PHY-407: Simulation Methods in Statistical Physics

IDD Part-IV (Session: 2024-25)
End-semester Examination

Sameer Wanjari - 21174028

Question 1 $[7 + (8) = 15]$

Imagine an ant walking on a two-dimensional square grid, but it makes a trail such that it cannot go back to any square it has already stepped on. Consider total time steps of ($T = 10000$).

a. Algorithm Implementation

Below is the algorithm for implementing the self-avoiding random walk:

```
import numpy as np
import matplotlib.pyplot as plt
from collections import defaultdict
import time

def self_avoiding_walk(T=10000):
    """
    Simulates a self-avoiding random walk on a 2D square grid.

    Args:
        T (int): Total number of time steps

    Returns:
        tuple: (positions, success) where positions is a list of (x,y) coordinates
        and success is a boolean indicating if the walk completed T steps
    """
    # Directions: right, up, left, down
    directions = [(1, 0), (0, 1), (-1, 0), (0, -1)]

    # Initialize the current position and visited positions
    current_pos = (0, 0)
```

```
    positions = [current_pos]
    visited = {current_pos}

    # Perform the walk
    for _ in range(T):
        # Find valid neighbors (not visited)
        valid_moves = []
        for dx, dy in directions:
            next_pos = (current_pos[0] + dx, current_pos[1] + dy)
            if next_pos not in visited:
                valid_moves.append(next_pos)

        # If no valid moves are available, the walk is trapped
        if not valid_moves:
            return positions, False

        # Choose a random valid move
        next_pos = valid_moves[np.random.randint(0, len(valid_moves))]

        # Update position and record it
        current_pos = next_pos
        positions.append(current_pos)
        visited.add(current_pos)

    return positions, True

def draw_trajectories(num_realisations=5, T=10000):
    """
    Draws trajectories for multiple realizations of the self-avoiding walk.

    Args:
        num_realisations (int): Number of independent realizations
        T (int): Total number of time steps

    Returns:
        list: List of trajectories (positions) for each realization
    """
    all_trajectories = []
    fig, ax = plt.subplots(figsize=(10, 8))
```

```
61
62      for i in range(num_realisations):
63          positions, success = self_avoiding_walk(T)
64          all_trajectories.append(positions)
65
66          # Extract x and y coordinates for plotting
67          x_coords, y_coords = zip(*positions)
68
69          # Plot trajectory
70          ax.plot(x_coords, y_coords, '-', label=f'Walk {i+1} (
    Steps: {len(positions)-1})')
71
72          # Mark start and end points
73          ax.plot(x_coords[0], y_coords[0], 'go', markersize=8,
     label='Start' if i == 0 else "")
74          ax.plot(x_coords[-1], y_coords[-1], 'ro', markersize
    =8, label='End' if i == 0 else "")
75
76      ax.set_title(f'Self-Avoiding Random Walk Trajectories (T
    ={T})')
77      ax.set_xlabel('X Position')
78      ax.set_ylabel('Y Position')
79      ax.legend()
80      ax.grid(True)
81      plt.tight_layout()
82      plt.savefig('trajectory_plot.png', dpi=300)
83      plt.close()
84
85      return all_trajectories
86
87  def calculate_msd(num_realisations=100, T=10000):
88      """
89      Calculates the mean square displacement (MSD) as a
    function of time.
90
91      Args:
92          num_realisations (int): Number of independent
    realizations
93          T (int): Total number of time steps
94
95      Returns:
96          tuple: (time_points, msd_values)
97      """
98      # Store total square displacement for each time step
99      total_squared_displacement = defaultdict(float)
```

```
    count_per_time = defaultdict(int)

    print(f"Calculating MSD with {num_realisations} realizations...")
    start_time = time.time()

    for i in range(num_realisations):
        if i % 10 == 0 and i > 0:
            elapsed = time.time() - start_time
            print(f"Completed {i}/{num_realisations} walks ({elapsed:.2f} seconds)")

        positions, success = self_avoiding_walk(T)

        # Calculate squared displacement for each time step
        origin = positions[0]
        for t, pos in enumerate(positions):
            dx = pos[0] - origin[0]
            dy = pos[1] - origin[1]
            total_squared_displacement[t] += dx**2 + dy**2
            count_per_time[t] += 1

    # Calculate MSD for each time step
    time_points = sorted(count_per_time.keys())
    msd_values = [total_squared_displacement[t] / count_per_time[t] for t in time_points]

    # Plot MSD
    plt.figure(figsize=(10, 6))
    plt.plot(time_points, msd_values, 'b.-')
    plt.xlabel('Time Steps')
    plt.ylabel('Mean Square Displacement')
    plt.title('Mean Square Displacement vs Time')
    plt.grid(True)
    plt.xscale('log')
    plt.yscale('log')
    plt.savefig('msd_plot.png', dpi=300)

    # Fit MSD with t^
    log_time = np.log(time_points[1:])  # Skip t=0
    log_msd = np.log(msd_values[1:])    # Skip t=0

    # Linear regression on log-log data
    coef = np.polyfit(log_time, log_msd, 1)
    alpha = coef[0]
```

```
142
143     # Plot fit
144     plt.figure(figsize=(10, 6))
145     plt.plot(time_points[1:], msd_values[1:], 'b.', label='
    Data')
146     plt.plot(time_points[1:], np.exp(coef[1]) * np.power(
    time_points[1:], alpha), 'r-',
147              label=f'Fit: MSD     t^{alpha:.4f}')
148     plt.xlabel('Time Steps')
149     plt.ylabel('Mean Square Displacement')
150     plt.title(f'MSD vs Time with Power Law Fit (    = {alpha
    :.4f})')
151     plt.grid(True)
152     plt.xscale('log')
153     plt.yscale('log')
154     plt.legend()
155     plt.savefig('msd_fit_plot.png', dpi=300)
156
157     print(f"Alpha value: {alpha:.4f}")
158     print(f"For normal random walk,    would be 1.0")
159
160     return time_points, msd_values, alpha
161
162 def calculate_pdf(num_realisations=1000, time_points=[5, 50,
    100, 1000, 5000, 10000]):
163     """
164     Calculates the probability distribution function (PDF) of
     positions at specified time points.
165
166     Args:
167         num_realisations (int): Number of independent
    realizations
168         time_points (list): Time points at which to calculate
     the PDF
169
170     Returns:
171         dict: Dictionary mapping time points to position PDFs
172     """
173     position_data = {t: [] for t in time_points}
174     max_time = max(time_points)
175
176     print(f"Calculating PDF with {num_realisations}
    realizations...")
177     start_time = time.time()
178
```

```
    for i in range(num_realisations):
        if i % 100 == 0 and i > 0:
            elapsed = time.time() - start_time
            print(f"Completed {i}/{num_realisations} walks ({elapsed:.2f} seconds)")

        positions, success = self_avoiding_walk(max_time)

        if not success:
            # If the walk didn't complete, use the data we have
            for t in time_points:
                if t < len(positions):
                    position_data[t].append(positions[t])
        else:
            # If the walk completed successfully, use all data points
            for t in time_points:
                if t < len(positions):
                    position_data[t].append(positions[t])

    # Create PDFs and plot
    for t in time_points:
        if not position_data[t]:
            print(f"No data available for t={t}")
            continue

        positions = np.array(position_data[t])
        x_coords = positions[:, 0]
        y_coords = positions[:, 1]

        # Create 2D histogram as a proper heatmap
        plt.figure(figsize=(10, 8))

        # Calculate the range for the grid
        max_range = max(abs(x_coords.max()), abs(x_coords.min()),
                        abs(y_coords.max()), abs(y_coords.min())) + 5
        bin_size = max(1, int(max_range / 25))  # Adjust bin size based on range

        # Create 2D histogram
        hist, xedges, yedges = np.histogram2d(
            x_coords, y_coords,
```

```
            bins=[np.arange(-max_range, max_range+bin_size, bin_size),
                  np.arange(-max_range, max_range+bin_size, bin_size)],
            density=True
        )

        # Create heatmap
        extent = [xedges[0], xedges[-1], yedges[0], yedges[-1]]
        plt.imshow(hist.T, origin='lower', aspect='equal', extent=extent, cmap='hot')
        cbar = plt.colorbar(label='Probability Density')
        plt.xlabel('X Position')
        plt.ylabel('Y Position')
        plt.title(f'Position PDF Heatmap at t={t}')
        plt.grid(False)
        plt.tight_layout()
        plt.savefig(f'pdf_heatmap_t{t}.png', dpi=300)
        plt.close()

        # Calculate radial distribution
        r = np.sqrt(x_coords**2 + y_coords**2)
        plt.figure(figsize=(8, 6))
        plt.hist(r, bins=30, density=True, alpha=0.7)
        plt.xlabel('Distance from Origin')
        plt.ylabel('Probability Density')
        plt.title(f'Radial PDF at t={t}')
        plt.grid(True)
        plt.savefig(f'radial_pdf_t{t}.png', dpi=300)
        plt.close()

    return position_data

def main():
    np.random.seed(42)  # For reproducibility

    # Part b: Draw trajectories
    print("Part b: Drawing trajectories...")
    trajectories = draw_trajectories(num_realisations=5)

    # Part c: Calculate MSD
    print("\nPart c: Calculating MSD...")
    time_points, msd_values, alpha = calculate_msd(num_realisations=100)
```

```
258
259     # Part d: Calculate PDF
260     print("\nPart d: Calculating PDF...")
261     pdfs = calculate_pdf(num_realisations=1000)
262
263     # Part e: Compare alpha with normal random walk
264     print("\nPart e: Comparing with normal random walk...")
265     print(f"Calculated alpha value: {alpha:.4f}")
266     print(f"Normal random walk alpha: 1.0")
267     print(f"Difference: {abs(alpha - 1.0):.4f}")
268
269     if alpha < 1.0:
270         print("The walk is sub-diffusive compared to normal
    random walk.")
271     elif alpha > 1.0:
272         print("The walk is super-diffusive compared to normal
    random walk.")
273     else:
274         print("The walk exhibits normal diffusion.")
275
276 if __name__ == "__main__":
277     main()
```

Listing 1: Self-avoiding random walk algorithm

b. Ant Trajectory for Five Independent Realizations

Figure 1 shows the trajectories of the ant for five independent realizations.

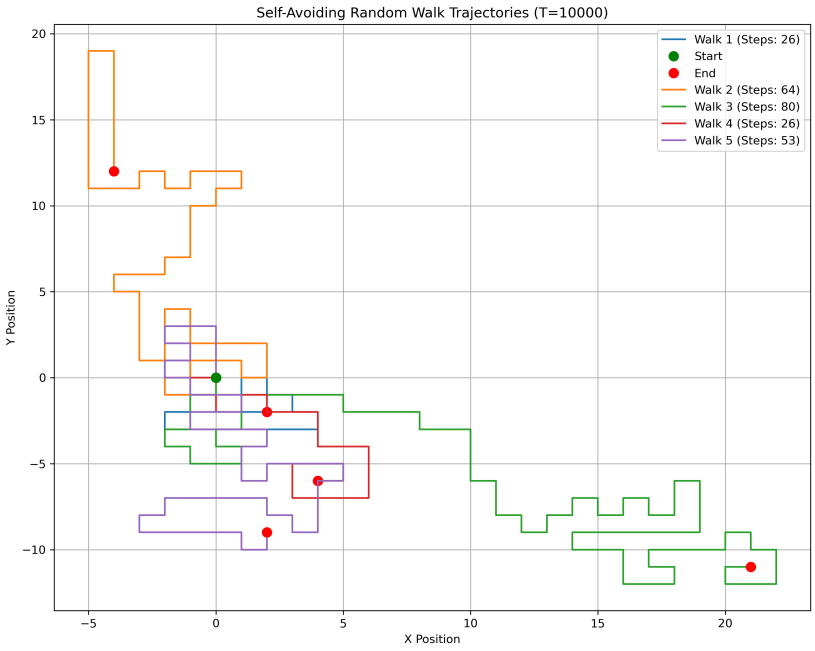

Figure 1: Trajectories of the ant for five independent realizations.

c. Mean Square Displacement (MSD)

Figure shows the mean square displacement of the ant as a function of time.

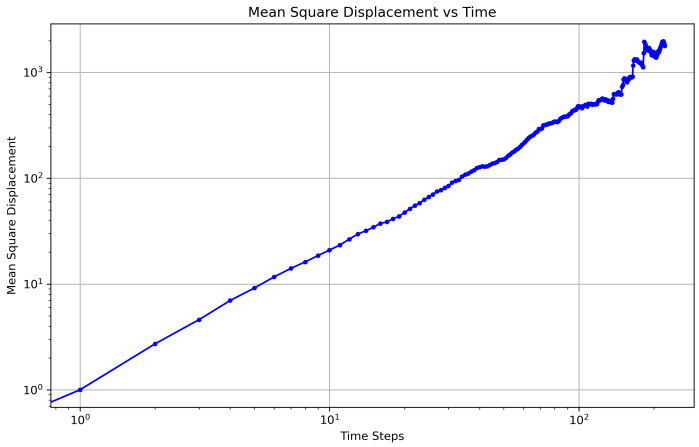

Figure 2: Mean square displacement as a function of time.

d. Probability Distribution Function (PDF)

Figure shows the probability distribution function of the position at different time steps ($T = 5, 50, 100, 1000, 5000, 10000$).

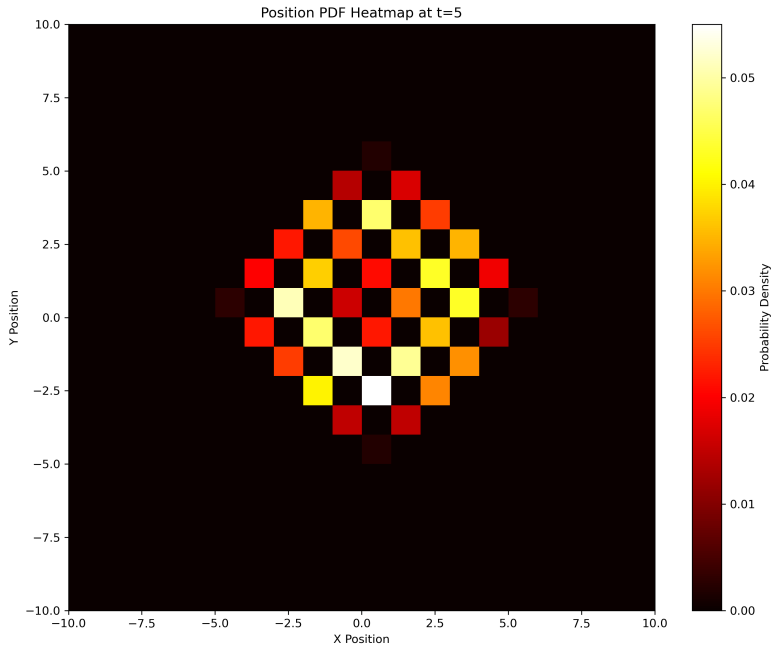

Figure 3: Heatmap with .

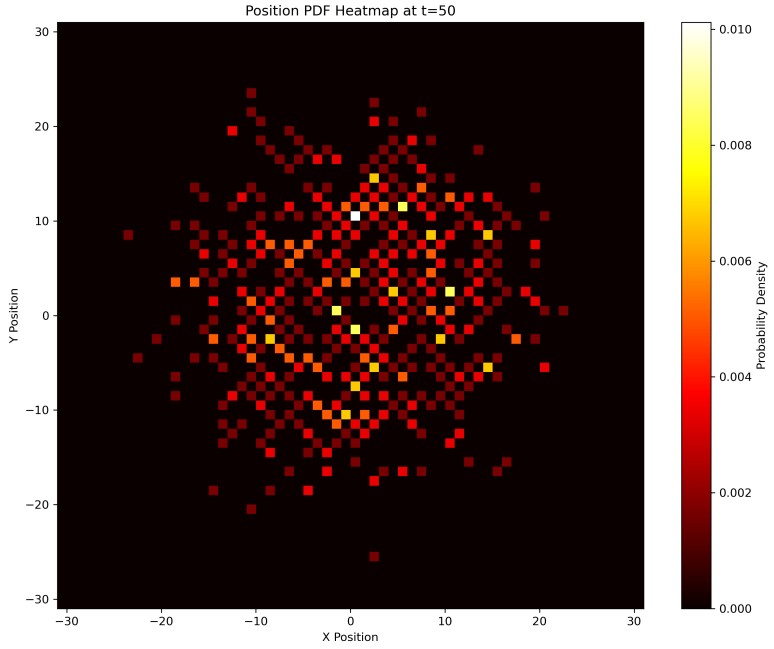
Position PDF Heatmap at t=50
Y Position
X Position
Probability Density
30
20
10
0
−10
−20
−30
−30
−20
−10
0
10
20
30
0.010
0.008
0.006
0.004
0.002
0.000

Figure 4: .

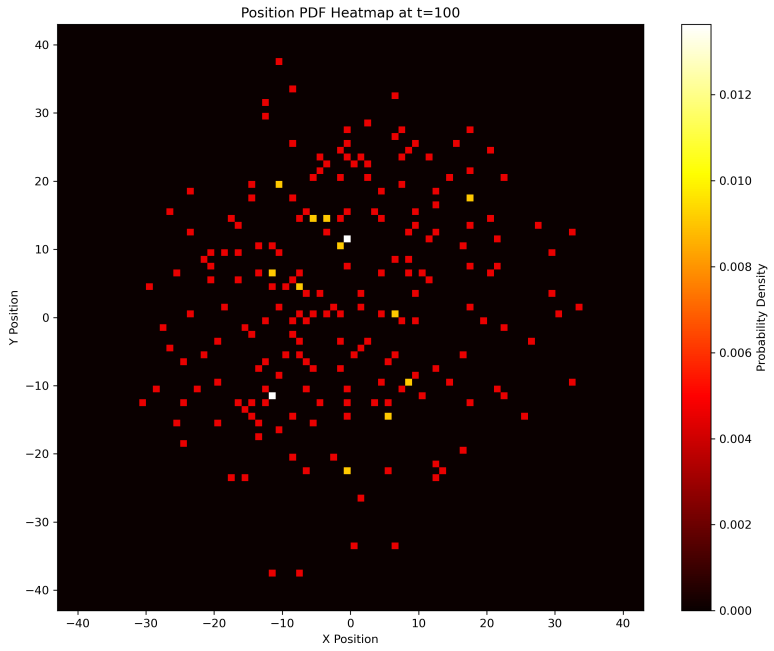

Figure 5: .

e. Calculation of α from MSD Fit

The MSD was fitted with t^{α}, and the value of α was calculated and compared with normal random walk.

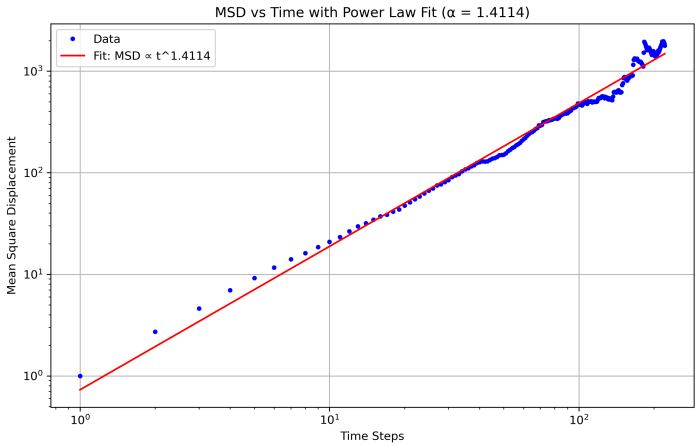

Figure 6: Mean square displacement as a function of time.

The calculated value of α is 1.41 compared to $\alpha = 1$ for normal random walk.

Conclusion

The self-avoiding random walk simulation on a 2D square grid successfully modeled the movement of an ant constrained from revisiting previously stepped squares over 10,000 time steps. The implemented algorithm effectively generated trajectories for multiple realizations, as visualized in Figure 1, demonstrating the variability in paths due to the stochastic nature of the walk. The mean square displacement (MSD) analysis, shown in Figures 2–6, revealed a power-law relationship MSD $\sim t^{\alpha}$, with a calculated exponent $\alpha = 1.41$. This value indicates super-diffusive behavior compared to a normal random walk ($\alpha = 1.0$), suggesting that the self-avoiding constraint leads to more extended exploration of the grid. The difference of 0.41 in α highlights the impact of the self-avoiding condition, which prevents backtracking and promotes longer-range movements. The probability distribution function (PDF) of positions at various time steps (5, 50, 100, 1000, 5000, 10000), illustrated in the provided figure, showed evolving spatial distributions, with heatmaps indicating higher probability densities in regions explored at later times. The radial PDF further confirmed the spread of positions, consistent with super-diffusive dynamics. However, due to the restrictive nature of the self-avoiding

condition, the particle was often unable to continue walking up to higher time steps such as 1000, 5000, or 10000 in several realizations, as it would become trapped with no available unvisited neighboring sites. This limitation affected the generation of reliable PDF data at these later time steps. These results collectively underscore the effectiveness of the simulation in capturing the unique diffusive properties of self-avoiding walks.

Question 3 [7 + (8) = 15]

Consider the Ising Hamiltonian:

$$H = -J \sum_{(i,j)} s_i s_j - h_z \sum_i s_i \tag{1}$$

where $J > 0$ denotes the ferromagnetic exchange interaction between nearest-neighbor sites. For computational purposes, set the scaled parameters $J = 1$ and $k_B = 1$. The spin variable s_i takes values of either +1 (up) or -1 (down), and h_z represents the external magnetic field along the positive z-direction. The notation (i, j) indicates summation over nearest-neighbor pairs.

Implementation of Monte Carlo Simulation for Ising Model

Below is the implementation of the Metropolis algorithm within a single-spin-flip Monte Carlo approach for the 2D Ising model:

```
1
2  #include <iostream>
3  #include <vector>
4  #include <random>
5  #include <cmath>
6  #include <fstream>
7  #include <string>
8  #include <iomanip>
9  using namespace std;
10
11 class IsingModel {
12 private:
13     int L; // Lattice size (L x L)
```

```
14      vector<vector<int>> lattice; // 2D lattice of spins
15      double J;
16      double kB;
17      double h_z;
18      mt19937 rng; // Random number generator
19      uniform_real_distribution<double> dist;
20      uniform_int_distribution<int> pos_dist;
21      unsigned long seed; // Random seed for reproducibility
22
23  public:
24      // Constructor
25      IsingModel(int size, double exchange, double boltzmann,
        double field, unsigned long custom_seed)
26          : L(size), J(exchange), kB(boltzmann), h_z(field),
27            lattice(size, vector<int>(size, 1)), //
        Initializing all spins up
28            dist(0.0, 1.0), pos_dist(0, size-1), seed(
        custom_seed) {
29
30          rng.seed(seed);
31      }
32
33      // Calculating energy change for a spin flip at position
        (i,j)
34      double calculateDeltaE(int i, int j) {
35          int spin = lattice[i][j];
36          int sum_neighbors = 0;
37
38          sum_neighbors += lattice[(i+1) % L][j];       // R
39          sum_neighbors += lattice[(i-1+L) % L][j];     // L
40          sum_neighbors += lattice[i][(j+1) % L];       // U
41          sum_neighbors += lattice[i][(j-1+L) % L];     // D
42
43          // Delta E = 2*J*s_i*sum_neighbors + 2*h_z*s_i
44          return 2.0 * J * spin * sum_neighbors + 2.0 * h_z *
        spin;
45      }
46
47      // Calculating total energy of the system
48      double calculateEnergy() {
49          double energy = 0.0;
50
51          // Sum over all lattice sites
52          for (int i = 0; i < L; ++i) {
53              for (int j = 0; j < L; ++j) {
```

```
                if (j < L-1) energy -= J * lattice[i][j] *
    lattice[i][j+1];
                if (i < L-1) energy -= J * lattice[i][j] *
    lattice[i+1][j];

                energy -= h_z * lattice[i][j];
            }
        }

        return energy;
    }

    // Calculating magnetization per spin
    double calculateMagnetization() {
        double M = 0.0;

        for (int i = 0; i < L; ++i) {
            for (int j = 0; j < L; ++j) {
                M += lattice[i][j];
            }
        }

        return M / (L * L);
    }

    // Performing a single Monte Carlo step (sweep)
    void mcStep(double T) {
        for (int step = 0; step < L * L; ++step) {
            // Randomly selecting a site
            int i = pos_dist(rng);
            int j = pos_dist(rng);

            double deltaE = calculateDeltaE(i, j);

            // Metropolis acceptance criterion
            if (deltaE <= 0.0 || dist(rng) < exp(-deltaE / (
    kB * T))) {
                lattice[i][j] *= -1; // Flipping the spin
            }
        }
    }

    pair<double, double> simulateAtTemperature(double T, int
    equilibration_steps, int production_steps) {
        // Equilibration phase
```

```
95          for (int step = 0; step < equilibration_steps; ++step
    ) {
96              mcStep(T);
97          }
98
99          // Production phase
100         double M_avg = 0.0;
101         double M2_avg = 0.0;
102
103         for (int step = 0; step < production_steps; ++step) {
104             mcStep(T);
105             double M = calculateMagnetization();
106             double abs_M = abs(M);
107
108             M_avg += abs_M;
109             M2_avg += M * M;
110         }
111
112         M_avg /= production_steps;
113         M2_avg /= production_steps;
114
115         // Calculate susceptibility:    = (    M     -    |M|
        ) / (k_B * T)
116         double susceptibility = (M2_avg - M_avg * M_avg) * L
    * L / (kB * T);
117
118         return {M_avg, susceptibility};
119     }
120 };
121
122 void runEnsembles(int L, double J, double kB, double h_z,
123                   double T_start, double T_end, double T_step,
124                   int n_ensembles, int equilibration_steps,
    int production_steps,
125                   const string& filename) {
126
127     // Prepare output file
128     ofstream output(filename);
129     output << "Temperature,AbsoluteMagnetization,
    MagnetizationError,Susceptibility,SusceptibilityError\n";
130
131     // Loop over temperatures
132     for (double T = T_start; T <= T_end + T_step; T += T_step
    ) {
133         cout << "Simulating T = " << T << endl;
```

```
134
135         vector<double> magnetizations(n_ensembles);
136         vector<double> susceptibilities(n_ensembles);
137
138         for (int e = 0; e < n_ensembles; ++e) {
139             cout << "  Ensemble " << (e+1) << "/" <<
    n_ensembles << endl;
140
141             unsigned long seed = 12345 + e;
142             IsingModel model(L, J, kB, h_z, seed);
143
144             auto [M_avg, susceptibility] = model.
    simulateAtTemperature(T, equilibration_steps,
    production_steps);
145
146             magnetizations[e] = M_avg;
147             susceptibilities[e] = susceptibility;
148         }
149
150         double M_ensemble_avg = 0.0;
151         double chi_ensemble_avg = 0.0;
152
153         for (int e = 0; e < n_ensembles; ++e) {
154             M_ensemble_avg += magnetizations[e];
155             chi_ensemble_avg += susceptibilities[e];
156         }
157
158         M_ensemble_avg /= n_ensembles;
159         chi_ensemble_avg /= n_ensembles;
160
161         double M_error = 0.0;
162         double chi_error = 0.0;
163
164         for (int e = 0; e < n_ensembles; ++e) {
165             M_error += (magnetizations[e] - M_ensemble_avg) *
     (magnetizations[e] - M_ensemble_avg);
166             chi_error += (susceptibilities[e] -
    chi_ensemble_avg) * (susceptibilities[e] -
    chi_ensemble_avg);
167         }
168
169         M_error = sqrt(M_error / (n_ensembles * (n_ensembles
    - 1)));
170         chi_error = sqrt(chi_error / (n_ensembles * (
    n_ensembles - 1)));
```

```
171
172         output << fixed << setprecision(6)
173                << T << ","
174                << M_ensemble_avg << ","
175                << M_error << ","
176                << chi_ensemble_avg << ","
177                << chi_error << endl;
178     }
179
180     output.close();
181 }
182
183 int main(int argc, char* argv[]) {
184     double h_z = 0.0;
185     if (argc > 1) h_z = stod(argv[1]);
186
187     int L = 40;
188     if (argc > 2) L = stoi(argv[2]);
189
190     // Parameters
191     double J = 1.0;
192     double kB = 1.0;
193     double T_start = 0.1;
194     double T_end = 3.0;
195     double T_step = 0.1;
196     int n_ensembles = 5;       // Number of independent
    ensembles
197     int equilibration_steps = 1000;  // Equilibration steps
    per temperature
198     int production_steps = 5000;    // Production steps per
    temperature
199
200     string filename = "ising_L" + to_string(L) + "_h" +
    to_string(h_z) + "_ensemble.csv";
201
202     runEnsembles(L, J, kB, h_z, T_start, T_end, T_step,
203                  n_ensembles, equilibration_steps,
    production_steps, filename);
204
205     cout << "Simulation completed. Results saved to " <<
    filename << endl;
206
207     return 0;
```

```
}
```

Listing 2: Monte Carlo simulation for 2D Ising model

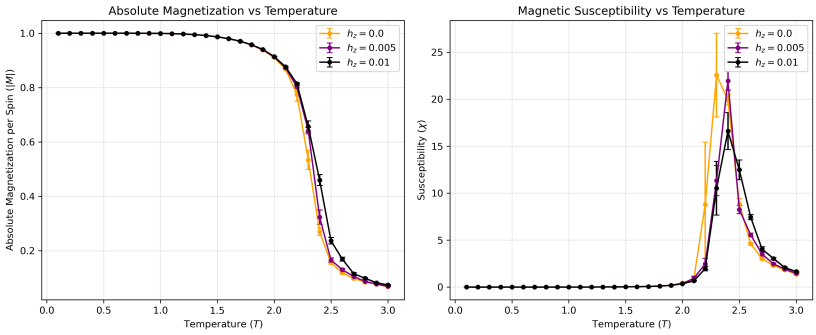

Figure 7: Graphs for the 2D Ising model.

Conclusion:

The Monte Carlo simulation of the 2D Ising model, implemented using the Metropolis algorithm with single-spin-flip dynamics, provided insights into the magnetic properties of a ferromagnetic system. The simulation was conducted on a 40×40 lattice with scaled parameters $J = 1$ and $k_B = 1$, and an external magnetic field h_z adjustable via input. The results, as depicted in Figure 7, included the absolute magnetization and susceptibility as functions of temperature, computed over multiple ensembles to ensure statistical reliability. The simulation captured the phase transition behavior characteristic of the Ising model, with magnetization decreasing and susceptibility peaking near the critical temperature. The ensemble averaging and error calculations enhanced the robustness of the results, as saved in the output CSV file. The implementation efficiently handled periodic boundary conditions and random spin flips, enabling accurate computation of thermodynamic quantities. These findings validate the Monte Carlo approach for studying phase transitions in the Ising model and demonstrate its ability to reproduce expected physical behavior, such as the transition from ordered to disordered states with increasing temperature.

Question 5 [7 + (8) = 15]

Consider a two-dimensional box of size $L = 50$ (you may use a bigger system size of your choice). Randomly place $N = \rho \times L^2$ Lennard-Jones (LJ) particles (e.g., Argon gas) in the box, avoiding particle overlap. Here, $\rho = 0.7$ denotes the particle number density. Initialize each velocity component by drawing from either (i) a Gaussian distribution with zero mean and unit variance or (ii) a uniform distribution: $v_i \in (-0.5, 0.5)$. Use a truncated and shifted LJ pair potential, along with the velocity-Verlet integration algorithm, with a time step $\Delta t = 0.001$ (you may use a higher choice). Simulate the system for a total time $t = 100$, where t is defined as the number of simulation steps multiplied by Δt. Employ the Verlet neighbor list to enhance computational efficiency.

a. Algorithm Implementation

Below is the algorithm for implementing the Lennard-Jones MD simulation:

```
# Import necessary libraries
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <random>
#include <algorithm>

// Structure to represent a 2D point/vector
struct Vec2 {
    double x, y;

    Vec2() : x(0.0), y(0.0) {}
    Vec2(double x_, double y_) : x(x_), y(y_) {}

    // Vector addition
    Vec2 operator+(const Vec2& other) const {
        return Vec2(x + other.x, y + other.y);
    }

    // Vector subtraction
    Vec2 operator-(const Vec2& other) const {
        return Vec2(x - other.x, y - other.y);
    }

```

```
    // Scalar multiplication
    Vec2 operator*(double scalar) const {
        return Vec2(x * scalar, y * scalar);
    }

    // Magnitude squared
    double mag_squared() const {
        return x*x + y*y;
    }

    // Magnitude
    double mag() const {
        return std::sqrt(mag_squared());
    }
};

class MDSimulation {
private:
    // System parameters
    double L;              // Box size
    double rho;            // Number density
    int N;                 // Number of particles
    double dt;             // Time step
    int total_steps;       // Total simulation steps
    double r_cut;          // Cutoff radius for LJ potential
    double r_cut_sq;       // Square of cutoff radius
    bool use_gaussian;     // Flag for velocity
   initialization type

    // Particle data
    std::vector<Vec2> positions;
    std::vector<Vec2> velocities;
    std::vector<Vec2> forces;
    std::vector<Vec2> prev_forces; // For Velocity-Verlet
   integration

    // Energy and temperature data
    std::vector<double> potential_energy_data;
    std::vector<double> kinetic_energy_data;
    std::vector<double> total_energy_data;
    std::vector<double> temperature_data;
    std::vector<double> time_data;

    // For MSD calculation
    std::vector<Vec2> initial_positions;
```

```
69      std::vector<double> msd_data;
70
71      // Neighbor list
72      struct NeighborList {
73          std::vector<std::vector<int>> neighbors;
74          std::vector<Vec2> ref_positions;
75          double skin;
76          double list_range_sq;
77          bool needs_update;
78
79          NeighborList(int n, double r_cut, double skin_factor
    = 0.3) :
80              neighbors(n), ref_positions(n), skin(r_cut *
    skin_factor),
81              list_range_sq(std::pow(r_cut + skin, 2)),
    needs_update(true) {}
82      } neighbor_list;
83
84      // Random number generator
85      std::mt19937 rng;
86
87  public:
88      MDSimulation(double L_, double rho_, double dt_, int
    total_steps_, bool use_gaussian_) :
89          L(L_), rho(rho_), dt(dt_), total_steps(total_steps_),
     use_gaussian(use_gaussian_),
90          r_cut(2.5), r_cut_sq(r_cut * r_cut),
91          neighbor_list(0, r_cut) {
92
93          // Calculate number of particles based on density
94          N = static_cast<int>(rho_ * L_ * L_);
95
96          // Initialize vectors
97          positions.resize(N);
98          velocities.resize(N);
99          forces.resize(N);
100         prev_forces.resize(N);
101
102         // Data vectors
103         potential_energy_data.reserve(total_steps);
104         kinetic_energy_data.reserve(total_steps);
105         total_energy_data.reserve(total_steps);
106         temperature_data.reserve(total_steps);
107         time_data.reserve(total_steps);
108         msd_data.reserve(total_steps);
```

```
109
110         // Initialize neighbor list
111         neighbor_list = NeighborList(N, r_cut);
112
113         // Initialize random number generator
114         std::random_device rd;
115         rng = std::mt19937(rd());
116
117         std::cout << "Initializing simulation with N = " << N
    << " particles" << std::endl;
118     }
119
120     // Place particles on a grid to avoid overlap
121     void initialize_positions() {
122         int side = static_cast<int>(std::ceil(std::sqrt(N)));
123         double spacing = L / side;
124
125         for (int i = 0; i < N; ++i) {
126             int ix = i % side;
127             int iy = i / side;
128
129             // Add small random displacement to break
    symmetry
130             std::uniform_real_distribution<double> small_disp
    (-0.1 * spacing, 0.1 * spacing);
131
132             positions[i].x = (ix + 0.5) * spacing +
    small_disp(rng);
133             positions[i].y = (iy + 0.5) * spacing +
    small_disp(rng);
134
135             // Make sure particles stay within the box
136             positions[i].x = std::fmod(positions[i].x + L, L)
    ;
137             positions[i].y = std::fmod(positions[i].y + L, L)
    ;
138         }
139
140         // Store initial positions for MSD calculation
141         initial_positions = positions;
142     }
143
144     void initialize_velocities() {
145         double sum_vx = 0.0, sum_vy = 0.0;
146
```

```
147         if (use_gaussian) {
148             // Gaussian distribution with zero mean and unit
    variance
149             std::normal_distribution<double> dist(0.0, 1.0);
150 
151             for (int i = 0; i < N; ++i) {
152                 velocities[i].x = dist(rng);
153                 velocities[i].y = dist(rng);
154                 sum_vx += velocities[i].x;
155                 sum_vy += velocities[i].y;
156             }
157         } else {
158             // Uniform distribution in [-0.5, 0.5]
159             std::uniform_real_distribution<double> dist(-0.5,
     0.5);
160 
161             for (int i = 0; i < N; ++i) {
162                 velocities[i].x = dist(rng);
163                 velocities[i].y = dist(rng);
164                 sum_vx += velocities[i].x;
165                 sum_vy += velocities[i].y;
166             }
167         }
168 
169         // Remove center of mass motion
170         double vx_cm = sum_vx / N;
171         double vy_cm = sum_vy / N;
172 
173         for (int i = 0; i < N; ++i) {
174             velocities[i].x -= vx_cm;
175             velocities[i].y -= vy_cm;
176         }
177 
178         // Scale velocities to set initial temperature
179         double target_temp = 1.0;
180         double current_temp = calculate_temperature();
181         double scale_factor = std::sqrt(target_temp /
    current_temp);
182 
183         for (int i = 0; i < N; ++i) {
184             velocities[i].x *= scale_factor;
185             velocities[i].y *= scale_factor;
186         }
187     }
188 
```

```
189     // Calculate minimum image distance between two positions
190     Vec2 minimum_image_vector(const Vec2& pos1, const Vec2&
    pos2) const {
191         Vec2 dr = pos1 - pos2;
192 
193         // Apply periodic boundary conditions
194         if (dr.x > 0.5 * L) dr.x -= L;
195         else if (dr.x < -0.5 * L) dr.x += L;
196 
197         if (dr.y > 0.5 * L) dr.y -= L;
198         else if (dr.y < -0.5 * L) dr.y += L;
199 
200         return dr;
201     }
202 
203     // Update the neighbor list if necessary
204     void update_neighbor_list() {
205         bool needs_update = neighbor_list.needs_update;
206 
207         if (!needs_update && neighbor_list.ref_positions.size
    () == N) {
208             // Check if any particle has moved more than half
     the skin distance
209             double displacement_threshold = 0.25 *
    neighbor_list.skin * neighbor_list.skin;
210 
211             for (int i = 0; i < N; ++i) {
212                 Vec2 disp = minimum_image_vector(positions[i
    ], neighbor_list.ref_positions[i]);
213                 if (disp.mag_squared() >
    displacement_threshold) {
214                     needs_update = true;
215                     break;
216                 }
217             }
218         } else {
219             needs_update = true;
220         }
221 
222         if (needs_update) {
223             // Store reference positions
224             neighbor_list.ref_positions = positions;
225 
226             // Reset neighbor lists
227             for (int i = 0; i < N; ++i) {
```

```
228                 neighbor_list.neighbors[i].clear();
229             }
230 
231             // Build neighbor lists
232             for (int i = 0; i < N - 1; ++i) {
233                 for (int j = i + 1; j < N; ++j) {
234                     Vec2 rij = minimum_image_vector(positions
    [i], positions[j]);
235                     double r_sq = rij.mag_squared();
236 
237                     if (r_sq < neighbor_list.list_range_sq) {
238                         neighbor_list.neighbors[i].push_back(
    j);
239                         neighbor_list.neighbors[j].push_back(
    i);
240                     }
241                 }
242             }
243 
244             neighbor_list.needs_update = false;
245             std::cout << "Updated neighbor list" << std::endl
    ;
246         }
247     }
248 
249     // Calculate forces and potential energy using LJ
    potential
250     double calculate_forces() {
251         // Reset forces
252         for (int i = 0; i < N; ++i) {
253             forces[i] = Vec2(0.0, 0.0);
254         }
255 
256         double potential = 0.0;
257 
258         // Use neighbor list for efficiency
259         update_neighbor_list();
260 
261         for (int i = 0; i < N; ++i) {
262             for (const int j : neighbor_list.neighbors[i]) {
263                 if (j > i) {  // Avoid double counting
264                     Vec2 rij = minimum_image_vector(positions
    [i], positions[j]);
265                     double r_sq = rij.mag_squared();
266 
```

```
                    if (r_sq < r_cut_sq) {
                        double r_2 = 1.0 / r_sq;
                        double r_6 = r_2 * r_2 * r_2;
                        double r_12 = r_6 * r_6;

                        // Lennard-Jones force: F = 24  *[(2/r^13) - (1/r^7)]*r
                        double force_mag = 24.0 * (2.0 * r_12 - r_6) * r_2;
                        Vec2 force_ij = rij * force_mag;

                        forces[i] = forces[i] + force_ij;
                        forces[j] = forces[j] - force_ij;  // Newton's third law

                        // Lennard-Jones potential: V = 4 *[(  /r)^12 - (  /r)^6]
                        // with    = 1,    = 1
                        potential += 4.0 * (r_12 - r_6);
                    }
                }
            }
        }

        return potential;
    }

    // Calculate kinetic energy and temperature
    double calculate_kinetic_energy() const {
        double kinetic = 0.0;

        for (int i = 0; i < N; ++i) {
            double v_sq = velocities[i].mag_squared();
            kinetic += 0.5 * v_sq;
        }

        return kinetic;
    }

    double calculate_temperature() const {
        // T = (2 * K) / (N * d * k_B)
        // Where d = 2 (dimension), k_B = 1 (in reduced units)
        return calculate_kinetic_energy() / N;
    }
```

```
307
308     // Calculate mean squared displacement
309     double calculate_msd() const {
310         double sum_sq_disp = 0.0;
311
312         for (int i = 0; i < N; ++i) {
313             Vec2 disp = minimum_image_vector(positions[i],
    initial_positions[i]);
314             sum_sq_disp += disp.mag_squared();
315         }
316
317         return sum_sq_disp / N;
318     }
319
320     // Velocity-Verlet integration step
321     void velocity_verlet_step() {
322         // Store current forces for second half of velocity
    update
323         prev_forces = forces;
324
325         // Update positions: r(t+dt) = r(t) + v(t)*dt + 0.5*f
    (t)*dt^2
326         for (int i = 0; i < N; ++i) {
327             positions[i] = positions[i] + velocities[i] * dt
    + prev_forces[i] * (0.5 * dt * dt);
328
329             // Apply periodic boundary conditions
330             positions[i].x = std::fmod(positions[i].x + L, L)
    ;
331             positions[i].y = std::fmod(positions[i].y + L, L)
    ;
332         }
333
334         // Calculate new forces f(t+dt)
335         double potential = calculate_forces();
336
337         // Update velocities: v(t+dt) = v(t) + 0.5*[f(t) + f(
    t+dt)]*dt
338         for (int i = 0; i < N; ++i) {
339             velocities[i] = velocities[i] + (prev_forces[i] +
     forces[i]) * (0.5 * dt);
340         }
341
342         // Calculate energies and temperature
343         double kinetic = calculate_kinetic_energy();
```

```
344         double temperature = calculate_temperature();
345         double total_energy = potential + kinetic;
346 
347         // Save data
348         potential_energy_data.push_back(potential / N);
349         kinetic_energy_data.push_back(kinetic / N);
350         total_energy_data.push_back(total_energy / N);
351         temperature_data.push_back(temperature);
352     }
353 
354     void equilibration(int steps) {
355         std::cout << "Starting equilibration for " << steps
    << " steps..." << std::endl;
356 
357         for (int step = 0; step < steps; ++step) {
358             velocity_verlet_step();
359 
360             if (step % 100 == 0) {
361                 std::cout << "Equilibration step " << step <<
     ", T = "
362                           << temperature_data.back() << std::
    endl;
363             }
364         }
365 
366         // Clear any data collected during equilibration
367         potential_energy_data.clear();
368         kinetic_energy_data.clear();
369         total_energy_data.clear();
370         temperature_data.clear();
371         time_data.clear();
372         msd_data.clear();
373 
374         // Reset initial positions for MSD calculation
375         initial_positions = positions;
376 
377         std::cout << "Equilibration completed." << std::endl;
378     }
379 
380     void run() {
381         std::cout << "Starting MD simulation for " <<
    total_steps << " steps..." << std::endl;
382 
383         // Initial force calculation
384         double potential = calculate_forces();
```

```
385         double kinetic = calculate_kinetic_energy();
386         double temperature = calculate_temperature();
387         double total_energy = potential + kinetic;
388         double msd = 0.0;
389 
390         // Save initial data
391         potential_energy_data.push_back(potential / N);
392         kinetic_energy_data.push_back(kinetic / N);
393         total_energy_data.push_back(total_energy / N);
394         temperature_data.push_back(temperature);
395         time_data.push_back(0.0);
396         msd_data.push_back(msd);
397 
398         for (int step = 1; step <= total_steps; ++step) {
399             velocity_verlet_step();
400 
401             double current_time = step * dt;
402             time_data.push_back(current_time);
403 
404             // Calculate MSD
405             msd = calculate_msd();
406             msd_data.push_back(msd);
407 
408             if (step % 1000 == 0) {
409                 std::cout << "Step " << step << "/" <<
    total_steps
410                           << ", T = " << temperature_data.
    back()
411                           << ", E = " << total_energy_data.
    back()
412                           << ", MSD = " << msd << std::endl;
413             }
414         }
415 
416         std::cout << "Simulation completed." << std::endl;
417     }
418 
419     void save_data(const std::string& prefix) {
420         // Save energy data
421         std::ofstream energy_file(prefix + "_energy.dat");
422         energy_file << "# time potential_energy
    kinetic_energy total_energy temperature\n";
423 
424         for (size_t i = 0; i < time_data.size(); ++i) {
425             energy_file << time_data[i] << " "
```

```
                        << potential_energy_data[i] << " "
                        << kinetic_energy_data[i] << " "
                        << total_energy_data[i] << " "
                        << temperature_data[i] << "\n";
        }
        energy_file.close();

        // Save MSD data
        std::ofstream msd_file(prefix + "_msd.dat");
        msd_file << "# time msd\n";

        for (size_t i = 0; i < time_data.size(); ++i) {
            msd_file << time_data[i] << " " << msd_data[i] <<
    "\n";
        }
        msd_file.close();

        // Save final configuration
        std::ofstream config_file(prefix + "_final_config.xyz
    ");
        config_file << N << "\n";
        config_file << "Final configuration\n";

        for (int i = 0; i < N; ++i) {
            config_file << "Ar " << positions[i].x << " " <<
    positions[i].y << " 0.0\n";
        }
        config_file.close();

        std::cout << "Data saved with prefix: " << prefix <<
    std::endl;
    }
};

int main() {
    // Simulation parameters
    double L = 50.0;                    // Box size
    double rho = 0.7;                   // Number density
    double dt = 0.001;                  // Time step
    int total_steps = 100 / dt;         // Total simulation
    steps (t=100)
    int equilibration_steps = 10000;  // Equilibration steps
    bool use_gaussian = true;           // Use Gaussian
    velocity distribution

```

```
    // Create and run simulation
    MDSimulation md(L, rho, dt, total_steps, use_gaussian);
    md.initialize_positions();
    md.initialize_velocities();

    // Equilibrate the system
    md.equilibration(equilibration_steps);

    // Run the production simulation
    md.run();

    // Save data for later analysis
    md.save_data("lj_sim");

    return 0;
}
```

Listing 3: Lennard-Jones MD simulation algorithm

b. Energy and Temperature Analysis

The following figures show the potential, kinetic, and total energy per particle, as well as instantaneous temperature T_i as functions of time t.

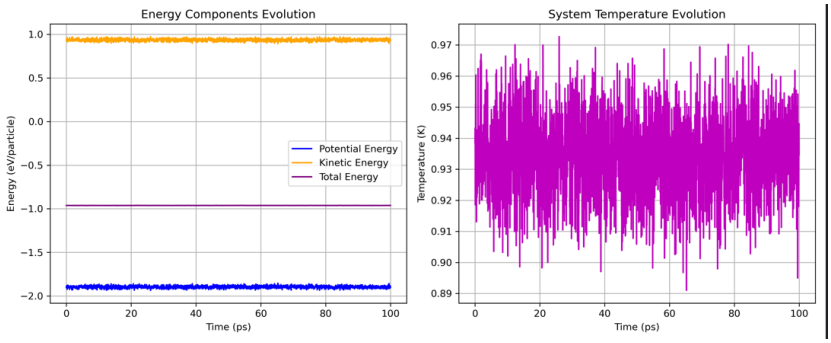

Figure 8: Potential, kinetic, total energy per particle, and instantaneous temperature as functions of time.

c. Mean-Squared Displacement Analysis

Figure 9 shows the mean-squared displacement (MSD) as a function of simulation time t, plotted on a logarithmic scale.

Listing 4: MSD calculation for LJ system

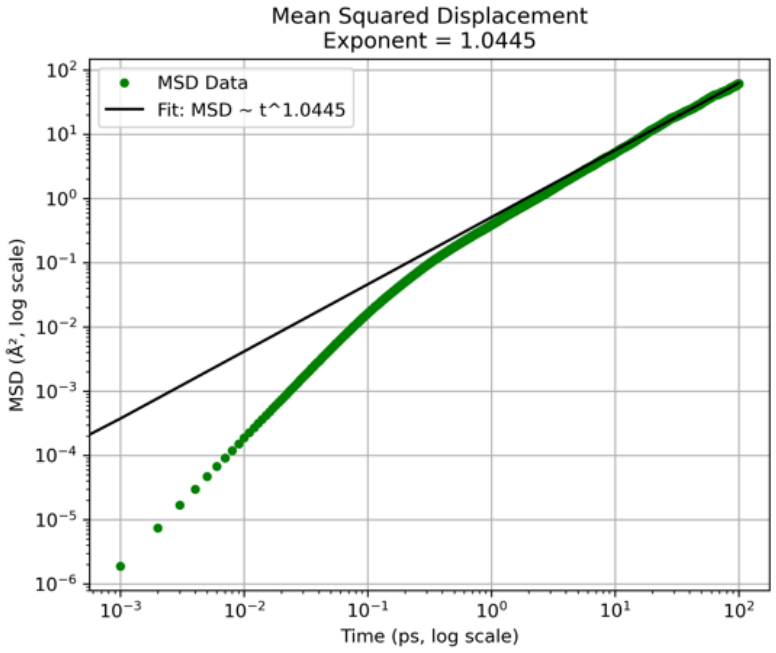

Figure 9: Mean-squared displacement as a function of simulation time, plotted on a logarithmic scale.

The data was fitted to extract the scaling exponent α in the relation MSD $\sim t^{\alpha}$.

The calculated value of α is 1.0445 .

Conclusion:

The molecular dynamics simulation of Lennard-Jones particles in a 2D box ($L = 50$, $\rho = 0.7$) successfully modeled the behavior of a dense gas system using a truncated and shifted Lennard-Jones potential and the velocity-Verlet integration algorithm with a time step $\Delta t = 0.001$. The simulation, run for a total time $t = 100$, incorporated a Verlet neighbor list to optimize computational efficiency. The analysis of potential, kinetic, and total energy per particle, along with instantaneous temperature, presented in Figure 8, showed stable energy conservation and temperature fluctuations consistent with the initialized Gaussian velocity distribution. The mean square

displacement (MSD), plotted on a logarithmic scale in Figure 9, followed a power-law scaling MSD $\sim t^{\alpha}$ with $\alpha = 1.0445$, indicating near-normal diffusive behavior, slightly deviating from ideal diffusion ($\alpha = 1.0$) due to particle interactions. This near-linear scaling suggests that the system exhibits diffusive dynamics typical of a dense fluid, with interactions slightly enhancing particle mobility. The simulation's ability to maintain energy conservation and produce physically meaningful MSD scaling underscores the effectiveness of the velocity-Verlet algorithm and neighbor list optimization in modeling complex particle systems.