Monte Carlo Simulations of a Lennard-Jones System

Project Report - Statistical Mechanics (PHY-3610-1)

Aditya Ramdasi, Dhruv Aryan

Instructor: Professor Bikram Phookun

Teaching Fellow: Philip Cherian

Submission Date: December 13, 2023

Abstract

The goal of this project was to perform Monte Carlo simulations of a 2D system of particles with the Lennard-Jones potential using the Metropolis algorithm. The radial distribution function was then used to observe phases in the system, as well as calculate energy and pressure and their temperature dependence for various densities. By comparing the results for pressure to the Van der Waals equation of state, we obtained the Van der Waals constants for such a system. The chemical potential and its temperature dependence for various densities were computed using the Widom insertion method.

I. Introduction

Statistical mechanics stands as a cornerstone in understanding the complex behavior of physical systems at the microscopic level, providing a bridge between the microscopic realm of particles and the macroscopic properties we observe in the world around us. It offers a theoretical framework to comprehend the thermodynamic properties of matter by investigating the statistical distribution of particles and their interactions.

In the pursuit of unraveling the intricacies of these systems, computational methods have become indispensable. The advent of powerful computing technologies has revolutionized our ability to simulate and comprehend the behavior of particles in diverse environments. Monte Carlo simulations, a key computational technique, have emerged as a valuable tool in this endeavor. This method, rooted in probabilistic sampling, allows us to explore the vast configuration space of a system and make statistical predictions about its behavior.

This project report delves into the application of Monte Carlo simulations in studying a Lennard-Jones system of particles. The Lennard-Jones potential, widely used to model van der Waals forces and steric interactions, presents an excellent test bed for understanding the equilibrium and dynamic properties of condensed matter systems. By harnessing the computational prowess of Monte Carlo simulations, this project aims to provide insights into the thermodynamic properties, phase transitions, and structural characteristics of the Lennard-Jones system, contributing to the broader understanding of statistical mechanics and the role of computation in unraveling the mysteries of the microscopic world.

i The Lennard-Jones Potential

The simplest way to model a system of particles moving inside a container is to assign each one of them an initial position and an initial velocity. Since there are no considerations made for the interactions between particles nor the size of those particles, this system will closely resemble a classical ideal gas. While a lot of interesting thermodynamics can be obtained by modeling such a classical ideal gas under various conditions, it is hardly a realistic system. Since the goal of this project is to study some thermodynamic properties of a realistic system, one of the most important things we should account for is the interaction between particles. This will be in the form of some potential between every pair of particles.

To model this interaction potential, we first decide upon some qualitative features that this potential should satisfy. Firstly, the atoms or molecules in a realistic gas contain protons in their nuclei and are surrounded by their electron clouds. Thus, two atoms can never get close beyond a certain distance to each other, due to the extremely strong (Pauli) repulsion between the protons and the electron clouds. To account for this, our potential should be highly repulsive at extremely short distances. Particles in a realistic system also experience weak (London dispersion) forces of attraction from each other at long distances, which implies that our potential should be mildly attractive at long distances. Lastly, for most realistic substances there exists an optimal intermolecular distance that corresponds to the lowest energy state. Fortunately, such an interaction potential had already been proposed by British scientist John Lennard-Jones in 1931. This potential takes on the following form -

$$V_{LJ}(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{1}$$

Here, r is the distance between two interacting particles, ϵ is the depth of the potential well (or the dispersion energy), and σ is the distance r at which the potential becomes zero.

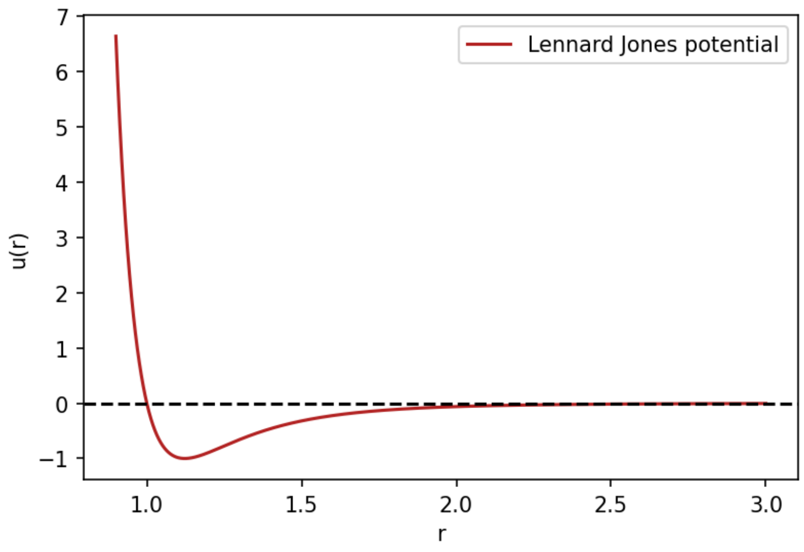


Figure 1: Plot of Lennard-Jones potential

We can see that this potential satisfies all the qualitative conditions described for intermolecular interactions. It is especially accurate for modeling noble gases like Helium, Neon, Argon, etc. due to the absence of any non-bonding free electrons. Thus, we use this potential to model the particle interactions in our system.

ii Van der Waals Equation of State

The Lennard-Jones potential can be used to computationally model a realistic system. Now, let us look at an analytical method of modelling a realistic system.

We have the following equation of state for an ideal gas:

$$PV = NT \tag{2}$$

Now, we make two modifications to the assumptions made for an ideal gas. An ideal gas consists of particles that have no volume. We can modify this by assuming that every particle now has a volume b, and so the *excluded* volume available for the gas to occupy then becomes $V - Nb$. Further, particles do not interact with each other in an ideal gas, but realistic systems consist of interacting particles. We thus add an interaction of aN^2/V^2. This gives us the following equation of state:

$$\left(P + a\frac{N^2}{V^2}\right)(V - Nb) = NT \tag{3}$$

In terms of the density $\rho = N/V$, we get

$$(P + a\rho^2)\left(\frac{1}{\rho} - b\right) = T \tag{4}$$

iii Chemical Potential

Denoted conventionally by μ, the chemical potential of a system can be defined in multiple slightly different ways depending on the context. In statistical mechanics, the chemical potential is defined as the "*Rate of change of free energy of a thermodynamic system G, F with respect to a change in the total number of particles (N) in the system*". In other words, the value of μ for a system gives us a measure of how hard it is to change the number of particles in the system at some constant temperature and volume or pressure (depending on which free energy). We can find the mathematical expression for μ in terms of the Helmholtz free energy F as follows -

$$\mu = -\left(\frac{\partial F}{\partial N}\right)_{V,T} \tag{5}$$

II. Techniques and Algorithms

In this project, three main computational techniques were used:

1. The Metropolis Algorithm
2. Radial Distribution Function
3. Widom Insertion Method

i The Metropolis Algorithm

The probability that a system is in a particular microstate m is given by

$$P(m) = \frac{1}{Z} e^{-\beta E_m} \tag{6}$$

where β is the inverse temperature, E_m is the energy of the microstate and Z is the partition function. To simulate the system, we can, in principle, directly sample from this distribution. However, in general, we do not know the partition function Z. Hence, we use a different technique known as the *Metropolis Algorithm*, a common algorithm used to perform Monte Carlo simulations. The algorithm is as follows:

- First perform a trial change in the positions of all particles. To do this, uniformly generate 2N random numbers δx_1,..., δx_N, δy_1,..., δy_N between $-\delta_0$ to δ_0, where N is the number of particles and δ_0 is the maximum trial displacement.
- Compute the resulting change in energy ΔE of the system.
- If $\Delta E \leq 0$, the trial change in positions is accepted.
- If $\Delta E > 0$, generate a uniform random number r between 0 and 1, and compute $w = e^{-\Delta E/T}$, where T is the temperature of the system.. If $r \leq w$, then the trial change is accepted. Otherwise, the trial change is rejected.
- Repeat a large number of times.

Surprisingly, we found the system to be quite sensitive to the value chosen for the maximum trial displacement δ_0. For values that are too low, the system takes a long time to equilibrate, and for values that are too high, most trial changes are rejected, leading to the system not reaching equilibrium at all and simply staying in its equilibrium configuration. In this project, we have used $\delta_0 = 0.015$.

ii Radial Distribution Function

To understand more about the structure of many-particle systems, it helps to understand how correlated the particles are to each other. The *radial distribution function* $g(\vec{r})$ is one measure of this correlation. It is defined as follows: in a system of N particles in volume V, if any particle is chosen to be at the origin, then the mean number of other particles between $\vec{r}$ and $\vec{r} + d\vec{r}$ is defined to be $\rho g(\vec{r})\,\mathrm{d}\vec{r}$, where ρ is the number density N/V. The normalisation condition is

$$\rho \int g(\vec{r})\,\mathrm{d}\vec{r} = N - 1 \approx N \tag{7}$$

For spherically symmetric interactions, such as the Lennard-Jones potential, $g(\vec{r})$ depends only on the $r = |\vec{r}|$.

$g(r)$ can be thought of as the ratio between the local number density and the global number density of the system. Hence, for an ideal gas, we expect that the local density is equal to the global density, so $g(r) = 1$ for all r. This also reflects the fact that there are no correlations between the particles. For the Lennard-Jones potential, we expect almost no particles to be very close to each other due to the highly repulsive forces between the particles at very low distances. Hence, we expect $\lim_{r\to 0} g(r) = 0$. We also know that the Lennard-Jones potential falls off at large distances, so we expect the particles to be uncorrelated at such large distances. Hence, we expect $\lim_{r\to\infty} g(r) = 1$.

We can use the radial distribution function to calculate other thermodynamic properties of interest, such as the mean potential energy per particle U/N and the mean pressure per particle P/N. These quantities are given by the following equations:

$$\frac{U}{N} = \frac{\rho}{2} \int g(r)u(r)\, \mathrm{d}\vec{r} \tag{8}$$

$$\frac{PV}{NT} = 1 - \frac{\rho}{2Td} \int g(r) \frac{\mathrm{d}u(r)}{\mathrm{d}r}\, \mathrm{d}\vec{r} \tag{9}$$

Here, d is the number of dimensions.

We compute the radial distribution function as follows:

- Choose one particle as the origin and compute distances to all other particles. These distances are binned with a bin width of dr.
- Compute the number of particles in each distance bin.
- Repeat this for all other particles chosen at the origin, and compute the average number $n(r)$ of particles in each distance bin.
- We normalise $n(r)$ by dividing it by the shell area, given by $\pi(r+dr)^2 - \pi r^2$, the number density ρ and $N/2$, which comes from the fact that there are a total of $N(N-1)/2$ distances considered. The result of this normalisation of $n(r)$ is $g(r)$.

The bin width dr should be small enough so that important features of $g(r)$ are seen, but large enough so that each bin has a significant enough contribution. In this project, we have used $dr = 0.025$.

iii Widom Insertion Method

We know that the chemical potential is given by

$$\mu = \left(\frac{\partial F}{\partial N}\right)_{V,T} = -T \lim_{N\to\infty} \ln \frac{Z_{N+1}}{Z_N} \tag{10}$$

where Z is the partition function. This comes from using $F = -T \ln Z$ and the limit definition of the derivative. The ratio Z_{N+1}/Z_N is the average of $e^{-\Delta E/T}$ over all possible states of the added particle with the added energy ΔE. Hence, we essentially need to add an imaginary particle and compute the change in energy ΔE to compute the chemical potential. This is the Widom insertion method. It is as follows:

- Run the Monte Carlo simulation until equilibrium is reached.
- In the next Monte Carlo step, choose a random position (x_0, y_0) in the lattice.
- Compute the change in energy ΔE that would have occurred if a particle was placed at the position (x_0, y_0).
- Compute $e^{-\Delta E/T}$.
- Repeat the above three steps for many Monte Carlo steps after equilibration and compute $\langle e^{-\Delta E/T}\rangle$.
- Compute $\mu = -T \ln\langle e^{-\Delta E/T}\rangle$.

Here, the μ calculated is the excess chemical potential, which only includes the chemical potential due to position, and does not include the chemical potential due to the momentum degrees of freedom, which is the chemical potential of an ideal gas.

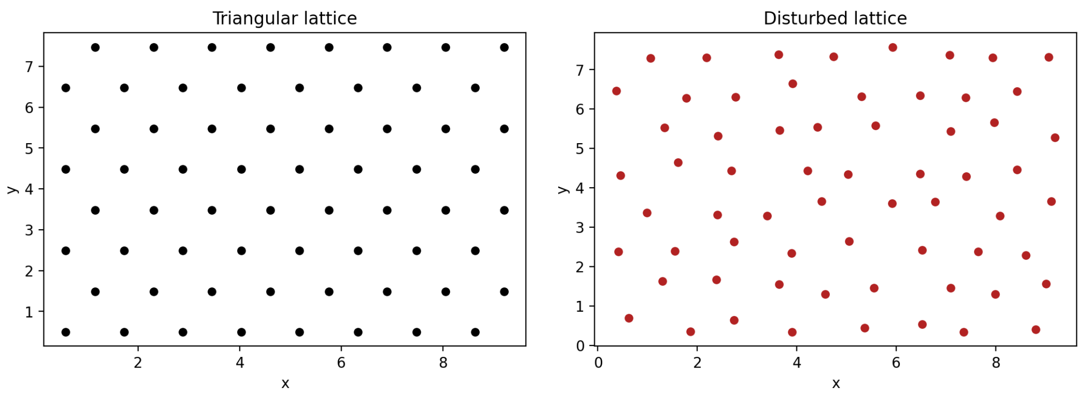
III. Setting Up the Simulations

i Initial Configurations

We run the Monte Carlo simulations on a triangular lattice of 8×8 particles with periodic boundary conditions. We use the following two initial configurations:

1. Perfect lattice
2. Disturbed lattice

The perfect lattice is simply a perfectly triangular lattice made of points placed at vertices of equilateral triangles. The disturbed lattice is created from the perfect triangular lattice by providing a randomized "kick" to each of the particles of the lattice. The magnitude of this randomized kick is set externally to be quite small, while the direction is random, which is what makes the kick random. In this project, we have generated a disturbed lattice using a displacement magnitude of 0.2 for the randomized kicks.

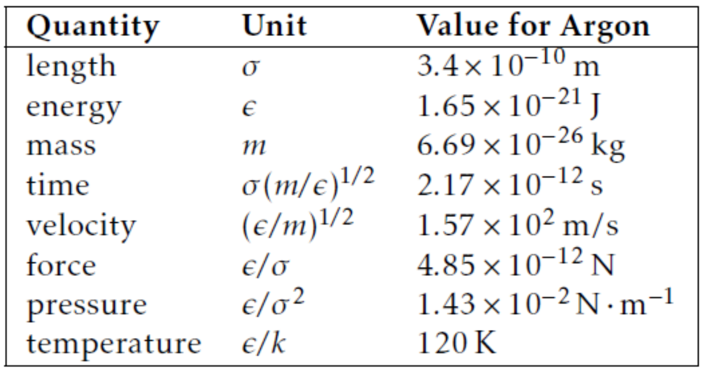


(a) Perfect triangular lattice (b) Disturbed triangular lattice

Figure 2: Initial configurations

ii Simulation Parameters and Units

The units in terms of which all the quantities in the simulation are expressed are given below. These help us to convert from our "code units" to the units used in the real-world -



Quantity	Unit	Value for Argon
length	σ	3.4×10^{-10} m
energy	ϵ	1.65×10^{-21} J
mass	m	6.69×10^{-26} kg
time	$\sigma(m/\epsilon)^{1/2}$	2.17×10^{-12} s
velocity	$(\epsilon/m)^{1/2}$	1.57×10^{2} m/s
force	ϵ/σ	4.85×10^{-12} N
pressure	ϵ/σ^2	$1.43 \times 10^{-2}\,\mathrm{N \cdot m^{-1}}$
temperature	ϵ/k	120 K

Figure 3: Units used in the simulation (Image Source: Tobochnik and Gould)

Since the units of various quantities in our code are clarified, we now list down the values of the parameters used in the simulation. It is important to note that all these quantities (which usually have physical units) are now mentioned in the units quoted above (e.g. T=3 signifies a temperature of $3 \times 120 = 360K$)

- The number of particles is $N = 64$, arranged in a 8×8 lattice. This is kept constant throughout all the simulations.
- The ratio between the length of the lattice in the y direction L_y and the length of the lattice in the x direction L_x (for all densities) $= \sqrt{3}/2$
- The density of the system is given by $\rho = N/V$, where V in this case is the **area** of the system since we are working in 2 dimensions. However, for consistency, we refer to it as a (2-dimensional) volume. The volume is given by $Lx \times Ly$, and we run the simulations for 13 different density values. These are motivated by a desire to cover a significant range of densities while also having the resolution to see sudden change in behavior at any potential critical value of densities. Thus, the ρ values simulated are - 3.492, 1.078, 0.873, 0.760, 0.558, 0.388, 0.285, 0.218, 0.172, 0.139, 0.115, 0.097, 0.082
- Temperature values ranging from $T = 0.1$ - 3 (in the units mentioned above). Number of temperature values used = 30 (equally distributed between 0.1-3)
- Magnitude of directionally-random "kicks" to create the the initial disturbed lattice = 0.2.
- Maximum trial displacement δ_0 in each Monte Carlo step = 0.015. (`maxdx`)
- Bin width dr for the radial distribution function = 0.025.
- Number of Monte Carlo steps for perfect lattice for low temperatures = 10,000
- Number of Monte Carlo steps for perfect lattice for high temperatures = 100,000
- Number of Monte Carlo steps for disturbed lattice for all temperatures = 10,000

The reason for choosing these values as the number of Monte Carlo steps is given by the number of steps required for the system to reach equilibrium at a particular initial configuration and a particular temperature, as discussed in Section 4.1.

IV. Results

i Stabilisation of Energy

We performed Monte Carlo simulations and obtained the energy at each Monte Carlo step for both initial configurations.

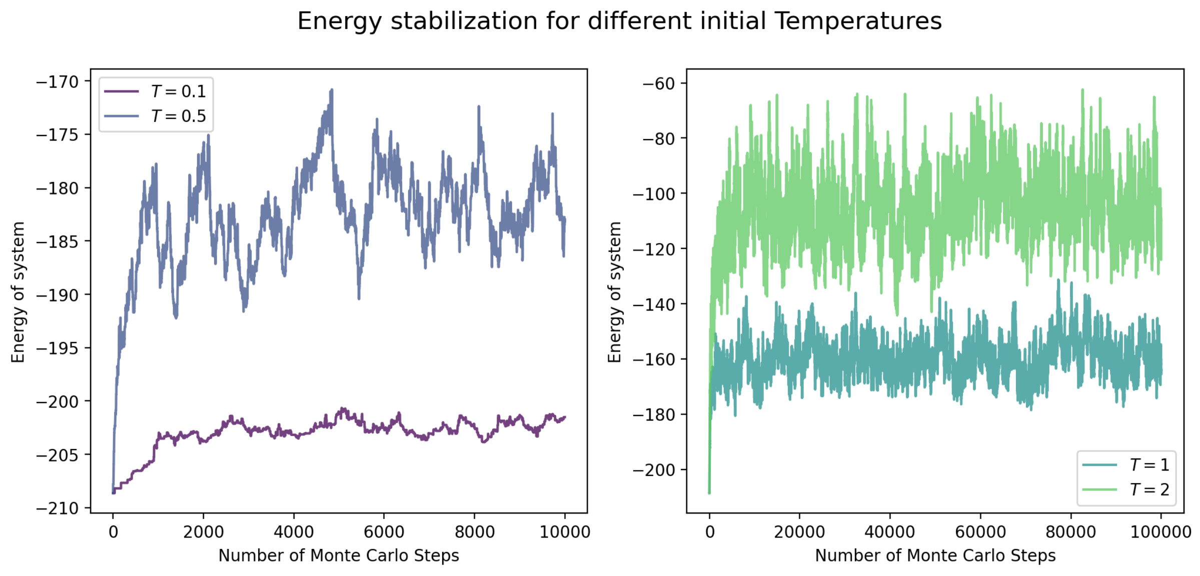


Figure 4: Energy stabilisation for perfect lattice initial configuration for T = 0.1, 0.5 (left), 1 and 2 (right). Notice the number of monte carlo steps requied by the higher temperature simulations (right) to reach equilibrium

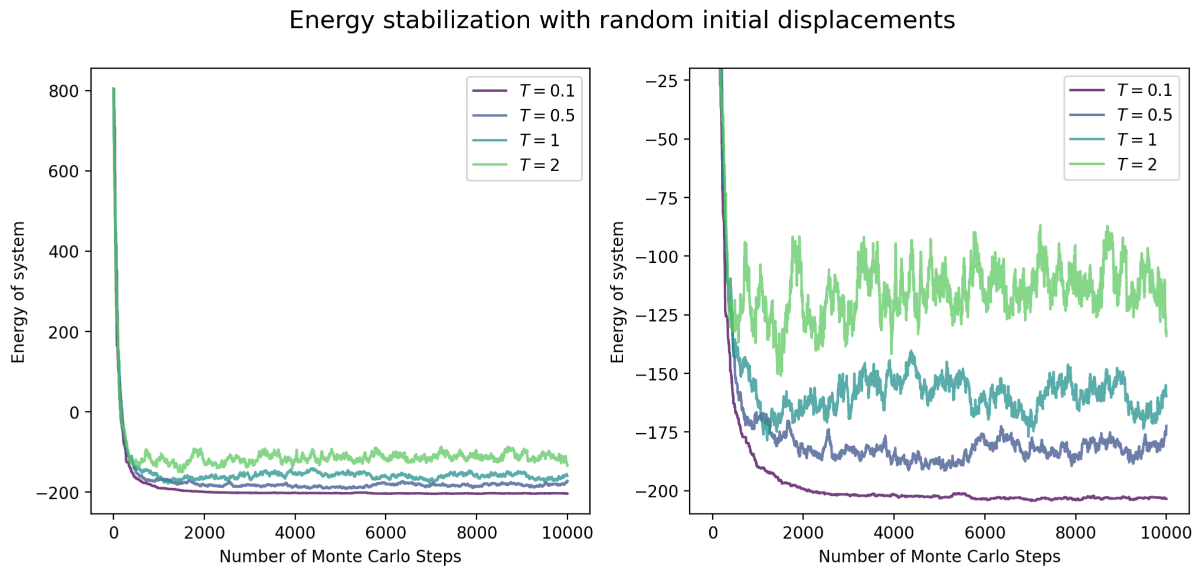


Figure 5: Energy stabilisation for perfect lattice initial configuration (displacement of magnitude 0.2) for T = 0.1, 0.5, 1 and 2. The graph on the right is the graph on the left zoomed-in at the equilibrium energies.

Interestingly, we see that the initial potential energy shoots up on slightly disturbing the lattice, even though the equilibrium energies are the same for the same temperature in both cases (see Figure 5).

As shown in Figure 4, for temperatures $T = 1, 2$ 100,000 Monte Carlo steps were required to see equilibration in the case of the perfect lattice initial configuration, while only 10,000 Monte Carlo steps were required to see equilibration for the same temperatures in the case of the disturbed lattice (see Figure 5). Hence, we can conclude that for the disturbed lattice initial configuration, the system reaches equilibrium much faster compared to the perfect lattice configuration. For this reason, to obtain all the other results of the project, we used the disturbed lattice as the starting configuration, with 10,000 Monte Carlo steps. We use the last 5000 Monte Carlo steps as equilibrium configurations while computing the radial distribution function, its derived quantities and the chemical potential μ.

ii Radial Distribution Function

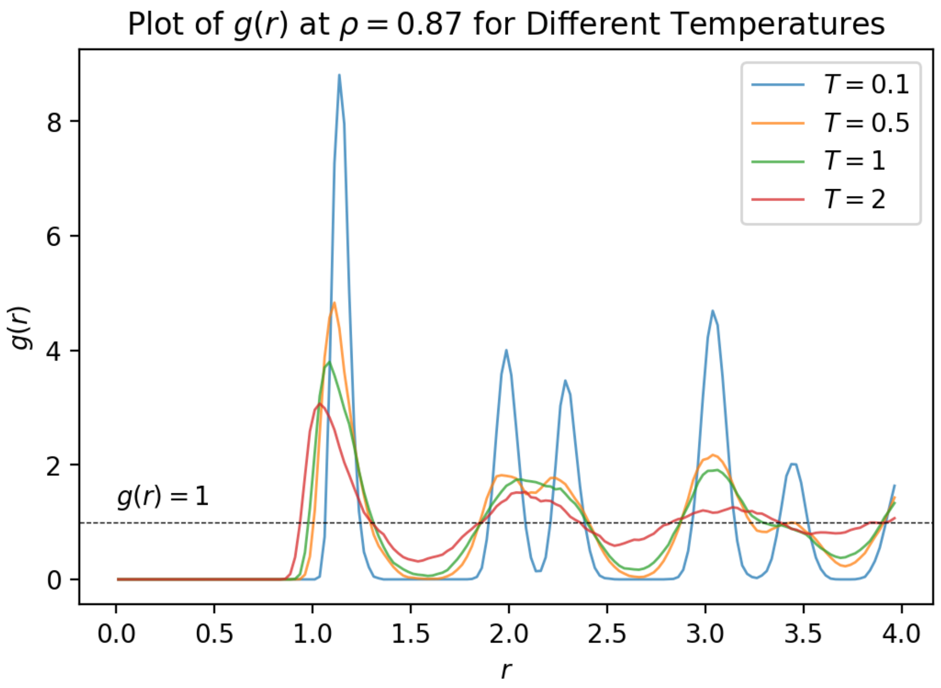


Figure 6: The radial distribution function for $\rho = 0.87$ and $T = 0.1$, 0.5, 1, 2.

From Figure 6, we see that for $\rho = 0.87$, $g(r)$ for all the temperatures approach 1 as r becomes larger, as expected, since the particles become uncorrelated at very large distances. Further, we see that for $T = 0.1$, we have sharp, high peaks. However, the higher the temperature, the more broad and low the peaks become. Sharp peaks correspond to periodicity and in the structure of the particles, so $T = 0.1$ must correspond to a solid, and the higher the temperature, the more fluid-like the particles behave, according to the features of the peaks of $g(r)$. Since there still seems to be some structure, the system seems to be a liquid for temperatures of $T = 0.5, 1$ and $T = 2$.

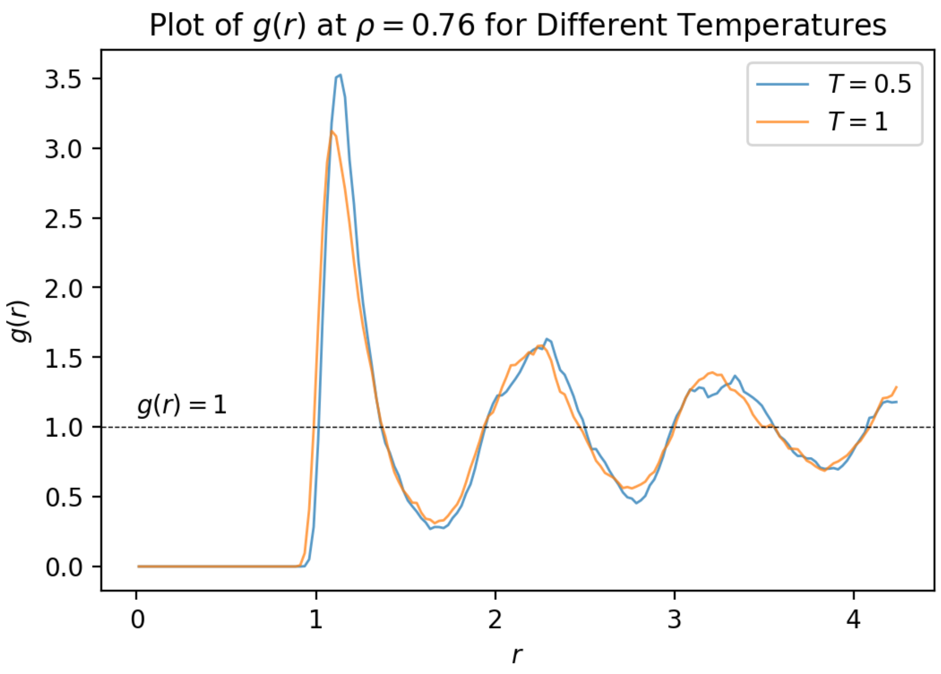


Figure 7: The radial distribution function for $\rho = 0.76$ and $T = 0.5$, 1.

These curves in Figure 7 also have broad, blunt peaks and are hence indicative of a liquid, since they too seem to still have some periodicity and structure.

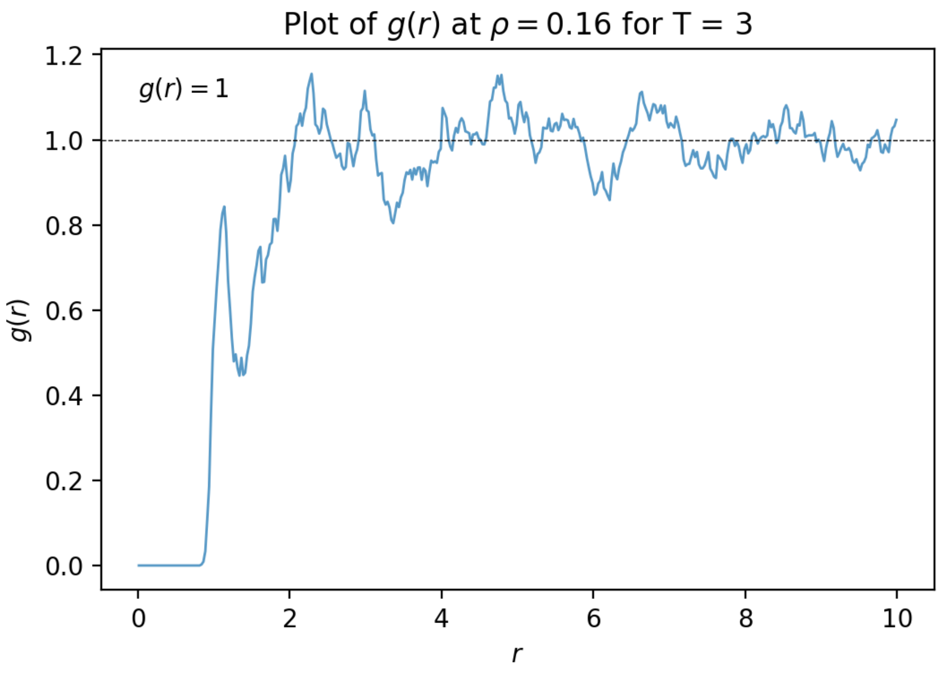


Figure 8: The radial distribution function for $\rho = 0.16$ and $T = 3$.

In Figure 8, it seems that $g(r)$ has lost all periodicity and structure, so here the system seems to be a gas. In particular, the density is low, so the system is a dilute gas. We see that the graph appears quite jagged, rather than smooth, and it is unclear why this is the case.

iii Energy and Pressure

Using the radial distribution function, we computed the energy and pressure of the system for a range of temperatures and densities.

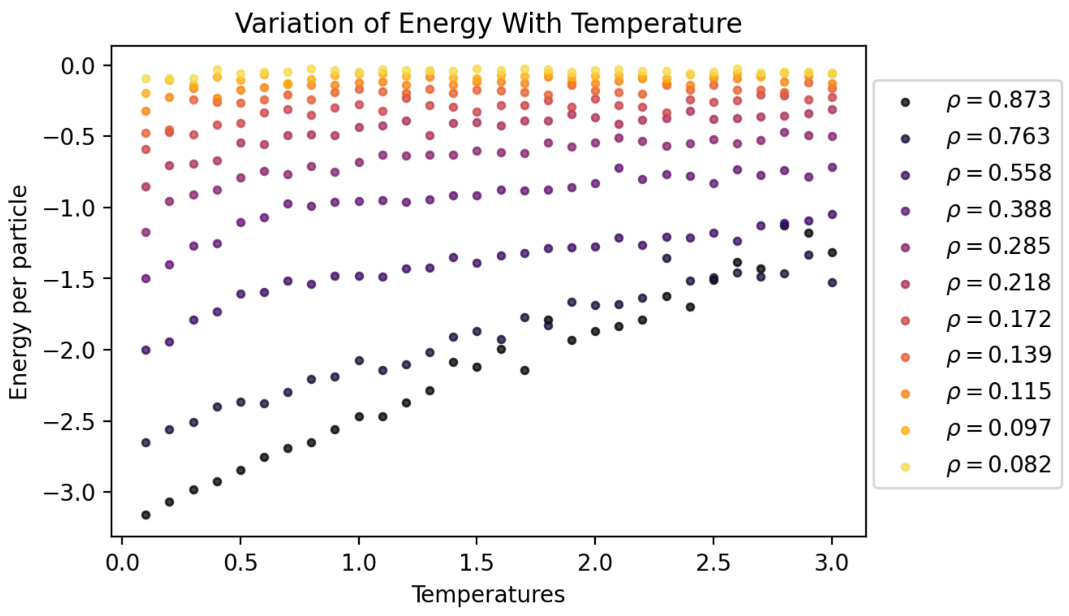


Figure 9: Plot of energy against temperature for various densities.

From this plot, we see that the energy becomes increasingly less negative and approaches 0 as both temperature as well as density become larger. From Figure 1, we can see that as r increases beyond the value of r corresponding to the minima, the potential energy should become less negative, and approaches 0. Lower density means that there is more separation between particles, and increasing the temperature should also have this effect, so energy is showing the expected trend.

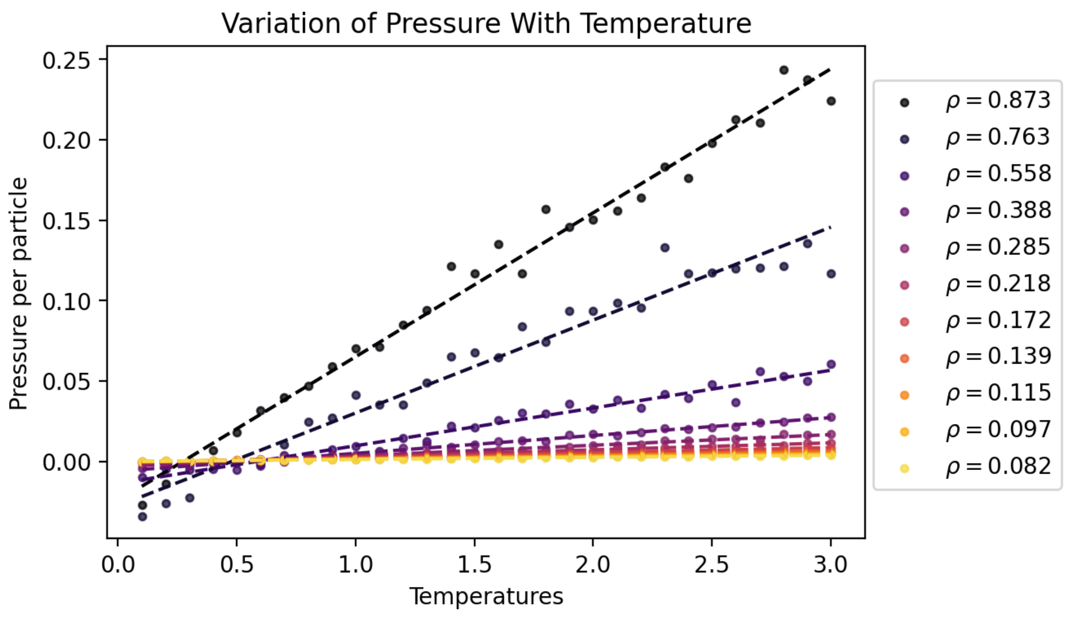


Figure 10: Plot of pressure against temperature for various densities.

From this plot, we see that the pressure seems to be increasing linearly with temperature and that the rate of this increase seems to increase with density. To further investigate the behaviour of the pressure, and the system in general, we model our system using the van der Waals equation of state.

iv Van der Waals Constants

From Equation (4), we have the following van der Waals equation of state:

$$(P + a\rho^2)\left(\frac{1}{\rho} - b\right) = T$$

Rewriting this and dividing both sides by the number of particles N, we have the following expression for the pressure per particle:

$$\frac{P}{N} = \frac{T}{N\left(\frac{1}{\rho} - b\right)} - \frac{a}{N}\rho^2 \tag{11}$$

This tells us that the graph of pressure against temperature should be a straight line, which is a result that we can see in Figure 10. If the graph of P against T has slope m_{PT} and y-intercept c_{PT}, then we have

$$\frac{1}{m_{PT}} = \frac{N}{\rho} - Nb \tag{12}$$

$$c_{PT} = -\frac{a}{N}\rho^2 \tag{13}$$

Hence, by plotting c_{PT} against ρ^2, and $1/m_{PT}$ against $1/\rho$, we can estimate the van der Waals parameters a and b, given that our system can be described by the van der Waals equation of state.

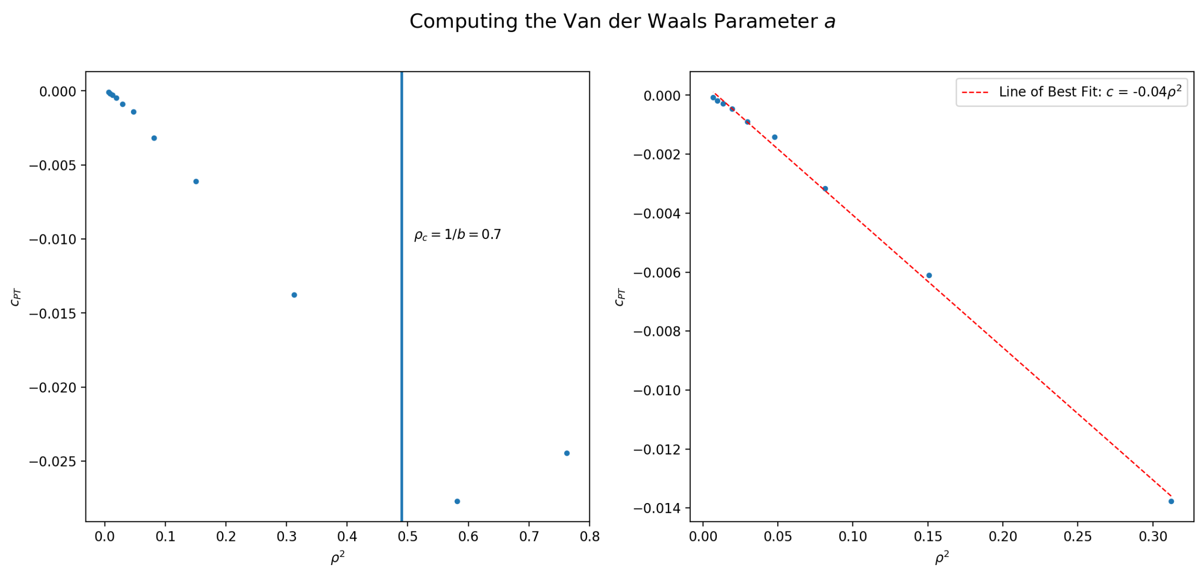


Figure 11: Plot of c_{PT} against ρ^2. The graph on the right shows only the linear part of the graph on the left.

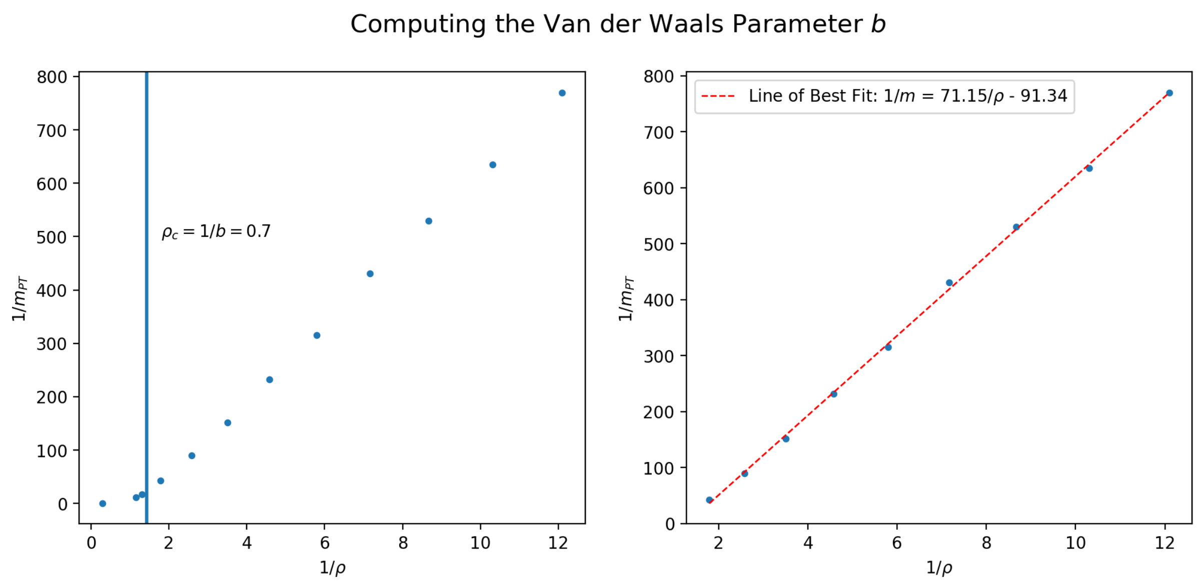


Figure 12: Plot of $1/m_{PT}$ against $1/\rho$. The graph on the right shows only the linear part of the graph on the left.

We find that our plots only seem to be linear for a particular range of densities. There seems to be a critical density above which the plots are not linear, i.e. they are not described by the van der Waals equation of state. This critical density is hypothesised to be the same critical density for which m_{PT} is undefined. This density is $1/b$. Hence, by estimating the value of b, we can estimate the value of the critical density ρ_c.

The slope of the line of best fit of the linear part of $1/m_{PT}$ against $1/\rho$ is 71.15 (see Figure 11). This value is close to the number of particles $N = 64$. Further, this line has a negative intercept. These two properties of the line indicate that our simulated system is indeed described by the van der Waals equation of state in this range of temperatures and densities.

The y-intercept of the graph is -91.34. Hence, from Equation (12), we get

$$b = -\frac{-91.34}{64} = 1.43 \tag{14}$$

This gives us the following critical density ρ_c:

$$\rho_c = \frac{1}{1.43} = 0.70 \tag{15}$$

From Figure 11, we see that the line of best fit of the linear part of the graph has a y-intercept of 0.00 (correct to 2 decimal places), and it has a negative slope. These two properties of the line help further confirm that our simulated system is described by the van der Waals equation of state in this range of temperatures and densities.

The slope of the graph is -0.04. Hence, from Equation (13), we get

$$a = -(64)(-0.04) = 2.87 \tag{16}$$

We used the `linregress` function from the `scipy.stats` module to obtain the errors in all the slopes and intercepts. Using these, we obtained the following error estimates:

$$\delta a = 0.36,\ \delta b = 0.09,\ \delta \rho_c = 0.09 \tag{17}$$

Hence, we get the following values of a, b and ρ_c:

$$a = 2.87 \pm 0.36,\ b = 1.43 \pm 0.09,\ \rho_c = 0.70 \pm 0.09 \tag{18}$$

It is this value of critical density at which the vertical line in Figure 11 is plotted. We can see that it is only beyond this critical density for which the van der Waals equation of state stops describing our system. This suggests that our hypothesis is true - computing $1/b$ does seem like a good way to estimate the critical density of our system.

As a check, we perform the same analysis for an ideal gas, which we simulate by using the same algorithms but without a potential.

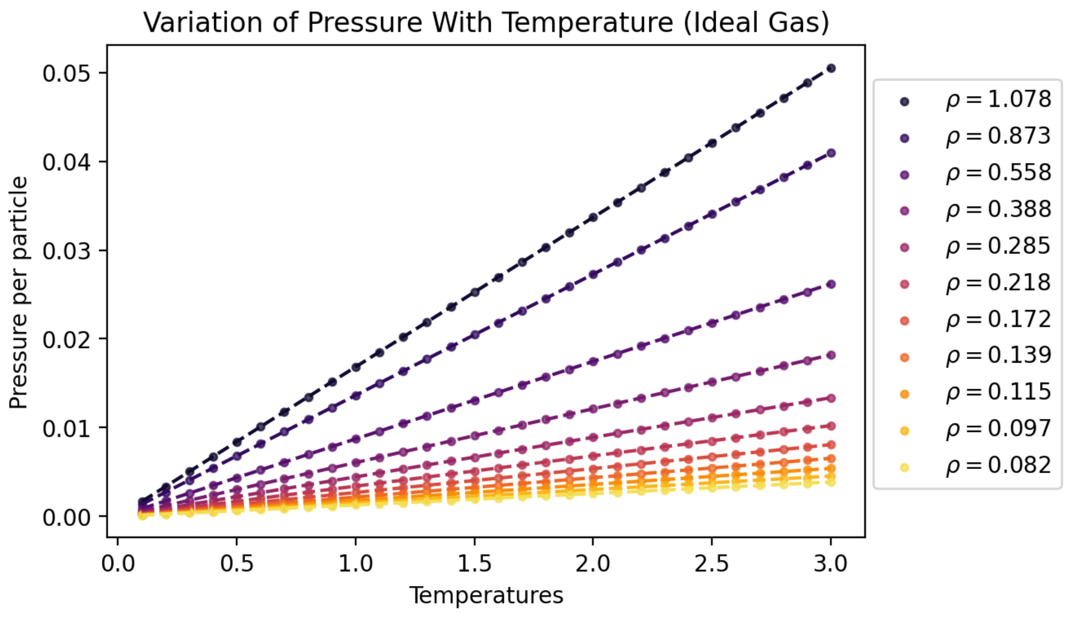


Figure 13: Plot of pressure against temperature for an ideal gas

We see that pressure varies linearly with temperature as expected, with the slope given by $1/V = 1/(L_xL_y)$. We now compute b for an ideal gas, expecting it to be 0.

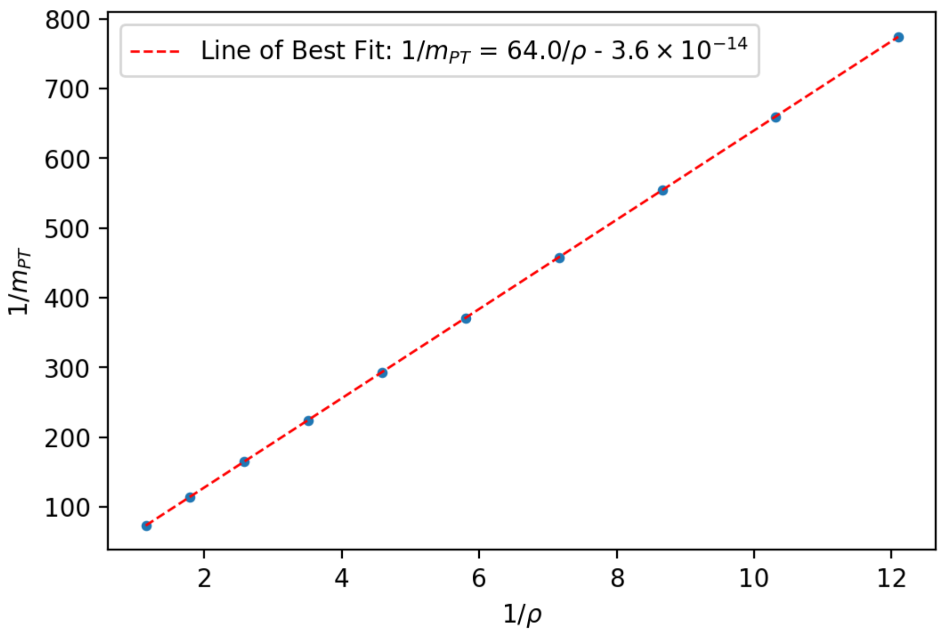


Figure 14: Plot of $1/m_{PT}$ against $1/\rho$ for an ideal gas.

The y-intercept of the graph is 3.6×10^{-14}. From Equation (12), we get

$$b = -\frac{3.6 \times 10^{-14}}{64} = -5.62 \times 10^{-16} \tag{19}$$

Hence, the value of b is found to be almost 0. Therefore, this method works. We have not done this analysis to find a because the y-axis contains very small numbers (of the order of 10^{-18}), so computational errors dominate. Thus, we only use the value of b as a check.

The fact that our code reproduces theoretically expected results for an ideal gas system when run with the correct conditions (turning off the Lennard Jones potential) acts as a reliability check for the code. This helps us ensure that any results obtained from these simulations for the Lennard Jones potential are genuine, and not artifacts of the computation itself. On getting this reassurance, we turn back on our potential, simulate the system, and look at the variation of the chemical potential of the system (μ) for different densities and temperatures in the next section.

v Chemical Potential

Using the Widom insertion method, we compute the chemical potential of our system for a range of temperatures and densities.

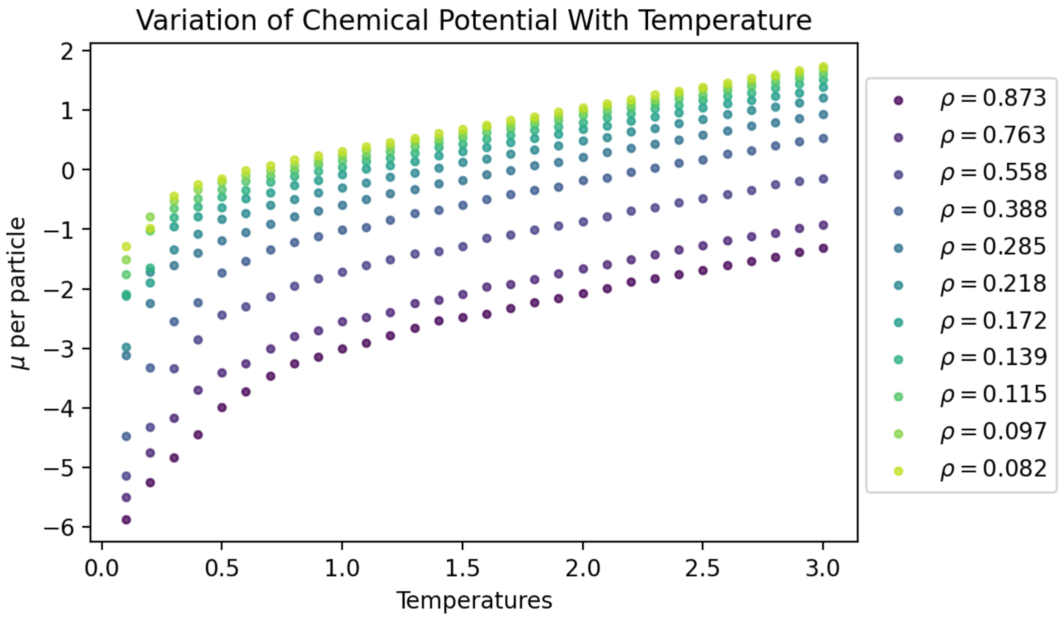


Figure 15: Plot of the chemical potential against temperature for various densities.

We can see that the chemical potential is mostly negative. However, it seems to increase for both higher densities and higher temperatures. This tells us that the system more readily accepts an added particle when it is at a lower temperature and density, and that it becomes more difficult in some sense to add a particle if the system has a higher temperature and density.

V. Conclusion

We performed Monte Carlo simulations using the Metropolis algorithm to simulate a Lennard Jones system, and by computing the radial distribution function for this system for a range of different densities and temperatures, we have shown that our system exhibits characteristics of both solids and fluids for different densities and temperatures. By then using the radial distribution function, we found both the energy and pressure as a function of temperature for various densities. Modelling our Lennard-Jones system of particles as a van der Waals gas, we used the values of pressure to compute the van der Waals constants, which we found to be the following:

$$a = 2.87 \pm 0.36 \qquad b = 1.43 \pm 0.09$$

Using this value of b, we estimated a critical density of our system to be $\rho_c = 0.70 \pm 0.09$. This was also found to be the density beyond which the van der Waals equation of state no longer described our system. For densities below the critical density, the van der Waals equation of state seems to describe the system well. We also used the Widom insertion method to compute the chemical potential for our system using over a range of temperatures for different densities, and from this we concluded that the system more readily accepts particles when it is at lower densities and temperatures.

VI. Challenges

1. **Problems in the Widom Insertion Technique for high densities:** The usual Widom Insertion function seems to only work for lower densities. On plotting the chemical potential μ against the temperature T for higher densities, we got highly fluctuating graphs. This may be because, at higher densities, the likelihood of randomly choosing a position very close to the position of another particle is much higher, due to which the resulting change in energy would be very high since the Lennard-Jones potential is very high at close distances. Hence, we made a modification: we only considered the potential energy contributions from particles greater than a distance of r/σ from the imaginary particle. While this seemed to have fixed the problem, allowing us to find μ for higher densities (as seen in Figure 15), it is difficult to say if this is a valid method of computing μ, especially since the Widom insertion method is primarily used for low densities.

2. **Long run-times for code:** Monte-Carlo simulations can in general become very time-consuming very quickly. Our initial difficulties of getting large numbers of Monte Carlo sweeps done over 64 particles repeatedly in relatively quick time were solved by `Numba` - a high-performance open-source JIT compiler for Python. This allowed us to do slightly more than very basic simulations. However, as we slowly increased the number of Monte-Carlo simulations and RDF calculations to expand the scope of the project (such as simulating for multiple temperatures and multiple densities), the code started taking longer runtimes even with Numba. We then. Thus, to avoid running the entire code every single time to do further analysis, we instead ran the appropriate big runs once and saved the E, P, μ data for every temperature and every density in separate `.csv` files. This simple technique allowed us to do all further analysis on the $E - T, P - T, \mu - T$ data much more quickly than running everything from scratch again.

3. **Implementing integral check for RDF:** The normalisation condition for the radial distribution function tells us that integrating the radial distribution function over the entire volume should give us $N - 1$, where N is the number of particles, so it should give us around the value of 63 in our case. However, when we performed this integration, we consistently got values of around 40. In order to check if our radial distribution function was working correctly, we compared the energy values computed using the radial distribution function to the mean of the energies computed at each Monte Carlo step in the Metropolis algorithm. These energies matched quite well, so it seemed that our radial distribution function was working properly.

VII. Future Work

1. **Animating the system:** We aim to animate our system so that we can see the change in the configuration and the radial distribution function over temperature and density
2. **Diffusion coefficient:** We also aim to compute a diffusion coefficient D by finding the mean-squared displacement of the particles, which is slightly difficult with periodic boundary conditions. We want to plot ρD against ρ, which should give us a critical density.
3. **Comparison with empirical data:** We would like to compare estimates produced by our simulation (such as the van der Waal constants) to empirical data of real systems such as Argon, Neon, Helium, etc.

Acknowledgements

We would like to thank Professor Bikram Phookun and our graduate assistant Philip Cherian for providing us with the opportunity to work on this project. We would also like to thank Philip Cherian for his guidance throughout the project. Lastly, we would like to thank our batchmates for insightful discussions and providing feedback.

References

[1] Tobochnik and Gould, *An Introduction to Computer Simulation Methods*.

Appendix

Listing 1: Jupyter Notebook - All function definitions

```
# Importing the necessary libraries

import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.dpi'] = 200
plt.rcParams['figure.facecolor']='w'
from matplotlib.animation import FuncAnimation
from numba import njit   # to speed up code
from scipy.stats import linregress

@njit
def LJpot(r, sigma=1, epsilon=1):                    # defining Lennard
    Jones Potential
    term = (sigma/r)**12 - (sigma/r)**6
    return 4*epsilon*term

@njit
```

```python
def LJforce(r, sigma=1, epsilon=1):              # defining Lennard Jones force
    term = 2*(sigma/r)**12 - (sigma/r)**6
    return (24*epsilon)*(term)/r


@njit
def get_pe(step_pos, Lx, Ly):

    '''
    Get the 2D Lennard-Jones potential energy for a system of `N` masses.

    Parameters:
    -----------
        pos : An `(N x 2)` numpy array of 2 elements storing the `x` and `y` values of each particle.

    Returns:
    --------
    The function returns one value:
        potential : The combined potential energy of the `N` masses.
    '''
    potential = 0
    for i in range(len(step_pos)-1):
        for j in range(i+1,len(step_pos)):
            new_dist = pbc_distance(step_pos[i]-step_pos[j],Lx=Lx,Ly=Ly)
            # Magnitude of actual distance
            r=np.sqrt((new_dist[0])**2+(new_dist[1]**2))

            potential += 4*((1/r)**12-(1/r)**6) # Summing potential energy between each pair of particles
    return potential


@njit
def pbc_distance(rij,Lx, Ly):

    '''
    Get the correct distance between two particles, accounting for periodic boundary conditions.

    Parameters:
    -----------
        rij : A numpy array of 2 elements storing the `x` and `y` values of the separation `rij`.

    Returns:
    --------
    The function returns one value:
        rij : The corrected r_ij, including the effects of periodic
       boundary conditions.
    '''

    # Computing the true distance, taking into account
    # periodic boundary conditions

    if abs(rij[0])>0.5*Lx:
        rij[0]=rij[0]-Lx*np.sign(rij[0])
    if abs(rij[1])>0.5*Ly:
        rij[1]=rij[1]-Ly*np.sign(rij[1])

    return rij


@njit
def init_pos(Lx, Ly, Nx, Ny):

    '''
    Assign the initial positions of the particles on a regular lattice

    Parameters:
    -----------
        Lx    : x-length of the lattice
        Ly    : y-length of the lattice
        Nx    : An integer number of particles in the x-direction
        Ny    : An integer number of particles in the y-direction

    Returns:
    --------
    The function returns one value:
        ipos : A numpy array of (N x 2) elements storing the x and y
       coordinates of N = Nx * Ny particles.
    '''

    N = Nx*Ny                                         # Total number of
       particles in the lattice

    dx = Lx/Nx                                        # Spacing between
       particles along the
    dy = Ly/Ny                                        # x and y axes

    ipos = np.zeros((N,2))                            # Empty array to
```

```
        store positions

    n=0                                                 # Counter to count
        the number of particles

    for x in range(Nx):                                 # Loop over all
        particles
        for y in range(0,Ny,2):
            ipos[n] = [dx/2 + dx*x, dy/2 + dy*y]        # Assign positions
                to each particle
            n+=1
        for y in range(1,Ny,2):
            ipos[n] = [dx + dx*x, dy/2 + dy*y]
            n+=1                                        # Increment counter

    return ipos

    @njit
def pos_randomizer(initpos, magn):
    '''
    Takes in positions of lattice points, gives each of them a 'kick'
        of fixed magnitude 'magn' in a
    uniformly random direction (random for every point) and returns the
        new position array

    '''

    randomized_pos = np.zeros(shape= (len(initpos), 2))

    for i in range(len(initpos)):

        r = np.random.uniform(0,2*np.pi)
        x = np.cos(r)*magn
        y = np.sin(r)*magn

        randomized_pos[i][0] = initpos[i][0] + x
        randomized_pos[i][1] = initpos[i][1] + y

    return randomized_pos

    @njit(parallel=True)
def onemcs(pos,PE,Nx,Ny,T,Lx, Ly, maxdx, idealgas=False):
    '''
    This function performs one Monte-Carlo "sweep" using the standard
        Metropolis algorithm, changing every particle's
    position with probability 1 if the trial change is accepted
    '''
```

```python

    N=Nx*Ny                                                # total number of particles = Nx * Ny

    delx=np.random.uniform(-maxdx,maxdx,size=(N,2))        # choosing random trial displacement

    new_pos = pos+delx

    # Periodic boundary conditions
    for i in range(N):
        if new_pos[i][0]>Lx:
            new_pos[i][0] = new_pos[i][0] - Lx

        if new_pos[i][0]<-Lx:
            new_pos[i][0] = new_pos[i][0] + Lx

        if new_pos[i][1]>Ly:
            new_pos[i][1] = new_pos[i][1] - Ly

        if new_pos[i][1]<-Ly:
            new_pos[i][1] = new_pos[i][1] + Ly


    if idealgas == False:                        # encoding condition to turn on/off potential (for ideal gas)
        dE=get_pe(new_pos,Lx,Ly) - PE

    else:
        dE = 0

    if dE<=0:                               # If the system loses energy or there is no change in the energy of the system
        pos=new_pos
        PE = PE + dE
    else:                                    # If the system gains energy
        r = np.random.uniform(0,1)
        prob = np.exp(-dE/T)
        if r<prob:
            pos=new_pos                  # The trial change in x is accepted
            PE = PE + dE


    return pos, PE

    @njit(parallel=True)
def simulate(n_mc, Lx, Ly, Nx, Ny, T, maxdx, initpos_random=False, pot=True, idealgas=False):
    '''
    Carries out multiple consecutive mcsweeps on the system
    '''
    # Defining quantities
    N = Nx*Ny
    V = Lx*Ly
    rho = N/V
    dr = 0.025

    # Defining arrays
    PE = np.zeros(n_mc)
    pos = np.zeros(shape=(n_mc,N,2))    # Arrays to store the history of the particles'
    dist = np.zeros(shape=(n_mc,N,2))

    ipos0 = init_pos(Lx, Ly, Nx, Ny)                                           # perfect lattice positions
    ipos0_random = pos_randomizer(ipos0, rand_init_disp_magn)                  # randomized initial positions

    if initpos_random == False:                          # for perfect lattice
        pos[0] = ipos0
        PE[0]  = get_pe(step_pos=ipos0,Lx=Lx, Ly=Ly)
    else:
        pos[0] = ipos0_random                                # for disturbed lattice
        PE[0]  = get_pe(step_pos=ipos0_random,Lx=Lx, Ly=Ly)

    # calling onemcs an 'n_mc' number of times and feeding in output of previous run as input for next run
    for mc in range(1,int(n_mc)):
        pos[mc],PE[mc] = onemcs(idealgas=idealgas, pos=pos[mc-1],PE=PE[mc-1],Nx=Nx,Ny=Ny,T=T,Lx=Lx,Ly=Ly,maxdx=maxdx)

    return pos, PE   # returns finaly position of energy of system

    @njit
def compute_rdf(pos, Lx, Ly, N, dr, rdf_accumulator):
    '''
    Computes rdf (n(r)) for a given array of positions

    '''

    # Accumulate data for n(r)
    for i in range(N - 1):
        # Loop over indices i from 0 to N-2
        for j in range(i + 1, N):
            # Loop over indices j from i+1 to N-1 (avoiding self-
               comparison and duplicate pairs)

            # Apply periodic boundary conditions to get the separation
            dx = pbc_distance(pos[i] - pos[j], Lx, Ly)

            # Calculate the squared distance between particles i and j
            r2 = dx[0]**2 + dx[1]**2

            # Calculate the distance between particles i and j
            r = np.sqrt(r2)

            # Determine the bin (shell) index for the RDF accumulator
            bin = int(r / dr)  # dr is the shell width

            # Increment the RDF accumulator for the corresponding bin
            rdf_accumulator[bin] += 1

    return rdf_accumulator


@njit
def normalize_rdf(N, Lx, Ly, number_rdf_measurements, dr,
    rdf_accumulator):
    '''
    Normalizes a given n(r) by dividing it by total number of bins and
        the correct shell area
    '''
    #
    density = N / (Lx * Ly)
    L = min(Lx, Ly)
    bin_max = int(L / (2 * dr))
    normalization = density * number_rdf_measurements * N / 2

    # Initialize arrays to store results
    bin_centers = np.zeros(bin_max)
    rdf_values = np.zeros(bin_max)

    for bin in range(bin_max):
        r = bin * dr
        shell_area = np.pi * ((r + dr)**2 - r**2)
        rdf = rdf_accumulator[bin] / (normalization * shell_area)

        # Store results in arrays
        bin_centers[bin] = dr * (bin + 0.5)
        rdf_values[bin] = rdf

    return bin_centers, rdf_values


# Checking that the integral of RDF gives back N-1
def check_RDF(r,g,dr, rho):
    summ = 0
    for i in range(len(r)):
        summ += g[i]*np.pi * ((r[i] + dr)**2 - r[i]**2)
    summ = rho*summ
    return summ

    @njit
def pressure_RDF(r,g,rho,dr,V,T, d=2, idealgas=False):
    '''
    Inputs: r, g(r), rho (density), dr

    Returns mean pressure per particle using the formula given.

    '''
    summ = 0

    if idealgas == False:
        for i in range(len(g)):
            summ += g[i]*LJforce(r[i])*dr*r[i]**2

    denom = T*d
    summ = summ*np.pi*(rho/denom)         # accounting for 2 dimensional
        dr

    P_per_N = (1 + summ)*(T/V)            # 1 + sum to account for sign
       of LJ force

    return P_per_N



@njit
def energy_RDF(r, g, rho, dr, idealgas=False):
    '''
    Inputs: r, g(r), rho (density), dr

    Returns mean (potential) energy per particle using the formula
       given.
```

```python

    '''
    summ = 0

    if idealgas == False:
        for i in range(len(g)):
            summ += g[i]*LJpot(r[i])*dr*r[i]

    summ = summ*rho*np.pi        # accounting for 2 dimensional dr

    return summ

    @njit
def rdf_quantities(pos, dr, Lx, Ly, Nx, Ny, T, idealgas=False):

    N = Nx*Ny
    V = Lx*Ly
    rho = N/V

    # Setting up all initial quantities and arrays

    L = min(Lx, Ly)
    bin_max = int(L / (2 * dr))
    rdf_accumulator = np.zeros(int((0.5*np.sqrt(Lx**2+Ly**2))/dr))
    number_rdf_measurements = 0

    # to calculate RDF only from equilibrium configurations, the last
       50% of the mcsweeps are averaged over
    for mc in range(int(0.5*len(pos)),len(pos)):
        rdf_accumulator = compute_rdf(pos[mc], Lx, Ly, N, dr,
            rdf_accumulator)
        number_rdf_measurements+=1
    r, g = normalize_rdf(N, Lx, Ly, number_rdf_measurements, dr,
       rdf_accumulator)

    # Computing energy
    RDF_pe = energy_RDF(r, g, rho, dr, idealgas=idealgas)
    # Computing pressure
    RDF_pres = pressure_RDF(r,g,rho,dr,V,T, d=2, idealgas=idealgas)

    return r, g, RDF_pe, RDF_pres    # returns array of r,g and values
       of energy and pressure computed using rdf

    @njit
def wid(pos, T, Lx, Ly, Nx, Ny, idealgas=False):
    '''
    Uses the Widom Insertion Method to compute the chemical potential
```

```
        of the system.

    '''
    N = Nx*Ny
    exp = np.zeros(len(pos))                    # initializing array to zero

    for mc in range(int(0.5*len(pos)), len(pos)):              # averaging over only equilbrium configurations

        # Choose a random position inside the volume of box
        randpos = np.array([np.random.uniform(0,1)*Lx, np.random.uniform(0,1)*Ly])

        # Compute dE
        dE = 0
        # use potential only if idealgas is false (potential is tured on)
        if idealgas==False:
            for i in range(N):                                  # for each partice
                new_dist = pbc_distance(pos[mc][i]-randpos,Lx=Lx,Ly=Ly)    # find x,y distance of test particle from that particle
                r=np.sqrt((new_dist[0])**2+(new_dist[1]**2))           # find magnitude of that distance

                if r > 1:                     # only add this particle to dE if it is more than 1sigma distance away
                    dE += LJpot(r)
        # if potential is turned off (ideal gas scenario)
        else:
            dE = 0        # just set dE to zero

        exp[mc] = np.exp(-dE/T)              # store the relevant quantity in exp after every mc sweep, for every particle

    # Compute chemical potential using formula
    mu = -T*np.log(np.mean(exp))

    return mu              # return calculate chemical potential
```