Hubbard模型的行列式蒙特卡洛模拟

吴晋渊 18307110155

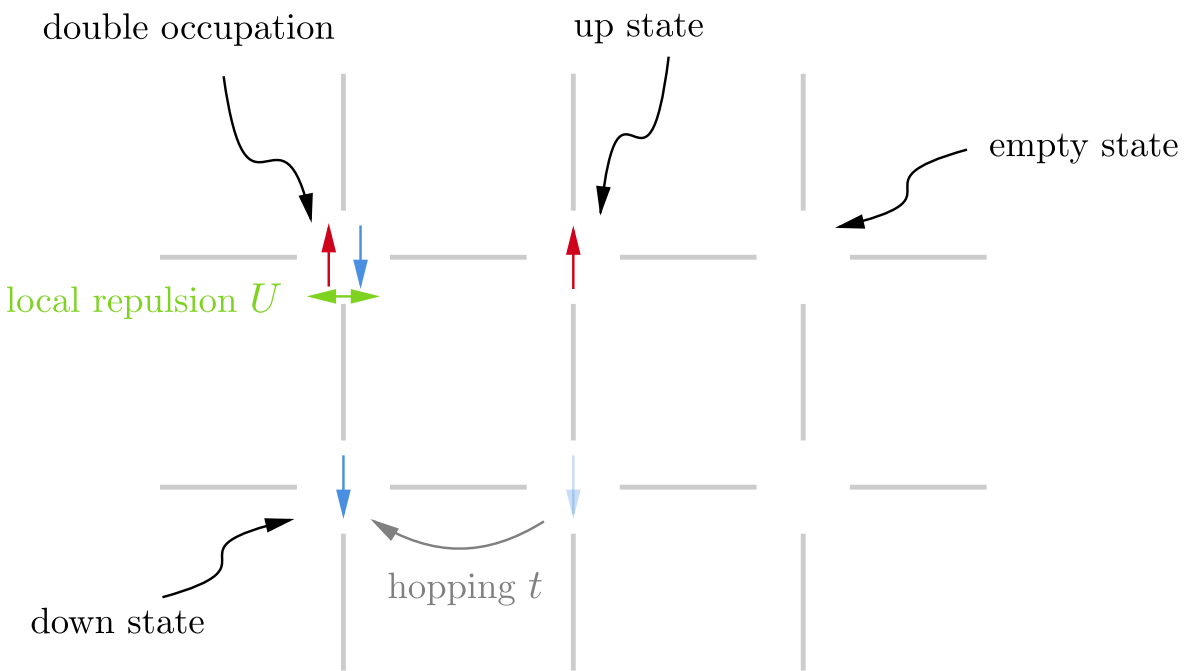
复旦大学物理学系

2022

二维正方晶格Hubbard模型

Hubbard模型

$$H = -t \sum_{\langle i,j \rangle} c^{\dagger}_{i\sigma} c_{j\sigma} + U \sum_{i} n_{i\uparrow} n_{j\downarrow} \tag{1}$$



二维正方晶格Hubbard模型

Hubbard模型

$$H = -t \sum_{\langle i,j \rangle} c^{\dagger}_{i\sigma} c_{j\sigma} + U \sum_{i} n_{i\uparrow} n_{j\downarrow} \tag{2}$$

- 最为著名的强关联电子模型
  - 半填充⇒金属-海森堡模型转变（Mott转变）
  - 掺杂⇒t-J模型：可能的高温超导机制，类似于铜基超导的赝能隙
- 不可能解析研究（高维无Bethe ansatz可用，微扰论失效），需要蒙特卡洛模拟
  - 量子系统，无离散场构型 ⇒ 解决方案：离散路径积分，引入N个虚时间点，做Trotter分解，d维量子系统=$d+1$维经典统计系统

$$\begin{aligned} Z = \operatorname{tr} \mathrm{e}^{-\beta H} &= \sum_{\{\sigma_1,\ldots,\sigma_N\}} \prod_{n} \langle \sigma_n | \mathrm{e}^{-\Delta\tau H} | \sigma_{n+1} \rangle \\ &= \sum_{\sigma} \prod_{n} \mathrm{e}^{-\Delta\tau H[\sigma_n,\sigma_{n+1}]} . \end{aligned} \tag{3}$$

  - 费米子算符无经典对应 ⇒ 设法把费米子自由度变成玻色自由度

朴素的DQMC

离散Hubbard-Stratonovich变换

- 用玻色场取代费米场
- 物理图像：用每个格点上的总自旋（或者别的费米算符二次型）取代费米自由度

$$\mathrm{e}^{-\Delta\tau U\sum_{\boldsymbol{i}}(n_{\boldsymbol{i}\uparrow}-1/2)(n_{\boldsymbol{j}\downarrow}-1/2)} \simeq \text{const}\times\sum_{s_1,s_2,\ldots,s_n=\pm1}\mathrm{e}^{\alpha\sum_{\boldsymbol{i}}s_{\boldsymbol{i}}(n_{\boldsymbol{i}\uparrow}-n_{\boldsymbol{i}\downarrow})}, \quad (4)$$

$$N\Delta\tau=\beta,\quad \cosh(\alpha)=\mathrm{e}^{\Delta\tau U/2}. \quad (5)$$

- 随后可以“积掉”电子自由度，只留下$s_{\boldsymbol{i},\tau}$

$$\mathrm{tr}\,\mathrm{e}^{-c_i^\dagger A_{ij}c_j}\,\mathrm{e}^{-c_i^\dagger B_{ij}c_j}\cdots=\det\left(1+\mathrm{e}^{-\mathbf{A}}\,\mathrm{e}^{-\mathbf{B}}\cdots\right). \quad (6)$$

- 这个方法称为**determinant quantum Monte Carlo (DQMC)**

朴素的的DQMC

- 定义

$$\mathbf{B}_{\mathbf{s}}^{\sigma}(\tau) = \mathrm{e}^{\sigma\alpha\ \mathrm{diag}\mathbf{s}_{\tau}}\,\mathrm{e}^{-\Delta\tau\mathbf{T}}, \quad \mathbf{B}_{\mathbf{s}}^{\sigma}(\tau_2,\tau_1) = \prod_{n=n_1+1}^{n_2}\mathbf{B}_{\mathbf{s}}^{\sigma}(\tau), \tag{7}$$

  其中$\sigma = \uparrow,\downarrow := \pm 1$，$\mathbf{s}_{\tau} = [s_{i,\tau}]$，s为一个$\{s_{i,\tau}\}$构型的简写，T为一次量子化动能矩阵

- 可推导出配分函数（用于计算接受率，DQMC的命名由来）

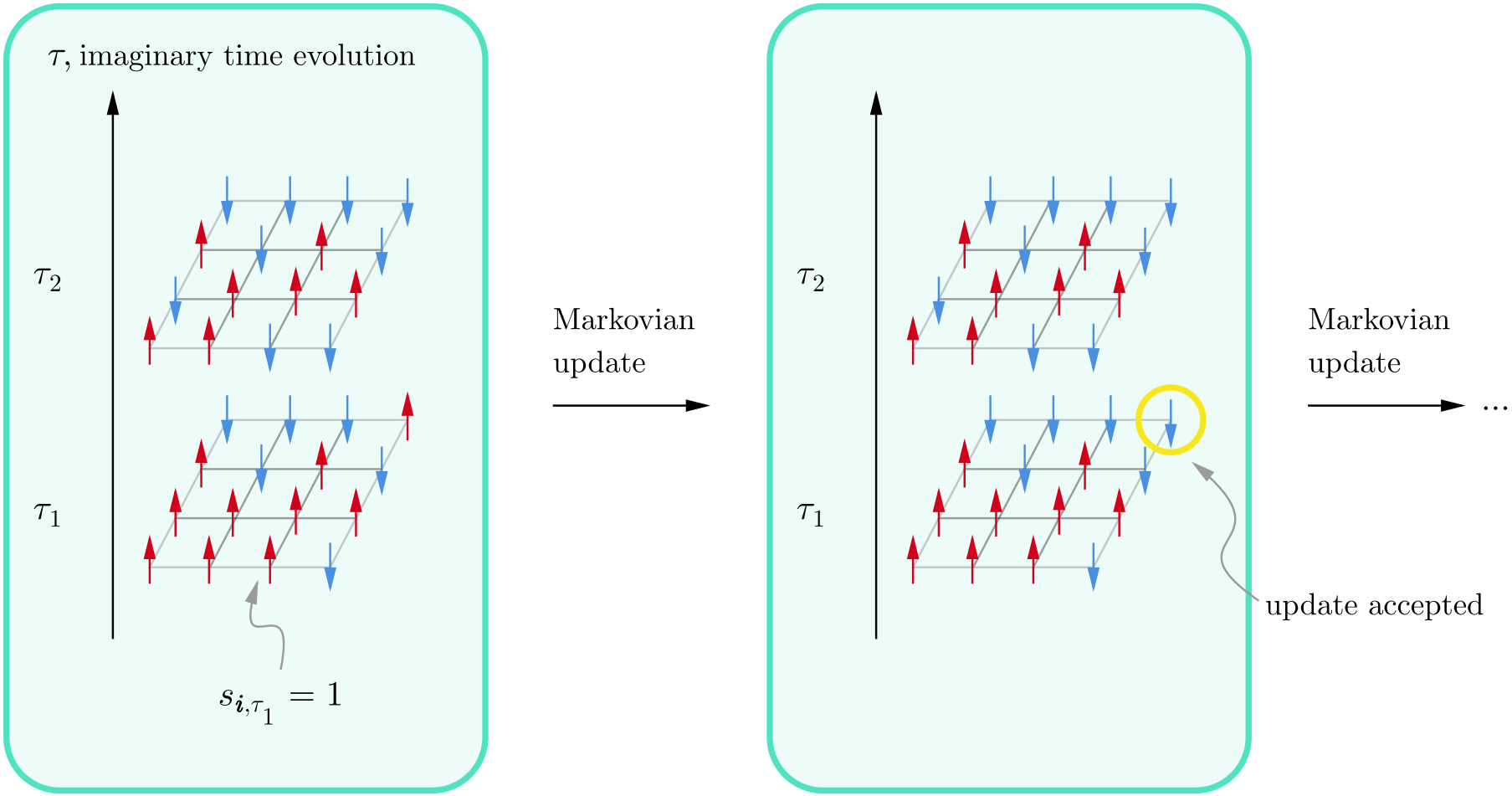
$$Z = \sum_{\mathbf{s}} Z[\mathbf{s}], \quad Z[\mathbf{s}] = \det\left(1 + \prod_{\sigma=\uparrow,\downarrow}\prod_{n=1}^{m}\mathbf{B}_{\mathbf{s}}^{\sigma}(\tau)\right), \tag{8}$$

- 以及等时格林函数（用于计算物理量期望值）

$$\mathbf{G}_{ij}^{\sigma}(\tau) = \langle c_{i\sigma}c_{j\sigma}^{\dagger}\rangle_{\tau} = (1 + \mathbf{B}_{\mathbf{s}}^{\sigma}(\tau,0)\mathbf{B}_{\mathbf{s}}^{\sigma}(\beta,\tau))_{i,j}^{-1}. \tag{9}$$

- 计算$\mathbf{B}^{\sigma}(\tau_1,\tau_2)$时由于有连乘，需要做数值稳定

朴素的DQMC



a system configuration in DQMC (what is stored when running the program)

朴素DQMC

物理量期望值为

$$\langle O \rangle = \sum_{\mathbf{s}} p(\mathbf{s}) \langle O \rangle_{\mathbf{s}}, \tag{10}$$

$p(\mathbf{s})$通过蒙特卡洛采样，而$\langle O \rangle_{\mathbf{s}}$遵从Wick定理

- 双占据：

$$\langle n_{i\uparrow} n_{i\downarrow} \rangle_{\mathbf{s}} = (1 - \langle c_{i\uparrow} c_{i\uparrow}^{\dagger} \rangle_{\mathbf{s}})(1 - \langle c_{i\downarrow} c_{i\downarrow}^{\dagger} \rangle_{\mathbf{s}}). \tag{11}$$

- 磁化关联函数：

$$\begin{aligned} \langle s_i^z s_j^z \rangle_{\mathbf{s}} &= \frac{1}{4} \langle (c_{i\uparrow}^{\dagger} c_{i\uparrow} - c_{i\downarrow}^{\dagger} c_{i\downarrow})((c_{j\uparrow}^{\dagger} c_{j\uparrow} - c_{j\downarrow}^{\dagger} c_{j\downarrow})) \rangle_{\mathbf{s}} \\ &= \frac{1}{4} (\langle c_{i\uparrow}^{\dagger} c_{i\uparrow} \rangle_{\mathbf{s}} \langle c_{j\uparrow}^{\dagger} c_{j\uparrow} \rangle_{\mathbf{s}} + \langle c_{i\downarrow}^{\dagger} c_{i\downarrow} \rangle_{\mathbf{s}} \langle c_{j\downarrow}^{\dagger} c_{j\downarrow} \rangle_{\mathbf{s}} \\ &\quad + \langle c_{i\uparrow}^{\dagger} c_{j\uparrow} \rangle_{\mathbf{s}} \langle c_{i\uparrow} c_{j\uparrow}^{\dagger} \rangle_{\mathbf{s}} + \langle c_{i\downarrow}^{\dagger} c_{j\downarrow} \rangle_{\mathbf{s}} \langle c_{i\downarrow} c_{j\downarrow}^{\dagger} \rangle_{\mathbf{s}} \\ &\quad - \langle c_{i\downarrow}^{\dagger} c_{i\downarrow} \rangle_{\mathbf{s}} \langle c_{j\uparrow}^{\dagger} c_{j\uparrow} \rangle_{\mathbf{s}} - \langle c_{j\downarrow}^{\dagger} c_{j\downarrow} \rangle_{\mathbf{s}} \langle c_{i\uparrow}^{\dagger} c_{i\uparrow} \rangle_{\mathbf{s}}). \end{aligned} \tag{12}$$

基于格林函数的DQMC

- 朴素DQMC性能开销大头：$\mathbf{B}(\tau_1, \tau_2)$矩阵连乘
- 虚时间点τ，空间格点$\boldsymbol{i}$处发生一次更新，则$\mathbf{B}$矩阵只有如下变化：

$$\mathbf{B}^{\sigma}_{\mathbf{s}'}(\tau) = \left(1 + \boldsymbol{\Delta}^{\boldsymbol{i},\sigma}\right) \mathbf{B}^{\sigma}_{\mathbf{s}}(\tau), \tag{13}$$

其中$\boldsymbol{\Delta}^{\boldsymbol{i},\sigma}$对角，除了$\boldsymbol{i}$号元素为$\mathrm{e}^{-2\sigma\alpha s_{i,\tau}} - 1$外其余为零

- 可以手动推导出

$$R = \prod_{\sigma=\uparrow,\downarrow} \underbrace{\det\left(1 + \boldsymbol{\Delta}^{\boldsymbol{i},\sigma}(1 - \mathbf{G}^{\sigma}_{\mathbf{s}}(\tau))\right)}_{R^{\sigma}}, \tag{14}$$

$$\mathbf{G}^{\sigma}_{\mathbf{s}'}(\tau) = \mathbf{G}^{\sigma}_{\mathbf{s}}(\tau) - \frac{1}{R^{\sigma}} \mathbf{G}^{\sigma}_{\mathbf{s}}(\tau) \boldsymbol{\Delta}^{\boldsymbol{i},\sigma}(1 - \mathbf{G}^{\sigma}_{\mathbf{s}}(\tau)). \tag{15}$$

- 接受率的计算和格林函数的更新其实无需$\mathbf{B}$矩阵！
- 可以将格林函数作为system configuration的一部分，初始化时从$\mathbf{s}$计算一次，之后似乎不必再计算任何$\mathbf{B}$矩阵连乘

基于格林函数的DQMC

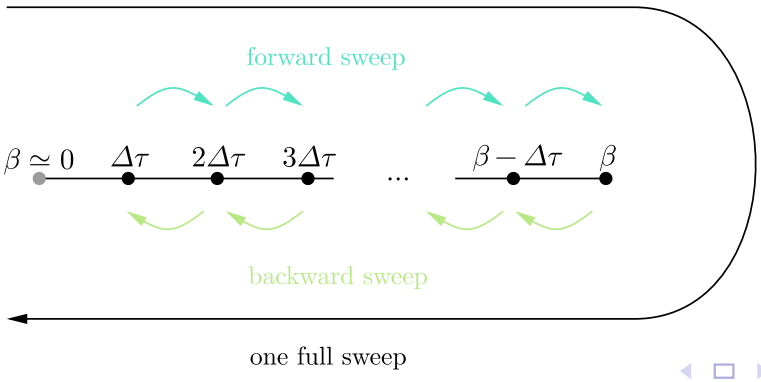
算法纲要：

- 系统构型：s，目前所处的虚时间点τ，τ处的格林函数$\mathbf{G}^\sigma$
- 初始化：初始$s_{i,\tau}$构型，计算各虚时间点的格林函数
- 给定τ，翻转$s_{i,\tau}$：接受率为(14)，如果成功，将$s_{i,\tau}$变更正负号，并且用(15)计算新的$\mathbf{G}^\sigma(\tau)$，$\mathbf{G}^\sigma \leftarrow \mathbf{G}^\sigma_{s'}(\tau)$
- 已经完成τ点处全部格点的翻转，使用

$$\mathbf{G}^\sigma(\tau+\Delta\tau) = \mathbf{B}^\sigma(\tau+\Delta\tau)\mathbf{G}^\sigma(\tau)\mathbf{B}^\sigma(\tau+\Delta\tau)^{-1} \tag{16}$$

从τ点转移到$\tau \pm \Delta\tau$点，$\mathbf{G}^\sigma \leftarrow \mathbf{G}^\sigma(\tau \pm \Delta\tau)$，$\tau \leftarrow \tau \pm \Delta\tau$，然后去翻转新的$\tau$处的$s_{i,\tau}$

- 一个sweep：



格林函数DQMC

进一步的改进：

- 通过矩阵乘法更新格林函数会导致矩阵计算误差指数放大
  - 隔一定步数仍然需要重新计算格林函数（称为wrap）
- 在时间点τ上的更新不会影响别的虚时间点上的$\mathbf{B}^\sigma(\tau')$
  - 将不同虚时间点的$\mathbf{B}^\sigma(\tau)$保存起来，避免在wrap的时候重复计算
- 用$\mathbf{G}^\sigma(\tau)$算$\mathbf{G}^\sigma(\tau \pm \Delta\tau)$时需要左乘、右乘$\mathbf{B}^\sigma$和$(\mathbf{B}^\sigma)^{-1}$
  - 求逆操作耗时长，直接使用

$$\mathbf{B}^\sigma(\tau)^{-1} = \mathrm{e}^{\Delta\tau\mathbf{T}}\,\mathrm{e}^{-\sigma\alpha\ \mathrm{diag}\mathbf{s}_\tau} \tag{17}$$

  - $\mathrm{e}^{\pm\Delta\tau\mathbf{T}}$计算用时长，$\mathbf{T}$计算过程中不变，应该预先算好
  - 左乘、右乘$\mathrm{e}^{\pm\sigma\alpha\ \mathrm{diag}\mathbf{s}_\tau}$应使用为对角矩阵优化的矩阵乘法

实现细节

目标：

- 模块化、封装：避免全局变量传参数
  - 被修改的变量、被调用的函数是非常透明的
  - 便于调试
- 代码复用
  - 可以将DQMC代码整合进更大的程序中，如研究Hubbard模型电子和Ising自旋耦合
  - 便于改动晶格类型

实现细节

将一个DQMC任务中用到的东西打包在一个结构体中，内容包括：

- 模型参数：t, U, β，lattice
- 问题规模：虚时间点总数N（记作n_imtimes），经过几个虚时间点做wrap（n_wrap）
- 预先算好的物理量：$\Delta\tau$, α, $\mathbf{T}$, $e^{\Delta\tau\mathbf{T}}$（记作expT），$e^{-\Delta\tau\mathbf{T}}$（记作expmT）
- 系统构型，包括s（辅助场构型），以及B矩阵
- 格林函数$\mathbf{G}^\sigma(\tau)$不在其中
  - 如果要有，那么目前的虚时间点也必须放在结构体HubbardDQMC中，而mutable struct性能不佳
  - 可以给sweep函数传回调函数来访问$\mathbf{G}^\sigma$

```
struct HubbardDQMC{L <: AbstractLattice}
    t::Float64
    U::Float64
    β::Float64
    Δτ::Float64
    n_imtimes::Int64
    n_wrap::Int64
    α::Float64

    lattice::L

    # s[τ, i]
    s::Matrix{Int64}

    # T[i, j]
    T::Matrix{Float64}
    # exp(Δτ * T)
    expT::Matrix{Float64}
    # exp(- Δτ * T)
    expmT::Matrix{Float64}

    id::Matrix{Float64}
    B_up_storage::Array{Float64, 3}
    B_dn_storage::Array{Float64, 3}
end
```

实现细节

格点信息单独定义，尽可能一般性；从1开始给每个格点编号

- `site_list[i, :]`：给出i号格点的坐标
- `inverse_list[x, y]`：给出(x, y)位置处的格点的编号
- `neighbor_list[i, :]`：给出能够称为i号格点的近邻格点的全部格点
- `neighbor_list[ineighbor_list_indices[1]]`给出i号格点的最近邻格点，将1换成2给出次近邻格点，等等

```
struct SquareLattice2D <: AbstractLattice
    n_sites::Int

    # Two dimensional, site_list[i, 1] = x, site_list[i, 2] = y
    site_list::Matrix{Int}

    # inverse_list[x, y] = i
    inverse_list::Matrix{Int}

    # neighbor_list[i, 1:4] gives the nearest neighbors, while
    neighbor_list::Matrix{Int}

    neighbor_list_indices
end
```

实现细节

格点初始化示例，类似地可以定义三角格子，六角格子等：

```
function SquareLattice2D(L::Integer)
    n_sites = L^2
    inverse_list = zeros(Int64, (L, L))
    site_list = zeros(Int64, (n_sites, 2))

    for i in 1:L
        for j in 1:L
            site_list[(i-1)*L+j, :] = [i, j]
            inverse_list[i, j] = (i - 1) * L + j
        end
    end

    neighbor_list = zeros(Int64, (n_sites, 8))
    for i in 1:L
        for j in 1:L
            neighbor_list[inverse_list[i, j], 1] = inverse_list[i, back_into_range(j+1, L)]
            neighbor_list[inverse_list[i, j], 2] = inverse_list[back_into_range(i+1, L), j]
            neighbor_list[inverse_list[i, j], 3] = inverse_list[i, back_into_range(j-1, L)]
            neighbor_list[inverse_list[i, j], 4] = inverse_list[back_into_range(i-1, L), j]
            neighbor_list[inverse_list[i, j], 5] = inverse_list[back_into_range(i+1, L), back_into_range(j+1, L)]
            neighbor_list[inverse_list[i, j], 6] = inverse_list[back_into_range(i+1, L), back_into_range(j-1, L)]
            neighbor_list[inverse_list[i, j], 7] = inverse_list[back_into_range(i-1, L), back_into_range(j-1, L)]
            neighbor_list[inverse_list[i, j], 8] = inverse_list[back_into_range(i-1, L), back_into_range(j+1, L)]
        end
    end

    SquareLattice2D(n_sites, site_list, inverse_list, neighbor_list, square_lattice_2D_neighbor_list_indices)
end
```

实现细节

计算的中间步骤完全透明，通过HubbardDQMC结构体和其它变量传递数据

```
function accept_rate_up(model::HubbardDQMC, G_up::Matrix{Float64}, τ, i)
    1 + Δ_up(model, τ, i)[i, i] * (1 - G_up[i, i])
end

function accept_rate_dn(model::HubbardDQMC, G_dn::Matrix{Float64}, τ, i)
    1 + Δ_dn(model, τ, i)[i, i] * (1 - G_dn[i, i])
end

function accept_rate(model::HubbardDQMC, G_up::Matrix{Float64}, G_dn::Matrix{Float64}, τ, i)
    accept_rate_up(model, G_up, τ, i) * accept_rate_dn(model, G_dn, τ, i)
end

function G_update(model::HubbardDQMC, G_up, G_dn, τ, i)
    s_τ = model.s
    B_up_storage = model.B_up_storage
    B_dn_storage = model.B_dn_storage
    copy!(G_up, G_up - G_up * Δ_up(model, τ, i) * (I - G_up) / accept_rate_up(model, G_up, τ, i))
    copy!(G_dn, G_dn - G_dn * Δ_dn(model, τ, i) * (I - G_dn) / accept_rate_dn(model, G_dn, τ, i))
    s_τ[τ, i] *= -1
    B_up_storage[:, :, τ] = B_up(model, τ)
    B_dn_storage[:, :, τ] = B_dn(model, τ)
end
```

效果

运行界面

```
Welcome to a simple DQMC simulation of Hubbard model.

Initialization completed.
We are simulating with
  t      =   1.0
  U      =   8.0
  beta   =   4.0
  dtau   =   0.05
  alpha  =   0.653732997987334
on a 4 * 4 2d lattice and with 80 time steps.

Heating up for 256 steps.
Heating up completed.

Observables:
Bin 1 completed.
E_kin      =   -0.9687630355749639 ± 0.33127357236689514
double_occ =   0.05310866943682016 ± 0.022527760888213445
mag        =   0.11132605943340927 ± 0.07612943910110066

Observables:
Bin 2 completed.
E_kin      =   -0.8916161993979214 ± 0.32920849296173
double_occ =   0.04991756543479846 ± 0.022149779620090345
mag        =   0.07070575529071395 ± 0.06922846274872253

Observables:
```

输出结果

```
Bin average data:
=============================================================
E_kin            double_occ      magnetization
=============================================================
-0.9687630356    0.0531086694    0.1113260594
-0.8916161994    0.0499175654    0.0707057553
-0.9003367259    0.0486302172    0.0410485615
-0.8389155985    0.0478881422    0.0451339080
-0.9308848780    0.0509861957    0.1028641983

Binning results:
-------------------------------------------------------------
E_kin      =   -0.9061032874760621 ± 0.04821225404278734
double_occ =   0.05010615800636402 ± 0.0020568849869937057
mag        =   0.07421569649323544 ± 0.03223369605369777

julia>
```

计算结果和许霄琰老师的示例代码一致。

效果

- 参数：$t = 1.0$, $\beta = 4.0$, $\Delta\tau = 0.05$
- 运行结果（红色）和许霄琰老师的示例代码的结果（蓝色）一致
- 随着U增大能量增大，同时涨落也增大
- 随着U增大双占据下降