一维 Bose-Hubbard 模型的量子相变

陈骜 程楠 童业恒

摘 要：

Bose-Hubbard 模型在零温时展现出特殊的 Mott 绝缘----超流量子相变。本次研究中，我们使用了一种全新的量子蒙特卡洛方法（随机序列展开）来模拟一维 Bose-Hubbard 模型，得到了该模型在$0 < \frac{\mu}{U} < 1.5$，$0 < \frac{z\omega}{U} < 0.3$范围内的相图。之后，我们从平均场理论角度对该相图做出了分析。

目录

一　引言

Hubbard 模型是强关联物理中一个重要的基本模型，它在理解晶体磁性，Mott 绝缘和超导现象中扮演着重要角色。近年来研究发现，该模型的玻色子对应物——Bose-Hubbard 模型，在数学上有着比传统 Hubbard 模型更简单的结构[1,2,3]，并且在零温时展现出特殊的 Mott 绝缘-超流量子相变。Bose-Hubbard 模型中的玻色子可以对应于实际体系中的在超导岛间发生 Josephson 隧穿的电子库珀对，衬底上运动的氦原子，或者光子晶体中的冷原子。因此，研究 Bose-Hubbard 模型对于深入理解超导，超流和冷原子体系中的物理现象有着重要作用。在本次研究中，我们使用了一种全新的量子蒙特卡洛方法[4,5,6]（随机序列展开）模拟一维 Bose-Hubbard 模型，得到了该模型在$0 < \frac{\mu}{U} < 1.5$，$0 < \frac{Z\omega}{U} < 0.3$范围内的相图。之后，我们平均场理论角度对该相图做出了理论分析。

二　一维 Bose-Hubbard 模型

体系哈密顿量为

$$H = -\omega \sum_{\langle i,j \rangle} \left(b_i^\dagger b_j + b_j^\dagger b_i \right) - \mu \sum_i n_i + \frac{U}{2} \sum_i n_i (n_i - 1)$$

其中，$\langle i,j \rangle$表示对最近邻格点求和，$b_i^\dagger$是i格点的产生算符，b_j是j格点的湮灭算符，两者满足对易关系$\left[b_i, b_j^\dagger \right] = \delta_{ij}$；$n_i = b_i^\dagger b_i$是$i$格点的粒子数算符。哈密顿中的第一项允许玻色子在不同格点间的跃迁；第二项中的μ表示粒子的化学势；最后一项表示同一格点上玻色子之间的相互作用，假定只存在两体相互作用，并且这个相互作用是排斥的[1]。

首先考虑$\omega = 0$的极限情形，此时哈密顿量在 Fock 空间中是对角化的，体系可以严格求解。当$\frac{\mu}{U}$不是整数时，体系基态非简并，每个格点上占据的粒子数都是$n_0(\frac{\mu}{U})$，其中

$$n_0\left(\frac{\mu}{U}\right) = \begin{cases} 0，\text{如果} \dfrac{\mu}{U} < 0 \\ 1，\text{如果 } 0 < \dfrac{\mu}{U} < 1 \\ \cdots\cdots \\ n，\text{如果} n-1 < \dfrac{\mu}{U} < n \end{cases}$$

[1] 容易看出，若$U \le 0$，体系将不存在一个最低能态，因为增加一个玻色子总会使能量降低。

当$\frac{\mu}{U}=n$为整数时，体系基态简并，每个格点上可以占据n个或$n+1$个粒子。

现在，考虑体系处于某个固定的化学势$n-1<\frac{\mu}{U}<n$的情形。$\omega=0$时，体系的基态中每个格点占据粒子数为n。记$\alpha\equiv\frac{\mu}{U}-n+\frac{1}{2}$，则某个格点再产生一个玻色子，体系能量增加$\delta E_p=(\frac{1}{2}-\alpha)U$；某个格点湮灭一个玻色子，体系能量增加$\delta E_h=(\frac{1}{2}+\alpha)U$。

再考虑ω远小于δE_h和δE_p的微扰论情形，近似认为基态的粒子数分布与$\omega=0$时相同。此时体系中每增加（减少）一个玻色子，体系约有$-\omega+\delta E_{h(p)}$的能量增加，因此体系倾向于保持每个格点上粒子数不变（ω是小量），此时体系处于 Mott 绝缘态。当ω逐渐增加使得$-\omega+\delta E_{h(p)}$小于零时，玻色子在相邻格点之间跃迁不会引起能量上升，因此体系将处于超流态。

由以上分析可以看出，在以$\frac{\omega}{U}$为横坐标，$\frac{\mu}{U}$为纵坐标的相图中，会有一部分区域每个格点上粒子占据数不随体系状态改变，这个区域对应于体系的 Mott 绝缘相，相图中体系平均粒子占据数会随体系状态改变的区域对应于超流相。

三　量子蒙特卡洛模拟

1. 随机级数展开（SSE）算法

对于可以映射为$d+1$维经典模型的d维量子统计模型，传统的量子蒙特卡洛方法（即“世界线”算法）是直接在对应的$d+1$维经典模型中进行经典蒙特卡洛模拟。但这种处理方法往往会需要多次对模拟参数取极限值。在实际模拟中，当遇到需要取极限的情况时，就需要多次改变参数的值，直到系统行为不再出现明显变化为止。这样做会大大增加模拟的耗时。例如，对于哈密顿量为

$$H=-J\sum_i\sigma_i^z\sigma_{i+1}^z-h_x\sum_i\sigma_i^x$$

的一维横场伊辛模型，其配分函数为

$$Z=\mathrm{tr}\left(e^{-\beta H}\right)=\lim_{\Delta\tau\to0}\mathrm{tr}(e^{-H\Delta\tau})^{\beta/\Delta\tau}$$

该模型可以映射为 2 维空间中的经典各向异性伊辛模型，其相互作用系数分别为

$$K_1=J\Delta\tau\qquad K_2=-\frac{1}{2}\ln h_x\Delta\tau$$

在用“世界线”算法进行模拟时，需要取两步极限：第一步为$\Delta\tau \to 0$，使得虚时维度上的截面无穷小；第二步为$\beta \to \infty$的零温极限，使得模拟结果能够反映量子相变过程。

为了改进“世界线”算法需要取多次极限的这一不足，我们利用随机级数展开（SSE）算法对 Bose-Hubbard 模型进行研究。该算法避免了“世界线”算法中的$\Delta\tau \to 0$极限，使得模拟的实际操作中只需要取一步零温极限$\beta \to \infty$即可。随机级数展开算法的详细介绍见附录 A。

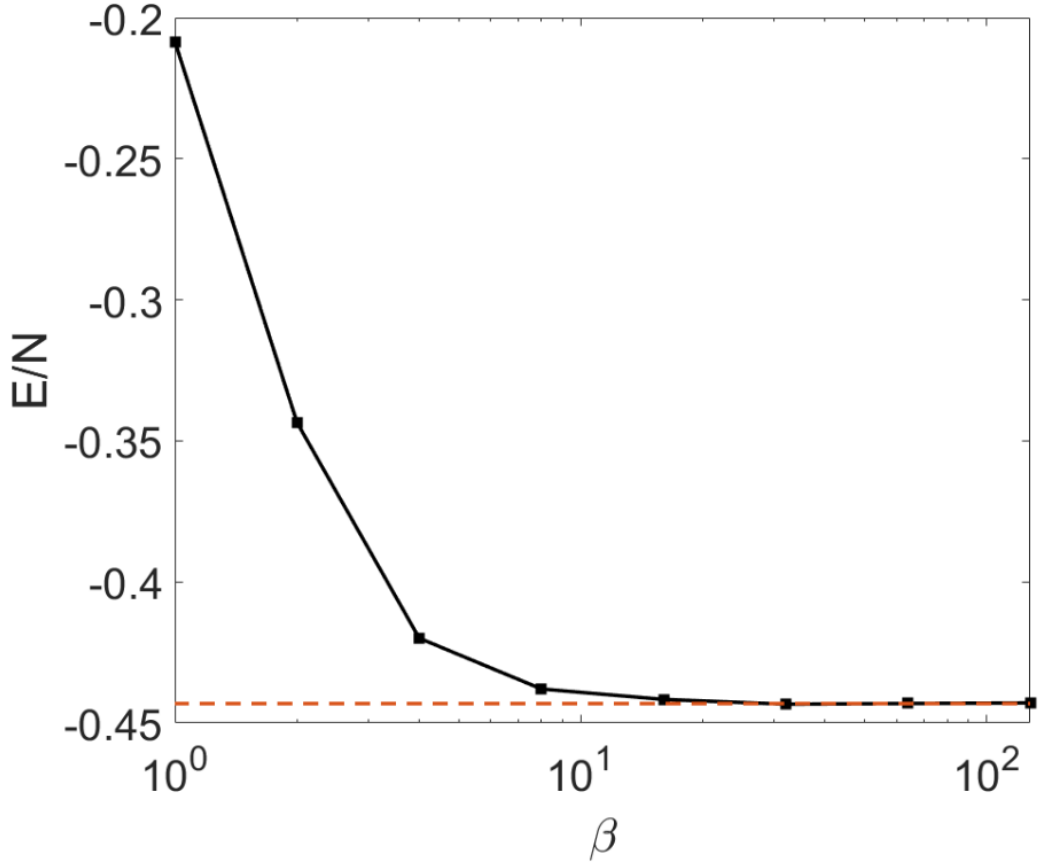


图1. 随机级数展开算法求解一维反铁磁自旋−1/2海森堡模型的基态，图中虚线为基态能量的理论值

为展示随机级数展开的效果，我们以一维反铁磁自旋-1/2的海森堡模型为例，具体的算法实现步骤可参见附录 B，代码见附录 D。如下图所示，随着$\beta \to \infty$，系统的能量逐渐达到稳定，最后固定在基态能量的理论值$e_0 = -\ln 2 + 1/4$附近。

这里也明显可以看出在模拟中取极限的困难。由于需要多次取值直到系统行为不变，期间会浪费大量的计算量。如果使用“世界线”算法进行两次极限的选取，其计算量将远大于随机级数展开算法中的一次极限。因此，后者为我们之后研究 Bose-Hubbard 模型的量子相变提供了很好的手段。

实际中我们还使用了循环更新的算法[4]来进一步提升程序效率，可参见附录 D、E 中的程序。

2. Bose-Hubbard 模型的相图

利用随机级数展开方法，可以得到巨正则系综下一维 Bose-Hubbard 模型的行为。我们选取一个足够高的β，固定$\omega = 1$，改变玻色子的相互作用能量U以及化学势μ，通过观察系统每个格点平均占据数ρ的变化得到系统的相图，如图 2 所示。

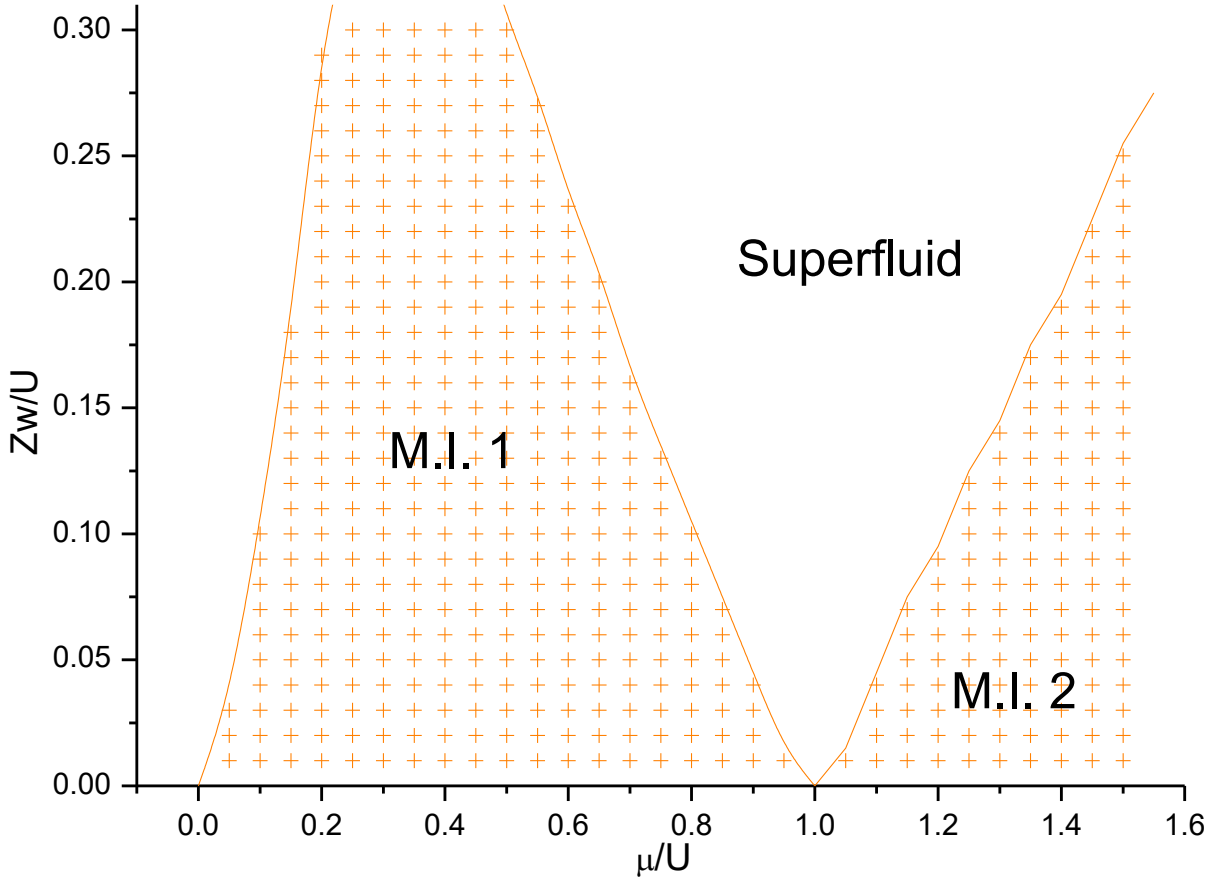


图2. SSE 模拟得到的相图。M.I. 1 表示粒子占据数为1的 Mott 绝缘态，M.I. 2 表示粒子占据数为2的 Mott 绝缘态，其他区域（Superfluid）表示超流态。

图中，有两个区域ρ的值固定为 1 和 2 保持不变，这是 Mott 绝缘相。在其他区域，ρ的值随系统参数变化而连续变化，这是超流相。

四　基于平均场近似的理论分析

平均场理论的核心在于假设体系波动很小，体系其他粒子对给定粒子的作用可以用一个平均场来代替。原本的哈密顿量在这种近似下不同的粒子间的耦合便会解除。用 Z 表示最近邻格点数，最近邻格点对给定格点的等价相互作用为ψ，体系平均场下的哈密顿量为[7]

$$H_{MF} = -\omega Z \sum_i \left(b_i^\dagger \psi + b_i \psi^*\right) - \mu \sum_i n_i + \frac{U}{2} \sum_i n_i (n_i - 1)$$

体系原本的哈密顿量H有$U(1)$对称性，即在$b_i \to e^{-i\varphi} b_i$下哈密顿量保持不变。平均场近似下，若$\psi \neq 0$，体系$U(1)$对称性破缺，因此，$\psi$可以看成体系的序参量，体系 Mott 绝缘-超流相变对应于体系$U(1)$对称性的自发破缺[7]。

原则上，序参量取值由自洽方程决定，即求解序参量为ψ时H_{MF}的基态，然后计算$\omega Z \frac{\sum_{i=1}^{N}\langle b_i\rangle}{N}$，$\omega Z \frac{\sum_{i=1}^{N}\langle b_i\rangle}{N}$在平均场意义下是最近邻格点对给定格点的平均作用，因此可以得到自洽方程为$\psi \equiv Z\omega \frac{\sum_{i=1}^{N}\langle b_i\rangle}{N}$。但是自洽方程的求解需要知道体系的基态波函数，平均场近似后体系哈密顿量在福克空间仍不是对角化的，求解基态波函数十分困难。我们从另外的角度考虑给定状态下序参量的取值。

考虑此基态下记平均场哈密顿量的基态能量为$E_{MF}(\psi)$，则系统原本的哈密顿量在平均场近似下的基态上的能量

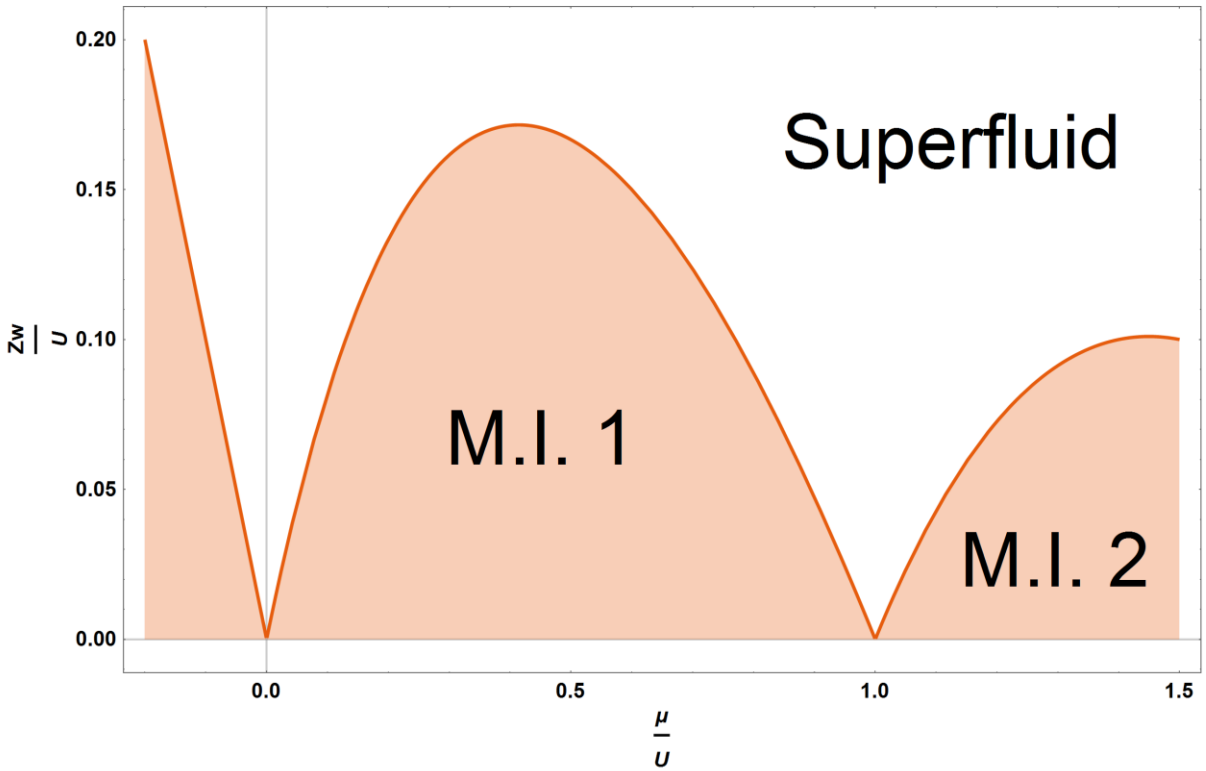
$$\frac{E_0}{N} = \frac{E_{MF}(\psi)}{N} + Z\omega\psi\psi^*$$

做微扰序参量的取值应使$\frac{E_0}{N}$极小，用标准的微扰方法处理H_{MF}，得到

$$\frac{E_0}{N} = \frac{E_{00}}{N} + r|\psi|^2 + \mathcal{O}(|\psi|^4)$$

其中

$$r = \chi_0\left(\frac{\mu}{U}\right)\left[1 - Z\omega\chi_0\left(\frac{\mu}{U}\right)\right]$$



$$\chi_0\left(\frac{\mu}{U}\right)=\frac{n_0\left(\frac{\mu}{U}\right)+1}{Un_0\left(\frac{\mu}{U}\right)-\mu}+\frac{n_0\left(\frac{\mu}{U}\right)}{\mu-U\left[n_0\left(\frac{\mu}{U}\right)-1\right]}$$

图3.平均场近似下的相图

$r>0$时，E_0在$\psi=0$时取最小值，体系处于 Mott 绝缘相；$r<0$时，E_0在$\psi\neq 0$时取最小值，体系处于超流相。因此，相图中$r=0$的曲线对应于相变点。图 3 是平均场理论预言的相图。平均场理论预言的相图和蒙特卡洛模拟得到的相图定性上符合得很好，但是在确定相变点位置上，平均场理论得到的结果与模拟结果有较大偏差，这是由于相变点附近体系涨落较大造成的，运用统计场论的方法可以对相图有着更精确的分析[8,9]。

五　参考文献

[1] Fisher, M.P.A., Weichman, P.B., Grinstein, G., and Fisher, D.S. (1989) *Phys. Rev.* B **40**, 546

[2] Ma, M., Halperin, B. I., Lee, P. A. (1986) *Phys. Rev.* B **34,** 3136

[3] Giamarchi, T., Schulz, H. J. (1988) *Phys. Rev.* B. **37**, 325

[4] Sandvik, A. W. (1999) *Phys. Rev.* B **59**, R14157

[5] Syljuasen, O. F., Sandvik, A. W. (2002) *Phys. Rev.* E **66**, 046701

[6] Sengupta, P., Sandvik, A. W., and Campbell, D. K. (2002) *Phys. Rev.* B **65** 155113

[7] Sachdev, Subir (2011). *Quantum phase transitions*. Cambridge University Press. ISBN 9780521514682. OCLC 693207153

[8] Freericks, J. K., and Monien, H. (1994) *Europhys. Lett.* **26** 545

[9] Freericks, J. K., and Monien, H. (1996) *Phys. Rev.* B **53**, 2691

附录 A　随机级数展开算法

该算法通过改写量子统计系统的配分函数，将对态的求和转化为对算符的求和，并且避免了“世界线”算法中的$\Delta\tau \to 0$极限，使得模拟中只需要取一步零温极限$\beta \to \infty$。

对配分函数 Z 的改写如下[4]，

$$Z = \mathrm{tr}\left(e^{-\beta H}\right) = \mathrm{tr}\left(\sum_{n=0}^{\infty}\frac{\beta^n}{n!}(-H)^n\right) = \sum_{n=0}^{\infty}\frac{\beta^n}{n!}\sum_{\alpha_0}\langle\alpha_0|(-H)^n|\alpha_0\rangle$$

其中Σ_{α_0}是对一组单位正交基的求和。低温情况下，假设系统基态能量为E_0，则配分函数中第 n 项的贡献约为$(-\beta E_0)^n/n!$，因此高阶项对配分函数的贡献可忽略不计。我们设定一个截止阶数 M 使得$\beta|E_0|/M \lesssim 1$（即$M \gtrsim \beta|E_0|$），

$$Z = \sum_{n=0}^{M}\frac{\beta^n}{n!}\sum_{\alpha_0}\langle\alpha_0|(-H)^n|\alpha_0\rangle = \sum_{n=0}^{M}\frac{\beta^n}{n!}\sum_{\{\alpha\}_n}\langle\alpha_0|-H|\alpha_{n-1}\rangle\ldots\langle\alpha_2|-H|\alpha_1\rangle\langle\alpha_1|-H|\alpha_0\rangle$$

其中$\Sigma_{\{\alpha\}_{n-1}}$是对从$\alpha_0$到$\alpha_{n-1}$的$n$组基的求和。在配分函数第 n 阶的展开中，乘积中共有 n 个矩阵元$\langle\alpha_n|-H|\alpha_{n-1}\rangle$。为保持不同阶展开之后乘积中矩阵元个数一致，可以引入单位矩阵 I 进行占位，例如

$$\sum_{\{\alpha\}_2}\langle\alpha_0|-H|\alpha_1\rangle\langle\alpha_1|-H|\alpha_0\rangle = \sum_{\{\alpha\}_3}\langle\alpha_0|I|\alpha_2\rangle\langle\alpha_2|-H|\alpha_1\rangle\langle\alpha_1|-H|\alpha_0\rangle$$

$$= \frac{1}{C_3^1}\sum_{\{S\}_3}\sum_{\{\alpha\}_3}\langle\alpha_0|S_3|\alpha_2\rangle\langle\alpha_2|S_2|\alpha_1\rangle\langle\alpha_1|S_1|\alpha_0\rangle$$

其中$S_n = I$或$-H$，$\Sigma_{\{S\}_3}$是对$S_n = I$或$-H$的情况求和。出现C_3^1是为了除掉不同的 I 和$-H$的组合产生的系数。将$n = 0,1,\ldots,M-1$阶的展开系数按上述方式统一改写为 M 个矩阵元的乘积，配分函数可写为

$$Z = \sum_{\{S\}_M}\frac{\beta^n(M-n)!}{M!}\sum_{\{\alpha\}_M}\langle\alpha_0|S_M|\alpha_{M-1}\rangle\ldots\langle\alpha_2|S_2|\alpha_1\rangle\langle\alpha_1|S_1|\alpha_0\rangle$$

该式中 n 的涵义变为 M 个 S 矩阵中$-H$的个数。将$-H$拆解为多个矩阵H_a的和，使得在给定的基底下，对于任意一个右矢$|\alpha\rangle$，仅有一个左矢$\langle\beta|$使得矩阵元$\langle\beta|H_a|\alpha\rangle \neq 0$，即

$$-H = \sum_a H_a \qquad \sum_{\langle\alpha'|}\langle\alpha'|H_a|\alpha\rangle = \langle\beta|H_a|\alpha\rangle$$

一种常见的满足该条件的H_a为对角矩阵，仅当$\langle\beta| = \langle\alpha|$时才有$\langle\beta|H_a|\alpha\rangle \neq 0$。现在配分

函数变为

$$Z=\sum_{\{S\}_M}\frac{\beta^n(M-n)!}{M!}\sum_{\{\alpha\}_M}\langle\alpha_0|S_M|\alpha_{M-1}\rangle\dots\langle\alpha_2|S_2|\alpha_1\rangle\langle\alpha_1|S_1|\alpha_0\rangle \qquad S_m=\{I,\sum H_a\}$$

$$=\sum_{\{S\}_M}\frac{\beta^n(M-n)!}{M!}\sum_{\alpha_0}\langle\alpha_0|S_M|\alpha_{M-1}\rangle\dots\langle\alpha_2|S_2|\alpha_1\rangle\langle\alpha_1|S_1|\alpha_0\rangle \qquad S_m=\{I,H_1,H_2,\dots\}$$

在最后一步中，我们将矩阵元的乘积$\prod_m\langle\alpha_m|\sum_a H_a|\alpha_{m-1}\rangle$展开为$\sum_{\{H_a\}}\prod_m\langle\alpha_m|H_a|\alpha_{m-1}\rangle$，这样$S_m$的涵义就相应的改变了。展开之后，由于每个右矢只对应一个使矩阵元非零的左矢，因此α_1到α_{M-1}的所有求和项中最多只有一项不为零，对$\{\alpha\}_M$的求和可以缩减为只对α_0求和，而其他的α_m视为自动满足使矩阵元不为零的要求。

这一配分函数 Z 改写后的形式说明在模拟中对所有的算符$\{S\}_M$和初始态α_0按照权重$\beta^n(M-n)!/M!$进行抽样即可得到我们所需的系统行为。需要注意的是这里所有的矩阵元都必须大于 0，使得配分函数 Z 中没有负数项。

当可观测量$\hat{O}$是对角矩阵时，$\hat{O}|\alpha_0\rangle=\langle\alpha_0|\hat{O}|\alpha_0\rangle|\alpha_0\rangle\equiv O(\alpha_0)|\alpha_0\rangle$，其期望值为

$$\langle\hat{O}\rangle=\frac{1}{Z}\mathrm{tr}\left(e^{-\beta H}\hat{O}\right)=\frac{1}{Z}\sum_{\alpha_0}\langle\alpha_0|e^{-\beta H}\hat{O}|\alpha_0\rangle=\frac{1}{Z}\sum_{\alpha_0}O(\alpha_0)\langle\alpha_0|e^{-\beta H}|\alpha_0\rangle$$

$$=\frac{1}{Z}\sum_{\{S\}_M}\frac{\beta^n(M-n)!}{M!}\sum_{\alpha_0}O(\alpha_0)\langle\alpha_0|S_M|\alpha_{M-1}\rangle\dots\langle\alpha_2|S_2|\alpha_1\rangle\langle\alpha_1|S_1|\alpha_0\rangle$$

$$=\langle O(\alpha_0)\rangle$$

因此我们可以直接测量$|\alpha_0\rangle$态下$\hat{O}$的期望值。当$\hat{O}$为非对角矩阵时，其期望值需要另行推导。一个常见的非对角可观测量为哈密顿量$\hat{H}$，它的期望值为

$$E=\langle H\rangle=\frac{1}{Z}\sum_{\alpha_0}\langle\alpha_0|e^{-\beta H}H|\alpha_0\rangle=-\frac{1}{Z}\sum_{n=0}^{\infty}\frac{\beta^n}{n!}\sum_{\alpha_0}\langle\alpha_0|(-H)^{n+1}|\alpha_0\rangle$$

$$=-\frac{1}{Z}\sum_{n=1}^{\infty}\frac{\beta^n}{n!}\frac{n}{\beta}\sum_{\alpha_0}\langle\alpha_0|(-H)^n|\alpha_0\rangle=-\frac{1}{Z}\sum_{n=0}^{\infty}\frac{\beta^n}{n!}\frac{n}{\beta}\sum_{\alpha_0}\langle\alpha_0|(-H)^n|\alpha_0\rangle$$

$$=-\frac{1}{Z}\sum_{\{S\}_M}\frac{\beta^n(M-n)!}{M!}\frac{n}{\beta}\sum_{\alpha_0}\langle\alpha_0|S_M|\alpha_{M-1}\rangle\dots\langle\alpha_2|S_2|\alpha_1\rangle\langle\alpha_1|S_1|\alpha_0\rangle=-\frac{\langle n\rangle}{\beta}$$

其中 n 与之前的定义一致，是不为单位矩阵 I 的算符个数。

在模拟过程中，看似需要取$M\to\infty$和$\beta\to\infty$两步极限，但实际上对基态温度E_0作估计后选取一个固定的$M\gtrsim\beta|E_0|$即可，因此只需要取$\beta\to\infty$一步极限。

附录 B　随机级数展开算法在海森堡模型中的实现

随机级数展开算法在海森堡模型中最易实现，所以我们以此为例来说明使用该算法的具体步骤。一维反铁磁自旋-1/2海森堡模型的哈密顿量为

$$H = J\sum_{i=1}^{N}\left(S_i^z S_{i+1}^z + S_i^x S_{i+1}^x + S_i^y S_{i+1}^y\right) = -\sum_{i=1}^{N}\left[\left(\frac{1}{4} - S_i^z S_{i+1}^z\right) - \frac{1}{2}\left(S_i^+ S_{i+1}^- + S_i^- S_{i+1}^+\right)\right] + \frac{N}{4}$$

其中 N 为系统的自旋总数，耦合强度$J \equiv 1$，上升算符S_i^+与下降算符S_i^-的定义为$S_i^{\pm} = S_i^x \pm iS_i^y$。提出常数项$N/4$是为了使$1/4 - S_i^z S_{i+1}^z$矩阵元的值为正。

取S^z的本征态为基，这样就可以定义 N 个对角算符

$$H_{1,i} = \frac{1}{4} - S_i^z S_{i+1}^z \qquad\qquad i = 1,2,\dots,N$$

以及 N 个非对角算符

$$H_{2,i} = -\frac{1}{2}\left(S_i^+ S_{i+1}^- + S_i^- S_{i+1}^+\right) \qquad i = 1,2,\dots,N$$

使得在忽略哈密顿量常数项的前提下，有

$$H = -\sum_{i=1}^{N}\left(H_{1,i} + H_{2,i}\right) = -\sum_{a=1}^{2N} H_a$$

$$S = \left\{I, H_{1,1}, H_{1,2}, \dots, H_{1,N}, H_{2,1}, H_{2,2}, \dots, H_{2,N}\right\}$$

算符的所有非零矩阵元为

$$\left\langle \uparrow_i\downarrow_{i+1} \left| H_{1,i} \right| \uparrow_i\downarrow_{i+1} \right\rangle = \frac{1}{2} \qquad \left\langle \downarrow_i\uparrow_{i+1} \left| H_{1,i} \right| \downarrow_i\uparrow_{i+1} \right\rangle = \frac{1}{2}$$

$$\left\langle \uparrow_i\downarrow_{i+1} \left| H_{2,i} \right| \downarrow_i\uparrow_{i+1} \right\rangle = -\frac{1}{2} \qquad \left\langle \downarrow_i\uparrow_{i+1} \left| H_{2,i} \right| \uparrow_i\downarrow_{i+1} \right\rangle = -\frac{1}{2}$$

$H_{2,i}$的矩阵元看似为负数不满足配分函数每一项都为正数的要求，但实际上非对角元的总个数必须为偶数，矩阵元中负号相消，因此配分函数 Z 的每一项仍为正数。

以这些算符为基础，该算法共有三个步骤：

1. 更新对角算符。I 与对角算符$H_{1,i}$按权重更新，改变算符总个数。

2. 更新非对角算符。对角算符$H_{1,i}$与非对角算符$H_{2,i}$按权重更新。这一步使用了循环更新（loop update）算法。

3. 更新初始态$|\alpha_0\rangle$

具体的步骤流程在此不赘述，可参见文献[4,5]。

附录 C　随机级数展开算法在 Bose-Hubbard 模型中的实现

一维 Bose-Hubbard 模型的哈密顿量为

$$H = -\omega\sum_{i=1}^{N}\left(b_i^\dagger b_{i+1} + b_i b_{i+1}^\dagger\right) - \mu\sum_{i=1}^{N} n_i + \frac{U}{2}\sum_{i=1}^{N} n_i(n_i-1)$$

$$= -\sum_{i=1}^{N}\left\{\frac{\mu}{2}(n_i + n_{i+1}) + \frac{U}{4}[4 - n_i(n_i-1) - n_{i+1}(n_{i+1}-1)] + b_i^\dagger b_{i+1} + b_i b_{i+1}^\dagger\right\} + NU$$

其中$\omega \equiv 1$。已假设U较大使得每个格点的填充数最多为 2。

选取每个格点的粒子填充数作为基矢，由此定义 N 个对角算符

$$H_{1,i} = \frac{\mu}{2}(n_i + n_{i+1}) + \frac{U}{4}[4 - n_i(n_i-1) - n_{i+1}(n_{i+1}-1)] \quad i = 1,2,\dots,N \quad n_i = 0,1,2$$

和$2N$个非对角算符

$$H_{2,i} = b_i^\dagger b_{i+1} \qquad H_{3,i} = b_i b_{i+1}^\dagger \qquad i = 1,2,\dots,N \quad n_i = 0,1,2$$

使得在忽略哈密顿量常数项的前提下，有

$$H = -\sum_{i=1}^{N}\left(H_{1,i} + H_{2,i} + H_{3,i}\right) = -\sum_{a=1}^{3N} H_a$$

$$S = \left\{I, H_{1,1}, H_{1,2}, \dots, H_{1,N}, H_{2,1}, H_{2,2}, \dots, H_{2,N}, H_{3,1}, H_{3,2}, \dots, H_{3,N}\right\}$$

算符的矩阵元数量较多，在此不赘述。在本模型中算法的实现步骤与在海森堡模型中类似，共有三步：

1. 更新对角算符。
2. 用循环更新算法更新非对角算符。
3. 更新初始态$|\alpha_0\rangle$

对角算符的更新方法与参考文献[4,5]中提到的类似。而非对角算符和初始态的更新可以参见文献[6]的附录 A。该文献中描述的方法是针对 Fermi-Hubbard 模型的，但对 Bose-Hubbard 模型也同样适用。

附录 D　海森堡模型的随机级数展开算法代码

```
//---------------------------------------------------------------------------
// Compiler: Microsoft Visual Studio 2017
// Programmer: CHEN Ao
// File name: SSE_Heisenberg.h
//---------------------------------------------------------------------------

#pragma once

#include<iostream>
#include<fstream>
#include<vector>
#include<set>
#include<math.h>
#include<stdlib.h>
#include<time.h>
using namespace std;

#define BOTTOM false
#define TOP true

#define BOTTOM_LEFT 0
#define BOTTOM_RIGHT 1
#define TOP_LEFT 2
#define TOP_RIGHT 3

#define IDENTITY 0
#define DIAGONAL 1
#define OFF_DIAGONAL 2

class Leg;
class Operator;
class Lattice;

class Leg {
private:
    bool type;                              // TOP or BOTTOM
    bool Is_in_cluster;                 // indicate whether this leg has been put
into a cluster
    int spin_position;              // position in x direction
    int time_position;              // position in imaginary t direction
    Operator* attaching_operator; // the operator this leg belongs to
    Leg* neighbor_leg;              // the leg next to this leg in the same
operator
    Leg* linked_leg;                    // the leg linked with this leg

public:
    Leg();
    bool get_type() const;
    int get_time_position() const;
    Operator* get_operator() const;
    void add_to_cluster(vector<bool>& spin);
    void clear_cluster();
    void initialize_time_position(int Time_Position, bool Type, Operator* op,
Leg* Neighbor);
    void change_spin_position(int Spin_Position);
    void change_linked_leg(Leg* Linked_Leg);
    friend bool operator<(const Leg& v1, const Leg& v2);

};

class Operator {
```

```
private:
    int type;                               // identity, diagonal or off-diagonal
    int flipped_times;            // the times that this operator has been
flipped
    int spin_position;            // position in x direction ( left one )
    int time_position;            // position in imaginary t direction ( bottom
one )
    vector<Leg> leg;              // 4 verteces of every operator

public:
    Operator();
    void initialize_time_position(int Time_Position);
    int get_type() const;
    int get_spin_position() const;
    int get_time_position() const;
    void add_flipped_times();
    void flip();
    void create_diagonal_operator(int Spin_Position, int N);
    void delete_operator();
    Leg* get_leg(int place);
};

class Lattice {
private:
    double beta;          // 1/kT
    int N;                        // total number of spins
    int M;                        // total number of time slices
    int n;                        // total number of operators
    vector<bool> spin; // spin in the initial state alpha_0
    vector<Operator> operators;       // operators of every row
    struct compare { bool operator()(const Leg* v1, const Leg* v2) const; };
    vector<set<Leg*, compare>> leg_in_colomn;       // vertex of every colomn

    void link_leg(set<Leg*, compare>::iterator& leg_iter, set<Leg*, compare>&
colomn);
    void create_diagonal_operator(int Spin_Position, int Time_Position);
    void delete_diagonal_operator(int Time_Position);
    void diagonal_update();
    void operator_loop_update();

public:
    Lattice(int n, int m);
    void change_beta(double Beta);
    void step();
    double energy();
};
```

```
//-------------------------------------------------------------------------
// Compiler: Microsoft Visual Studio 2017
// Programmer: CHEN Ao
// File name: lattice.cpp
//-------------------------------------------------------------------------

#include"SSE_Heisenberg.h"

Lattice::Lattice(int n, int m) {
    N = n;
    M = m;
    leg_in_colomn.resize(N);

    operators.resize(M);
    for (int i = 0; i < M; i++) {
        operators[i].initialize_time_position(i);
    }

    spin.resize(N);
    for (auto iter = spin.begin(); iter != spin.end(); iter++)
        *iter = rand() % 2;
}

void Lattice::change_beta(double Beta) {
    beta = Beta;
}

bool Lattice::compare::operator()(const Leg* v1, const Leg* v2) const {
    return *v1 < *v2;
}

void Lattice::create_diagonal_operator(int Spin_Position, int Time_Position) {
    n++;
    Operator* op = &operators[Time_Position];
    set<Leg*, compare>* left_column = &leg_in_colomn[Spin_Position];
    set<Leg*, compare>* right_column = &leg_in_colomn[(Spin_Position + 1) % N];

    op->create_diagonal_operator(Spin_Position, N);
    auto iter_left_bottom =
left_column->insert(op->get_leg(BOTTOM_LEFT)).first;
    auto iter_left_top = left_column->insert(iter_left_bottom,
op->get_leg(TOP_LEFT));
    auto iter_right_bottom =
right_column->insert(op->get_leg(BOTTOM_RIGHT)).first;
    auto iter_right_top = right_column->insert(iter_right_bottom,
op->get_leg(TOP_RIGHT));

    link_leg(iter_left_bottom, *left_column);
    link_leg(iter_left_top, *left_column);
    link_leg(iter_right_bottom, *right_column);
    link_leg(iter_right_top, *right_column);
}

void Lattice::delete_diagonal_operator(int Time_Position) {
    n--;
    Operator* op = &operators[Time_Position];
    set<Leg*, compare>* left_column = &leg_in_colomn[op->get_spin_position()];
    set<Leg*, compare>* right_column = &leg_in_colomn[(op->get_spin_position()
+ 1) % N];

    auto iter_left = left_column->find(op->get_leg(BOTTOM_LEFT));
    iter_left = left_column->erase(iter_left);
    iter_left = left_column->erase(iter_left);
    if (!left_column->empty()) {
        if (iter_left == left_column->end())
```

```
            iter_left = left_column->begin();
        link_leg(iter_left, *left_column);
    }

    auto iter_right = right_column->find(op->get_leg(BOTTOM_RIGHT));
    iter_right = right_column->erase(iter_right);
    iter_right = right_column->erase(iter_right);
    if (!right_column->empty()) {
        if (iter_right == right_column->end())
            iter_right = right_column->begin();
        link_leg(iter_right, *right_column);
    }

    op->delete_operator();
}

void Lattice::link_leg(set<Leg*, compare>::iterator& leg_iter, set<Leg*,
compare>& colomn) {
    if ((*leg_iter)->get_type() == TOP) {
        if (leg_iter == (--colomn.end()))
            (*leg_iter)->change_linked_leg(*colomn.begin());
        else {
            auto copy = leg_iter;
            copy++;
            (*leg_iter)->change_linked_leg(*copy);
        }
    }
    else {
        if (leg_iter == colomn.begin())
            (*leg_iter)->change_linked_leg(*(--colomn.end()));
        else {
            auto copy = leg_iter;
            copy--;
            (*leg_iter)->change_linked_leg(*copy);
        }
    }
}

void Lattice::diagonal_update() {
    vector<bool> bond(N);
    vector<int> bond_position(N);

    for (int i = 0; i < N; i++) {
        bond[i] = (spin[i] != spin[(i + 1) % N]);
    }
    for (auto op_iter = operators.begin(); op_iter !=
operators.end();op_iter++) {
        int type = op_iter->get_type();
        if (type == OFF_DIAGONAL) {
            bond[(op_iter->get_spin_position() - 1 + N) % N]
= !bond[(op_iter->get_spin_position() - 1 + N) % N];
            bond[(op_iter->get_spin_position() + 1) % N]
= !bond[(op_iter->get_spin_position() + 1) % N];
        }
        else {
            int Nb = 0;
            for (int position = 0; position < N; position++) {
                if (bond[position]) {
                    bond_position[Nb] = position;
                    Nb++;
                }
            }

            if (type == IDENTITY) {
                if ((double)(2 * (M - n)*rand()) / (Nb*RAND_MAX) < beta)       //
```

```
add (double) here when beta is small

    create_diagonal_operator(bond_position[rand()%Nb],op_iter->get_time_positio
n());
            }
            else {
                if (Nb*rand()*beta < 2 * RAND_MAX*(M - n + 1))
                    delete_diagonal_operator(op_iter->get_time_position());
            }
        }
    }
}

void Lattice::operator_loop_update() {
    for (auto column_iter = leg_in_colomn.begin(); column_iter !=
leg_in_colomn.end(); column_iter++) {
        for (auto leg_iter = column_iter->begin(); leg_iter !=
column_iter->end(); leg_iter++) {
            (*leg_iter)->add_to_cluster(spin);
        }
    }

    for (auto op_iter = operators.begin(); op_iter != operators.end();
op_iter++) {
        op_iter->flip();
    }

    for (int i = 0; i < N; i++) {
        if (leg_in_colomn[i].empty() && rand() % 2)
            spin[i] = !spin[i];
    }
}

void Lattice::step() {
    diagonal_update();
    operator_loop_update();
}

double Lattice::energy() {
    return -n / beta / N + 0.25;
}
```

```cpp
//-------------------------------------------------------------------------
// Compiler: Microsoft Visual Studio 2017
// Programmer: CHEN Ao
// File name: operator.cpp
//-------------------------------------------------------------------------

#include"SSE_Heisenberg.h"
Operator::Operator() {
    type = IDENTITY;
    flipped_times = 0;
    leg.resize(4);
}
void Operator::initialize_time_position(int Time_Position) {
    time_position = Time_Position;
    leg[BOTTOM_LEFT].initialize_time_position(Time_Position, BOTTOM, this,
&leg[BOTTOM_RIGHT]);
    leg[BOTTOM_RIGHT].initialize_time_position(Time_Position, BOTTOM, this,
&leg[BOTTOM_LEFT]);
    leg[TOP_LEFT].initialize_time_position(Time_Position + 1, TOP, this,
&leg[TOP_RIGHT]);
    leg[TOP_RIGHT].initialize_time_position(Time_Position + 1, TOP, this,
&leg[TOP_LEFT]);
}
int Operator::get_type() const {
    return type;
}
int Operator::get_spin_position() const {
    return spin_position;
}
int Operator::get_time_position() const {
    return time_position;
}
void Operator::add_flipped_times() {
    flipped_times++;
}
void Operator::flip() {
    if (flipped_times % 2) {
        if (type == DIAGONAL)
            type = OFF_DIAGONAL;
        else {
            type = DIAGONAL;
        }
    }
    flipped_times = 0;
    for (auto leg_iter = leg.begin(); leg_iter != leg.end(); leg_iter++) {
        leg_iter->clear_cluster();
    }
}
void Operator::create_diagonal_operator(int Spin_Position, int N) {
    type = DIAGONAL;
    spin_position = Spin_Position;
    leg[BOTTOM_LEFT].change_spin_position(Spin_Position);
    leg[TOP_LEFT].change_spin_position(Spin_Position);
    leg[BOTTOM_RIGHT].change_spin_position((Spin_Position + 1)%N);
    leg[TOP_RIGHT].change_spin_position((Spin_Position + 1)%N);
}

void Operator::delete_operator() {
    type = IDENTITY;
}

Leg* Operator::get_leg(int place) {
    return &(leg[place]);
}
```

```cpp
//---------------------------------------------------------------------------
// Compiler: Microsoft Visual Studio 2017
// Programmer: CHEN Ao
// File name: leg.cpp
//---------------------------------------------------------------------------
#include"SSE_Heisenberg.h"
Leg::Leg() {
    Is_in_cluster = false;
}
bool Leg::get_type() const {
    return type;
}
int Leg::get_time_position() const {
    return time_position;
}
Operator* Leg::get_operator() const {
    return attaching_operator;
}
void Leg::add_to_cluster(vector<bool>& spin) {
    if (!Is_in_cluster) {
        bool flip_judgement = (bool)(rand() % 2);
        Leg* current_leg = this;
        do {
            current_leg->Is_in_cluster = true;
            current_leg = current_leg->neighbor_leg;
            current_leg->Is_in_cluster = true;
            if (flip_judgement) {
                current_leg->attaching_operator->add_flipped_times();
                if ((current_leg->type == TOP && current_leg->time_position >
current_leg->linked_leg->time_position)
                    || (current_leg->type == BOTTOM && current_leg->time_position
< current_leg->linked_leg->time_position))
                    spin[current_leg->spin_position]
= !spin[current_leg->spin_position];
            }
            current_leg = current_leg->linked_leg;
        } while (!current_leg->Is_in_cluster);
    }
}
void Leg::clear_cluster() {
    Is_in_cluster = false;
}
void Leg::initialize_time_position(int Time_Position, bool Type, Operator* op,
Leg* Neighbor) {
    time_position = Time_Position;
    type = Type;
    attaching_operator = op;
    neighbor_leg = Neighbor;
}
void Leg::change_spin_position(int Spin_Position) {
    spin_position = Spin_Position;
}
void Leg::change_linked_leg(Leg* Linked_Vertex) {
    linked_leg = Linked_Vertex;
    Linked_Vertex->linked_leg = this;
}

bool operator<(const Leg& v1, const Leg& v2) {
    if (v1.time_position < v2.time_position)
        return true;
    else if (v1.time_position == v2.time_position && v1.type == TOP)
        return true;
    else
        return false;
}
```

```cpp
//---------------------------------------------------------------------------
// Compiler: Microsoft Visual Studio 2017
// Programmer: CHEN Ao
// File name: driver.cpp
//---------------------------------------------------------------------------

#include"SSE_Heisenberg.h"

int main() {
    srand((unsigned int)time(NULL));
    ofstream energy_file("test.txt");


    for (double beta = 1; beta < 200; beta *= 2) {
        double energy = 0;

        for (int test = 0; test < 10; test++) {
            Lattice lattice(100, 10000);
            lattice.change_beta(beta);

            for (int t = 0; t < 100; t++)
                lattice.step();
            for (int t = 0; t < 1000; t++) {
                lattice.step();
                energy += lattice.energy();
            }

        }

        cout<< beta << '\t' << energy / (1000 * 10) << '\n';
        energy_file << beta << '\t' << energy / (1000 * 10) << '\n';
    }

    energy_file.close();

    return 0;
}
```

附录 E　Bose-Hubbard 模型的随机级数展开算法代码

```
//-------------------------------------------------------------------------
// Compiler: Microsoft Visual Studio 2017
// Programmer: CHEN Ao
// File name: SSE_Hubbard.h
//-------------------------------------------------------------------------

#pragma once

#include<iostream>
#include<fstream>
#include<vector>
#include<set>
#include<math.h>
#include<stdlib.h>
#include<time.h>
using namespace std;

#define BOTTOM_LEFT 0
#define BOTTOM_RIGHT 1
#define TOP_LEFT 2
#define TOP_RIGHT 3

#define IDENTITY 0
#define DIAGONAL 1
#define LEFT 2
#define RIGHT 3

#define SQRT_2 1.41421356237

extern double beta, U, mu;
extern int probability_table[3][3][3][3][3];
extern int site_occupation_probability[2];

class Leg;         // leg is the verteces of an operator, every operator has
four legs
class Operator;        // operator ( identity, diagonal and off-diagonal )
class Lattice;     // all information in the simulation

class Leg {
private:
    int type;                  // type (position) of the leg
    int occupation_number;         // the number of bosons in this leg
    int new_occupation_number;         // used to store new value of occupation
number before the update applies
    bool Is_starting_point;        // mark whether this leg is the starting point
of the loop update
    int spin_position;         // position in x direction
    int time_position;         // position in imaginary t direction
    Operator* attaching_operator;
    Leg* linked_leg;
public:
    Leg();
    int get_type() const;
    int get_time_position() const;
    int get_spin_position() const;
    Operator* get_operator() const;
    int get_occupation_number() const;
    Leg* get_linked_leg() const;
    void change_starting_status(bool new_status);
    bool Not_starting_point() const;
    void change_occupation_number(int n);
    void change_occupation_number(const Leg& leg);
```

```
    void random_new_occupation();
    void change_new_occupation(int n);
    void update_new_occupation();
    void remove_new_occupation();
    void initialize(int Time_Position, Operator* op);
    void change_spin_position(int Spin_Position);
    void change_linked_leg(Leg* Linked_Leg);
    friend bool operator<(const Leg& v1, const Leg& v2);
};
class Operator {
private:
    int type;              // identity, diagonal, left or right
    bool Is_changed;           // indicate whether this operator is changed
during loop update
    int spin_position;     // position in x direction ( left one )
    int time_position;     // position in imaginary t direction ( bottom one )
    vector<Leg> leg;           // 4 verteces of every operator
public:
    Operator();
    void initialize_time_position(int Time_Position);
    int get_type() const;
    int get_spin_position() const;
    int get_time_position() const;
    bool get_update_status() const;
    void create_diagonal_operator(int Spin_Position, int N);
    void delete_operator();
    void change_update_status(bool new_status);
    void type_update();
    Leg* loop_update(Leg* p_leg);
    Leg* get_leg(int place);
};
class Lattice {
private:
    int N;                     // total number of sites
    int M;                     // total number of time slices
    int n;                     // total number of operators
    vector<int> site_occupation;  // occupation number of the sites of alpha_0
    vector<Operator> operators;   // operators of every row
    struct compare_leg { bool operator()(const Leg* v1, const Leg* v2)
const; };
    vector<set<Leg*, compare_leg>> leg_in_colomn;   // vertex of every colomn
    void link_leg(set<Leg*, compare_leg>::iterator& leg_iter, set<Leg*,
compare_leg>& colomn);
    void create_diagonal_operator(vector<Operator>::iterator& op, int
Spin_Position);
    void delete_diagonal_operator(vector<Operator>::iterator& op);
    void diagonal_update();
    void operator_loop_update();
    void site_occupation_update();
public:
    Lattice(int n, int m);
    void step();
    double energy() const;
    double rho() const;
    int get_n() const;
    void output_configuration(ofstream& file) const;           // output the
occupation number of every site
    friend ostream& operator<<(ostream& os, Lattice& lattice);// output all
operators and legs
};
void parameter_renew(double Beta, double interaction, double
chemical_potential);
double matrix_element(int n0, int n1, int operator_type);
double matrix_element(const vector<Operator>::iterator& op);
double matrix_element(int n0, int n1, int n2, int n3);
```

```cpp
//-------------------------------------------------------------------------
// Compiler: Microsoft Visual Studio 2017
// Programmer: CHEN Ao
// File name: constants.cpp
//-------------------------------------------------------------------------

#include"SSE_Hubbard.h"

double beta;           // 1/kT
double U;                 // interaction in bosons
double mu;             // chemical potential

double e1, e2, e3, e4, e5;

int probability_table[3][3][3][3][3] = { 0 };
int site_occupation_probability[2];

void parameter_renew(double Beta, double interaction, double
chemical_potential) {
    beta = Beta;
    U = interaction;
    mu = chemical_potential;
    e1 = mu / 2 + U;
    e2 = mu + U / 2;
    e3 = mu + U;
    e4 = mu * 3 / 2 + U / 2;
    e5 = 2 * mu;

    for (int n0 = 0; n0 < 3; n0++)
        for (int n1 = 0; n1 < 3; n1++)
            for (int n2 = 0; n2 < 3; n2++)
                for (int n3 = 0; n3 < 3; n3++) {
                    if (n0 + n1 == n2 + n3)
                        continue;
                    double weight[4];
                    weight[0] = matrix_element(n2 + n3 - n1, n1, n2, n3);
                    weight[1] = matrix_element(n0, n2 + n3 - n0, n2, n3);
                    weight[2] = matrix_element(n0, n1, n0 + n1 - n3, n3);
                    weight[3] = matrix_element(n0, n1, n2, n0 + n1 - n2);
                    double total_weight = weight[0] + weight[1] + weight[2] +
weight[3];
                    double current_weight = 0.0;
                    for (int new_leg = 0; new_leg < 3; new_leg++) {
                        current_weight += weight[new_leg];
                        probability_table[n0][n1][n2][n3][new_leg] =
(int)(current_weight / total_weight*(RAND_MAX + 1) + 0.5);
                    }
                }
    double site_occupation_weight[3] = { 1.0, exp(beta*mu), exp(2 * beta*mu) };
    double total_weight = site_occupation_weight[0] + site_occupation_weight[1]
+ site_occupation_weight[2];
    site_occupation_probability[0] = (int)(site_occupation_weight[0] /
total_weight*(RAND_MAX + 1) + 0.5);
    site_occupation_probability[1] = (int)((site_occupation_weight[0] +
site_occupation_weight[1]) / total_weight*(RAND_MAX + 1) + 0.5);

}

double matrix_element(int n0, int n1, int operator_type) {
    if (n0 < 0 || n0>2 || n1 < 0 || n1>2)
        return 0.0;
    switch (operator_type) {
    case DIAGONAL:
        switch (n0 + n1) {
        case 0:
```

```
            return U;
        case 1:
            return e1;
        case 2:
            if (n0 == 1)
                return e3;
            else
                return e2;
        case 3:
            return e4;
        case 4:
            return e5;
        }
    case LEFT:
        if (n0 <= n1) {
            switch (n0 + n1) {
            case 1:
                return 1.0;
            case 2:
                return SQRT_2;
            case 3:
                return 2.0;
            }
        }
        break;
    case RIGHT:
        if (n0 >= n1) {
            switch (n0 + n1) {
            case 1:
                return 1.0;
            case 2:
                return SQRT_2;
            case 3:
                return 2.0;
            }
        }
        break;
    }
    return 0.0;
}

double matrix_element(const vector<Operator>::iterator& op) {
    return matrix_element(op->get_leg(BOTTOM_LEFT)->get_occupation_number(),
        op->get_leg(BOTTOM_RIGHT)->get_occupation_number(), op->get_type());
}

double matrix_element(int n0, int n1, int n2, int n3) {
    if (n0 + n1 != n2 + n3)
        return 0.0;
    if (n2 < 0 || n2>2 || n3 < 0 || n3>2)
        return 0.0;
    switch (n0 - n2) {
    case 0:
        return matrix_element(n0, n1, DIAGONAL);
    case -1:
        return matrix_element(n0, n1, LEFT);
    case 1:
        return matrix_element(n0, n1, RIGHT);
    default:
        return 0.0;
    }
}
```

```
//-------------------------------------------------------------------------
// Compiler: Microsoft Visual Studio 2017
// Programmer: CHEN Ao
// File name: lattice.cpp
//-------------------------------------------------------------------------

#include"SSE_Hubbard.h"

Lattice::Lattice(int n, int m) {
    N = n;
    M = m;
    leg_in_colomn.resize(N);

    operators.resize(M);
    for (int i = 0; i < M; i++) {
        operators[i].initialize_time_position(i);
    }

    site_occupation.resize(N);
    for (auto iter = site_occupation.begin(); iter != site_occupation.end();
iter++)
        *iter = rand() % 3;
}

bool Lattice::compare_leg::operator()(const Leg* v1, const Leg* v2) const {
    return *v1 < *v2;
}

void Lattice::create_diagonal_operator(vector<Operator>::iterator& op, int
Spin_Position) {
    n++;
    set<Leg*, compare_leg>* left_column = &leg_in_colomn[Spin_Position];
    set<Leg*, compare_leg>* right_column = &leg_in_colomn[(Spin_Position + 1) %
N];

    op->create_diagonal_operator(Spin_Position, N);
    auto iter_left_bottom =
left_column->insert(op->get_leg(BOTTOM_LEFT)).first;
    auto iter_left_top = left_column->insert(iter_left_bottom,
op->get_leg(TOP_LEFT));
    auto iter_right_bottom =
right_column->insert(op->get_leg(BOTTOM_RIGHT)).first;
    auto iter_right_top = right_column->insert(iter_right_bottom,
op->get_leg(TOP_RIGHT));

    link_leg(iter_left_bottom, *left_column);
    link_leg(iter_left_top, *left_column);
    link_leg(iter_right_bottom, *right_column);
    link_leg(iter_right_top, *right_column);
}

void Lattice::delete_diagonal_operator(vector<Operator>::iterator& op) {
    n--;
    set<Leg*, compare_leg>* left_column =
&leg_in_colomn[op->get_spin_position()];
    set<Leg*, compare_leg>* right_column =
&leg_in_colomn[(op->get_spin_position() + 1) % N];

    auto iter_left = left_column->find(op->get_leg(BOTTOM_LEFT));
    iter_left = left_column->erase(iter_left);
    iter_left = left_column->erase(iter_left);
    if (!left_column->empty()) {
        if (iter_left == left_column->end())
            iter_left = left_column->begin();
        link_leg(iter_left, *left_column);
```

```
    }

    auto iter_right = right_column->find(op->get_leg(BOTTOM_RIGHT));
    iter_right = right_column->erase(iter_right);
    iter_right = right_column->erase(iter_right);
    if (!right_column->empty()) {
        if (iter_right == right_column->end())
            iter_right = right_column->begin();
        link_leg(iter_right, *right_column);
    }

    op->delete_operator();
}

void Lattice::link_leg(set<Leg*, compare_leg>::iterator& leg_iter, set<Leg*,
compare_leg>& colomn) {
    if ((*leg_iter)->get_type() == TOP_LEFT || (*leg_iter)->get_type() ==
TOP_RIGHT) {
        if (leg_iter == (--colomn.end())) {
            (*leg_iter)->change_linked_leg(*colomn.begin());

    (*leg_iter)->change_occupation_number(site_occupation[(*leg_iter)->get_spin
_position()]);
        }
        else {
            auto copy = leg_iter;
            copy++;
            (*leg_iter)->change_linked_leg(*copy);
            if (colomn.size() == 2)

    (*leg_iter)->change_occupation_number(site_occupation[(*leg_iter)->get_spin
_position()]);
            else
                (*leg_iter)->change_occupation_number(**copy);
        }
    }
    else {
        if (leg_iter == colomn.begin()) {
            (*leg_iter)->change_linked_leg(*(--colomn.end()));

    (*leg_iter)->change_occupation_number(site_occupation[(*leg_iter)->get_spin
_position()]);
        }
        else {
            auto copy = leg_iter;
            copy--;
            (*leg_iter)->change_linked_leg(*copy);
            if (colomn.size() == 2)

    (*leg_iter)->change_occupation_number(site_occupation[(*leg_iter)->get_spin
_position()]);
            else
                (*leg_iter)->change_occupation_number(**copy);
        }
    }
}

void Lattice::diagonal_update() {
    vector<int> Site_Occupation(site_occupation);

    for (auto op_iter = operators.begin(); op_iter != operators.end();
op_iter++) {
        int type = op_iter->get_type();
        if (type == LEFT) {
            Site_Occupation[op_iter->get_spin_position()]++;
```

```
            Site_Occupation[(op_iter->get_spin_position() + 1) % N]--;
        }
        else if (type == RIGHT) {
            Site_Occupation[op_iter->get_spin_position()]--;
            Site_Occupation[(op_iter->get_spin_position() + 1) % N]++;
        }
        else if (type == DIAGONAL) {
            if (N*rand()*matrix_element(op_iter) < (M - n + 1)*RAND_MAX / beta)
                delete_diagonal_operator(op_iter);
        }
        else
        {
            int spin_position = rand() % N;
            int n1 = Site_Occupation[spin_position], n2 =
Site_Occupation[(spin_position + 1) % N];
            if (rand()*(M - n) / matrix_element(n1, n2, DIAGONAL) < N*RAND_MAX *
beta)
                create_diagonal_operator(op_iter, spin_position);
        }
    }
}

void Lattice::operator_loop_update() {
    if (n == 0)
        return;
    for (int total_visited_operator = 0; total_visited_operator < 2 * M;) {
        int operator_number = rand() % n, op_position = 0;
        auto op_iter = operators.begin();
        while (op_iter->get_type() == IDENTITY)
            op_iter++;
        while (op_position < operator_number) {
            op_iter++;
            if (op_iter->get_type() != IDENTITY)
                op_position++;
        }
        Leg* p_leg = op_iter->get_leg(rand() % 4);
        Leg* p_starting_leg = p_leg;
        Operator* p_op = &(*op_iter);
        p_starting_leg->change_starting_status(true);
        p_starting_leg->get_linked_leg()->change_starting_status(true);
        int loop_length = 0;

        p_starting_leg->random_new_occupation();
        do {
            p_op->change_update_status(true);
            p_leg = p_op->loop_update(p_leg);
            p_op = p_leg->get_operator();
            loop_length++;
            total_visited_operator++;
        } while (p_leg->Not_starting_point() && loop_length < 50*M);
        if (p_leg->Not_starting_point()) {
            p_starting_leg->change_starting_status(false);
            p_starting_leg->get_linked_leg()->change_starting_status(false);
            for (auto iter = operators.begin(); iter != operators.end(); iter++)
{
                if (iter->get_update_status()) {
                    iter->change_update_status(false);
                    for (int leg_number = 0; leg_number < 4; leg_number++)
                        iter->get_leg(leg_number)->remove_new_occupation();
                }
            }
            return;
        }
        else {
            p_starting_leg->change_starting_status(false);
```

```
            p_starting_leg->get_linked_leg()->change_starting_status(false);
            for (auto iter = operators.begin(); iter != operators.end(); iter++)
{
                if (iter->get_update_status()) {
                    iter->change_update_status(false);
                    iter->type_update();
                    for (int leg_number = 0; leg_number < 4; leg_number++) {
                        Leg* leg = iter->get_leg(leg_number);
                        leg->update_new_occupation();
                        if ((leg->get_type() == TOP_LEFT || leg->get_type() ==
TOP_RIGHT) &&
                            leg->get_time_position() >
leg->get_linked_leg()->get_time_position())
                            site_occupation[leg->get_spin_position()] =
leg->get_occupation_number();
                    }
                }
            }
        }

    }
}

void Lattice::site_occupation_update() {
    for (int i = 0; i < N; i++) {
        if (leg_in_colomn[i].empty()) {
            int random = rand();
            if (random < site_occupation_probability[0])
                site_occupation[i] = 0;
            else if (random < site_occupation_probability[1])
                site_occupation[i] = 1;
            else
                site_occupation[i] = 2;
        }
    }
}

void Lattice::step() {
    diagonal_update();
    operator_loop_update();
    site_occupation_update();
}

double Lattice::energy() const {
    return -n / beta / N + U;
}

double Lattice::rho() const {
    int total_boson = 0;
    for (auto iter = site_occupation.begin(); iter != site_occupation.end();
iter++)
        total_boson += *iter;
    return (double)total_boson / N;
}

int Lattice::get_n() const {
    return n;
}

void Lattice::output_configuration(ofstream& file) const {
    vector<int> Site_Occupation(site_occupation);

    for (int time_position = 0; time_position < M;time_position++) {
        const Operator* op = &operators[time_position];
```

```
        for (int spin_position = 0; spin_position < N; spin_position++)
            file << Site_Occupation[spin_position] << '\t';
        file << '\n';

        int type = op->get_type();

        switch (type) {
        case LEFT:
            Site_Occupation[op->get_spin_position()]++;
            Site_Occupation[(op->get_spin_position() + 1) % N]--;
            break;
        case RIGHT:
            Site_Occupation[op->get_spin_position()]--;
            Site_Occupation[(op->get_spin_position() + 1) % N]++;
            break;
        }

    }
}

ostream& operator<<(ostream& os, Lattice& lattice) {

    for (int time_position = lattice.M - 1; time_position >= 0; time_position-
-) {
        Operator* op = &lattice.operators[time_position];

        for (int position = 0; position < op->get_spin_position(); position++)
            os << '\t';
        os << op->get_leg(TOP_LEFT)->get_occupation_number() << '\t'
            << op->get_leg(TOP_RIGHT)->get_occupation_number() << '\n';
        for (int position = 0; position < op->get_spin_position(); position++)
            os << '\t';
        switch (op->get_type()) {
        case LEFT:
            os << "<--------";
            break;
        case RIGHT:
            os << "-------->";
            break;
        case DIAGONAL:
            os << "---------";
        }
        os << '\n';
        for (int position = 0; position < op->get_spin_position(); position++)
            os << '\t';
        os << op->get_leg(BOTTOM_LEFT)->get_occupation_number() << '\t'
            << op->get_leg(BOTTOM_RIGHT)->get_occupation_number() << '\n';
    }

    for (auto iter_site = lattice.site_occupation.begin(); iter_site !=
lattice.site_occupation.end(); iter_site++)
        os << *iter_site << '\t';
    os << '\n';
    return os;
}
```

```cpp
//------------------------------------------------------------------------
// Compiler: Microsoft Visual Studio 2017
// Programmer: CHEN Ao
// File name: operator.cpp
//------------------------------------------------------------------------

#include"SSE_Hubbard.h"

Operator::Operator() {
    type = IDENTITY;
    leg.resize(4);
    Is_changed = false;
}
void Operator::initialize_time_position(int Time_Position) {
    time_position = Time_Position;
    leg[BOTTOM_LEFT].initialize(Time_Position, this);
    leg[BOTTOM_RIGHT].initialize(Time_Position, this);
    leg[TOP_LEFT].initialize(Time_Position + 1, this);
    leg[TOP_RIGHT].initialize(Time_Position + 1, this);
}
int Operator::get_type() const {
    return type;
}
int Operator::get_spin_position() const {
    return spin_position;
}
int Operator::get_time_position() const {
    return time_position;
}
bool Operator::get_update_status() const {
    return Is_changed;
}
void Operator::create_diagonal_operator(int Spin_Position, int N) {
    type = DIAGONAL;
    spin_position = Spin_Position;
    leg[BOTTOM_LEFT].change_spin_position(Spin_Position);
    leg[TOP_LEFT].change_spin_position(Spin_Position);
    leg[BOTTOM_RIGHT].change_spin_position((Spin_Position + 1) % N);
    leg[TOP_RIGHT].change_spin_position((Spin_Position + 1) % N);
}
void Operator::delete_operator() {
    type = IDENTITY;
}
void Operator::change_update_status(bool new_status) {
    Is_changed = new_status;
}
void Operator::type_update() {
    switch (leg[TOP_LEFT].get_occupation_number() -
leg[BOTTOM_LEFT].get_occupation_number()) {
    case 0:
        type = DIAGONAL; break;
    case -1:
        type = RIGHT; break;
    case 1:
        type = LEFT;
    }
}
Leg* Operator::loop_update(Leg* p_leg) {
    int n[4] = { leg[0].get_occupation_number(),leg[1].get_occupation_number(),
        leg[2].get_occupation_number(),leg[3].get_occupation_number() };
    int* probability = probability_table[n[0]][n[1]][n[2]][n[3]];
    int p = rand();
    Leg* flipped_leg;
    if (p < probability[0])
        flipped_leg = &leg[0];
```

```
    else if (p < probability[1])
        flipped_leg = &leg[1];
    else if (p < probability[2])
        flipped_leg = &leg[2];
    else
        flipped_leg = &leg[3];
    int previous_occupation = flipped_leg->get_occupation_number();
    int type = flipped_leg->get_type();
    if (type / 2)
        flipped_leg->change_new_occupation(n[0] + n[1] - n[5 - type]);
    else
        flipped_leg->change_new_occupation(n[2] + n[3] - n[1 - type]);
    flipped_leg->get_linked_leg()->change_new_occupation(flipped_leg->get_occup
ation_number());
    return flipped_leg->get_linked_leg();
}
Leg* Operator::get_leg(int place) {
    return &(leg[place]);
}
```

```cpp
//-------------------------------------------------------------------------
// Compiler: Microsoft Visual Studio 2017
// Programmer: CHEN Ao
// File name: leg.cpp
//-------------------------------------------------------------------------

#include"SSE_Hubbard.h"
Leg::Leg() {
    Is_starting_point = false;
}
int Leg::get_type() const {
    return type;
}
int Leg::get_time_position() const {
    return time_position;
}
int Leg::get_spin_position() const {
    return spin_position;
}
Operator* Leg::get_operator() const {
    return attaching_operator;
}
int Leg::get_occupation_number() const {
    return new_occupation_number;
}
Leg* Leg::get_linked_leg() const {
    return linked_leg;
}
void Leg::change_occupation_number(int n) {
    occupation_number = n;
    new_occupation_number = n;
}
void Leg::change_occupation_number(const Leg& leg) {
    occupation_number = leg.occupation_number;
    new_occupation_number = occupation_number;
}
void Leg::random_new_occupation() {
    new_occupation_number = (new_occupation_number + rand() % 2 + 1) % 3;
}
void Leg::change_new_occupation(int n) {
    new_occupation_number = n;
}
void Leg::update_new_occupation() {
    occupation_number = new_occupation_number;
}
void Leg::remove_new_occupation() {
    new_occupation_number = occupation_number;
}
void Leg::change_starting_status(bool new_status) {
    Is_starting_point = new_status;
}
bool Leg::Not_starting_point() const {
    return !Is_starting_point;
}
void Leg::initialize(int Time_Position, Operator* op) {
    time_position = Time_Position;
    attaching_operator = op;
    for (int i = 0; i < 4; i++) {
        if (op->get_leg(i) == this)
            type = i;
    }
}
void Leg::change_spin_position(int Spin_Position) {
    spin_position = Spin_Position;
}
```

```
void Leg::change_linked_leg(Leg* Linked_Vertex) {
    linked_leg = Linked_Vertex;
    Linked_Vertex->linked_leg = this;
}
bool operator<(const Leg& v1, const Leg& v2) {
    if (v1.time_position < v2.time_position)
        return true;
    else if (v1.time_position == v2.time_position && (v1.type == TOP_LEFT ||
v1.type == TOP_RIGHT) )
        return true;
    else
        return false;
}
```

```cpp
//--------------------------------------------------------------------------
// Compiler: Microsoft Visual Studio 2017
// Programmer: CHEN Ao
// File name: driver.cpp
//--------------------------------------------------------------------------

#include"SSE_Hubbard.h"

int main() {
    srand((unsigned int)time(NULL));
    ofstream rho_file("rho.txt");
    ofstream n_file("n.txt");
    for (double x = 0.01; x < 0.305; x+=0.01) {
        for (double y = 0.00; y < 1.51; y += 0.05) {
            parameter_renew(50 * x, 2 / x, 2 * y / x);
            double rho = 0;
            int n = 0;
            for (int test = 0; test < 10; test++) {
                Lattice lattice(16, (int)(22*beta*(mu+U)));
                for (int t = 0; t < 100; t++) {
                    lattice.step();
                }
                for (int t = 0; t < 1000; t++) {
                    lattice.step();
                    rho += lattice.rho();
                    n += lattice.get_n();
                }
            }
            // use n/M to indicate whether the size of the system is too small
            cout << x << '\t' << y << '\t' << rho / (1000 * 10) << '\t' <<
(double)n / (1000 * 10) / (22 * beta*(mu + U)) << '\n';
            rho_file << x << '\t' << y << '\t' << rho / (1000*10) << '\n';
            n_file << x << '\t' << y << '\t' << (double)n / (1000 * 10) / (22 *
beta*(mu + U)) << '\n';
        }
    }
    rho_file.close();
    n_file.close();
    return 0;
}
```

附录 F　格点占据数动态变化绘制代码

```
//--------------------------------------------------------------------------
// Compiler: Microsoft Visual Studio 2017
// Programmer: CHEN Ao
// File name: driver.cpp
// Use this file to replace the "driver.cpp" file in Appendix E
//--------------------------------------------------------------------------
#include"SSE_Hubbard.h"
int main() {
srand((unsigned int)time(NULL));
ofstream constants_file("constants.dat");
ofstream configuration_file("configuration.dat");
int N = 100, M = 2000;
int total_time = 100;
int output_interval = 1;
double beta = 0.5, U = 20, mu = 10;
constants_file << M << '\n' << total_time / output_interval << '\n' << U <<
'\n' << mu;
Lattice lattice(N, M);
parameter_renew(0.5, 20, 10);
for (int t0 = 0; t0 < 100; t0++)
    lattice.step();
for (int t = 0; t < total_time; t++) {
    lattice.step();
    if (t%output_interval == 0)
        lattice.output_configuration(configuration_file);
}
constants_file.close();
configuration_file.close();
return 0;
}
```

```
%-------------------------------------------------------------------------
% Programmed in MATLAB R2017a
% Programmer: CHEN Ao
% File name: draw_gif.m
%-------------------------------------------------------------------------
constants=importdata('constants.dat');
L=constants(1);
image_number=constants(2);
U=constants(3);
mu=constants(4);
filename=sprintf('SSE_Hubbard %d-%02d-%02d %02d-%02d-%02d.gif',fix(clock));
A0=cell([image_number 1]);
map=cell([image_number 1]);
configuration=importdata('configuration.dat');
MyMap=[ 0 0 0
       0.5 0.5 0.5
       1 1 1];
for i=1:image_number
    image=figure;
    set(image,'Color','w');
    set(image,'position',[0 0 1000 1000]);
    set(image,'visible','off');
    h=pcolor(configuration((i-1)*L+1:i*L,:));
    set(gca,'FontSize',30);
    title(sprintf('U = %.0f  mu = %.0f',U,mu));
    caxis([0 2]);
    colormap(MyMap);
    yticks([500 1000 1500 2000])
    set(h,'LineStyle','none');
    [A0{i},map{i}] = rgb2ind(frame2im(getframe(image)),256);
end

imwrite(A0{1},map{1},filename,'gif','LoopCount',Inf,'DelayTime',0.1);
for i=2:image_number
    imwrite(A0{i},map{i},filename,'gif','WriteMode','append','DelayTime',0.1);
end
winopen(filename);
```