Programming and Data Analysis Final Assessment Report

Wen Hans Tan sl22121@bristol.ac.uk

March / April 2024

1 Introduction

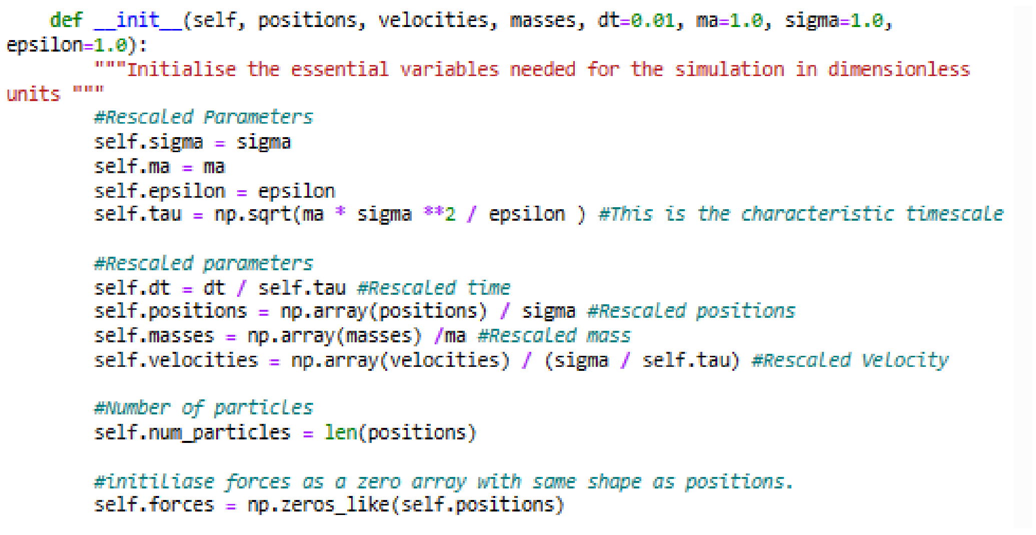
For my attempt at tackling this assignment, I've seperated each task (1,2,3) into seperate Python files. I used OOP (Object Orientated Programming). The general idea was to create **3 different classes** in each .py file and to code by inheritance as I progressed through the assignment. For example, the class in Task_2.py is a child class which inherits all the methods and properties from Task_1.py which is the parent class. As suggested by the instructions, I dissected the problem from simulating the N-body code, establishing boundaries, and analysing the effects of pressure with varying temperature or volume.

This report looks at examining the behaviour of argon/normal atoms at varying independent variables (volume / temperature). This particular idea can be tied to the conformity of the ideal gas law. Understanding these relationships is essential for practical reasons, because they are foundational for various engineering applications.

For clarification, I did my best in covering all aspects of good coding practice which includes readability, reusability, and understandability. Disclaimer: some minor corners have to be cut due to time constraint.

2 Producing N-body Code - Task 1

2.1 2 Particle Scenario with Simple Parameters



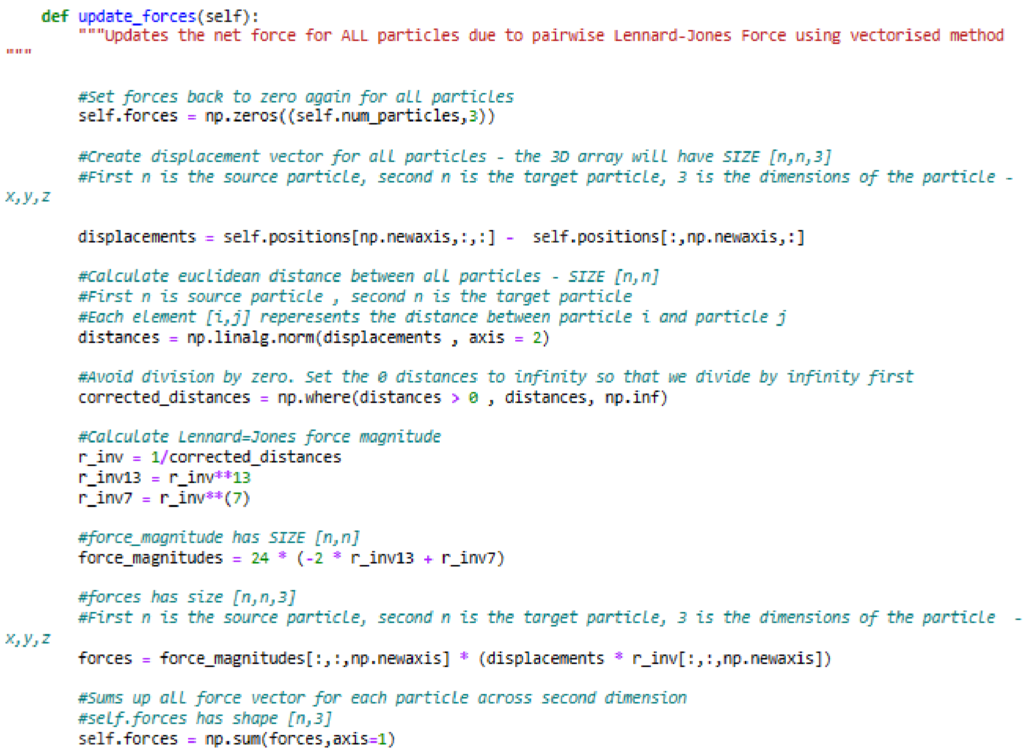
```
    def __init__(self, positions, velocities, masses, dt=0.01, ma=1.0, sigma=1.0,
epsilon=1.0):
        """Initialise the essential variables needed for the simulation in dimensionless
units """
        #Rescaled Parameters
        self.sigma = sigma
        self.ma = ma
        self.epsilon = epsilon
        self.tau = np.sqrt(ma * sigma **2 / epsilon ) #This is the characteristic timescale

        #Rescaled parameters
        self.dt = dt / self.tau #Rescaled time
        self.positions = np.array(positions) / sigma #Rescaled positions
        self.masses = np.array(masses) /ma #Rescaled mass
        self.velocities = np.array(velocities) / (sigma / self.tau) #Rescaled Velocity

        #Number of particles
        self.num_particles = len(positions)

        #initiliase forces as a zero array with same shape as positions.
        self.forces = np.zeros_like(self.positions)
```

Figure 1: Initalising attributes for dimensionless units

Firstly, I created the class NBodySimulation_T1 with many instance attributes like self.sigma and self,ma using the parameters : positions, velocities, masses, dt , m_a, sigma, and epsilon. This is extremely for reusability since each simulation demands for different parameters and initialisation.m_a, epsilon, and sigma have been preemptively set to 1 for convenience - if the user wants to simulate a simple simulation.

Task 1 serves as a foundation for the rest of the code in this assessment. With that, there's a lot of calculations that goes with it. However, two methods make up for the bulk of the calculations, Due to time constraint, I will be explaining my choice of code for both of the methods, namely $update\_forces$ and $verlet\_leap\_frog\_integration\_step$.

2.1.1 "update_forces" method



```
        def update_forces(self):
            """Updates the net force for ALL particles due to pairwise Lennard-Jones Force using vectorised method
"""

            #Set forces back to zero again for all particles
            self.forces = np.zeros((self.num_particles,3))

            #Create displacement vector for all particles - the 3D array will have SIZE [n,n,3]
            #First n is the source particle, second n is the target particle, 3 is the dimensions of the particle -
x,y,z

            displacements = self.positions[np.newaxis,:,:] -  self.positions[:,np.newaxis,:]

            #Calculate euclidean distance between all particles - SIZE [n,n]
            #First n is source particle , second n is the target particle
            #Each element [i,j] reperesents the distance between particle i and particle j
            distances = np.linalg.norm(displacements , axis = 2)

            #Avoid division by zero. Set the 0 distances to infinity so that we divide by infinity first
            corrected_distances = np.where(distances > 0 , distances, np.inf)

            #Calculate Lennard=Jones force magnitude
            r_inv = 1/corrected_distances
            r_inv13 = r_inv**13
            r_inv7 = r_inv**(7)

            #force_magnitude has SIZE [n,n]
            force_magnitudes = 24 * (-2 * r_inv13 + r_inv7)

            #forces has size [n,n,3]
            #First n is the source particle, second n is the target particle, 3 is the dimensions of the particle  -
x,y,z
            forces = force_magnitudes[:,:,np.newaxis] * (displacements * r_inv[:,:,np.newaxis])

            #Sums up all force vector for each particle across second dimension
            #self.forces has shape [n,3]
            self.forces = np.sum(forces,axis=1)
```

Figure 2: Update force method which is inserted into the verlet integration step method.

In this method, we calculate and update the net forces for all particles. I have **vectorised** the code to save a lot of time. For example: If there is 3 particles; then, particle 1 has forces acting on it from the rest of the particles.

Below are further explanation of the method (Pictures included outside of the box because Overleaf won't allow me to):

1) Set all net forces to 0: Firstly, we need to set all forces acting on particles to 0 to establish consistency.

2) Calculate displacements: $self.positions$ in itself has size [n,3] where n is the number of particles and 3 is the dimension. I calculated the displacement 3D array which has a size of [n,n,3]. $self.positions[np.newaxis, :, :]$ reshapes it from [n,3] to [1,n,3]. Similarly, $self.positions[:, np.newaxis, :]$ has shape [n,1,3].

Due to broadcasting, the first array repeats its single slice n times, essentially copying the whole dataset again n times. **This transforms the array from [1,n,3] to [n,n,3].** Similarly, the second array has all of the rows in each slice repeat n times, essentially having one slice with n rows of the first particle's trajectory, one slice with n rows of the second particle's trajectory and so on. **This transforms the array from [n,1,3] to [n,n,3]**.

In the end, each element trajectories[i,j,:] contains the displacement vector from **particle i** to **particle j.**

3) Distance Calculation and Correction: Next, we obtain the relative distance for each particle. I put axis=2 to initiate the calculation for every row in each slice. The relative distance to a particle's self will be 0. Since that value is required for the denominator for some calculations, we temporarily set these values to ∞.

4) Lennard-Jones Magnitude and force: We, calculate the Lennard-Jones magnitude and forces based on the formulas given. The resulting force_magnitude has size [n,n]. With the idea of broadcasting, we expand all variables to be 3D in order to calculate "force". So, forces[i,j,3] represents the force exerted from particle j to particle i,

5) Summing Up Forces from all particles: For each column in every slice, we sum up all the numbers to obtain the net result of forces acting on a particular particle. So, $self.forces[i, 3]$ represents the 3 forces from all particles(except i) (x,y,z) acting on particle i.

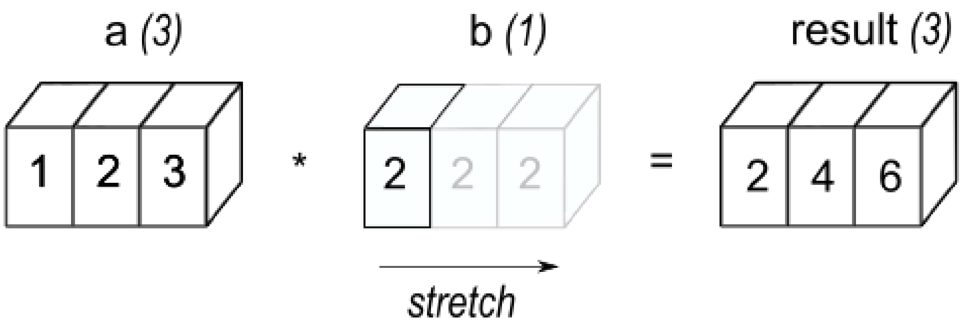


Figure 3: A simple explanation of what broadcasting looks like in order to explain Step 2. Taken from [1]

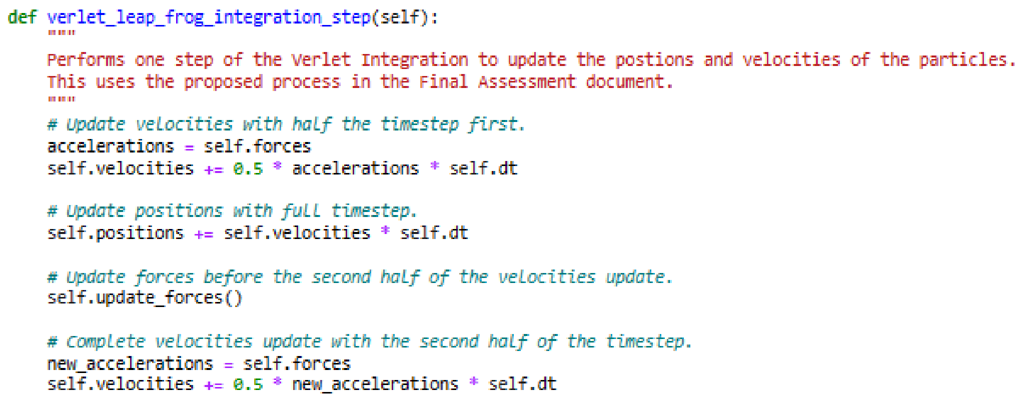
$$V_{\mathrm{LJ}}(r) = 4\left[\left(\frac{1}{r}\right)^{12} - \left(\frac{1}{r}\right)^{6}\right], \tag{6}$$

while Eq. (2) reduces to

$$\vec{F}_{pj} = 24\left[-2\left(\frac{1}{r}\right)^{13} + \left(\frac{1}{r}\right)^{7}\right]\vec{r}_{jp}, \tag{7}$$

Figure 4: Equation 6 is the Lennard-Jones Magnitude. Equation 7 is the Lennard-Jones Force. Taken from the assignment document.

2.1.2 "verlet_leap_frog_integration_step" method



```
def verlet_leap_frog_integration_step(self):
    """
    Performs one step of the Verlet Integration to update the postions and velocities of the particles.
    This uses the proposed process in the Final Assessment document.
    """
    # Update velocities with half the timestep first.
    accelerations = self.forces
    self.velocities += 0.5 * accelerations * self.dt

    # Update positions with full timestep.
    self.positions += self.velocities * self.dt

    # Update forces before the second half of the velocities update.
    self.update_forces()

    # Complete velocities update with the second half of the timestep.
    new_accelerations = self.forces
    self.velocities += 0.5 * new_accelerations * self.dt
```

Figure 5: One Verlet Integration step to update the velocities, positions, accelerations and forces.

At each half time-step, we update the velocities and positions accordingly. With the new conditions, I use the new positions and called the previous method to update forces (and hence acceleration since we are using dimensionless units) for that particle. At the remaining half time-step, I calculated the velocities with new accelerations.

2.1.3 Explaining the Plots

Listing 1: Oscilllation Behaviour

```
sim = NBodySimulation_T1(
    positions=[[0, 0, 0], [1.2, 0, 0]],# Place one particle at the origin and the other on the x-axis
    velocities=[[0.8, 0, 0], [-1.0, 0, 0]],# Start with both particles at rest or with some initial
        velocity
    masses=[1, 1],# Masses of the particles - standard mass
    dt=0.001 # Time step for the simulation
)
```

```
steps = 10000
sim.plot_relative_distance(steps)
```

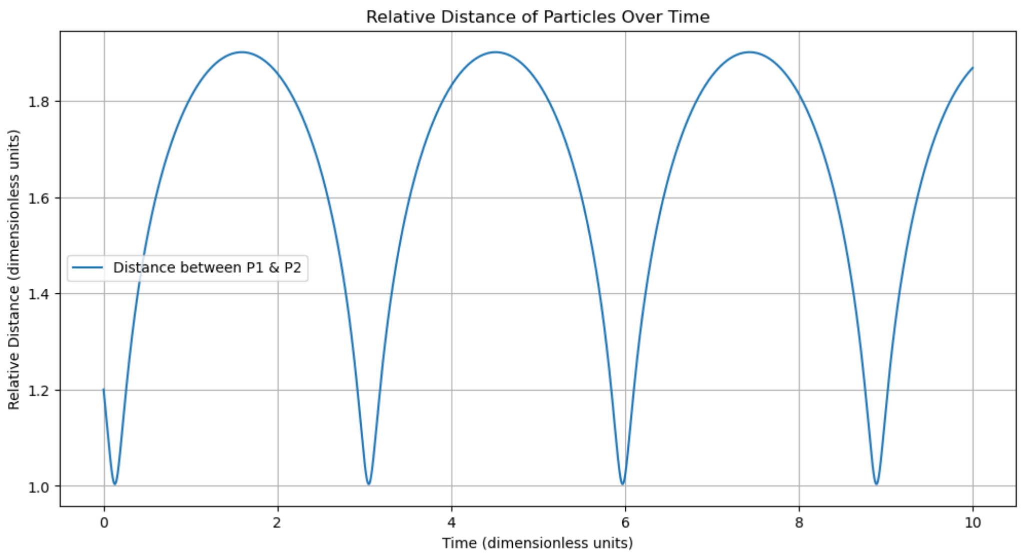


Figure 6: Oscillating behaviour between 2 particles

The code and diagram above demonstrates that it's possible to achieve an oscillating behaviour between 2 particles if the initial conditions are set correctly. As recommended in the document, I've assigned the particles to be seperated with an order of sigma (in this case, sigma = 1), or else the repulsive forces will be very large. I also have assigned the time-step to be very small to ensure that calculations will be stable enough to output a reliable graph.

The particles are also moving away from each other initially. However, the relative distance begins to decrease initially, proving that the Lennard-Jones potential is in works. Also, when particles are closer than the equilibrium distance , the repulsive part of the potential is very steep, meaning there's a very strong force pushing them apart. When particles are further away from equilibrium distance, the attractive part of the potential is shallower, meaning the force pulling them together is weaker.

The particles spend more time further apart. This can be verified from Figure 6 by examining the peaks and the troughs. Based on figure 6, the width of the peaks appear to be slightly wider than the troughs.

Listing 2: Non-Oscillating Behaviour

```
sim = NBodySimulation_T1(
    positions=[[0, 0, 0], [0.9, 0, 0]],# Place one particle at the origin and the other on the x-axis
    velocities=[[0.8, 0, 0], [-1.0, 0, 0]],# Start with both particles at rest or with some initial
        velocity
    masses=[1, 1],# Masses of the particles - standard mass
    dt=0.01# Time step for the simulation
)

steps= 1000
sim.plot_relative_distance(steps)
```

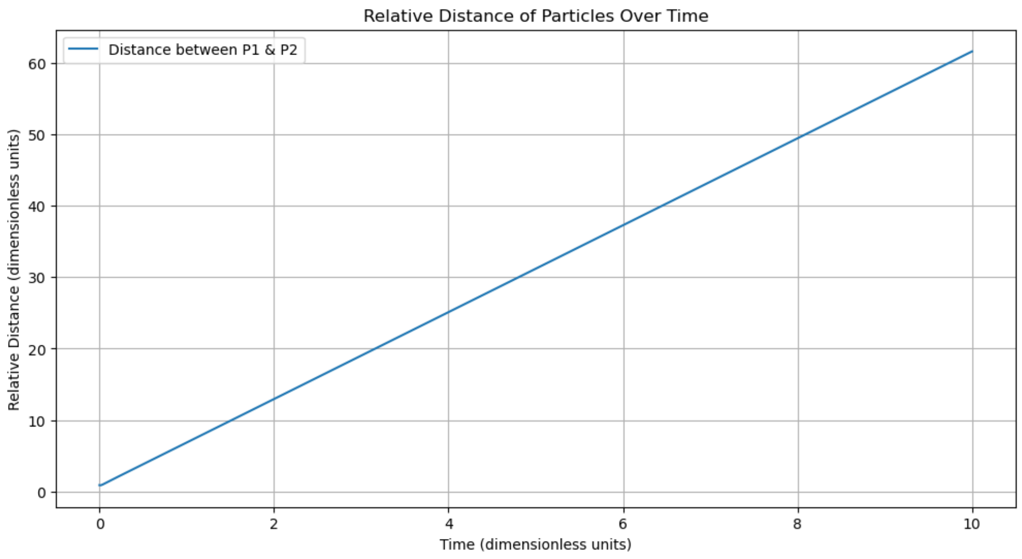


Figure 7: Relative Distances between the particles increases indefinitely.

For this simulation, I have just changed the position of one particle so that the initial relative distance starts off to be short.(from [1.2,0,0] to [0.9,0,0]) This is because the initial repelling force is so strong that both particles drift off indefinitely. Hence, we should be **MINDFUL** of the initial values such as positions, and velocities.

2.2 2 Argon Particles in a Gas System

Explanation has been given in the Jupyter Notebook. We can also obtain the characteristic timescale and characteristic speed for other type of gas particles, given we know values like ϵ, σ and m_a.

2.3 General N-Particles in a Gas System

So far , we have examined how the simulation will behave under 2 particles. What would happen if there was N number of particles? Here, we will examine what it would be like for more particles in the system.

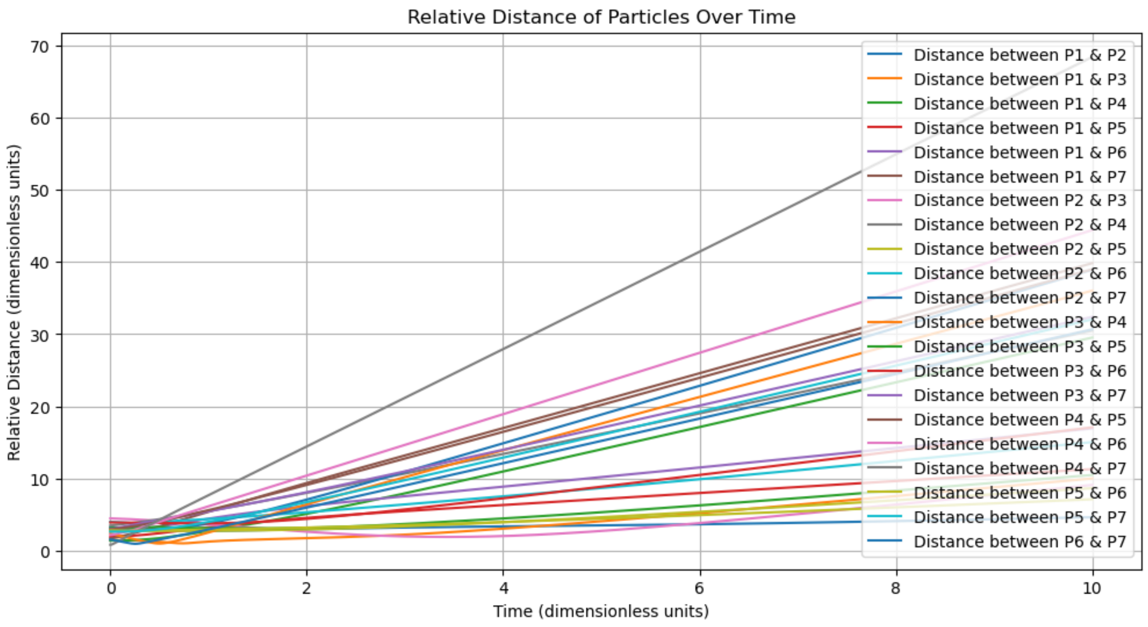


Figure 8: The trajectories of 7 particles overtime.

More details of code and relative distance between particles can be observed in the Jupyter Notebook. In general, it is extremely hard to obtain an oscillating behaviour when we assign the initial positions of the particles randomly. Given the right conditions (positions and velocities) , we can consistently obtain oscillations as observed in Figure 9.

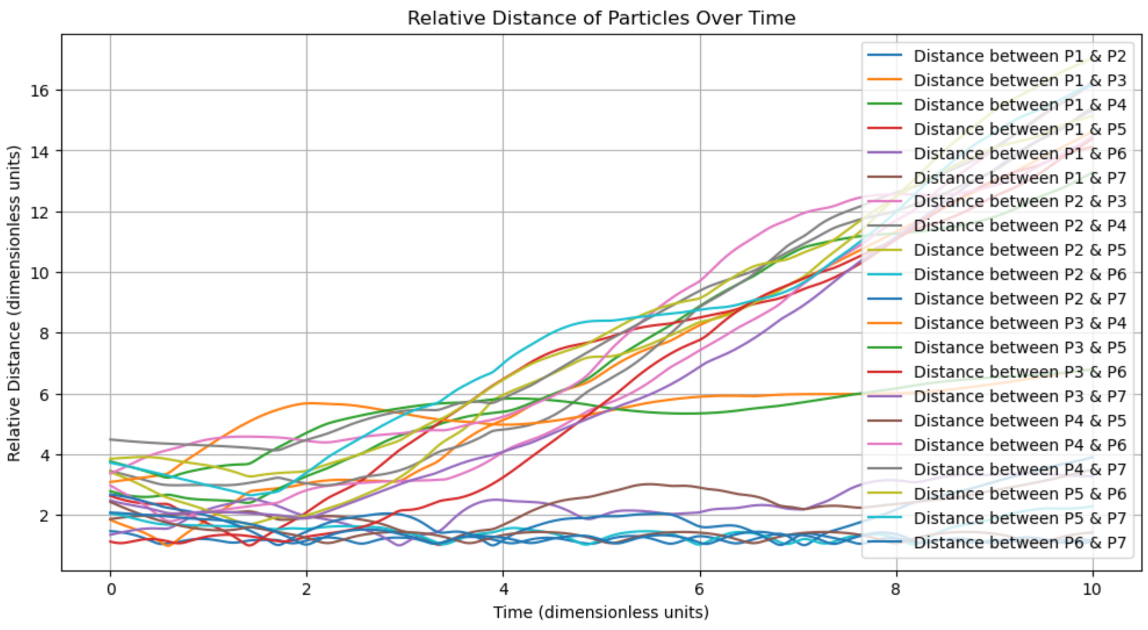


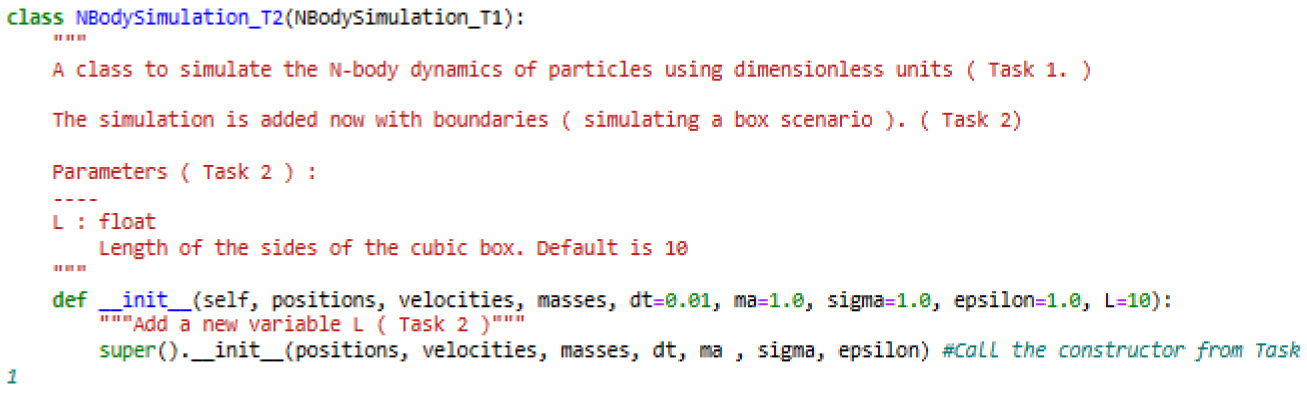
Figure 9: Oscillating trajectories of 7 particles overtime.

Here, in this case, I've set a random seed of 30 and have slightly changed the values for positions, velocities, and masses but are still within the range of values. In conclusion, to properly achieve the intended Lennard-Jones interaction, we have to be **CAREFUL** how the particles are initialised.

3 Particles in a box - Task 2

3.1 Additional Code

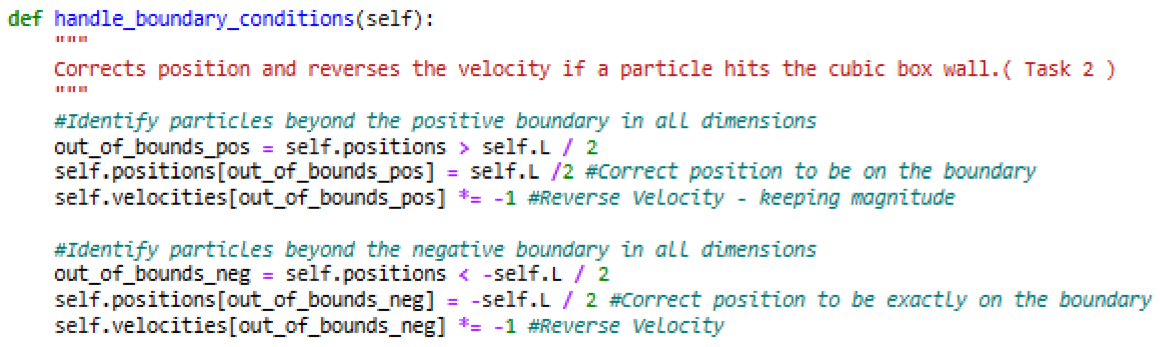
Since Task 2 demands that we introduce perfectly elastic boundaries to our simulation, I figured that creating a new child class from the parent class of Task 1 would be a great idea because users will be able to understand the progression and changes for different types of simulations (boundaries or no boundaries).



```
class NBodySimulation_T2(NBodySimulation_T1):
    """
    A class to simulate the N-body dynamics of particles using dimensionless units ( Task 1. )

    The simulation is added now with boundaries ( simulating a box scenario ). ( Task 2)

    Parameters ( Task 2 ) :
    ----
    L : float
        Length of the sides of the cubic box. Default is 10
    """
    def __init__(self, positions, velocities, masses, dt=0.01, ma=1.0, sigma=1.0, epsilon=1.0, L=10):
        """Add a new variable L ( Task 2 )"""
        super().__init__(positions, velocities, masses, dt, ma , sigma, epsilon) #Call the constructor from Task
1
```

Figure 10: Inheritance Code

This way, the class "NBodySimulation_T2" inherits all of the methods and attributes from Task 1's class "NBodySimulation_T1". This removes clutter and helps direct our focus on integrating boundary conditions.

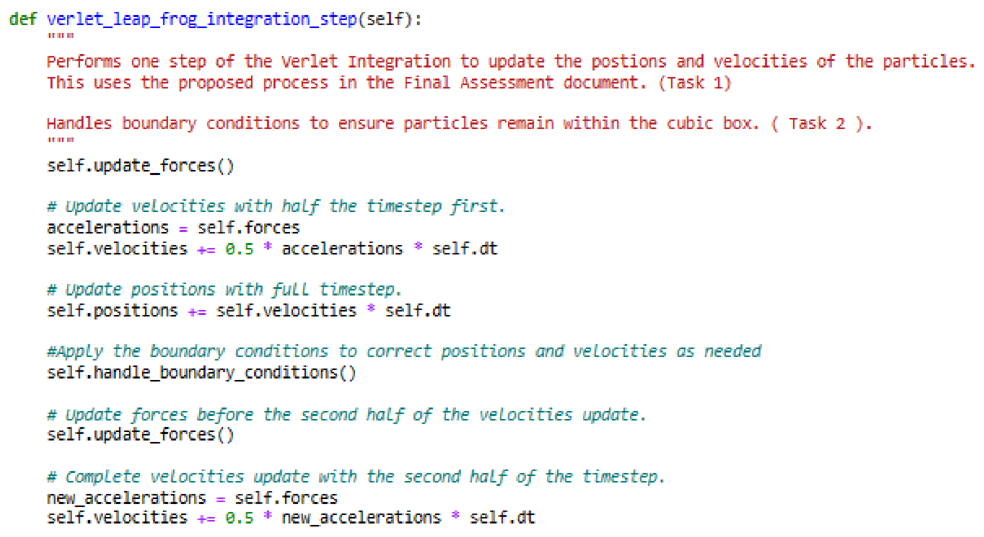


```python
def handle_boundary_conditions(self):
    """
    Corrects position and reverses the velocity if a particle hits the cubic box wall.( Task 2 )
    """
    #Identify particles beyond the positive boundary in all dimensions
    out_of_bounds_pos = self.positions > self.L / 2
    self.positions[out_of_bounds_pos] = self.L /2 #Correct position to be on the boundary
    self.velocities[out_of_bounds_pos] *= -1 #Reverse Velocity - keeping magnitude

    #Identify particles beyond the negative boundary in all dimensions
    out_of_bounds_neg = self.positions < -self.L / 2
    self.positions[out_of_bounds_neg] = -self.L / 2 #Correct position to be exactly on the boundary
    self.velocities[out_of_bounds_neg] *= -1 #Reverse Velocity
```

Figure 11: Setting the boundary conditions

We establish that the cube has side length of L, and the centre to be at 0. So, the boundaries are situated at -L/2 and L/2. The intuitive way is to identify the positions of the particles which exceed the boundary. If it is, we adjust the particles which are mispositioned to be exactly on the boundary and velocity to be reversed only.

This is **NOT** 100 percent **ACCURATE** because the time wasted within a time-step could have been utilised to travel further within the cube. Due to time constraint, I decided not to venture through this. In order to avoid this issue, we can make **dt**, but we might be at risk of lack of computational power (something which we will explore later on).



```python
def verlet_leap_frog_integration_step(self):
    """
    Performs one step of the Verlet Integration to update the postions and velocities of the particles.
    This uses the proposed process in the Final Assessment document. (Task 1)

    Handles boundary conditions to ensure particles remain within the cubic box. ( Task 2 ).
    """
    self.update_forces()

    # Update velocities with half the timestep first.
    accelerations = self.forces
    self.velocities += 0.5 * accelerations * self.dt

    # Update positions with full timestep.
    self.positions += self.velocities * self.dt

    #Apply the boundary conditions to correct positions and velocities as needed
    self.handle_boundary_conditions()

    # Update forces before the second half of the velocities update.
    self.update_forces()

    # Complete velocities update with the second half of the timestep.
    new_accelerations = self.forces
    self.velocities += 0.5 * new_accelerations * self.dt
```

Figure 12: Updated Verlet Leap Integration Method

Although this method is present in our parent class (Task 1), we still have to call and update this method in order to insert the "handle_boundary_condition" method which corrects particles that have crossed the boundary. Otherwise, particles will drift away past the boundary.

3.2 Trajectories Over-Time

Listing 3: Trajectory within a cubic box with boundary

```python
positions = [[3,1,2]]
velocities = [[-0.1,0.2,0.1]] #Moving towards the left wall while going slightly upwards
masses = [1.0,1.0]
num_steps = 10000

#Assume simple parameters for this model
sim_bound = NBodySimulation_T2(positions,velocities,masses)
sim_bound.sim_particle_bouncing(num_steps)
```

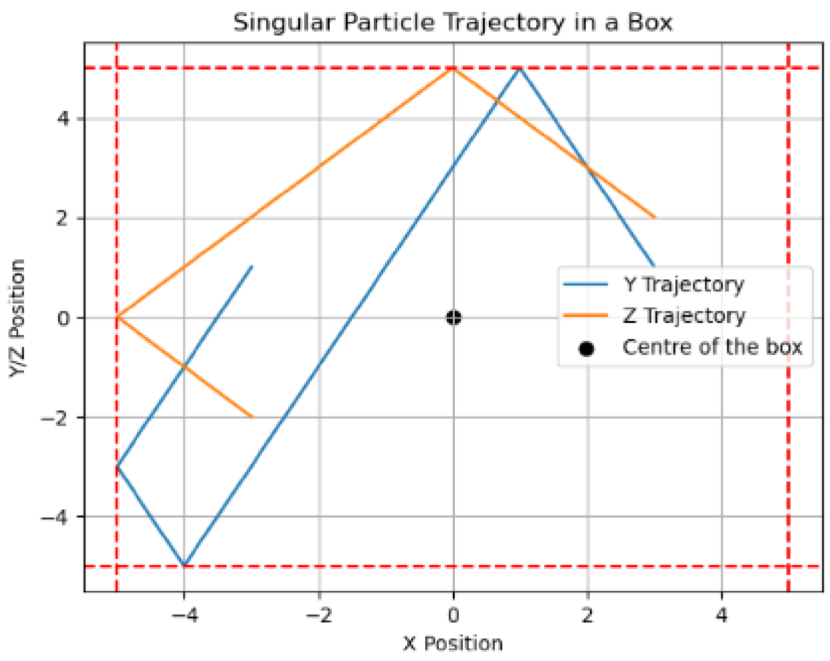


Figure 13: Various position of particles rebounding off the wall

As examined from Figure 13, it is noted that the particle rebounds off the walls, but how do we actually know that it behaves as expected?

There are 2 attributes which we can confirm :

1) Perfect Elasticity: Velocity Component of gas particle is reversed, but scalar remains unchanged. **2) Kinetic Energy:** Remains unchanged since there's only one particle in the system.

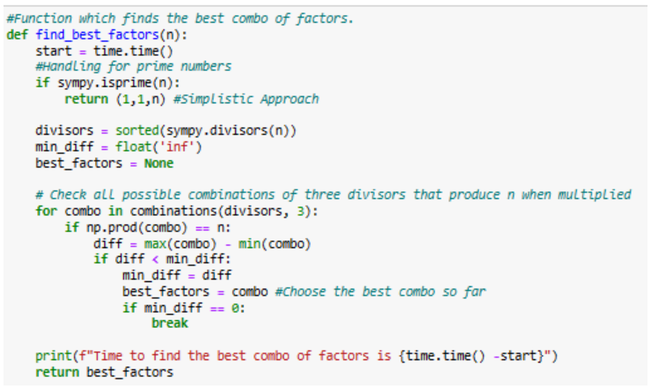
Since there is sufficient explanation , graphs , and proof present in this section of the Jupyter Notebook, I shall leave that to be.

4 Investigate! - Task 3

4.1 Initialising Initial Positions:

In this section, we are called to investigate the effects of pressure with changes in volume of the box and temperature of the gas. Again, we create another class which inherits all the attributes and methods from the parent class which is from Task 2. Please refer the "init" function from "Task_3.py" for further reference.

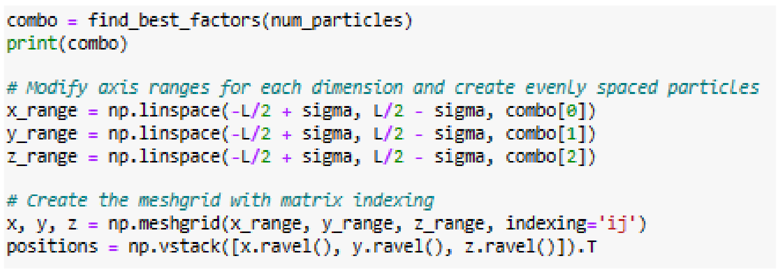
Recalling the hint of seperating the particles at least an order of σ away, we have to create evenly spaced particles which involves finding the best factors and implementing a meshgrid from numpy.



```python
#Function which finds the best combo of factors.
def find_best_factors(n):
    start = time.time()
    #Handling for prime numbers
    if sympy.isprime(n):
        return (1,1,n) #Simplistic Approach

    divisors = sorted(sympy.divisors(n))
    min_diff = float('inf')
    best_factors = None

    # Check all possible combinations of three divisors that produce n when multiplied
    for combo in combinations(divisors, 3):
        if np.prod(combo) == n:
            diff = max(combo) - min(combo)
            if diff < min_diff:
                min_diff = diff
                best_factors = combo #Choose the best combo so far
                if min_diff == 0:
                    break

    print(f"Time to find the best combo of factors is {time.time() -start}")
    return best_factors
```

Figure 14: Finding Best Factor based on (n) number of particles.

We aim to obtain 3 factor numbers only where the difference between the biggest and smallest number is minimised as much as possible. The reason behind it will soon make sense as soon as I explain the following code.



```python
combo = find_best_factors(num_particles)
print(combo)

# Modify axis ranges for each dimension and create evenly spaced particles
x_range = np.linspace(-L/2 + sigma, L/2 - sigma, combo[0])
y_range = np.linspace(-L/2 + sigma, L/2 - sigma, combo[1])
z_range = np.linspace(-L/2 + sigma, L/2 - sigma, combo[2])

# Create the meshgrid with matrix indexing
x, y, z = np.meshgrid(x_range, y_range, z_range, indexing='ij')
positions = np.vstack([x.ravel(), y.ravel(), z.ravel()]).T
```

Figure 15: Distribution of particles using factors from combo

This setup is crucial to ensure that our system starts in a state of equilibrium or near equilibrium which is typical in natural systems.

1) Combo: The function described earlier calculates the optimal subdivision of the space along each axis to acommodate (n) - number of particles.

2) Evenly Spaced Points: For each axis, we generate a sequence of evenly spaced particles using "np.linspace". The parameters "-L/2 + sigma" and "L/2 - sigma" define the boundaries of the space, which prevents initialisaiton too close to the wall.

3) Meshgrid: We then construct a 3D grid with matrix indexing, where we create a structured set of points where each point corresponds to a position of a particle in the simulation.

4) Flatten and Transpose: Finally, we **flatten** each of the 3D coordinate array using .ravel() and **vertically stack** using .vstack(). After transposing the array, it transforms it to shape (num_particles,3), where each row corresponds to the 3D coordinate of a particle.

4.2 Proving Isotropy:

$$P_x = \frac{1}{A\Delta t} \sum_{i\,\text{crossing}} m_i v_{x,i},$$

Figure 16: Formula for pressure in a specific direction.

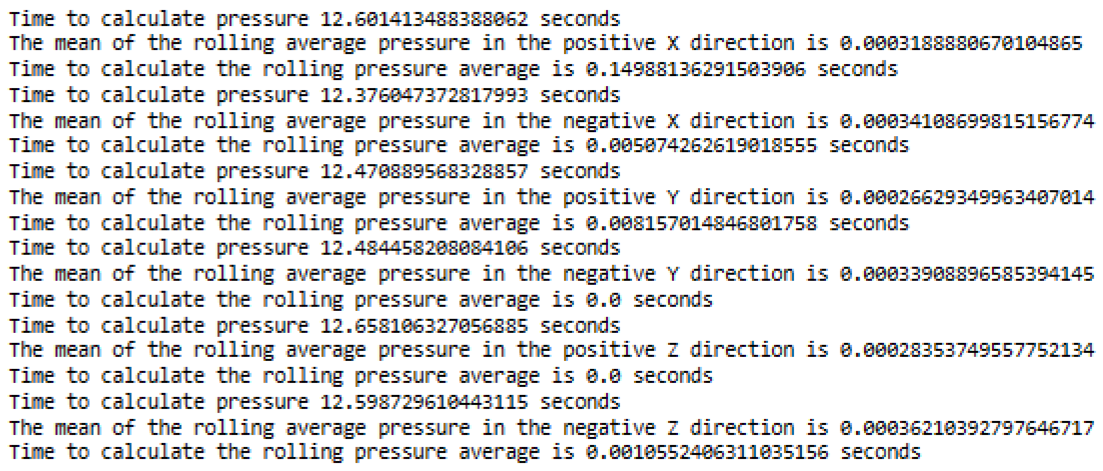
Assuming pressure is isotropic,in theory, $P_x = P_y = P_z = \frac{1}{3}P$ (where P is the total pressure). I also have to consider pressure moving in both positive and negative directions. Hence, I have to verify this to ensure that my simulation is correct. Since there are 3 axes as well as the positive and negative directions to consider, we have to calculate the pressure 6 times. This makes sense because it's a cubic simulation.

Below is the code which calculates directional pressure and the **rolling average**, followed by the output of the code. Also known as a moving average, it is **defined** as a series of data points created using a series of average of different intervals using the entire dataset.

Listing 4: Proving Isotropy

```python
# Calculate pressures for a sufficient number of steps to ensure equilibrium
num_steps = 10000
window_size = 200

pressures_positive_x , rolling_average_x_positive_pressure =
    sim.calculate_directional_pressure(num_steps, 'X', 'positive', window_size, output = 'Yes')
pressures_negative_x , rolling_average_x_negative_pressure =
    sim.calculate_directional_pressure(num_steps, 'X', 'negative', window_size, output = 'Yes')
pressures_positive_y , rolling_average_y_positive_pressure =
    sim.calculate_directional_pressure(num_steps, 'Y', 'positive', window_size, output = 'Yes')
```

```
pressures_negative_y , rolling_average_y_negative_pressure =
    sim.calculate_directional_pressure(num_steps, 'Y', 'negative', window_size, output = 'Yes')
pressures_positive_z , rolling_average_z_positive_pressure =
    sim.calculate_directional_pressure(num_steps, 'Z', 'positive', window_size, output = 'Yes')
pressures_negative_z , rolling_average_z_negative_pressure =
    sim.calculate_directional_pressure(num_steps, 'Z', 'negative', window_size, output = 'Yes')
```



```
Time to calculate pressure 12.601413488388062 seconds
The mean of the rolling average pressure in the positive X direction is 0.0003188880670104865
Time to calculate the rolling pressure average is 0.14988136291503906 seconds
Time to calculate pressure 12.376047372817993 seconds
The mean of the rolling average pressure in the negative X direction is 0.00034108699815156774
Time to calculate the rolling pressure average is 0.005074262619018555 seconds
Time to calculate pressure 12.470889568328857 seconds
The mean of the rolling average pressure in the positive Y direction is 0.00026629349963407014
Time to calculate the rolling pressure average is 0.008157014846801758 seconds
Time to calculate pressure 12.484458208084106 seconds
The mean of the rolling average pressure in the negative Y direction is 0.00033908896585394145
Time to calculate the rolling pressure average is 0.0 seconds
Time to calculate pressure 12.658106327056885 seconds
The mean of the rolling average pressure in the positive Z direction is 0.00028353749557752134
Time to calculate the rolling pressure average is 0.0 seconds
Time to calculate pressure 12.598729610443115 seconds
The mean of the rolling average pressure in the negative Z direction is 0.00036210392797646717
Time to calculate the rolling pressure average is 0.0010552406311035156 seconds
```

Figure 17: Rolling average pressure in all 6 directions.

The mean of the rolling average are relatively close, but not identical. This is expected since pressure has a stochastic behaviour. As a metric for determining isotropy, I decided to round the values to the closest 1 s.f. (significant figure). In the end, I obtained 5 0.0003 values and 1 0.0004 value. This time, we are unable to obtain a side which is unanimous to the rest ; sometimes, this can be for 2 or 3 sides. In order to increase accuracy, we can increase the number of steps but we must be aware of computational time.

4.3 Plotting Stochastic Behaviour:

Since the initial velocities and positions of the particles are assigned randomly, this introduces an element of stochasticity to the system.The exact trajectories of the particles are not deterministic. Small changes in initial conditions can lead to **significantly different** outcomes due to the chaotic nature. Since this exists in our simulation and in practice, the stochastic nature is inevitable. Also, the use of a *finite* time-step to integrate the equations of motion means our class can only approximate the behaviour between time steps.

```
sim.plot_pressure_direction(pressure , num_steps, direction, plane_side)
```

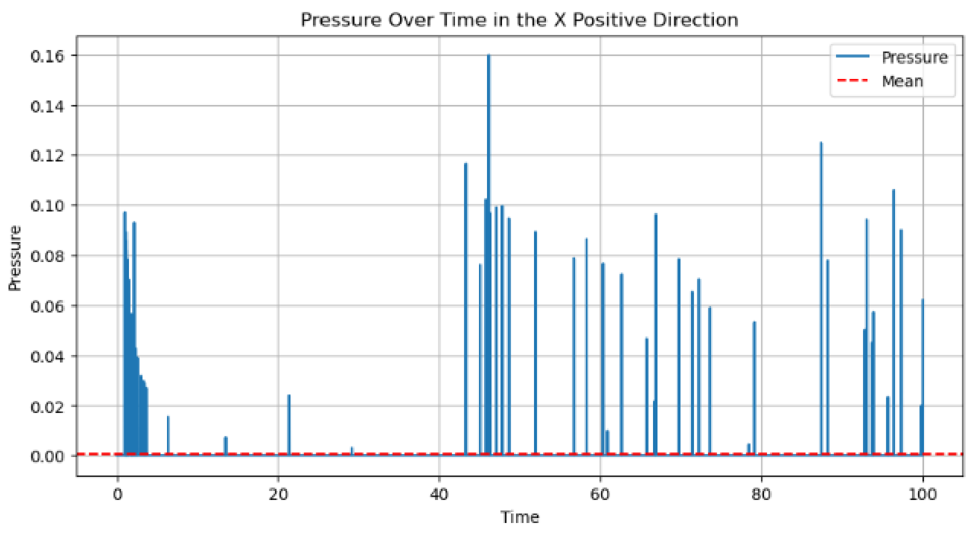


Figure 18: Pressure over time in X Positive Direction

Establishing that pressure is isotropic, we can plot any of the pressure in any of the 6 sides and we still get a similar but not exact result since it's stochastic. In this case, I have used the pressure in the X-positive direction as my dependent variable. Similarly, this can also be done for temperature.

```
#Compute one for kinetic energy
sim.plot_kinetic_energy(kinetic_energy, num_steps)
```

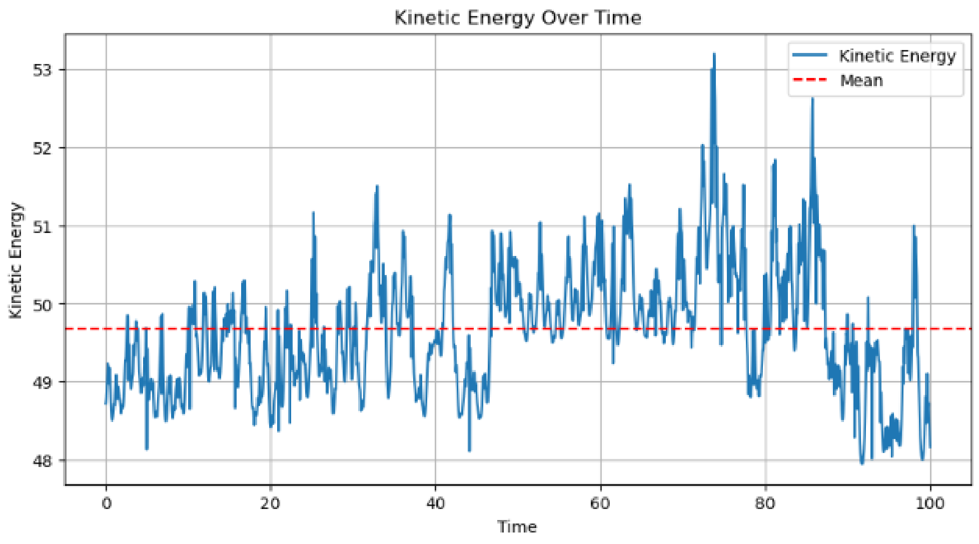


Figure 19: Kinetic energy over time

Side of the cubic simulation is not given as a parameter in this method unlike "plot_pressure_direction". The dashed line representing average kinetic energy looks relatively stable compared to the actual kinetic energy line. This shows that while individual measurements are highly variable, the overall system's kinetic energy tends to hover around an average value in the long term, indicating state of equilibrium.

Both diagrams exhibit significant variability in kinetic energy/pressure over time. Instead of a smooth curve, they exhibit a jagged, irregular pattern, indicating that it changes unpredictably within a certain range. The randomness could be attributed to energy exchange between particles, where collisions can transfer kinetic energy between them, leading to fluctuations in individual particles.

To prove isotropy, we used the rolling average as a metric for comparison. They are useful mainly for the following reasons:

1) Noise Reduction: Help smooth out short-term fluctuations and highlight longer-term trends.

2) Visibility of Trends: By smoothing out variations, rolling averages make it easier to identify trends that may not be so clear in raw data.

3) Data Stabilization: Can help stabilise a time series by averaging out irregular components. Useful if there is a unexpected peak or drop , or has seasonal variance.

Despite all the advantages, rolling averages can introduce a lag between actual data and smoothed data, which might not be suitable for super time-precision analysis. Moreover, then can smooth over important short-term variations which might be important for some analysis. In this project, however, we are more interested in identifying underlying patterns. Hence, **rolling averages will be used** to plot out any **remaining graphics** as they clearly show longer-term trends or cycles.

4.4 Plotting smoothed graphs:

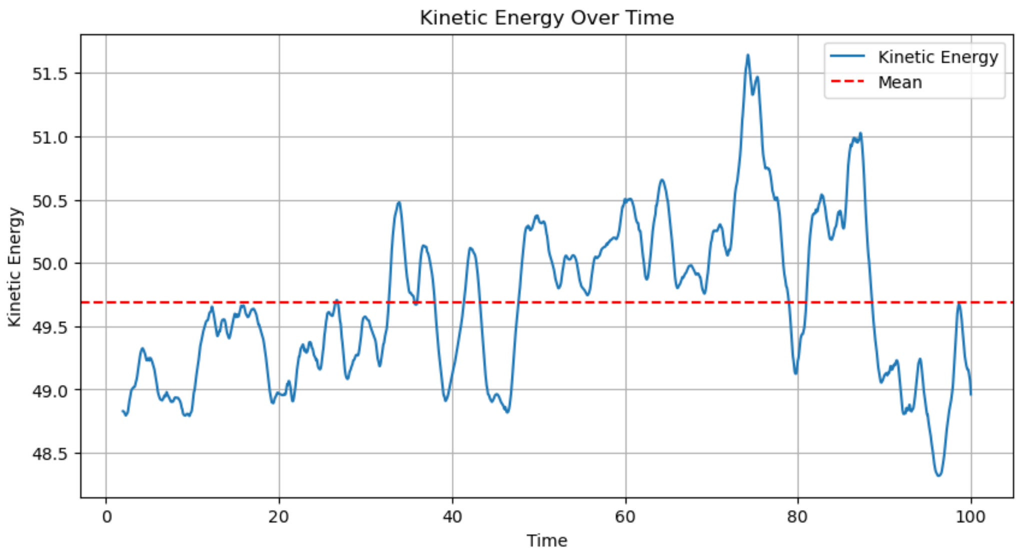
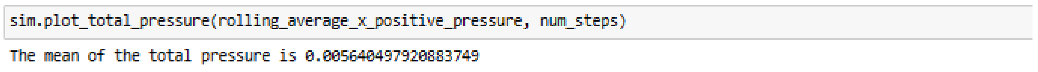


Figure 20: Smoothed Kinetic Energy Over time

Trends start to look smoother and much clearer. Image of pressure against time not included as it is in Jupyter Notebook. Also, I plotted out the total pressure against time graph since the pressure in our simulation is isotropic. Essentially, the total pressure is 3 times as big as any pressure in any one of the 6 directions.



```
sim.plot_total_pressure(rolling_average_x_positive_pressure, num_steps)
```

```
The mean of the total pressure is 0.005640497920883749
```

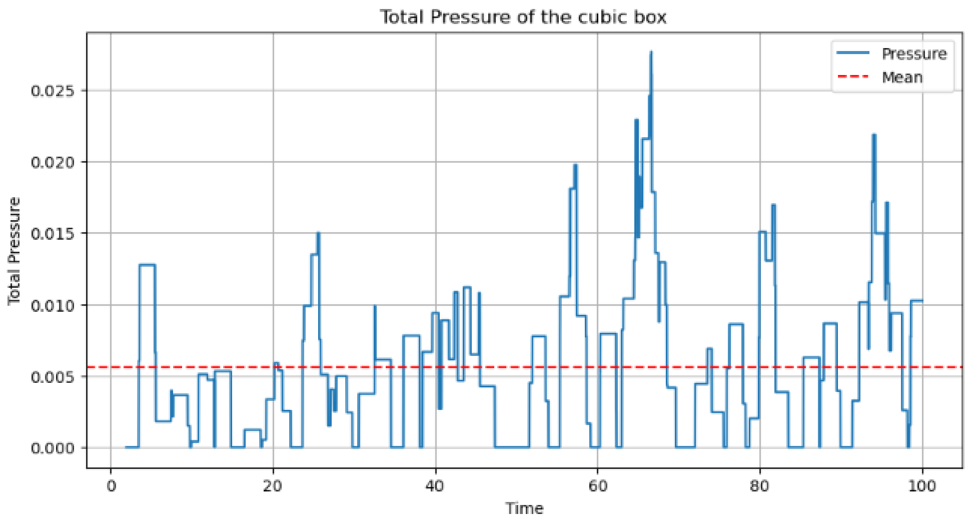


Figure 21: Total pressure against time.

4.5 Relationship of pressure, volume and temperature:

4.5.1 Simulations with simple parameters:

In this section, I decided to initiate my simulation with simple parameters to examine and test any underlying relationship. With numerous experimenting, the "dt" interval of (0.01 , 0.1) and "num_particle" interval of (50,200) gives the greatest balance of good computational speed and clear trends. Due to time constraint, I was not able to create a plot for this.

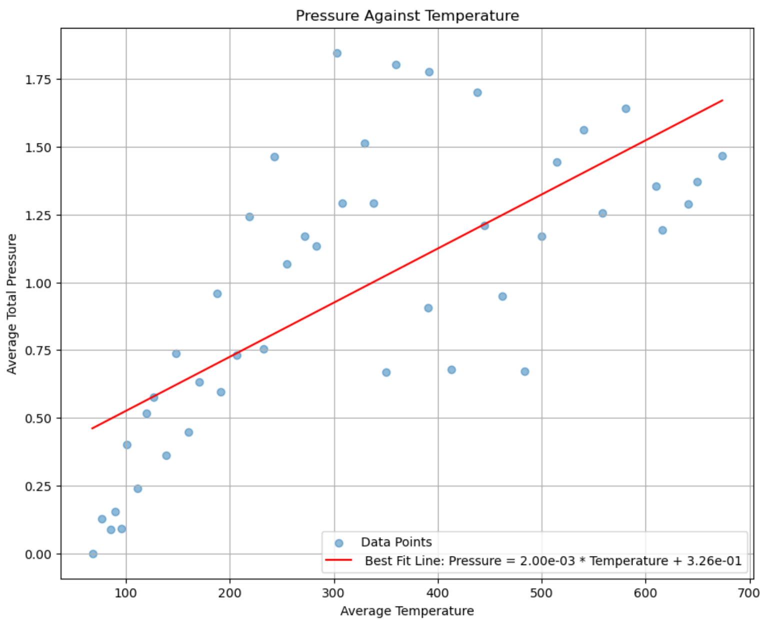


Figure 22: Total pressure against temperature

This is a positive linear relationship between pressure and the average rolling temperature. As expected, as the average temperature rises , the rate at which the particles hit the surface the cube increase ; hence, total pressure increases overall.

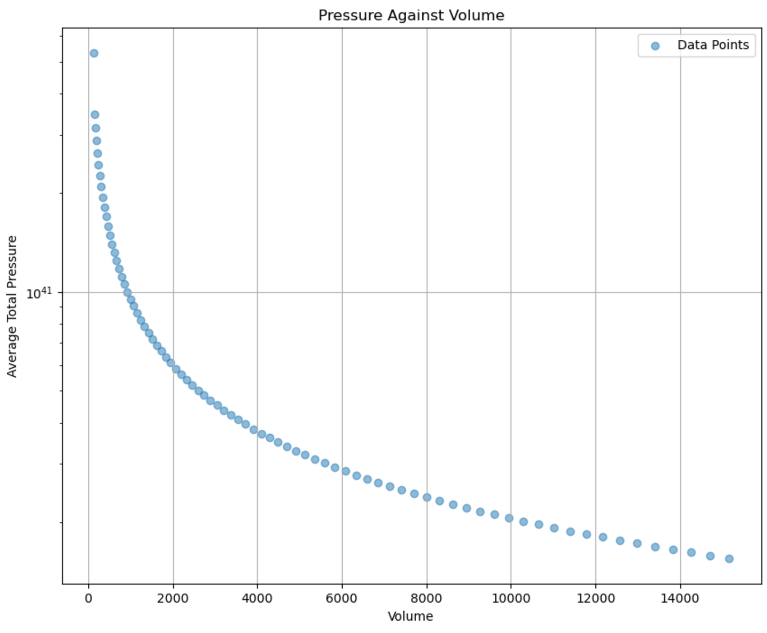


Figure 23: Total pressure against volume

As expected, the graph is a logarithm function. When volume approaches 0, total pressure approaches ∞ . **More** graphs about the **argon particles** can be referred in the Jupyter Notebook.

NOTE: All plots utilised the **mean of the rolling average** as the dependent variable.

References

[1] Broadcasting &x2014; NumPy v1.26 Manual — numpy.org. https://numpy.org/doc/stable/user/basics.broadcasting.html. [Accessed 22-04-2024].