') L
            write(*,*) '
                !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
            ,
71          write(*,'(4a20)') 'T', 'm', 'chi', 'C'
72      end if
73      T = 2.25d0
74      T_c = 0
75      m_c = 0
76      chi_max = 0
77      C_max = 0
78      allocate(Lattice(0:L - 1,0:L - 1))
79      Lattice = 1
80
81      do while (T < T_final)
82          beta = 1.d0 / kB / T
83          Padd = 1 - exp(-2 * beta * J)
84          M = 0.d0
85          M_sqr = 0.d0
86          E = 0.d0
87          E_sqr = 0.d0
88
89          ! warming up
90          do i =1, n_warmup
91              call EVOLUTION(lattice, L, Padd)
92          end do
93
94          ! evolution and measurement
95          do i = 1, n_evol
96              ! flip the cluster
97              call EVOLUTION(lattice, L, Padd)
98              ! magnetization
99              M = M + abs(sum(lattice))
100             ! square of magnetization
101             M_sqr = M_sqr + sum(lattice)**2
102             ! system energy
103             E_tmp = 0.d0
104             do x = 0, L - 2
105                 do y = 0, L - 2
106                     E_tmp = E_tmp + lattice(x, y) * (lattice(x + 1, y) + lattice(x,
                            y + 1))
107                 end do
108             end do
109             do x = 0, L - 2
110                 E_tmp = E_tmp + lattice(x, L - 1) * (lattice(x + 1, L - 1) +
                        lattice(x, 0))
111             end do
112             do y = 0, L - 2
```

```
113                     E_tmp = E_tmp + lattice(L - 1, y) * (lattice(L - 1, y + 1) +
                            lattice(0, y))
114                 end do
115                 E_tmp = E_tmp + lattice(L - 1, L - 1) * (lattice(0, L - 1) + Lattice(L
                        - 1, 0))
116                 E_tmp = - E_tmp * J
117                 E = E + E_tmp
118                 ! square of system energy
119                 E_sqr = E_sqr + E_tmp**2
120             end do
121             ! gather results
122             call MPI_REDUCE(M, M_ave, 1, MPI_REAL8, MPI_SUM, 0, MPI_COMM_WORLD, rc)
123             call MPI_REDUCE(M_sqr, M_sqr_ave, 1, MPI_REAL8, MPI_SUM, 0, MPI_COMM_WORLD,
                    rc)
124             call MPI_REDUCE(E, E_ave, 1, MPI_REAL8, MPI_SUM, 0, MPI_COMM_WORLD, rc)
125             call MPI_REDUCE(E_sqr, E_sqr_ave, 1, MPI_REAL8, MPI_SUM, 0, MPI_COMM_WORLD,
                    rc)
126             if (id == 0) then
127                 ! average magnetization
128                 M_ave = M_ave / dble(ntasks * n_evol)
129                 ! average of square of magnetization
130                 M_sqr_ave = M_sqr_ave / dble(ntasks * n_evol)
131                 ! susceptibility
132                 chi = beta * (M_sqr_ave - M_ave**2)
133                 ! average system energy
134                 E_ave = E_ave / dble(ntasks * n_evol)
135                 ! average of square of system energy
136                 E_sqr_ave = E_sqr_ave / dble(ntasks * n_evol)
137                 ! specific heat
138                 C = kB * beta**2 / dble(L * L) * (E_sqr_ave - E_ave**2)
139                 ! print and write results of this temperature
140                 write(*,'(4f20.10)') T, M_ave / dble(L * L), chi, C
141                 write(1,'(4f20.10)') T, M_ave / dble(L * L), chi, C
142
143                 ! find temperature, magnetization, susceptibility, specific heat at
                        critical point
144                 if (chi > chi_max) then
145                     m_c = M_ave / dble(L * L)
146                     chi_max = chi
147                     T_c = T
148                     C_max = C
149                 end if
150             end if
151             ! next temperature
152             T = T + dT
```

```
        end do

        ! print and write results about critical temperature
        if (id == 0) then
            write(*,*) '
                !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
                '
            write(*,'(a5,a15,3a20)') 'L', 'T_c', 'm_c', 'chi_max', 'C_max'
            write(*,'(i5,f15.10,3f20.10)') L, T_c, m_c, chi_max, C_max
            write(*,*) '
                !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
                '
            write(*,*)
            write(1,*)
            write(2,'(i10,4f20.10)') L, T_c, m_c, chi_max, C_max
            close(1)
            close(2)
        end if
        ! next lattice size
        L = L + dL
        deallocate(Lattice)
    end do

    ! done with MPI
    call MPI_FINALIZE(rc)
end program main

subroutine EVOLUTION(lattice, L, Padd)
    ! flip a cluster
    implicit none
    ! lattice size
    integer, intent(in) :: L
    ! spins
    integer, intent(inout) :: lattice(0:L - 1, 0:L - 1)
    ! probability of adding a spin to cluster
    real(8), intent(in) :: Padd
    real(8) :: r
    ! coordinate of seed spin and neighboring spin
    integer :: x, y, x_neighbor, y_neighbor, i
    ! spins in cluster
    integer :: cluster(L * L, 2)
    ! # of seed and cluster spin in cluster, spin direction in cluster
    integer :: n_seed, n_cluster, cluster_spin
    ! whether a spin is in cluster
    logical :: notincluster
```

```
194
195     ! choose a seed spin randomly, add it to cluster and spin it
196     call RANDOM_NUMBER(r)
197     x = floor(r * dble(L))
198     call RANDOM_NUMBER(r)
199     y = floor(r * dble(L))
200     n_cluster = 1
201     n_seed = 1
202     cluster(1,1) = x
203     cluster(1,2) = y
204     cluster_spin = lattice(x, y)
205     lattice(x, y) = -lattice(x, y)
206
207     ! add neighboring spin to cluster and flip them
208     do while (n_seed <= n_cluster)
209         x = cluster(n_seed, 1)
210         y = cluster(n_seed, 2)
211         n_seed = n_seed + 1
212
213         ! choose a neighboring spin
214         x_neighbor = modulo(x - 1, L)
215         y_neighbor = y
216         ! exam if this neighboring spin has the same spin as seed spin
217         if (lattice(x_neighbor, y_neighbor) == cluster_spin) then
218             call RANDOM_NUMBER(r)
219             if (r < Padd) then
220                 ! exam if this neighboring spin has already been in cluster
221                 notincluster = .true.
222                 do i = n_cluster, 1, -1
223                     if ((cluster(i, 1) == x_neighbor) .and. (cluster(i, 2) ==
                            y_neighbor)) then
224                         notincluster = .false.
225                         exit
226                     end if
227                 end do
228                 if (notincluster .eqv. .true.) then
229                     ! add this neighboring spin to cluster
230                     n_cluster = n_cluster + 1
231                     cluster(n_cluster, 1) = x_neighbor
232                     cluster(n_cluster, 2) = y_neighbor
233                     ! flip this neighboring spin
234                     lattice(x_neighbor, y_neighbor) = -lattice(x_neighbor, y_neighbor)
235                 end if
236             end if
237         end if
```

```
238
239         x_neighbor = modulo(x + 1, L)
240         ! y_neighbor = y
241         if (lattice(x_neighbor, y_neighbor) == cluster_spin) then
242             call RANDOM_NUMBER(r)
243             if (r < Padd) then
244                 notincluster = .true.
245                 do i = n_cluster, 1, -1
246                     if ((cluster(i, 1) == x_neighbor) .and. (cluster(i, 2) ==
                            y_neighbor)) then
247                         notincluster = .false.
248                         exit
249                     end if
250                 end do
251                 if (notincluster .eqv. .true.) then
252                     n_cluster = n_cluster + 1
253                     cluster(n_cluster, 1) = x_neighbor
254                     cluster(n_cluster, 2) = y_neighbor
255                     lattice(x_neighbor, y_neighbor) = -lattice(x_neighbor, y_neighbor)
256                 end if
257             end if
258         end if
259
260         x_neighbor = x
261         y_neighbor = modulo(y - 1, L)
262         if (lattice(x, y_neighbor) == cluster_spin) then
263             call RANDOM_NUMBER(r)
264             if (r < Padd) then
265                 notincluster = .true.
266                 do i = n_cluster, 1, -1
267                     if ((cluster(i, 1) == x_neighbor) .and. (cluster(i, 2) ==
                            y_neighbor)) then
268                         notincluster = .false.
269                         exit
270                     end if
271                 end do
272                 if (notincluster .eqv. .true.) then
273                     n_cluster = n_cluster + 1
274                     cluster(n_cluster, 1) = x_neighbor
275                     cluster(n_cluster, 2) = y_neighbor
276                     lattice(x_neighbor, y_neighbor) = -lattice(x_neighbor, y_neighbor)
277                 end if
278             end if
279         end if
280
```

```
281         ! x_neighbor = x
282         y_neighbor = modulo(y + 1, L)
283         if (lattice(x_neighbor, y_neighbor) == cluster_spin) then
284             call RANDOM_NUMBER(r)
285             if (r < Padd) then
286                 notincluster = .true.
287                 do i = n_cluster, 1, -1
288                     if ((cluster(i, 1) == x_neighbor) .and. (cluster(i, 2) ==
                            y_neighbor)) then
289                         notincluster = .false.
290                         exit
291                     end if
292                 end do
293                 if (notincluster .eqv. .true.) then
294                     n_cluster = n_cluster + 1
295                     cluster(n_cluster, 1) = x_neighbor
296                     cluster(n_cluster, 2) = y_neighbor
297                     lattice(x_neighbor, y_neighbor) = -lattice(x_neighbor, y_neighbor)
298                 end if
299             end if
300         end if
301     end do
302 end subroutine EVOLUTION
```