A Computational Approach into Particle Interactions Using Lennard-Jones (LJ) Potential and Metropolis Monte-Carlo Simulation

Christian Fernandez, David Houshangi, Seungho Yoo, Yash Maheshwaran, Yejin Yang

Department of Chemistry, College of Chemistry, University of California at Berkeley,

Abstract

This study investigates particle aggregation and equilibrium structures using the Lennard–Jones (LJ) potential combined with the Metropolis Monte Carlo algorithm. Simulations were implemented in both Python and C++ to balance ease of analysis with computational efficiency. By systematically varying the repulsive (a) and attractive (b) LJ parameters, temperature (T), and particle number (N), we examined how these factors influence clustering behavior and equilibrium stability. Results show that stronger attraction (larger b) produces compact, stable clusters, while stronger repulsion (larger a) disrupts aggregation. Higher temperature increases cluster breakup probability, whereas higher density promotes aggregation. Comparative runs between Python and C++ produced qualitatively consistent results under identical parameters. These findings highlight how parameter tuning and algorithmic design choices jointly determine equilibrium structures and sampling accuracy, offering guidance for extending such models to more realistic molecular systems.

Keywords: Metropolis Monte Carlo, Lennard-Jones potential, molecular simulation, particle interactions, equilibrium states, statistical mechanics, phase transitions, temperature effects, Python, C++ programming

1. Introduction

Understanding how particles interact under specific thermodynamic conditions is a central challenge in physics and chemistry [1]. Predicting the collective behavior of many interacting particles is essential for explaining and designing processes in fields such as materials science, chemical engineering, and nanotechnology. However, directly solving the equations of motion for large particle systems is computationally expensive and analytically intractable, especially when interactions are complex. This difficulty creates the need for simplified yet physically meaningful models that can capture essential features of particle behavior while remaining computationally feasible.

The Lennard–Jones (LJ) potential is a widely used model for non-bonded interactions, balancing short-range electron-shell repulsion with longer-range Van der Waals attraction [2]. However, when particles come very close, the steep repulsive term of the LJ potential produces extremely large forces, which in a direct Newtonian integration can cause particles to be "kicked out" violently from the system. Conversely, particles that are far apart experience negligible forces, leading to very slow changes. These effects can hinder the system from efficiently sampling physically relevant configurations.

The Metropolis Monte Carlo method addresses these issues by focusing on whether a proposed move is accepted rather than explicitly integrating particle trajectories. In this algorithm, we start with an initial arrangement of particles and repeatedly propose small random displacements. Each move is accepted or rejected based on the change in potential energy and the Boltzmann probability distribution at a given temperature T [3]. If the energy change ΔU is negative, the move is always accepted; if $\Delta U>0$, the move is accepted with probability $e^{-\Delta U/T}$. This temperature-dependent acceptance criterion allows the system to explore a wider configuration space, overcoming local minima at

higher *T* and converging toward equilibrium structures at lower *T*. This way, the method prevents particles from getting unrealistically close while still allowing some higher energy moves so the system can explore more possibilities.

In this study, we implement the simulation in both Python and C++ and compare the results to verify that both yield consistent physical conclusions under identical parameters. Python enables rapid prototyping, easy parameter tuning, and direct visualization of results, while C++ offers greater computational speed for large-scale simulations. By running the same model in both environments and comparing their outcomes, we assess the robustness and reproducibility of our conclusions independent of programming language or implementation details.

2. Methods

Lennard-Jones Potential: The Lennard-Jones potential models (eq. 1) the pairwise interaction energy between neutral particles as a function of the distance between them:

$$\Phi(r) = 4\varepsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}] \quad \text{(eq.1)}$$

Where:

ϕ(r): The potential energy that exists between two non-bonded particles that are separated by r.

***σ*:** The interparticle potential is zero at a finite distance. It roughly depicts the diameter of the particle or the point at which attractive and repulsive forces balance each other out.

***ε*:** The potential well's depth. It indicates how strongly particles are attracted to one another; a higher ε indicates a stronger interaction.

***r*:** The distance between the centers of two particles.

The model can be reduced in complexity by introducing parameters ***a*** and ***b*** which combine the constants in the earlier equation (eq. 1) to form:

$$\Phi(r) = (\frac{a}{r})^{12} - (\frac{b}{r})^{6} \quad \text{(eq.2)}$$

Where:

***a*:** The parameter associated with repulsive forces at short distances that stem from electron shell repulsion.

***b*:** The parameter associated with attractive forces at longer distances that stem from Van der Waals forces.

Since the repulsion term is raised to a much higher power than the attraction term, particles are dominated by repulsive forces at short distances, causing the calculated potential to increase dramatically.

Monte Carlo Simulation: Monte Carlo techniques utilize random sampling to model the likelihood of output configurations within systems that are too complex to derive analytical solutions. They randomly vary system parameters that allow for the analysis of trends within the outcome states. The Monte Carlo simulation can then be combined with various algorithms to define how the specific simulation is run, which in this case, is the Metropolis algorithm.

Metropolis Algorithm: The Metropolis algorithm at its core determines whether a system changes state according to some acceptance criteria which is generated from the context of the problem. When utilized with Monte Carlo simulation, the algorithm randomly proposes new system states, for example, a new sample or change in sample state. These new system states are evaluated via the acceptance criteria which are derived from a probability distribution

that characterizes the system dynamics. Accepted changes within the system are saved as the current system state, and the entire process is repeated for any number of desired iterations.

3. Implementation

The simulation of particle movement was done by combining the Lennard-Jones potential with the Monte Carlo Simulation using the Metropolis Algorithm. Each particle's position is tracked along the x and y axes, and its energy is calculated based on its distances from all other particles, stored in distance matrices.

For Python the simulation stores particle positions in two one-dimensional arrays: *X[i]* stores the x-coordinate of particle *i* and *Y[i]* gives us the y-coordinate of particle *i*. A third array is created for energy: *Utot[i]* stores the total energy of particle *i* due to all other particles.

In the C++ implementation, particle positions are stored in *std::vector<double> x* and *y*, while total energy for each particle is stored in *std::vector<double> U*.

Potential Energy Calculation: The total potential for particle *i* is computed using the Lennard-Jones potential equation utilizing the earlier distance matrices, as shown in *eq.2*.

Self-interactions are explicitly set to zero. In Python, the pairwise potentials are computed inside the DisToPotential() function, which uses Potential() to apply the Lennard–Jones equation to all particle pairs and then sums the interactions to produce *Utot[i]* for each particle. In C++, this is done by total_potential_per_particle(), which computes the Lennard-Jones potential for a single particle relative to the other particles and stores these results in *U[i]*.

Metropolis Monte Carlo Simulation: The Metropolis algorithm determines whether to accept or reject proposed moves for all particles in the system in each iteration of the simulation. In Python, every particle is displaced in both the x and y directions by normal distribution with mean of 0 and standard deviation of *sigma*. After new positions are created, the Metropolis algorithm is implemented to determine whether the new particle positions (trial configuration) are to be accepted or rejected based on independent ρ values assigned to each particle. In the Python implementation, these displacements are generated in the form of arrays (*dx* and *dy*) to update all particle positions simultaneously. The potential energy for the new trial configuration (Utri) is calculated using the same Lennard-Jones potential function as for the original configuration (*Uold)* of the particles. The difference in energy between *Utri* and *Uold* represents the energy change caused by the proposed change in particle position.

In C++, the particle position displacements for each particle are created inside of a loop. The trial positions are stored in *xt* and *yt*. The particle position direction is then chosen randomly for each particle using rng.choice_pm() to pick +*delta* or -*delta*. The potential energy for the trial configuration. (*U_new_i*) is computed by recalculating only the trial particle's energy with the total_potential_per_particle() function, and comparing it to the original stored energy *U[i]*.

Acceptance Rules: If the change in energy is negative (the new state has lower energy), the new particle position move is automatically accepted as it makes the system more stable. If the change in energy is positive (the new state has higher energy), the move is accepted only with probability $e^{-\Delta U/T}$, where T is the temperature. Determining particle movements on probabilistic outcomes allows for particles to explore more configurations and prevents them from getting stuck in a local minimum space. The probabilistic decision is made by comparing $e^{-\Delta U/T}$ to a uniformly distributed random number, ρ, between 0 and 1 for each particle. In Python, if a move is accepted, the particle's new position is stored in *X[i]* and *Y[i]* values are updated to the trial values. All particles are proposed to move simultaneously, but acceptance is evaluated per particle. The entire process allows for the simulation to run in accordance with the Boltzmann distribution, while simultaneously allowing for both energy-lowering and occasional energy-raising moves, depending on thermodynamic conditions. In C++, if the move is accepted, the *x[i], y[i],* and *U[i]* values are updated inside the loop.

4. Evaluation

We analyze the particle movement over time under the default setting using Python simulations, followed by an examination of how changes in the interaction parameters (*a, b*), particle number (*N*), and temperature (*T*) affect aggregation patterns and overall system behavior. Finally, we present a comparative analysis of the results obtained from the C++ implementation.

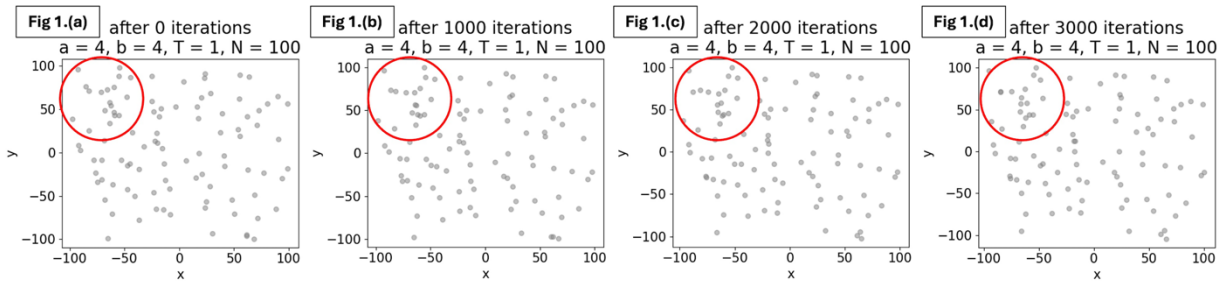

Figure 1. Particle configurations from Python simulation under the default setting (*a*=4, *b*=4, *T*=1, *N*=100) at different iterations: (a) initial state (0 iterations), after (b) 1000 iterations, (c) 2000 iterations, and (d) 3000 iterations. The red circles highlight a representative particle cluster.

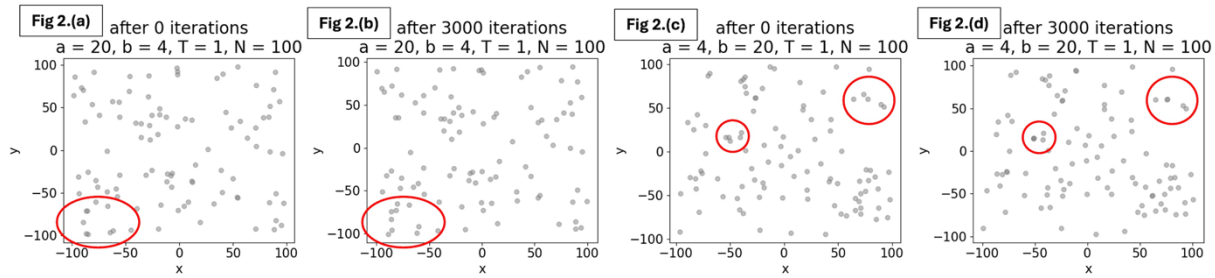

Figure 2. Particle configurations from Python simulation with different a and b. (a, b) *a*=20, *b*=4, *T*=1, *N*=100 at initial state and after 3000 iterations, (c, d) *a*=4, *b*=20, *T*=1, *N*=100 at initial state and after 3000 iterations.

Under the default setting ($a = 4$, $b = 4$, $T = 1$, $N = 100$), aggregation patterns remain stable over iterations (from 0 to 3000th iteration), as shown in the red circles of Fig. 1(a–d). This indicates that the particle clusters have reached equilibrium. When the Lennard-Jones parameters are varied, however, the degree of aggregation changes. Increasing *a* from 4 to 20 makes the repulsive force much stronger, so particles push each other away and the clusters gradually break apart. This can be seen by comparing the initial and after 3000 iterations (Fig. 2(a, b)). The effect will be especially noticeable when particles are close together (not shown here), because the repulsive term a/r^{12} becomes dominant in short distance. On the other hand, increasing *b* from 4 to 20 strengthens the attractive force between particles. As shown in initial and after 3000 iterations (Fig. 2(c, d)), this leads to tighter clustering and helps the aggregates stay preserved even after many iterations.

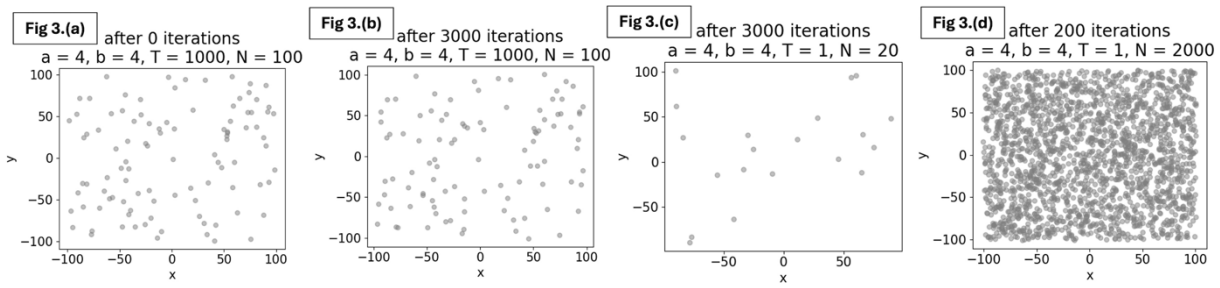

Figure 3. Particle configurations from Python simulation with different T and N. (a, b) a=4, b=4, T=1000, N=200 at initial state and after 3000 iterations, (c) a=4, b=20, T=1, N=20 after 3000 iterations, and (d) a=4, b=20, T=1, N=2000 after 200 iterations.

T and N also have a clear impact on aggregation behavior. Comparing $T = 1$ at initial and after 3000 iterations in Fig. 1 (a, d) with those at $T = 1000$ in Fig. 3 (a, b) shows that at high temperature, aggregation does not maintain its structure over iterations. The increased thermal energy raises the probability of accepting moves that break clusters apart, leading particles to move more freely and remain more dispersed. In contrast, at lower T (Fig 1. a, d) clusters remain relatively stable for much longer. Varying N changes the system density, which affects how often particles collide. At low N ($N = 20$ in Fig. 3c) even after 3000 iterations, the density is so low that particles rarely meet, and less visible aggregation occurs. At high N even after only 200 iterations ($N = 2000$ in Fig. 3d), collisions are frequent, and clustering becomes much stronger due to the increased influence of attractive forces. The $N = 2000$ simulation was stopped early at 200 iterations due to large N causing a long runtime.

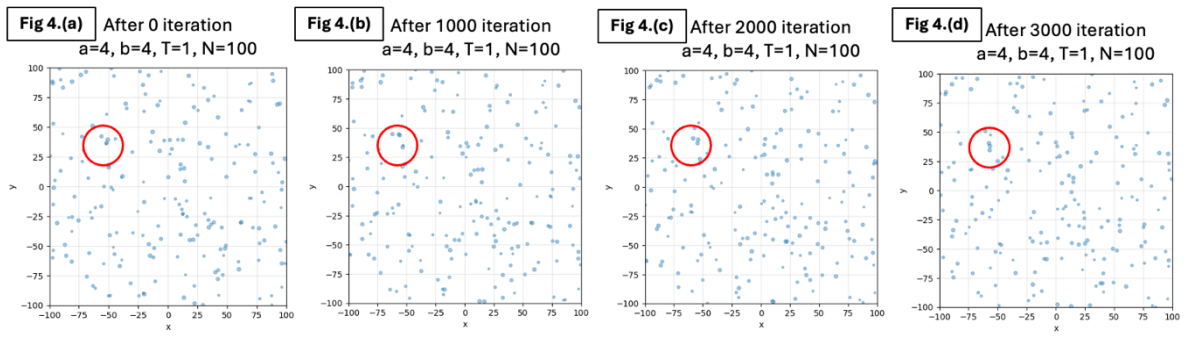

Figure 4. Particle configurations from C++ simulation under a=4, b=4, T=1, N=100 at different iterations: (a) initial state (0 iterations), (b) after 1000 iterations, (c) after 2000 iterations, and (d) after 3000 iterations. The red circles highlight a representative particle cluster.

With $a = 4$, $b = 4$, $T = 1$, $N = 100$, the C++ simulation shows that the degree and pattern of aggregation remain largely unchanged from the initial state to the 3000th iteration (Fig. 4). The clusters highlighted in red circles persist over time, indicating that the aggregated regions stay stable. The overall aggregation behavior is qualitatively similar to that observed in Python under the same parameters.

5. Discussion

Result Interpretation: Under a default set of parameters, particles started in a random, evenly spread arrangement. As the simulation progressed, they moved under both repulsive and attractive forces. Small groups formed, merged, and eventually stabilized into clusters. Once equilibrium was reached, the structure remained stable.

Aggregation mainly depends on the Lennard–Jones parameters a and b, the temperature T, and the density set by N. Parameter a controls how sharply particles repel when very close, while b controls how strongly they attract. Larger b creates deeper potential wells (larger ε), making clusters more stable. Conversely, larger a increases repulsion, making particles spread out and reducing aggregation. These patterns appeared clearly in the results: stronger attraction made compact, stable clusters, while stronger repulsion broke them apart.

In the Metropolis algorithm, the temperature T strongly influences the acceptance of particle moves. At high T, even moves that increase the system's potential energy are often accepted, allowing particles to move more freely and remain more dispersed. At low T, such energy-increasing moves are rarely accepted, so particles tend to stay in energy-minimizing configurations, leading to stable clustering. This temperature effect is realized through the acceptance rule, which depends on the change in potential energy (ΔU): if ΔU is negative (the move lowers the system's energy), the move is always accepted; if ΔU is positive, the acceptance probability decreases according to the Boltzmann factor, with the reduction becoming more severe at lower T.

The number of particles (N) defines the system's density (ρ), which controls how often particles encounter each other. Higher density (large N) increases collisions and promotes aggregation, while lower density (small N) reduces collisions and makes aggregation less likely. The C++ implementation produced qualitatively similar aggregation behaviors under same parameters, confirming the consistency of the results across platforms.

Limitations of the Model: Our simulation uses the Lennard–Jones potential with the Metropolis Monte Carlo method to model particle interactions. This approach captures important thermodynamic behavior but does not fully represent real molecular systems. In reality, particles can interact in more complex ways, with multiple particles influencing each other simultaneously, long-range forces such as electrostatic attraction or repulsion. These are not included in the Lennard–Jones model [4]. The Monte Carlo method also does not track the actual motion of particles over time, because it does not integrate Newton's equations of motion step by step. Instead, it generates a random new position and checks whether it should be accepted based on the energy change and the Boltzmann probability. As a result, the simulation produces only a sequence of configurations that follow the correct equilibrium distribution, with no real clock. The order of moves in Monte Carlo steps does not correspond to physical time, so it cannot show how long it takes for a particle to diffuse or relax, only which configurations are likely at equilibrium [5].

Effect of Moving All Particles Simultaneously: In the classical Metropolis Monte Carlo, trial moves usually shift only one particle or a small group at a time. This is essential for keeping detailed balance and ergodicity in Markov chain sampling [6–8]. In our study, however, all particles were moved at once in each update. This makes the code simpler and may seem to require fewer updates, but it has drawbacks. Most importantly, it can break the detailed balance and ergodicity assumptions. The acceptance/rejection step is designed for symmetric, localized moves so that transition probabilities are well-defined [6, 7]. Moving all particles together, however, can create correlations between steps, reduce statistical independence, and lower the accuracy of the Markov chain.

Potential Improvements to the Simulation: Several strategies could address these limitations discussed above. Instead of only moving one particle at a time or all particles at once, a mixed approach can be used. Most of the time, single-particle moves are made, but occasionally several particles or the entire system are moved together. This helps the particles explore space more quickly while still keeping detailed balance [8]. Another option is Hybrid Monte Carlo (HMC), which moves particles by briefly following Newton's equations so they travel smoothly and continuously before applying the usual Monte Carlo acceptance test. This allows particles to move farther in a natural way while keeping a high acceptance rate, making the simulation explore configurations more efficiently [9].

6. Reference

[1] Frenkel, D. and Smit, B., 2023. Understanding molecular simulation: from algorithms to applications. Elsevier.

[2] Allen, M.P. and Tildesley, D.J., 1987. Computer simulation of liquids.

[3] Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H. and Teller, E., 1953. Equation of state calculations by fast computing machines. The journal of chemical physics, 21(6), pp.1087-1092.

[4] Allen, M.P. and Tildesley, D.J., 1987. Computer simulation of liquids.

[5] Frenkel, D. and Smit, B., 1957. Understanding molecular simulation. San Diego: Academic Press.

[6] Müller, F., Christiansen, H., Schnabel, S. and Janke, W., 2023. Fast, hierarchical, and adaptive algorithm for Metropolis Monte Carlo simulations of long-range interacting systems. Physical Review X, 13(3), p.031006.

[7] Luijten, E., 2006. Introduction to cluster Monte Carlo algorithms. In Computer Simulations in Condensed Matter Systems: From Materials to Chemical Biology Volume 1 (pp. 13-38). Berlin, Heidelberg: Springer Berlin Heidelberg.

[8] Plascak, J.A., Ferrenberg, A.M. and Landau, D.P., 2002. Cluster hybrid Monte Carlo simulation algorithms. Physical Review E, 65(6), p.066702.

[9] Duane, S., Kennedy, A.D., Pendleton, B.J. and Roweth, D., 1987. Hybrid monte carlo. Physics letters B, 195(2), pp.216-222.

[10] Haji, S.H. and Abdulazeez, A.M., 2021. Comparison of optimization techniques based on gradient descent algorithm: A review. PalArch's Journal of Archaeology of Egypt/Egyptology, 18(4), pp.2715-2743.