CHEM 272 Final Project
Joseph Gill, Leah Sutherland,
Sibhi Sakthivel, Katrina Reyes,
Snigdha Chanduri, Nathalie Murphy

Simulations for Movement of Particles in a Lennard-Jones Potential Using Python and C++

ABSTRACT

In this project, we simulate two-dimensional particle motion driven by the Lennard-Jones potential in Python and C++ using the Metropolis Monte Carlo algorithm. The model captures short-range repulsion and long-range attraction and produces characteristic equilibrium distance and clustering. In Python, NumPy is used to calculate distances between particles and total potential energy of the system (U_{tot}), with moves accepted or rejected via the Boltzmann criteria. In C++, we use an object-oriented design to calculate pairwise and total energies, integrating the Metropolis method for updates. The results show that higher particle density and sufficient iterations promote clustering, while temperature influences the equilibrium between ordered and random motion. The study demonstrates the ability of both implementations to reproduce the motion of many particles in a Lennard-Jones potential and suggest improvements.

INTRODUCTION

Understanding the mechanisms underlying how particles interact with one another is essential to advancing knowledge in many diverse scientific fields, including drug development, condensed matter physics, and chemical engineering. One of the most widely used models for exploring these interactions is the Lennard–Jones potential. This model simulates the net force between two molecules as the combination of a short-range repulsion and a long-range attraction. These opposing forces establish a characteristic equilibrium distance between particles and provide valuable insights into molecular organization, phase transitions, and the structural stability of materials. In this project, we model the two-dimensional movement of particles interacting through Lennard-Jones potential. This simulation utilizes the Metropolis Monte Carlo algorithm, which approximates mathematical models to simulate complex systems by implementing random sampling statistical techniques[1].

At times, we cannot model a real-world system with an analytical solution due to the complexity of the scenario. Another problem that may arise is that though a solution exists, there are singularities and effects of randomness, making standard simulations not possible. This situation is faced when modelling how particles move in a potential (U) using the Lennard-Jones potential. Though the model has an analytical solution, we have a singularity. When the distance between two particles (r) is zero, the repulsion between those two particles approaches infinity. Here, that scenario is modeled using the Monte Carlo and Metropolis methods.

METHODS

Within the repeated random sampling of the Monte Carlo method implemented in these models, the Metropolis algorithm determines whether a particle moves to a different position. For each iteration of the simulations, random movements are suggested for each particle, and whether those movements are made permanent depend on the total potential energy they would experience at the next position. The individual potential energy of every particle is calculated using a simplified version of the Lennard-Jones potential equation:

$$\phi = \frac{a}{r^{12}} - \frac{b}{r^{6}}$$

Then the total potential for each particle is calculated by summation of the individual potentials each particle feels from all others in the system. This accounts for the cumulative effects of attractive and repulsive forces from all of the other particles acting on each particle. The distance between two particles is represented by the variable 'r.' In this simulation, the particles belong to a two-dimensional space. Calculating the distance between two particles is modeled by the equation:

$$r_{ij} = \sqrt{(x(i) - x(j))^2 + (y(i) - y(j))^2}$$

where the x and y coordinate of each particle is represented by their corresponding index i or j.

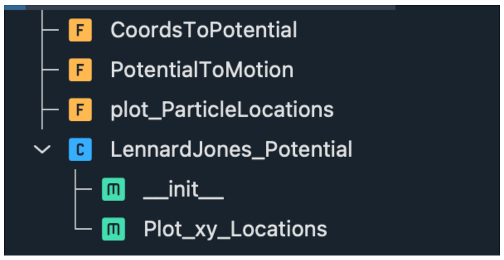
Comparison of the total potentials of each particle before and after suggested random movement will determine if the particles move or undergo further scrutiny. If the movement results in a reduction in the total potential energy of the system (U_{tot}), the particle will move.

If the movement of a particle results in an increase in potential energy, simulation of random particle movement (Brownian motion) will determine movement. The probability of a particle moving (p_{move}) in this condition is proportional to the Boltzmann factor:

$$p_{move} \approx e^{(-dU_{tot}(x,y)\,/\,T)}$$

where dU_{tot} refers to the differences in the potential energy of the system before and after the movement of the particle. The variable 'T' refers to the temperature of the system. Thus, the probability of a particle moving is based on the difference in potential energy and temperature. As dU_{tot} increases in value, p_{move} approaches 0. A random number selected between 0 and 1 determines whether the movement of a particle will occur. If that random number is less than p_{move}, then the particle moves. Otherwise, it remains in place. This matches endogenous conditions as states with higher potential energies are difficult to attain.

IMPLEMENTATION – PYTHON

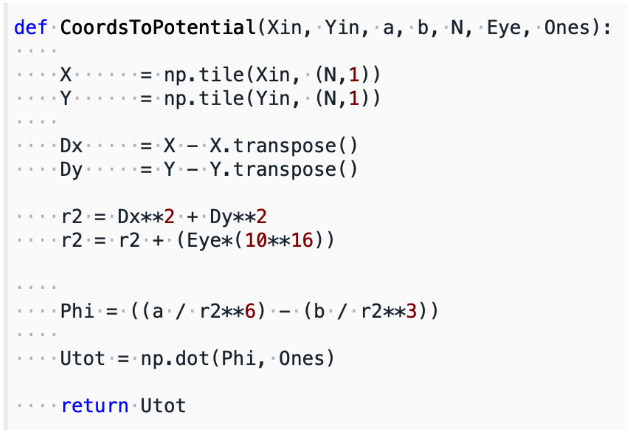


```
├ F CoordsToPotential
├ F PotentialToMotion
├ F plot_ParticleLocations
⌄ C LennardJones_Potential
  ├ m __init__
  └ m Plot_xy_Locations
```

The python package consists of three main functions that are called in various methods within one class. These functions depend on functionalities offered in the numpy and matplotlib.pyplot libraries, which are called once in the program header.

The function "CoordsToPotential" calculates and returns the distance-dependent total potential experienced by each particle in the simulation. CoordsToPotential accepts initial positions for each particle in a 2-dimensional space. The x and y coordinates are passed in as separate numpy arrays. The function also accepts two constants used in the potential equation, a and b, and the total number of particles in the simulation, N, as input parameters. It also passes the arguments "Eye" and "Ones." These are an identity and one's matrices, respectively, that are defined as such in a different function that calls CoordsToPotential.

The first operations of CoordsToPotential calculate the distance of each particle to all other particles using vector operations. The squares of the distances are summed and saved under the variable, r2. Since the distance between a molecule to itself is zero, the diagonals of the distance matrices are all zeros. In order to avoid dividing by zero when calculating the potentials, r2 is overwritten by adding itself to the Eye (identity matrix of size (N,N)) multiplied by 10^{16}. Then the potential each molecule feels from every other molecule individually is saved under the variable Phi.



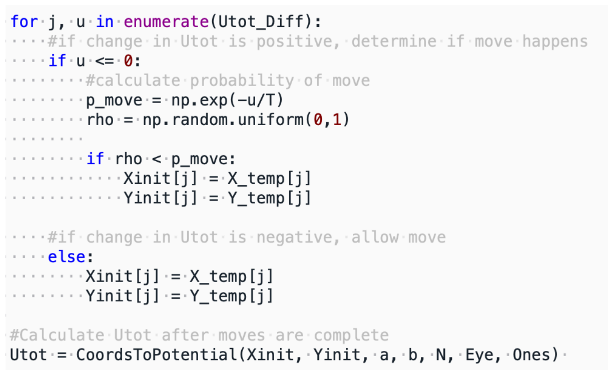
```
def CoordsToPotential(Xin, Yin, a, b, N, Eye, Ones):

    X      = np.tile(Xin, (N,1))
    Y      = np.tile(Yin, (N,1))

    Dx     = X - X.transpose()
    Dy     = Y - Y.transpose()

    r2 = Dx**2 + Dy**2
    r2 = r2 + (Eye*(10**16))


    Phi = ((a / r2**6) - (b / r2**3))

    Utot = np.dot(Phi, Ones)

    return Utot
```

Since r2 represents the squared sum of the differences (not the exact distance, which would be the square root of those sums), the exponents in the Lennard-Jones equation are adjusted accordingly. The final operation in CoordsToPotential calculates the sum of the potentials each particle feels from all other particles and returns the summed potentials in an array.

The next function, "PotentialToMotion," calls CoordsToPotential and converts the potentials to motions. This function accepts the number of iterations of the simulation, the number of particles, a and b (previously discussed above), temperature (T), and step size. This function also defines the Eye and Ones variables that are passed to CoordsToPotential. Next, it defines the initial x and y positions from a uniform distribution. The total potentials for N molecules at the initial positions are calculated using CoordsToPotential. Next, two empty arrays, one three-dimensional and one two-dimensional, are initialized. The three-dimensional array will store all the positional coordinates and total potentials from the entire simulation. The two-dimensional array is used to temporarily store those data per simulation iteration before they are added to the 3D matrix.

Next, a for loop that iterates through the range of iterations randomly suggests movements according to the user-defined step size. Two new variables, X_temp and Y_temp, are defined as the initial x and y positions plus the random steps. Then the total potentials for each molecule at the new, temporary locations are calculated using CoordsToPotential. Then a new array, Utot_Diff, is created and defined as the difference between total potentials at the previous locations and the temporary locations.

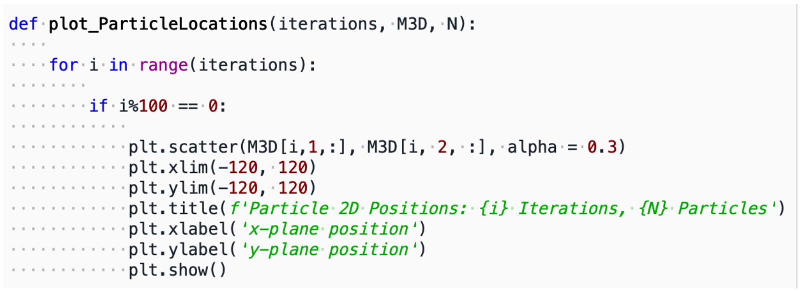
A nested for loop iterates through the indices and values of Utot_Diff using the enumerate function and uses the Metropolis method to determine if each particle will permanently undergo the temporary movement. If the value of Utot_Diff is less than or equal to zero, the probability of movement is calculated using the Boltzmann factor (which incorporates the user-defined temperature of the simulation), and the variable rho is randomly selected between 0 and 1 from a uniform distribution. The total potentials of all the molecules after movement are calculated again using CoordsToPotential.



```
for j, u in enumerate(Utot_Diff):
    #if change in Utot is positive, determine if move happens
    if u <= 0:
        #calculate probability of move
        p_move = np.exp(-u/T)
        rho = np.random.uniform(0,1)
        
        if rho < p_move:
            Xinit[j] = X_temp[j]
            Yinit[j] = Y_temp[j]

    #if change in Utot is negative, allow move
    else:
        Xinit[j] = X_temp[j]
        Yinit[j] = Y_temp[j]

#Calculate Utot after moves are complete
Utot = CoordsToPotential(Xinit, Yinit, a, b, N, Eye, Ones)
```

The total potentials, x positions, and y positions are stored in the previously defined 2D array. The last step of the outer for loop stores the 2D array in the index of the current iteration of the 3D array. After the outer for loop finishes for all iterations, PotentialToMotion returns the completed 3D array.

The function, "plot_ParticleLocations," plots the x and y positions of every particle. The function accepts the number of iterations, 3D array generated by PotentialToMotion, and number of particles as input arguments. This function plots the x and y positions for each molecule for every 100th iteration. The coordinates for plotting are pulled from the 3D array.



```
def plot_ParticleLocations(iterations, M3D, N):
    
    for i in range(iterations):
        
        if i%100 == 0:
            
            plt.scatter(M3D[i,1,:], M3D[i, 2, :], alpha = 0.3)
            plt.xlim(-120, 120)
            plt.ylim(-120, 120)
            plt.title(f'Particle 2D Positions: {i} Iterations, {N} Particles')
            plt.xlabel('x-plane position')
            plt.ylabel('y-plane position')
            plt.show()
```

After the functions are defined, the program defines a class, Lennard-Jones_Potential that calls them. The __init__ function defines the defaults for the input variables outlined above. It also runs the PotentialToMotion function and saves the returned 3D array to the variable, Matrix. A second method within the class calls the plotting function. Separating the simulation call from the plotting allows the program to generate each plot immediately after the previous, so the user is better able to visualize the continuous motion of the simulation.

IMPLEMENTATION – C++

Similarly, the aim in the C++ code is to simulate particle movement via the creation of a particle class, Lennard-Jones class, a main function that incorporates the Metropolis method, and configurations for plotting in MatPlotLib.

The particle class creates a constructor for the particle itself and dictates the particle's movement in the simulation area. 'Double x, y' works in conjunction with the constructor to provide randomized, uniform positions for the particle. The 'void move()' function applies the displacement to the particles, causing them to move about the simulation.

The purpose of the Lennard-Jones class is to calculate both the potential energy between each pair of particles as well as the total potential energy of the system. First, we calculate the straight-line distance between two points using the square root of the Euclidean Distance Formula:

```
double r = std::sqrt (dx * dx + dy * dy);
```

Next, the Lennard-Jones equation is used to determine the potential energy between both particles:

```
double r6 = std::pow(r, 6);
double r3 = std::pow(r, 3);
return (a / r6) - (b / r3);
```

Once the potential energy between two particles has been calculated, the potential energy of the entire system is determined in the 'potential_energy' function. First, the two 'for' loops are utilized to determine the indices over each pair of particles. All particles are accounted for by using 'particles.size().' Then, the distance between two particles along the x and y axes are determined by finding 'double dx' and 'double dy' accordingly:

```
double dx = particles[i].x - particles[j].x;
double dy = particles[i].y - particles[j].y;
```

Finally, in calling the Lennard-Jones function, all potential energies are added together to calculate the total energy of the system.

The main function sets the parameters for the Metropolis loop which is set to loop over each particle individually. First, the following line of code calls the current coordinates of each particle and calculates the potential energy of the system at these coordinates:

```
double old_x = system.particles[i].x;
double old_y = system.particles[i].y;
double old_energy = system.potential_energy();
```

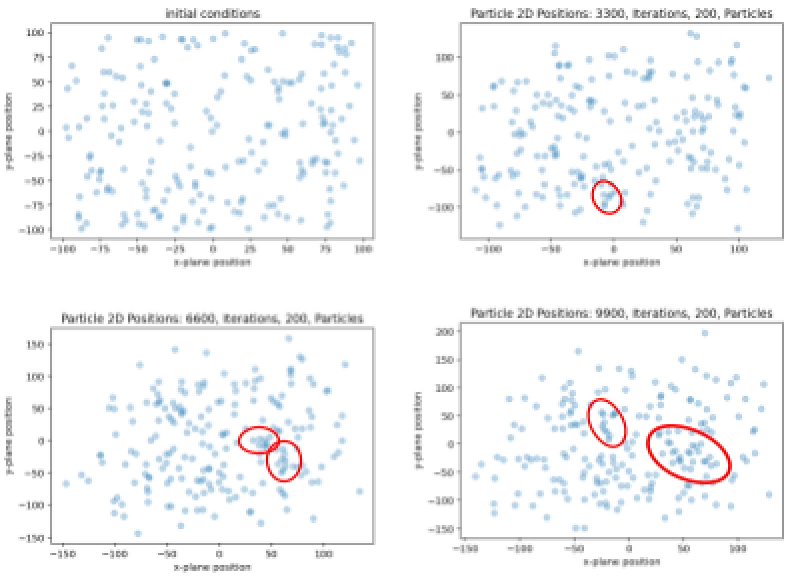
Then, the 'move' function causes the particles to move to new coordinates. Once the move happens, the change in potential energy is calculated accordingly:

```
double new_energy = system.potential_energy();
double deltaE = new_energy - old_energy;
```

Once the change in potential energy has been calculated, the Boltzmann acceptance rule is applied, where, if 'delta E' is greater than zero, then a random number will be used to calculate 'Pmove.' If the random number 'r' is higher than 'Pmove,' then the move will be rejected, and the particle will stay in its normal position.

EVALUATION

The plots generated from the python script exhibit increased clustering behavior. This is reflected by the initial conditions being more scattered as opposed to the 9900th iteration, which exhibits a more condensed configuration with clusters formed by several particles. The particles clustering together shows that the Metropolis algorithm incorporated correctly simulates a system that favors a total potential energy change in which there are optimal distances between particles that allow the balance of attractive and repulsive forces. The consistent movement of particles between each iteration suggests that the system is shifting towards these favorable changes.

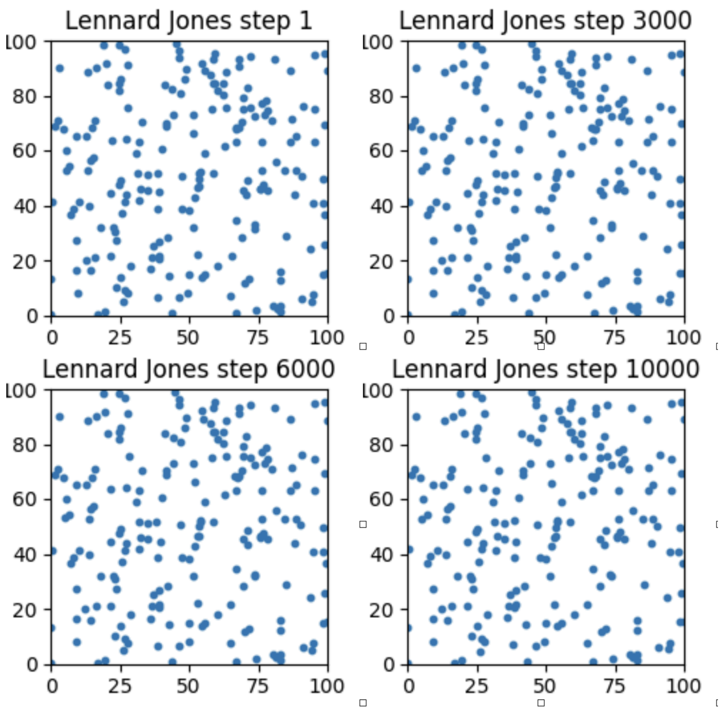


Motion of 200 particles generated from python code

The plots outputted from the C++ simulation did not exhibit significant clustering. This result is a combination of the number of iterations and the behavior of the Metropolis algorithm, which only moves a single particle in one iteration. Therefore, this C++ implementation only shows 10,000 particles moved, which is very small in comparison to the python implementation, which represents 200,000 particle movements. This low update rate means that many particles stay in positions close to their initial locations, preventing the system from reaching the equilibrium states where attractive forces could bring particles together into clusters.

In addition to the limited particle movement, the displacement parameter significantly impacts observed clustering. A step size that is too small results in insignificant particle movement, which would require more iterations for change to be observed. In contrast, setting the displacement at too large of a value would result in large particle jumps, representing unrealistic movement. It is essential for the displacement parameter to be set to an appropriate value to accurately model particle movement and clustering.

Overall, the lack of clustering observed in the C++ simulation can be explained by the low number of moves per particle, an iteration count too small to observe change over time, and parameter settings that did not promote particles to reach equilibrium states. Increasing the number of particle moves per iteration, running the simulation for more total iterations, and adjusting the displacement and temperature values would likely produce results that more closely reflect the clustering behavior observed in the Python implementation.



Motion of 200 particles generated from C++ code

DISCUSSION

Both programs outlined in this report successfully model the motion of any N number of neutral particles in a Lennard-Jones potential over any number of iterations. The plots of early iterations of the simulations show the random, uniform distributions in both the x and y directions. Conversely, the plots of later iterations demonstrate well the expected movement of particles under the intentional assumptions of the physical space (i.e. the particles are neutrally charged, and any impacts of differing masses are ignored). Clustering in various locations happens over time as a result of the optimum attractive forces of particles at or near the r_{min} distance from each other. Particles in clusters also do not travel even closer to each other because, though they may randomly take 'temporary' steps toward each other, the probability of actually making such steps is so low (as defined by the Boltzmann factor) that they do not make extremely energetically unfavorable moves. Additionally, the clusters emerge in different locations within the x-y space in each simulation run, demonstrating that while the motion of the particles is governed by predictable physics, some random nature to the system persists. Expectedly, particles that randomly travel away from other particles move back toward other particles less frequently because they are neither experiencing attraction to nor repulsion from the other particles in the space.

Though both programs meet the objective of this assignment, each has limitations and potential improvements. The python program accumulates and returns a 3-dimensional array containing all total potentials, x positions, and y positions for all particles for all iterations. This allows the end user to access the complete dataset generated in the simulation for any potential downstream uses or analyses. However, this functionality also slows the program down, with the most significant impacts observed for large numbers of iterations and large numbers of particles. If the user requires speed more than they do access to the complete dataset of the simulation, they may choose to use C++ given it typically runs significantly faster than the python program. However, the original C++ program that was generated for this project ran much slower than the Python program once it was tasked with looping through 200 particles over 10,000 iterations. Given the accommodation of these large parameters in one program, the total runtime for the C++ code lasted approximately one hour. Therefore, the final C++ program was set to loop over one particle at a time in order to improve the runtime and put less stress on the system.

The program could be improved with additional consideration toward the breadth and utility of the visual representations of data it offers. While the program does return the three-dimensional array of the complete simulation, it does not contain a built-in method to export the data. Additional methods to export the three-dimensional array and plots would improve the practicability of the program. Furthermore, there is currently no method to visualize total potentials over time. The lack of such a plotting routine is a limitation of the current program, and its addition would further strengthen the package.

Additionally, the program could be improved with enhanced flexibility for the user to define how they want the simulation executed. The current version assumes all particles in the field have the same mass. It could be improved by randomizing the masses and incorporating the effects of those masses on the movement. Additionally, the initial positions are hard-coded to be pulled from a uniform distribution. A potential future improvement would incorporate an option for the user to select what distribution they want the initial positions to be selected from.

Another improvement for the program would include the implementation of the 'fork()' system call in the C++ code. As of right now, the entire system of code is running in one thread, which continuously reuses the same memory space and renders the program inefficient. 'Fork()' works to create a new process, often referred to as the 'child.'[2] The 'child' creates a separate memory space from the 'parent' by using the 'copy-on-write' (COW) function. By utilizing the 'fork()' system call, this memory space would only be utilized when needed, thus improving the efficiency and runtime of the program.[3]

CONCLUSION

Overall, both the Python and C++ methods incorporated the Lennard-Jones potential and Metropolis algorithm to create a molecular dynamics simulation. While the Python code was able to successfully create a simulation that clearly encapsulated attractive and repulsive tendencies, the C++ code generated a simulation in which the movement of the particles was less significant. This can be attributed to the runtime of the thread in the C++ code, which can be improved upon by implementing more efficient processes in the code itself. Further work/studies on this topic can focus on how the C++ code calculates the change in potential energy, as well as how it incorporates positional changes for particles to further examine the effects of the Lennard-Jones equation and the Metropolis algorithm.

REFERENCES

[1] Harrison RL. Introduction To Monte Carlo Simulation. AIP Conf Proc. 2010 Jan 5;1204:17-21. doi: 10.1063/1.3295638. PMID: 20733932; PMCID: PMC2924739.
[2] Michigan Technological University. The fork() System Call. www.csl.mtu.edu. https://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html
[3] Lab: Copy-on-Write Fork for xv6. pdos.csail.mit.edu. https://pdos.csail.mit.edu/6.828/2019/labs/cow.html