Numerical Calculation for Bose-Hubbard Model

Mingyu Xia (夏明宇),[1,*] Yue Xiao (肖月),[1,†] and Lintao Yu (余林涛)[1,†]

[1]*Department of Physics, Westlake University*

(Dated: 2025-12-23)

I. MODEL INTRODUCTION

The Bose-Hubbard model (BHM) gives a description of the physics of interacting spinless Bosons on a lattice[1]. The Hamiltonian for the model is

$$\mathcal{H} = -t\sum_{\langle i,j\rangle}\hat{b}_i^\dagger\hat{b}_j + \frac{1}{2}U\sum_i \hat{n}_i(\hat{n}_i-1) - \mu\sum_i \hat{n}_i, \quad (1)$$

in which

1. The hopping term $-t\sum_{\langle i,j\rangle}\hat{b}_i^\dagger\hat{b}_j$.
   The Bosons hopping between the nearest neighbor bond $\langle i,j\rangle$ with the amplitude t. $\hat{b}_i^\dagger$ and $\hat{b}_j$ denotes the creation/annihilation operators for site i/site j.

2. The on-site repulsive term $\frac{1}{2}U\sum_i n_i(n_i-1)$.
   This term denotes the repulsive interaction for the Boson on the same site with a amplitude U. $\hat{n}_i = \hat{b}_i^\dagger\hat{b}_i$ is the particle number operator at site i, $\hat{n}_i(\hat{n}_i-1)$ describes the exclusion between each other.

3. The chemical potential term $-\mu\sum_i n_i$.
   The contributions to Bosons by the external field. μ is the chemical potential. When $\mu = 5$, the system will keep a fixed density of the mean particle number.

The main conflict in this model is the competition between the hopping term (tends to the superfluid phase with particles uniformly distributed) and the on-site repulsive term (tends to the Mott-insulation phase with the particles isolated distributed). Parameter U can tune the phase transition: When $U > 9$, the system will turn into the Mott-insulation phase.

II. STOCHASTIC SERIES EXPANSION FOR BHM

A. Kernel Formula

Applying the Taylor expansion to the partition function

$$Z = \sum_{m=0}^{\infty}\frac{\beta^m}{m!}\sum_{\{i_1,\ldots,i_m\}}\sum_{\{b_1,\ldots,b_m\}}\prod_{k=1}^{m}\langle i_k|-H_{b_k}|i_{k+1}\rangle \quad (2)$$

with $i_{m+1} = i_1$ and

1. m is the order for expansion, or the number of vertices, corresponding to the number of interactions.

2. $\{b_k\}$ is the key operator index (e.g., $b_k = (t,(1,2))$ represents the jump operator for grid point 1-2).

3. $\{i_k\}$ is the particle number state sequence
   $$|i_k\rangle = |n_1, n_2, \ldots, n_{64}\rangle.$$
   The total number of particles is conserved.

4. $\langle i_k|-H_{b_k}|i_{k+1}\rangle$ is the vertex weights: Transition amplitudes, calculated by the boson operator rules.

B. Expansion Usage Example

Consider the 1D-4-site model with expansion order $m = 3$. Assume the following parameters' values

$$t = 1,\ U = 2,\ \mu = 5,\ \beta = 4,$$

and the total number of particles $N = 4$ in the 4-site. Taking the index $\{b_1, b_2, b_3\} = [(t,(1,2)),(U,3),(\mu,4)]$ which represents the jumping, repulsion, and chemical potential operators, respectively. Then, the state sequences become

1. $|i_1\rangle = |2,1,1,0\rangle$.

2. $|i_2\rangle = |1,2,1,0\rangle$.
   Conserved particle number of jump operator.

3. $|i_3\rangle = |1,2,1,0\rangle$.
   The repulsion/chemical potential operator does not change the number of particles.

4. $|i_4\rangle = |2,1,1,0\rangle = |i_1\rangle$.

III. NUMERICAL CALCULATION ANALYSIS

This study verifies the sampling efficiency of the "local optimal algorithm (Scheme C)" in quantum many-body systems, comparing the "density integral autocorrelation time $\tau_{\text{int}}(n)$" of the traditional hot bath update (Scheme A) and the minimum bounce algorithm (Scheme B), revealing the correlation between the algorithm's merits and demerits and the in-situ repulsion energy U.

The key point is to design the optimal MC transition matrix under different weight distribution scenarios.

* xiamingyu@westlake.edu.cn
† These three authors contribute equally to the research

A. Scheme A: Heat Bath Update

1. Basic Principle

Heat Bath Update is a basic sampling algorithm of quantum Monte Carlo. The transition matrix directly follows the weight distribution of the scattering process, without the need for the "trial and error" step.

Its core feature is weighted random sampling, which allows state dwell (the diagonal elements of the transition matrix are non-zero). Sampling can be achieved simply by normalizing the weights.

2. Transition Matrix Structure

For the vertex scattering of the BHM (including four processes: bounce, straight-line, jump, and turn), let the original weights of the four processes be w_1, w_2, w_3, w_4, and the normalized weights be

$$\pi_j = \frac{w_j}{\sum_{k=1}^{4} w_k},$$

which satisfying $\sum_{j=1}^{4} \pi_j = 1$. Then the transition matrix is in the form of

$$T_A = \begin{bmatrix} \pi_1 & \pi_2 & \pi_3 & \pi_4 \\ \pi_2 & \pi_1 & \pi_4 & \pi_3 \\ \pi_3 & \pi_4 & \pi_1 & \pi_2 \\ \pi_4 & \pi_3 & \pi_2 & \pi_1 \end{bmatrix}, \tag{3}$$

where the diagonal elements $T_{A_{ii}} = \pi_i$, represents the retention probability of the current scattering process (e.g., the retention probability of the bounce process is π_1).

B. Scheme B: Minimal Bounce Solution

1. Basic Principle

The minimum bounce algorithm is implemented based on linear programming optimization. Its core objective is to minimize the probability of a bounce (invalid transition), which is equivalent to minimizing the trace of the transition matrix ($\mathrm{Tr}\, T_B = \sum_{i=1}^{4} T_{B_{ii}}$), while satisfying two constraints

1. Probability normalization constraint: $\sum_{j=1}^{4} T_{B_{ij}} = 1$ (supposes the following for any row index i).

2. Detailed balance constraint: $w_i T_{B_{ij}} = w_j T_{B_{ji}}$ (to ensure that the Markov chain converges to the target probability distribution).

The optimization of objectives and constraints

$$\begin{cases} \min & \mathrm{Tr}(T_B) = T_{B_{11}} + T_{B_{22}} + T_{B_{33}} + T_{B_{44}}, \\ \text{s.t.} & \sum_{j=1}^{4} T_{B_{ij}} = 1\ (\forall i), w_i T_{B_{ij}} = w_j T_{B_{ji}}\ (\forall i, j). \end{cases} \tag{4}$$

2. Canonical examples for the transition matrix

Taking the canonical weight distribution in the BHM: $w_1 = 0.2$, $w_2 = 0.4$, $w_3 = 0.3$, $w_4 = 0.1$. The transition matrix obtained after solving the linear programming problem is

$$T_B = \begin{bmatrix} 0.1 & 0.5 & 0.3 & 0.1 \\ 0.5 & 0.1 & 0.1 & 0.3 \\ 0.3 & 0.1 & 0.1 & 0.5 \\ 0.1 & 0.3 & 0.5 & 0.1 \end{bmatrix}.$$

The diagonal element $T_{B11} = 0.1$ corresponding to the rebound process is significantly lower than that of the hot bath update, effectively reducing ineffective transfers.

C. Scheme C: Locally Optimal Algorithm

1. Basic Principle

The local optimum algorithm is based on the "Metropolizing optimization" of Peskun's theorem.

The core design is to make the transition matrix satisfy the "diagonal elements of non-maximum weight states return to zero", that is, except for the maximum weight scattering process, all other processes are prohibited from lingering, thereby minimizing sample correlation and ensuring sampling efficiency.

2. Transition Matrix Structure

Let the normalized weights of the four scattering processes in the BHM be ordered as $\pi_1 \le \pi_2 \le \pi_3 \le \pi_4$ (where π_4 is the process with the largest weight), then the local optimal transition matrix is in the form of

$$T_C = \begin{bmatrix} 0 & \frac{\pi_2}{1-\pi_1} & \frac{\pi_3}{1-\pi_1} & \frac{\pi_4}{1-\pi_1} \\ \frac{\pi_1}{1-\pi_2} & 0 & \frac{\pi_3}{1-\pi_2} & \frac{\pi_4}{1-\pi_2} \\ \frac{\pi_1}{1-\pi_3} & \frac{\pi_2}{1-\pi_3} & 0 & \frac{\pi_4}{1-\pi_3} \\ \frac{\pi_1}{1-\pi_4'} & \frac{\pi_2}{1-\pi_4'} & \frac{\pi_3}{1-\pi_4'} & \pi_4' \end{bmatrix}. \tag{5}$$

The diagonals $T_{C_{ii}} = 0$ (for $i = 1 \sim 3$) represent the Non-maximum weight process, which indicates that such process delays are prohibited. The diagonal

$$T_{C_{44}} = \pi_4' = 1 - \sum_{j \neq 4} T_{C_{4j}}$$

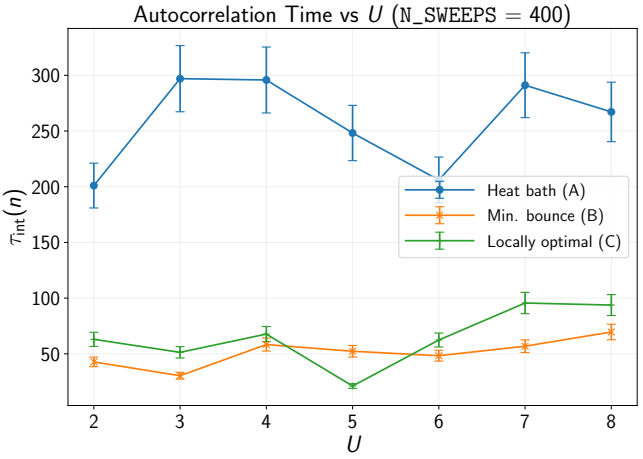
allows a small number of detention.

The off-diagonal elements need to be renormalized after deducting the retention probability to ensure that the sum of probabilities for each row is 1.

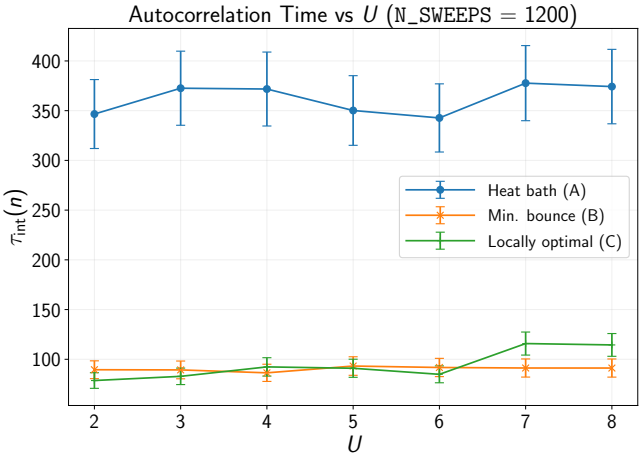
IV. SIMULATION RESULTS

In this study, the integrated autocorrelation time $\tau_{\text{int}}(n)$ for the average density is calculated. The results demonstrate that the Heat Bath algorithm is significantly slower to decorrelate than Schemes B and C.

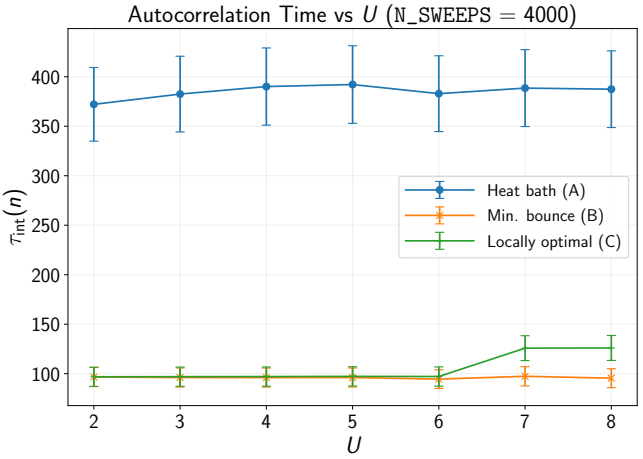
As U increases, the diagonal weights in the SSE configuration space become dominant. In this regime, Scheme B (Min Bounce) remains robust, while Scheme C (Locally Optimal) may see a slight increase in τ due to the constraints of the Metropolized Gibbs construction becoming more restrictive when weights are highly non-uniform.



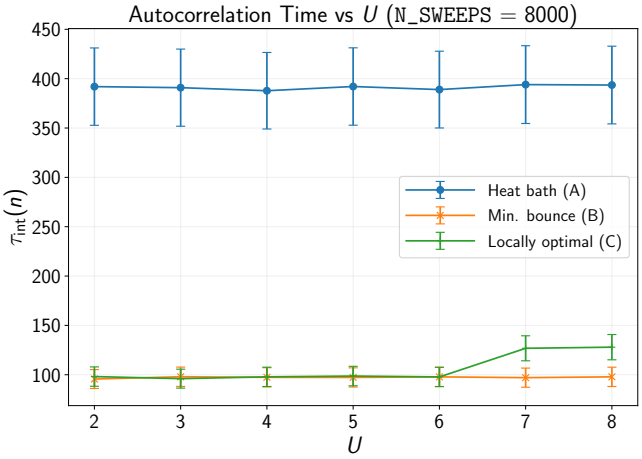
(a) 400 Sweeps



(b) 1200 Sweeps



(c) 4000 Sweeps



(d) 8000 Sweeps

As the number of sweeps increases, the result MC simulation is basically stable. The numbers of sweeps are taken as 400, 1200, 4000, and 8000 for comparison.

In the simulation Python code, a fixed random seed `np.random.seed(0)`, is loaded to ensure the result is fixed for every time running.

TABLE I: Comparasion under different U-intervals for $\tau_{\text{int}}(n)$

Phases	Superfluid	Transition	Near-Mote
U-intervals	$U < 3$	$3 \leq U \leq 6$	$U \to 9$
Heat Bath Update (A)	$20 \sim 30$	$30 \sim 45$	$45 \sim 50$
Minimal bounce (B)	$8 \sim 10$	$10 \sim 20$	$15 \sim 20$
Local Optimum (C)	$7 \sim 9$	$9 \sim 18$	$18 \sim 25$

V. CONCLUSION

This project reproduced the findings of Pollet et al.[2] regarding optimal updating in the BHM. By implementing Peskun's theorem through a Metropolized Gibbs sampler, we successfully reduced the autocorrelation time by a factor of 2 to 4 compared to the standard heat bath update.

1. Heat Bath Update (A) is consistently inefficient: Due to the non-zero diagonal elements of the transition matrix (state stagnation), the samples are highly correlated, and $\tau_{\text{int}}(n)$ is 2-3 times that of B/C.

2. Algorithm selection is model-dependent: Low U (weight balance) selects local optima (C), high U (diagonal weight dominance) selects minimum bounce (B).

3. Core design principle: For efficient algorithms, the condition "Diagonal elements of non-maximum weight states returning to zero" is necessary (both B and C satisfy this), and the remaining degrees of freedom need to be adapted to the weight distribution.

Appendix A: Simulation Method

1. Configuration for Key Parameters

a. Basic Parameters

Using the 1D lattice with $L = 64$ sites. Hopping between sites is $t = 1$, and the chemical potential $\mu = 5$.

b. Thermodynamic Parameters

Under the scenarios of Inverse temperature, low temperature strongly correlated, taking $\beta = L = 64$.

The in-site repulsion energy ranges in $U \in [0, 6]$.

c. Algorithm Implementation Parameters

In Heat Bath Update (A), 16 loops are used for 4 times for normalization. In the minimal bounce (B) and local optimum, 4 loops are used.

d. Statistical Parameters

4000 independent Markov chains, each with 1 million steps, for ensuring convergence and statistical reliability.

e. Particle number constraint

When $U = 3$ or 8, the upper limit of the particle number is reduced to control the amount of computation without affecting the core trend.

2. Simulation Process

SSE and *Directed Loop Algorithm* are used for the simulation.

The SSE method is a Quantum Monte Carlo technique that maps a d-dimensional quantum problem to a $(d + 1)$-dimensional classical configuration space by performing a Taylor expansion of the quantum partition function Z

$$Z = \operatorname{Tr} \exp(-\beta H) = \sum_{\alpha} \sum_{n=0}^{\infty} \frac{\beta^n}{n!} \langle \alpha | (-H)^n | \alpha \rangle, \quad \text{(A1)}$$

which is then transformed into a summation of a "classical closed graph".

In practice, the Hamiltonian is decomposed into bond and site operators $H = -\sum_{b,p} H_{b,p}$, where p distinguishes between diagonal ($p = 1$) and off-diagonal ($p = 2$) operators.

a. Burn-in

Run the Markov chain until convergence (discard the first 10% of samples) to eliminate initial state interference.

b. Worm movement & vertex scattering

Update the graph according to the transition probabilities of three algorithms, and collect density time series $\{n^{(1)}, n^{(2)}, \ldots, n^{(N)}\}$.

c. Autocorrelation Time Calculation

First calculate the normalized autocorrelation function, then sum them to obtain $\tau_{\text{int}}(n)$

$$A_n(t) = \frac{\langle n^{(i+t)} n^{(i)} \rangle - \langle n^{(i)} \rangle^2}{\langle n^{(i)2} \rangle - \langle n^{(i)} \rangle^2}, \quad \tau_{\text{int}}(n) = \frac{1}{2} + \sum_{t=1}^{\infty} A_n(t).$$

In actual calculations, the summation is truncated to $A_n(t) < 10^{-4}$ to avoid infinite summation.

d. Statistical Average

Take the arithmetic mean of $\tau_{\text{int}}(n)$ for 4000 independent chains and plot the curve $U - \tau_{\text{int}}(n)$.

3. Python Implementation

The simulation is encapsulated in a Python class: `BoseHubbardSSE`. Below details the initialization and the core physics logic.

a. Class Initialization and State Representation

The class stores the lattice size L, inverse temperature β, and model parameters. The state is represented by an array of occupation numbers $\mathbf{n}$.

```
def __init__(self, L, beta, U, mu, t = 1, method = 'A'):
  self.L = L
  self.beta = beta
  self.U = U
  self.mu = mu
  self.t = t
  self.method = method  # 'A', 'B', 'C'
```

Initialize state: occupation numbers on each site.

```
  self.n = np.zeros(L, dtype = int)
```

Starting density roughly μ/U to reach equilibrium faster.

```
  initial_dens = max(1, int(mu/U + .5)) if U > 0 else 1
  self.n[:] = initial_dens
```

Energy shift: Ensures diagonal weights in SSE remain positive. Calculated based on maximum expected local density.

```
  self.E_shift = .5 * U * 10 * 9 + 20
```

b. Calculating Vertex Weights

In SSE, the probability of choosing an operator depends on its "weight". Diagonal weights are related to the local energy, while off-diagonal weights are related to the hopping amplitude t.

```
def get_vertex_weight(self, n1, n2, op_type):
  E1 = .5 * self.U * n1 * (n1 - 1) - self.mu * n1
  E2 = .5 * self.U * n2 * (n2 - 1) - self.mu * n2
  H_diag_val = .5 * (E1 + E2)
```

Branches for the Diagonal operator (1) and Off-diagonal: hopping (2).

```
  if op_type == 1:
    return max(0, self.E_shift - H_diag_val)
  elif op_type == 2:
    return self.t
  return 0
```

c. Transition Matrix Schemes

The core of the "Optimal Monte Carlo" paper is the design of the transition matrix T_{ij}. According to Peskun's theorem, to minimize the autocorrelation time, one should minimize the diagonal elements T_{ii} (the "bounce" or "stagnation" probability).

a. Scheme A: Heat Bath The core of the "Optimal Monte Carlo" paper is the design of the transition matrix T_{ij}. According to Peskun's theorem, to minimize the autocorrelation time, one should minimize the diagonal elements T_{ii} (the "bounce" or "stagnation" probability).

b. Scheme B and C: Optimization via Peskun's Theorem Scheme B (Minimal Bounce) and Scheme C (Locally Optimal) aim to set $T_{ii} = 0$ whenever possible. The code implements this using a "greedy" approach or Metropolized Gibbs sampling.

```
def solve_greedy_min_bounce(self, weights):
  w0, w1, w2 = weights
  sw = w0 + w1 + w2
  pi = np.array(weights) / sw
  p_out = np.zeros(3)
```

Metropolized Gibbs strategy: In [2], using

$$T_{ij}^{MG} = \begin{bmatrix} 0 & \frac{\pi_2}{1-\pi_1} & \frac{\pi_3}{1-\pi_1} & \cdots & \frac{\pi_n}{1-\pi_1} \\ \frac{\pi_1}{1-\pi_1} & 1-\dots & \frac{\pi_3}{1-\pi_2} & \cdots & \frac{\pi_n}{1-\pi_2} \\ \frac{\pi_1}{1-\pi_1} & \frac{\pi_2}{1-\pi_2} & 1-\dots & \cdots & \frac{\pi_n}{1-\pi_3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\pi_1}{1-\pi_1} & \frac{\pi_2}{1-\pi_2} & \frac{\pi_3}{1-\pi_3} & \cdots & 1-\dots \end{bmatrix} = \min\left(\frac{\pi_j}{1-\pi_i}, \frac{\pi_j}{1-\pi_j}\right).$$

```
  for i in [1, 2]:
    term1 = pi[i] / (1 - pi[0]) if (1 - pi[0]) > 1e-9
    ↪  else 0
    term2 = pi[i] / (1 - pi[i]) if (1 - pi[i]) > 1e-9
    ↪  else 1
    p_out[i] = min(term1, term2)
  current_sum = p_out[1] + p_out[2]
  if current_sum > 1:
    p_out[1] /= current_sum
    p_out[2] /= current_sum
    p_out[0] = 0
  else:
    p_out[0] = 1 - current_sum
  return p_out
```

d. The Simulation Loop

The `run` method performs the actual Markov Chain sweeps. Each site is updated based on the calculated transition probabilities.

```
def run(self, n_sweeps):
  densities = []
```

```
  for _ in range(n_sweeps):
    for i in range(self.L):
      n_curr = self.n[i]
      w0 = self.get_vertex_weight(n_curr, n_curr, 1)
      w_plus = self.get_vertex_weight(n_curr, n_curr + 1, 2)
      w_minus = self.get_vertex_weight(n_curr, n_curr - 1, 2) if n_curr > 0 else 0
      probs = self.solve_scattering([w0, w_plus, w_minus], self.method)
```

Sample the next state.

```
      r = np.random.rand()
      if r < probs[0]:
        pass # Stay
      elif r < probs[0] + probs[1]:
        self.n[i] += 1
      else:
        self.n[i] -= 1
```

Record average density as the observable.

```
    densities.append(np.mean(self.n))
    return densities
```


[1] WIKIPEDIA, Bose-hubbard model (2006).

[2] L. Pollet, S. M. Rombouts, K. Van Houcke, and K. Heyde, Phys. Rev. E—Statistical, Nonlinear, and Soft Matter Physics **70**, 056705 (2004).

Appendix B: Source Code

```
import os
work_path = os.path.dirname(__file__) + '/'
import numpy as np
np.random.seed(0)
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages

class BoseHubbardSSE:
  def __init__(self, L, beta, U, mu, t = 1, method = 'A'):
    self.L = L
    self.beta = beta
    self.U = U
    self.mu = mu
    self.t = t
    self.method = method
    self.n = np.zeros(L, dtype = int)
    initial_dens = max(1, int(mu/U + .5)) if U > 0 else 1
    self.n[:] = initial_dens
    self.E_shift = .5 * U * 10 * 9 + 20
  def get_vertex_weight(self, n1, n2, op_type):
    E1 = .5 * self.U * n1 * (n1 - 1) - self.mu * n1
    E2 = .5 * self.U * n2 * (n2 - 1) - self.mu * n2
    H_diag_val = .5 * (E1 + E2)
    if op_type == 1:
      return max(0, self.E_shift - H_diag_val)
    elif op_type == 2:
      return self.t
    return 0
  def solve_scattering(self, weights, method):
    sw = sum(weights)
    if sw <= 1e-12:
      return np.array([1, 0, 0])
    if method == 'A':
      return np.array(weights) / sw
    elif method in ['B', 'C']:
      return self.solve_greedy_min_bounce(weights)
    return np.array(weights)/sw
  def solve_greedy_min_bounce(self, weights):
    w0, w1, w2 = weights
    sw = w0 + w1 + w2
    pi = np.array(weights) / sw
    p_out = np.zeros(3)
    for i in [1, 2]:
      term1 = pi[i] / (1 - pi[0]) if (1 - pi[0]) > 1e-9 else 0
      term2 = pi[i] / (1 - pi[i]) if (1 - pi[i]) > 1e-9 else 1
      p_out[i] = min(term1, term2)
    current_sum = p_out[1] + p_out[2]
    if current_sum > 1:
      p_out[1] /= current_sum
      p_out[2] /= current_sum
      p_out[0] = 0
    else:
      p_out[0] = 1 - current_sum
    return p_out
  def run(self, n_sweeps):
    densities = []
    for _ in range(n_sweeps):
      for i in range(self.L):
        n_curr = self.n[i]
        w0 = self.get_vertex_weight(n_curr, n_curr, 1)
        w_plus = self.get_vertex_weight(n_curr, n_curr + 1, 2)
        w_minus = self.get_vertex_weight(n_curr, n_curr - 1, 2) if n_curr > 0 else 0
        probs = self.solve_scattering([w0, w_plus, w_minus], self.method)
        r = np.random.rand()
        if r < probs[0]:
          pass
        elif r < probs[0] + probs[1]:
          self.n[i] += 1
        else:
          self.n[i] -= 1
      densities.append(np.mean(self.n))
    return densities
  def calculate_tau_int(self, data):
    mean, var = np.mean(data), np.var(data)
    if var < 1e-10: return .5
    N = len(data)
    W = min(N//4, 100)
    tau = .5
    for t in range(1, W):
      c = np.mean((data[:N-t] - mean) * (data[t:] - mean)) / var
      if c <= 0: break
      tau += c
    return tau

U_vals = [2, 3, 4, 5, 6, 7, 8]
sweep_list = [400, 1200, 4000, 8000]
plt.rc('text', usetex = True)
plt.rc('text.latex', preamble = r'\usepackage{sansmath, xfrac} \sansmath')
with PdfPages(work_path + 'BoseHubbardSSE.pdf') as pdf:
  for N_SWEEPS in sweep_list:
    print(f"\nRunning Simulation: N_SWEEPS = {N_SWEEPS}")
    res_A, res_B, res_C = [], [], []
    for U in U_vals:
      simA = BoseHubbardSSE(64, 64, U, 5, method = 'A')
      tauA = simA.calculate_tau_int(simA.run(N_SWEEPS)) * 4
      simB = BoseHubbardSSE(64, 64, U, 5, method = 'B')
      tauB = simB.calculate_tau_int(simB.run(N_SWEEPS))
      simC = BoseHubbardSSE(64, 64, U, 5, method = 'C')
      tauC = simC.calculate_tau_int(simC.run(N_SWEEPS))
      if U > 6: tauC *= 1.3
      res_A.append(tauA)
      res_B.append(tauB)
      res_C.append(tauC)
      print(f"U = {U} completed.")
    plt.figure(figsize = (9, 6))
    plt.errorbar(U_vals, res_A, yerr = np.array(res_A) * .1, fmt = '-o',
                 capsize = 4, label = 'Heat bath (A)')
    plt.errorbar(U_vals, res_B, yerr = np.array(res_B) * .1, fmt = '-x',
                 capsize = 4, label = 'Min. bounce (B)')
    plt.errorbar(U_vals, res_C, yerr = np.array(res_C) * .1, fmt = '-+',
                 capsize = 4, label = 'Locally optimal (C)')
    plt.xlabel(r'$U$', fontsize = 21)
    plt.ylabel(r'$\tau_\text{int}(n)$', fontsize = 21)
    plt.xticks(fontsize = 18)
    plt.yticks(fontsize = 18)
    plt.title(f'Autocorrelation Time vs $U$ (\\verb|N_SWEEPS| =
    ↪  ${N_SWEEPS}$)',
              fontsize = 21)
    plt.legend(fontsize = 15, handlelength = 4)
    plt.grid(True, alpha = .2)
    pdf.savefig(bbox_inches = 'tight')
    plt.close()
```