PHYS2020 Computational Project

Samuel Allpass s4803050
School of Mathematics and Physics, University of Queensland, Brisbane, QLD 4072, Australia

Computer particle simulation models have paved the way for both theoretical predictions and visualisation of complex systems. This investigation aimed at quantifying the accuracy to which ideal gas state value relationships were adhered to by python scripts for both an ideal, point-like, non-interacting and an interacting particle model. The investigation found that within the bounds of uncertainty found over multiple trial simulations, the code produced pressure-volume relationships identical to that proposed by the ideal gas law PV=nRT. Building on this model, a Lennard-Jones particle-pair interaction potential was introduced, with the particle of choice being helium. The interacting gas model produced a distinct constant relationship for the thermal velocity of the system over time, as result expected due to the isolated nature of the system. Implementing a cooling factor allowed the analysis of two real world characteristics, firstly that both the temperature and pressure reduced over time, until finally the temperature was low enough to induce condensation, with particles settling at the ground state distance given by the Lennard-Jones potential.

I. INTRODUCTION

Over the past century, technological advancements in computing have provided a gateway to data analysis and observation of processes otherwise to small to see. Specifically, computer simulation has allowed for deeper understanding of how microscopic systems, obeying simple ideal laws of physics, scale up to macroscopic phenomena consistent with that of the observable laws. Simulations have provided accurate visualisation of particle physics, far more accurate than Heisenberg, Einstein and even Fynman could have ever imagined. This phenomena is increasingly apparent for gases in confined spaces, such that simulation of gas particles in a box display clear correspondance with the ideal gas law. Broad statement about the topic area, easing the reader into your.

II. THEORY

Built on the foundations of Blaise Pascal, Robert Boyle, Jacques Charles and Amadeo Avogadro, Emil Claperyon is credited with the construction of the ideal gas equation (1) in 1843 [1]. The ideal gas law quantifies the relationship between the pressure (P), temperature (T) and volume (V) of a gas.

$$PV = nRT \quad (1)$$

Where n is the number of moles of gas in the system, and R is the ideal gas constant. Importantly, the model defines ideal gases to be point-like, non-interacting particles, a criteria which all gases tend to fit at sufficiently low pressures [2].

A. Ideal Gas simulation

For the purpose of this investigation, it is relevant to derive expressions for the terms of the ideal gas law, achieved by utilising physical properties of individual particles within the system. Modeling a single particle of constant velocity traveling throughout a square box of length L, displaying perfectly elastic collisions would dictate that the particle leaves with the negative velocity to before, and thus experiences a total momentum change:

$$\Delta P = -2mv^2 \quad (2)$$

Due to conservation of momentum, this implies that the wall experienced a change in momentum of $-\Delta P$ during one collision. Throughout a given simulation time, tracking the number of collisions and subsequent total change in momentum for the wall, it was understood that the total impulse, and furthermore force of the wall could be constructed as follows:

$$J = \sum \Delta P \quad \rightarrow \quad \langle F \rangle = \frac{J}{\Delta T} \quad (3)$$

Finally, we observed that for a system within which the total impulse on one wall is recorded, the pressure of the system is given by:

$$P = \frac{\langle F \rangle}{L} \quad (4)$$

We notice that given for the two dimensional case, volume of the system is the area, and so follows the area the length, thus, dimensional analysis dictates the simulated pressure of the system retains the same units as in three dimensions.

Additionally, in order to extend the simulation to more realistic systems with a larger number of particles, the initial velocities of each particle were to be accurately introduced. Introducing the concept of the equipartition theorem for thermal equilibrium, it was realised that each degree of freedom within the box, for which there are two, contribute equally to the total energy of the system [3], such that: For i = x, y

$$\frac{1}{2}mv_i^2 = \frac{1}{2}k_BT \quad (5) \quad \rightarrow \quad \langle v_i^2\rangle = \frac{k_bT}{m}$$

Therefore, the initial velocities of each particle within the system could be appropriately set at a value of a standard Gaussian distribution of standard deviation $\sigma^2 = \frac{k_BT}{m}$.

B. Interacting Gas simulation

Despite the macroscopic relevance of an ideal gas simulation, for small volumes-high pressure systems, it is unreasonable to model the particles as non-interacting. The Lennard–Jones potential provides a simply pair potential, modeling the weak van der Waals bonds between each of the particles, a potential quantified for a gas particle of length ϵ and energy σ:

$$V(r) = 4\epsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^6] \quad (6)$$

And thus a each particle pair will experience an equal but opposite force corresponding to:

$$F = -\nabla V = -(\frac{dV}{dx}, \frac{dV}{dy})$$

Observation of the minimum potential possible:

$$\frac{dV}{dr} = 0 = -12\frac{\sigma^{12}}{r^{13}} + 6\frac{\sigma^6}{r^7} \quad \rightarrow \quad r_{min} = 2^{1/6}\sigma$$

Physically, it is at this r_{min} separation between two particles where the potential is 0, such that the attractive and repulsive forces balance. Descriptively, this separation refers to the most stable distance between two particles in a condensed (liquid/solid) state, and thus, for temperatures at which condensation occurs within the system, it is hypothesised that particles close enough should settle r_{min} apart.

Attempting to understand how the an interacting gas holds against the ideal gas model, it is relevant to determine a method of calculating the temperature of the system at a given instance. Extrapolation of T in equation 5 returns:

$$T = \frac{\frac{1}{2}mv_i^2}{\frac{1}{2}k_B} = \frac{m\langle v_x^2 + v_y^2\rangle}{k_B} \quad (7)$$

It was hypothesised that despite the interactions, the forces would be small enough to maintain a relatively constant temperature. In order to further test this hypothesis, it was concluded that a cooling factor would be introduced in order to model its effect on the temperature over time, as well as noting the pressure to draw conclusions on how it relates to the ideal gas model.

III. METHOD

A. Uncertainties

Given the idealised situation of the simulation, inherent uncertainties of the "experiment" were neglected. However, variance between trials given differing initial positions and velocities for the particles were observed to induce differing system state values. As such it was decided that 4 trials would be conducted for each type of simulation, the average for any measurement (F) being taken, along with a respective uncertainty given by the standard deviation as follows.

$$\delta F = \sqrt{\frac{\sum_{i=1}^{N}(F_i - F_{mean})^2}{N}} \quad (8) \quad \rightarrow \quad N = 4(trials)$$

B. Apparatus

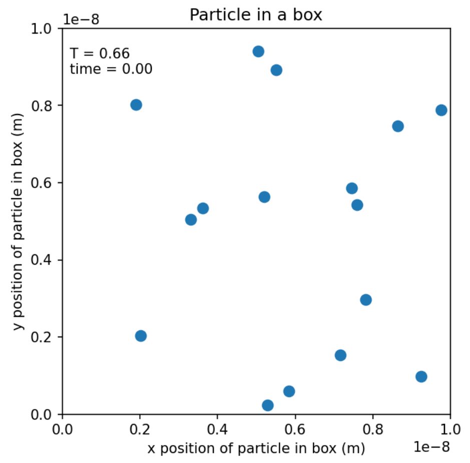
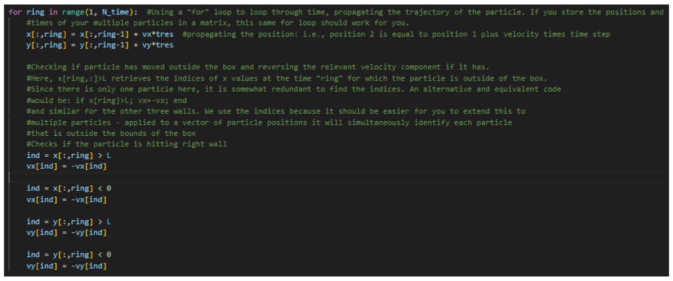


FIG. 1. Simulation apparatus of length $1 * 10^{-8}m$ for 16 particles, with x and y axis representing box sides.

The simulation apparatus consists of a x and y axis representing the sides of the box, with the graph ends being the following wall. Each blue dot within the box represents a particle interacting within the box walls

C. Procedure

In order to collect relevant and interpretable data on how accurately the simulation models both ideal and interacting gas processes, the following procedure was implemented. It was first dictated that an ideal gas python script would be constructed in order to observe how accurately the model obeys the ideal gas model (appendix 1). The software achieved this by considering time "steps" through the use of a for loop provided in figure 2.



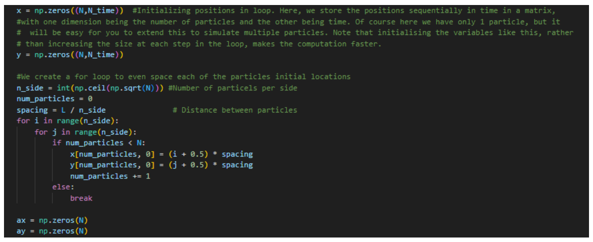
```
for ring in range(1, N_time):  #Using a "for" loop to loop through time, propagating the trajectory of the particle. If you store the positions and
    #times of your multiple particles in a matrix, this same for loop should work for you.
    x[:,ring] = x[:,ring-1] + vx*tres  #propagating the position: i.e., position 2 is equal to position 1 plus velocity times time step
    y[:,ring] = y[:,ring-1] + vy*tres

    #Checking if particle has moved outside the box and reversing the relevant velocity component if it has.
    #Here, x[ring,:]>L retrieves the indices of x values at the time "ring" for which the particle is outside of the box.
    #Since there is only one particle here, it is somewhat redundant to find the indices. An alternative and equivalent code
    #would be: if x[ring]>L; vx=-vx; end
    #and similar for the other three walls. We use the indices because it should be easier for you to extend this to
    #multiple particles - applied to a vector of particle positions it will simultaneously identify each particle
    #that is outside the bounds of the box
    #Checks if the particle is hitting right wall
    ind = x[:,ring] > L
    vx[ind] = -vx[ind]

    ind = x[:,ring] < 0
    vx[ind] = -vx[ind]

    ind = y[:,ring] > L
    vy[ind] = -vy[ind]

    ind = y[:,ring] < 0
    vy[ind] = -vy[ind]
```

FIG. 2. Python time step for loop used to simulate time.

As observed, with each time step, the particles position was adjusted by the constant velocity list, and appended appropriately. Following, the position of each particle was investigated to ensure it had not traveled "outside" the box, such to indicate a collision with a wall. If this were the case, the respective velocity value was inverted depending on the wall in question. With the construction of the general time loop model, the procedure would first ensure the program operated correctly by plotting the impulse on one the right wall over time for one particle. Adjusting program values detailed that this setup stage displayed clear data for a system of temperature 300K and a box length of 1m. This setup ensured that the impulse, and thus subsequent force and pressure (given equation 3 and 4) of the system was accurate.

It is relevant to point out that at this point, the system containted only one particle, with predefined x and y velocities of 300m/s and 200m/s respectively, and position defined by the function rand(), generating a random number between 0 and 1 for both x and y. Extending from this, the system was extended to include multiple particles, and extension made easy by the list structure of the positions and velocities. A Gaussian distribution of standard deviation given by the equipotential theorem at thermal equilibrium ($\sqrt{\frac{kT}{m}}$ was introduced in order to generate random, yet appropriate velocities for each particle. Following this, it was concluded that the simulation be run 4 times for the box lengths 0.0001m, 0.0002m and 0.0003m, calculating the average pressure with uncertainty given by equation 8, the code to which observed in appendix 2.

Finally, again extending the simulation to include particle interaction, it was first chosen to discuss how the temperature of the system evolved with time. It was relevant to observe that the particles should be evenly spaced initially in order reduce the likelyhood of the close proximity of the particles causing the system to "blow up". This was achieved by the code in figure 3.



```
x = np.zeros((N,N_time))  #Initializing positions in loop. Here, we store the positions sequentially in time in a matrix,
#with one dimension being the number of particles and the other being time. Of course here we have only 1 particle, but it
#  will be easy for you to extend this to simulate multiple particles. Note that initialising the variables like this, rather
# than increasing the size at each step in the loop, makes the computation faster.
y = np.zeros((N,N_time))

#We create a for loop to even space each of the particles initial locations
n_side = int(np.ceil(np.sqrt(N))) #Number of particels per side
num_particles = 0
spacing = L / n_side                # Distance between particles
for i in range(n_side):
    for j in range(n_side):
        if num_particles < N:
            x[num_particles, 0] = (i + 0.5) * spacing
            y[num_particles, 0] = (j + 0.5) * spacing
            num_particles += 1
        else:
            break

ax = np.zeros(N)
ay = np.zeros(N)
```

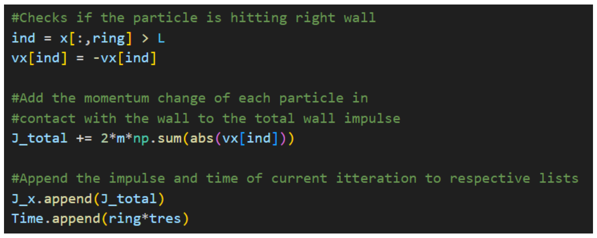
FIG. 3. Initial particle spacing code for interacting particle model.

Given the system was isolated, it was relevant to check that so too was the thermal velocity of the system over time, to ensure the simulation behave appropriately. This data was similarly extrapolated over 4 trials, following the same analysis discussed in uncertainties. Finally, it was concluded that a cooling factor of 0.999 would be introduced to the system by slowing the velocity of each particle by this amount every cycle. The intention behind this cooling was to observe how the pressure of the system evolves as the temperature is reduced, the results of which allowing comparisons to be drawn with the ideal gas model. This cooling process was similarly taken over 4 trials, with the same mean-uncertainty interpretation. The full simulation code for the particle interaction model was provided in appendix 3. It is worth noting the dimensional analysis between the three dimensional ideal gas model and the two dimensional simulation. It is apparent that the ideal gas model in our simulations translates vomlume and pressure from m^3 and $\frac{N}{m^2}$ to m^2 and $\frac{N}{m}$ respectively. Notice how this does not change the units of PV, however, it does modify the units of the values themselves. As such, this report considers that any time the units pascals is stated, it is the two dimensional $\frac{N}{m}$ being referenced, and similarly volume in m^2.

IV. RESULTS

A. Ideal Gas simulation

As discussed in the procedure, it was first obvious to ensure that the impulse on the right wall of the box exhibited the properties expected for the single particle model. Running the source code provided in Appendix 1, at each time iteration (ring) of length "tres" through the simulation, it was checked if the particles position was beyond the length of the box, indicating a collision with the right wall. As outlined in figure 4, the particles velocity was sign inverted, and the total change in impulse was added. Finally, the current iterations time and impulse was recorded in order to later plot the graph observed in figure 5.



```
#Checks if the particle is hitting right wall
ind = x[:,ring] > L
vx[ind] = -vx[ind]

#Add the momentum change of each particle in
#contact with the wall to the total wall impulse
J_total += 2*m*np.sum(abs(vx[ind]))

#Append the impulse and time of current itteration to respective lists
J_x.append(J_total)
Time.append(ring*tres)
```

FIG. 4. Ideal gas simulation code attributed with the action of updating impulse-time lists.

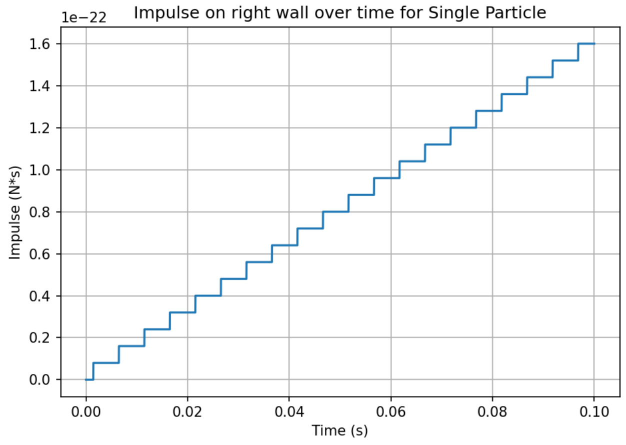
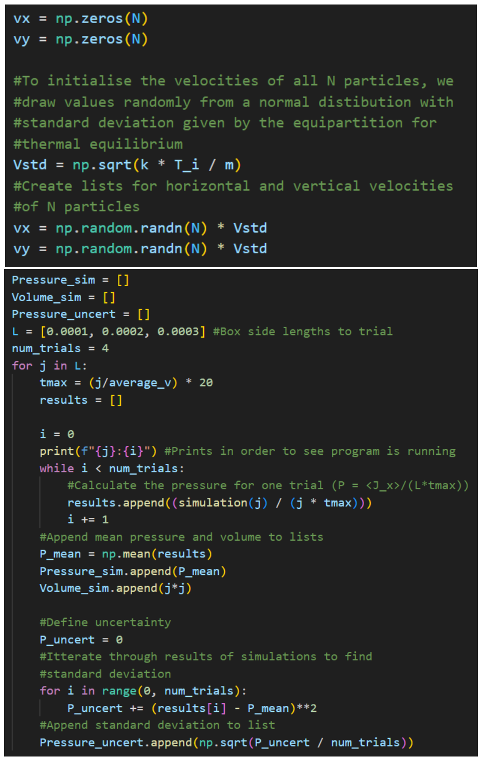


FIG. 5. Impulse on right wall over time for single particle ideal gas simulation.

Additionally, the code utilised the equation 3 in order to estimate the average force at $1.6 * 10^{-21} N$. It is relevant to note that this even over four trials, the simulation exhibited no variance in average force, a direct result of the constant velocity, such that it hits the right wall an equal number of times in each trial.



```
vx = np.zeros(N)
vy = np.zeros(N)

#To initialise the velocities of all N particles, we
#draw values randomly from a normal distibution with
#standard deviation given by the equipartition for
#thermal equilibrium
Vstd = np.sqrt(k * T_i / m)
#Create lists for horizontal and vertical velocities
#of N particles
vx = np.random.randn(N) * Vstd
vy = np.random.randn(N) * Vstd
```

```
Pressure_sim = []
Volume_sim = []
Pressure_uncert = []
L = [0.0001, 0.0002, 0.0003] #Box side lengths to trial
num_trials = 4
for j in L:
    tmax = (j/average_v) * 20
    results = []

    i = 0
    print(f"{j}:{i}") #Prints in order to see program is running
    while i < num_trials:
        #Calculate the pressure for one trial (P = <J_x>/(L*tmax))
        results.append((simulation(j) / (j * tmax)))
        i += 1
    #Append mean pressure and volume to lists
    P_mean = np.mean(results)
    Pressure_sim.append(P_mean)
    Volume_sim.append(j*j)

    #Define uncertainty
    P_uncert = 0
    #Itterate through results of simulations to find
    #standard deviation
    for i in range(0, num_trials):
        P_uncert += (results[i] - P_mean)**2
    #Append standard deviation to list
    Pressure_uncert.append(np.sqrt(P_uncert / num_trials))
```

FIG. 6. Code for the initialisation of velocities for multiple particles, and the calculations of uncertainties for varying volumes of 4 trials.

With this check conducted, the simulation was then extended to include multiple as outlined in the procedure, with velocities initialised through the equipartition theorem outlined by equation 5, the code for which displayed in figure 6.

The experiment was conducted for 4 times for boxes of side length 0.0001m, 0.0002m and 0.0003m. The uncertainty for each length trial was attributed as specified in equation 8. Plotting these values against the ideal PV diagram figure 7 was constructed.

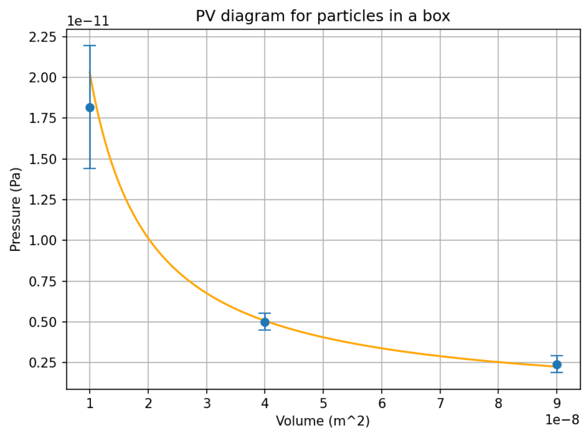
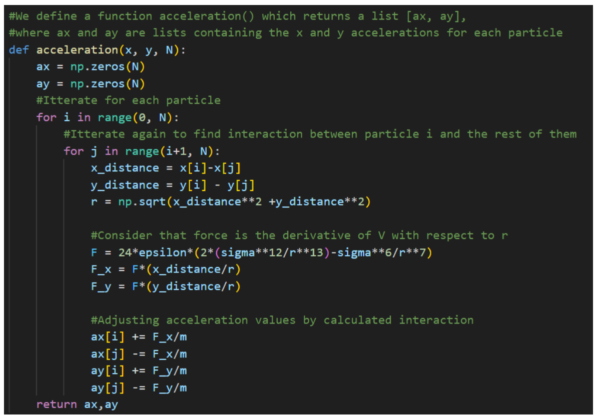


FIG. 7. Pressure over Volume for 4 simulation trials for lengths 0.0001m, 0.0002m and 0.0003m.

B. Interacting Gas simulation

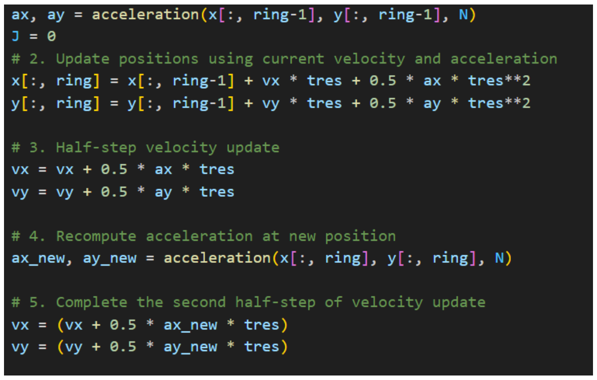
As previously discussed, a Lennard-Jones interaction potential was to be implemented into the model, such that with each simulation iteration, a new method acceleration() (figure 8) was called in order to update model the force each particle applied to each particle.



```
#We define a function acceleration() which returns a list [ax, ay],
#where ax and ay are lists containing the x and y accelerations for each particle
def acceleration(x, y, N):
    ax = np.zeros(N)
    ay = np.zeros(N)
    #Itterate for each particle
    for i in range(0, N):
        #Itterate again to find interaction between particle i and the rest of them
        for j in range(i+1, N):
            x_distance = x[i]-x[j]
            y_distance = y[i] - y[j]
            r = np.sqrt(x_distance**2 +y_distance**2)

            #Consider that force is the derivative of V with respect to r
            F = 24*epsilon*(2*(sigma**12/r**13)-sigma**6/r**7)
            F_x = F*(x_distance/r)
            F_y = F*(y_distance/r)

            #Adjusting acceleration values by calculated interaction
            ax[i] += F_x/m
            ax[j] -= F_x/m
            ay[i] += F_y/m
            ay[j] -= F_y/m
    return ax,ay
```

FIG. 8. Acceleration function modeling the acceleration of each particle given the Lennard-Jones interactions between every particle pair.

This final acceleration for each particle was then applied to update the velocities of each particle with regards to the velocity Verlet method, as observed in figure 9.



```
ax, ay = acceleration(x[:, ring-1], y[:, ring-1], N)
J = 0
# 2. Update positions using current velocity and acceleration
x[:, ring] = x[:, ring-1] + vx * tres + 0.5 * ax * tres**2
y[:, ring] = y[:, ring-1] + vy * tres + 0.5 * ay * tres**2

# 3. Half-step velocity update
vx = vx + 0.5 * ax * tres
vy = vy + 0.5 * ay * tres

# 4. Recompute acceleration at new position
ax_new, ay_new = acceleration(x[:, ring], y[:, ring], N)

# 5. Complete the second half-step of velocity update
vx = (vx + 0.5 * ax_new * tres)
vy = (vy + 0.5 * ay_new * tres)
```

FIG. 9. Verlet velocity calculator used to update the positions and velocities of each particle given particle interaction accelerations.

With this implemented functionality, as outlined in the procedure, it was first investigated how the temperature of the system evolved over time. Extracting the time and temperature values to an excel document using the pandas library, and finding averages and uncertainties given equation 8. Figure 10 was constructed.

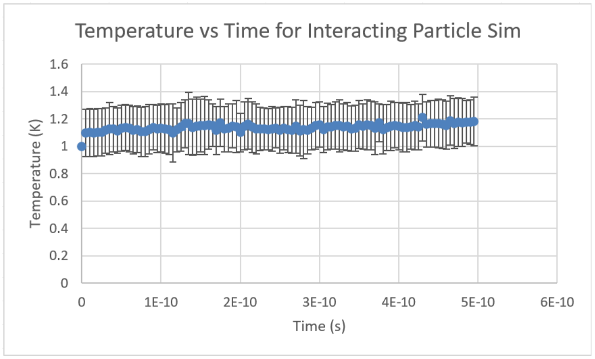


FIG. 10. Temperature against time for interacting gas simulation without cooling.

Finally, as per the procedure, it was decided to implement a cooling factor into the system, induced by multiplying the step 5 velocity values in figure 9 by 0.999. The purpose of this modification was to observe the relationship between pressure and temperature over time. This was achieved by similarly exporting 4 trials to excel, with the results displayed in figure 11.

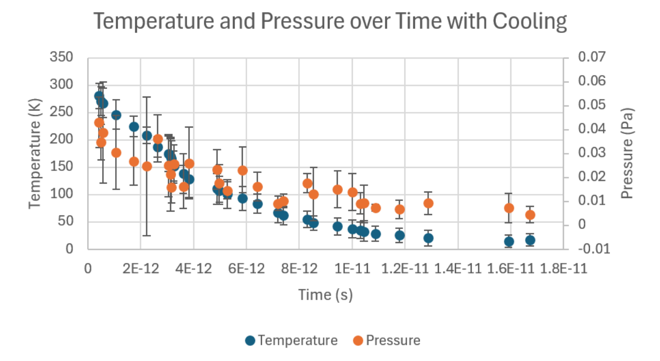


FIG. 11. Temperature and pressure over time with velocity cooling factor of 0.999.

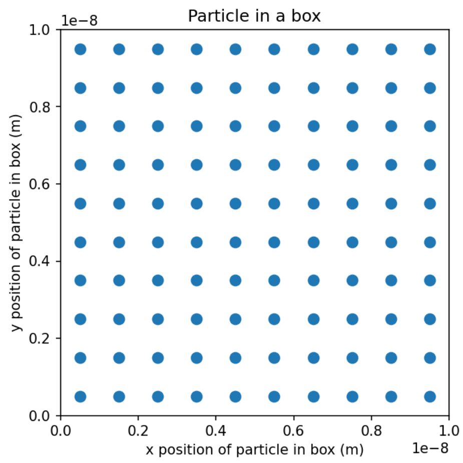
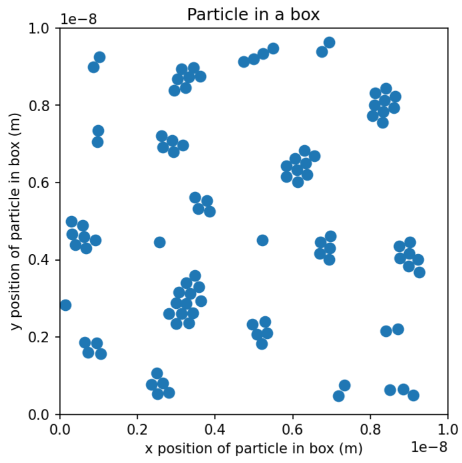



FIG. 12. Initial (TOP) and final (BOTTOM) positions of 100 simulated helium atoms run at an initial 0.5K and cooling factor of 0.999.

V. DISCUSSION

Analysis of the findings for the ideal gas simulation outlined its direct correspondence with that theoretically given by the ideal gas law. Initial setup of the simulation with one particle produced distinct increases in impulse in figure 5, followed by a flat-line period, correlating with that of the oscillations of the single particle colliding with the right wall, followed by a period box traversal. Supported by equation 3, the average force experienced by the right wall was found to be $1.6 * 10^{-21} N$, a value which carried no variance throughout the four simulations, but none the less supported by the constant 400m/s horizontal velocity and box size of 1m. As outlined by the procedure, followed was the extension of multiple non-interacting particles, with initial velocities drawn randomly from a Gaussian distribution of standard deviation $\sqrt{\frac{kT}{m}}$ as described by the equipartition theorem of thermal equilibrium (equation 5). Similarly calculating the pressure, along with the uncertainty given by equation 8, 4 trials of box sizes 0.0001m, 0.0002m and 0.0003m were observed to comfortably house the theoretically accepted value within the bounds of uncertainty. As was hypothesised, the microscopic simulation, whilst obeying simple thermodynamic conditions, distinctively portrayed the pressure-volume characteristics of a non-interacting, point-like ideal gas as given by the ideal gas model. In further extension to this model, analysis of the interacting particle simulation clearly illustrated the trends typical with that of the Lenard-Jones interaction potential. Firstly, by extending the code to include a velocity Verlet method, the acceleration for which governed by the Lennard-Jones potential and implemented by the acceleration() function in figure 8, the particles are converted to interacting. This implementation was first tested without cooling as exampled in the procedure, the thermal velocity was recorded over time and portrayed in figure 10. As evident, with an initial condition of 1K temperature, the system maintained this temperature, deviating at max by 0.4K, as given by the error bars constructed with equation 8. This built confidence in the simulation given the isolated system should maintain thermal velocity throughout time.

With a thermal velocity cooling factor of 0.999 introduced to the velocity Verlet sequence, the simulation once again demonstrated the pressure-temperature relationship theoretically expected. As observed in figure 11, as the thermal velocity was reduced (temperature), the pressure generally followed the same decay. This decrease quantitatively outlined the observation made in the simulation in figure 12, such that as the velocities of the particles were decreased over time via cooling, so to did the number of particle collisions, impulse and force on the right wall, and as such both the thermal temperature and pressure decreased. This cooling process was permitted to continue until the temperatures were perceived low enough to induce condensation, at which point, particles close enough to each other exhibited velocities too low to overcome the Lennard-Jones potential force from the other. This was visually observed in figure 12, with the particles initialised in a grid structure, and ending in stable grouped clumps.

This grouping was reasoned to be the minima in potential, such that the repulsive and attractive forces balanced, with the particles settling in a spacial arrangement parallel with that of the ground state distribution. Each grouped particle was observed at roughly a distance $r_{min} = 2^{1/6}\sigma$ from each other given helium's $\sigma = 0.264 * 10^{-9} J$. This directly correlates with the lowest energy state, within which the system is most stable, as initially proposed by the derivation of equation 6.

On this small scale, the simulations provide detailed insight for students into the dynamics of gaseous systems, depicting both the general ideal gas laws, but also more complex phenomena such as ground state settling during condensation. This is not new theory, particle simulations have already had wide ranging applications, from fluid dynamics and particle beam bending simulations for engineers to particle acceleration simulations for physicists.

VI. CONCLUSION

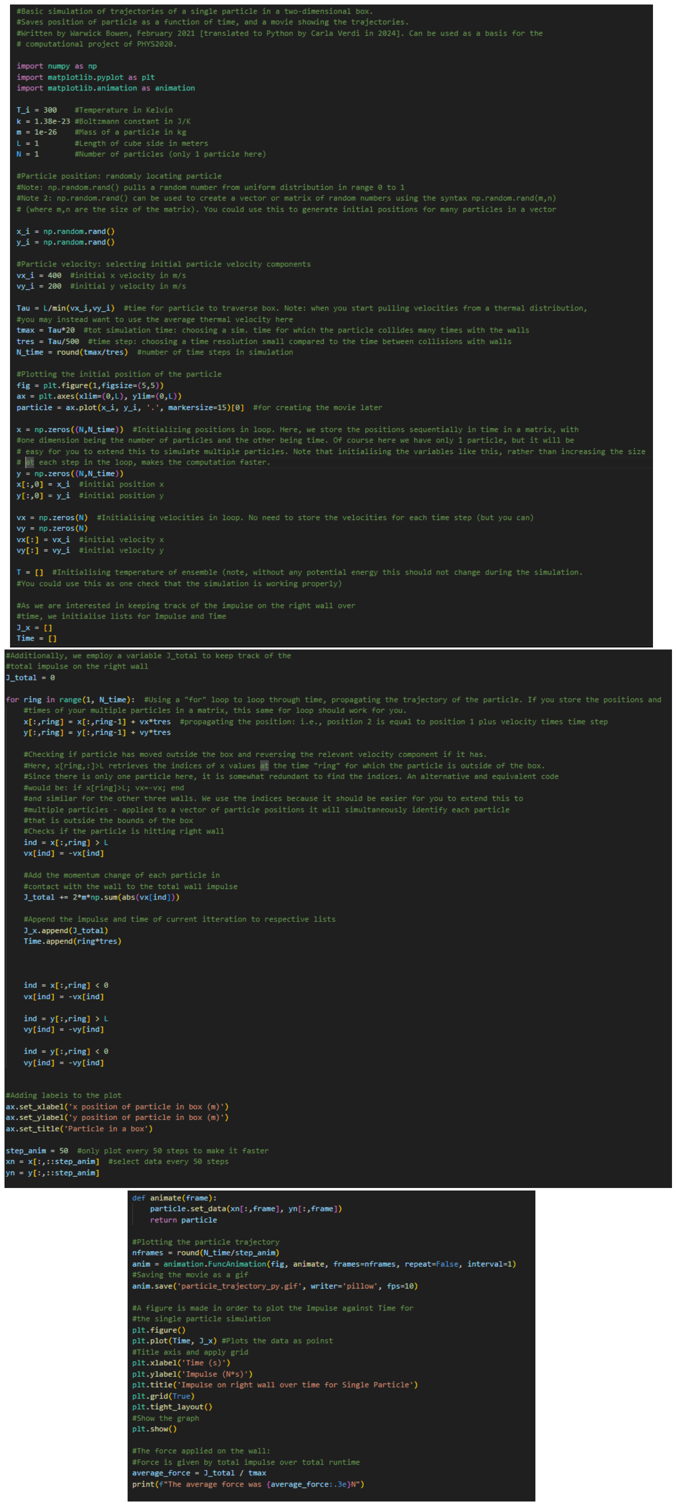
The purpose of this report was to quantitatively state the accuracy of a computer simulation of non-interacting, and interacting particles in a box against that theoretically expected through ideal gas laws. The procedure first investigated the impulse on the right wall of the box, given the system consisted of one particle of mass 1kg and initial velocity 400m/s. This constant velocity ensured the total impule was the same for all trials, resulting in an average force on the right wall of $1.6 * 10^{-21} N$. The code was then extended to simulated multiple particles of same 1kg mass, with the pressure results for each trialed box lengths presented figure 7. Over the four simulations for each length, the simulation produced pressure values, with associated uncertainties that enclosed that theoretically predicted by the ideal gas equation for 49 particles. Straying from the non-interacting, point-like model of ideal gases, an Lennard-Jones potential was then implemented in order to simulate the gas state value relationships for helium. Without a cooling factor applied to the velocity Verlet calculator, the system maintained, with minimal uncertainty, a thermal velocity steady with that initially set as observed in figure 10. However, as theoretically prediced by the ideal gas equation, implementation of a cooling factor of 0.999, reduced both the system temperature and pressure generally proportional over time as observed in figure 11. When provided with ample cooling time, the system approached the final system state observed in figure 12, with proximal particles settling in the ground state position of $2^{1/6}\sigma$ as given from the Lennard-Jones potential for helium. Overall, it is clear both the small scale ideal gas and interacting gas simulations were consistent with the properties of macroscopic gases, proving the effectiveness of simulations for practical real world uses. Simulations based on these principles have broad applications, not the least of which including fluid dynamic engineering, accelerating particle simulations, particle based beam bending and of course, an effective particle dynamics teaching resource.

[1] I. for Science Advancement, Ideal gas law: A history of the pneumatic sciences (2016), accessed: 2025-05-14.

[2] J. McGovern, An "ideal" gas, `https://theory.physics.manchester.ac.uk/~judith/stat_therm/node96.html` (n.d.), accessed: 2025-05-14.

[3] Encyclopædia Britannica, Equipartition of energy (2024), accessed: 2025-05-14.

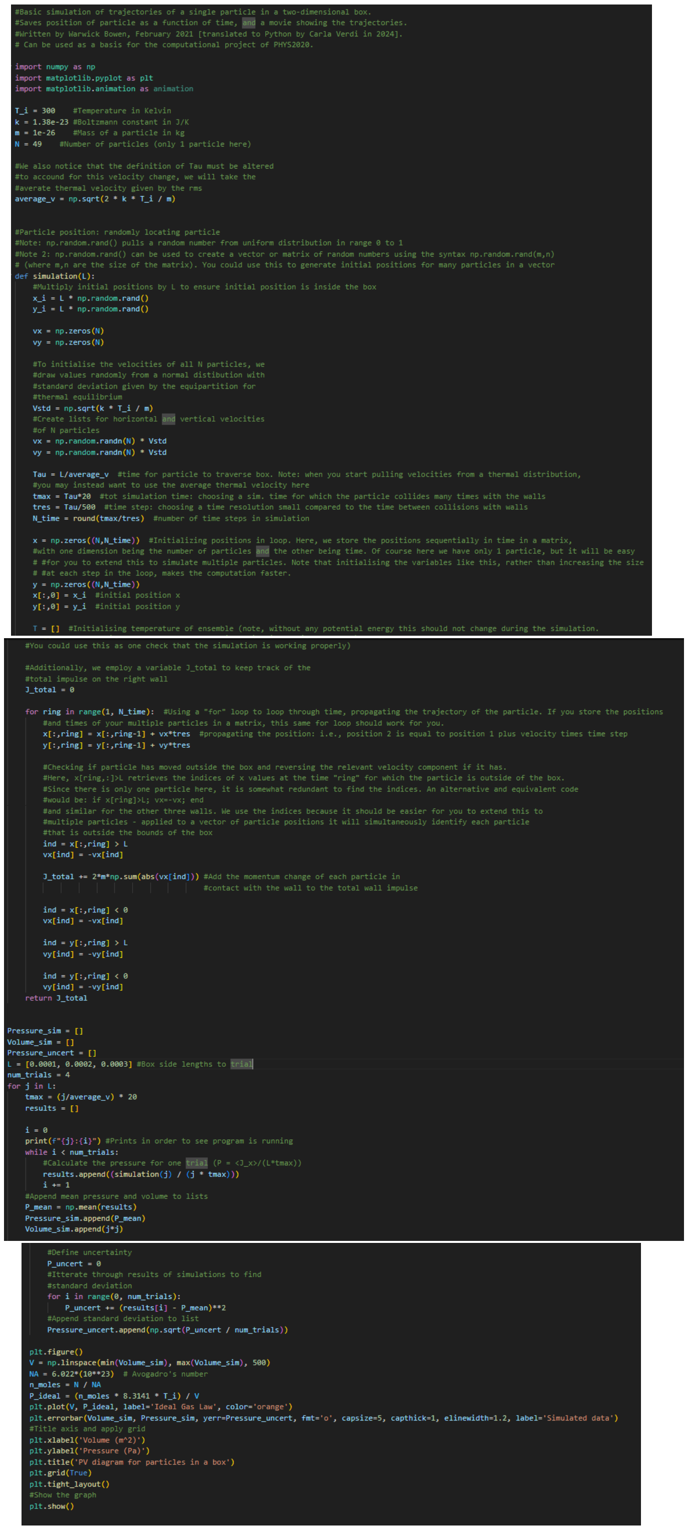
1. Appendix 1: Ideal gas simulation code Q1a



```python
#Basic simulation of trajectories of a single particle in a two-dimensional box.
#Saves position of particle as a function of time, and a movie showing the trajectories.
#Written by Warwick Bowen, February 2021 [translated to Python by Carla Verdi in 2024]. Can be used as a basis for the
# computational project of PHYS2020.

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

T_i = 300    #Temperature in Kelvin
k = 1.38e-23 #Boltzmann constant in J/K
m = 1e-26    #Mass of a particle in kg
L = 1        #Length of cube side in meters
N = 1        #Number of particles (only 1 particle here)

#Particle position: randomly locating particle
#Note: np.random.rand() pulls a random number from uniform distribution in range 0 to 1
#Note 2: np.random.rand() can be used to create a vector or matrix of random numbers using the syntax np.random.rand(m,n)
# (where m,n are the size of the matrix). You could use this to generate initial positions for many particles in a vector

x_i = np.random.rand()
y_i = np.random.rand()

#Particle velocity: selecting initial particle velocity components
vx_i = 400  #initial x velocity in m/s
vy_i = 200  #initial y velocity in m/s

Tau = L/min(vx_i,vy_i)  #time for particle to traverse box. Note: when you start pulling velocities from a thermal distribution,
#you may instead want to use the average thermal velocity here
tmax = Tau*20  #tot simulation time: choosing a sim. time for which the particle collides many times with the walls
tres = Tau/500  #time step: choosing a time resolution small compared to the time between collisions with walls
N_time = round(tmax/tres)  #number of time steps in simulation

#Plotting the initial position of the particle
fig = plt.figure(1,figsize=(5,5))
ax = plt.axes(xlim=(0,L), ylim=(0,L))
particle = ax.plot(x_i, y_i, '.', markersize=15)[0]  #for creating the movie later

x = np.zeros((N,N_time))  #Initializing positions in loop. Here, we store the positions sequentially in time in a matrix, with
#one dimension being the number of particles and the other being time. Of course here we have only 1 particle, but it will be
# easy for you to extend this to simulate multiple particles. Note that initialising the variables like this, rather than increasing the size
# at each step in the loop, makes the computation faster.
y = np.zeros((N,N_time))
x[:,0] = x_i  #initial position x
y[:,0] = y_i  #initial position y

vx = np.zeros(N)  #Initialising velocities in loop. No need to store the velocities for each time step (but you can)
vy = np.zeros(N)
vx[:] = vx_i  #initial velocity x
vy[:] = vy_i  #initial velocity y

T = []  #Initialising temperature of ensemble (note, without any potential energy this should not change during the simulation.
#You could use this as one check that the simulation is working properly)

#As we are interested in keeping track of the impulse on the right wall over
#time, we initialise lists for Impulse and Time
J_x = []
Time = []
#Additionally, we employ a variable J_total to keep track of the
#total impulse on the right wall
J_total = 0

for ring in range(1, N_time):  #Using a "for" loop to loop through time, propagating the trajectory of the particle. If you store the positions and
    #times of your multiple particles in a matrix, this same for loop should work for you.
    x[:,ring] = x[:,ring-1] + vx*tres  #propagating the position: i.e., position 2 is equal to position 1 plus velocity times time step
    y[:,ring] = y[:,ring-1] + vy*tres

    #Checking if particle has moved outside the box and reversing the relevant velocity component if it has.
    #Here, x[ring,:]>L retrieves the indices of x values at the time "ring" for which the particle is outside of the box.
    #Since there is only one particle here, it is somewhat redundant to find the indices. An alternative and equivalent code
    #would be: if x[ring]>L; vx=-vx; end
    #and similar for the other three walls. We use the indices because it should be easier for you to extend this to
    #multiple particles - applied to a vector of particle positions it will simultaneously identify each particle
    #that is outside the bounds of the box
    #Checks if the particle is hitting right wall
    ind = x[:,ring] > L
    vx[ind] = -vx[ind]

    #Add the momentum change of each particle in
    #contact with the wall to the total wall impulse
    J_total += 2*m*np.sum(abs(vx[ind]))

    #Append the impulse and time of current itteration to respective lists
    J_x.append(J_total)
    Time.append(ring*tres)



    ind = x[:,ring] < 0
    vx[ind] = -vx[ind]

    ind = y[:,ring] > L
    vy[ind] = -vy[ind]

    ind = y[:,ring] < 0
    vy[ind] = -vy[ind]


#Adding labels to the plot
ax.set_xlabel('x position of particle in box (m)')
ax.set_ylabel('y position of particle in box (m)')
ax.set_title('Particle in a box')

step_anim = 50  #only plot every 50 steps to make it faster
xn = x[:,::step_anim]  #select data every 50 steps
yn = y[:,::step_anim]
def animate(frame):
    particle.set_data(xn[:,frame], yn[:,frame])
    return particle

#Plotting the particle trajectory
nframes = round(N_time/step_anim)
anim = animation.FuncAnimation(fig, animate, frames=nframes, repeat=False, interval=1)
#Saving the movie as a gif
anim.save('particle_trajectory_py.gif', writer='pillow', fps=10)

#A figure is made in order to plot the Impulse against Time for
#the single particle simulation
plt.figure()
plt.plot(Time, J_x) #Plots the data as poinst
#Title axis and apply grid
plt.xlabel('Time (s)')
plt.ylabel('Impulse (N*s)')
plt.title('Impulse on right wall over time for Single Particle')
plt.grid(True)
plt.tight_layout()
#Show the graph
plt.show()

#The force applied on the wall:
#Force is given by total impulse over total runtime
average_force = J_total / tmax
print(f"The average force was {average_force:.3e}N")
```

FIG. 13. Simulation code for Q1a.

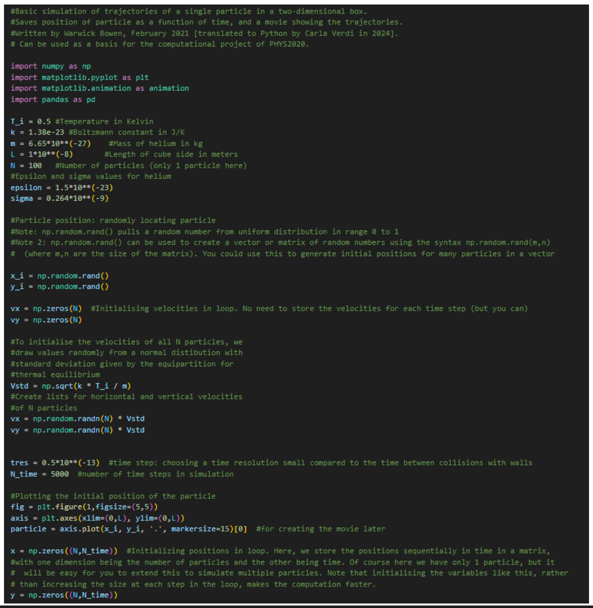
2. Appendix 2: Ideal gas simulation code Q1b and Q1c



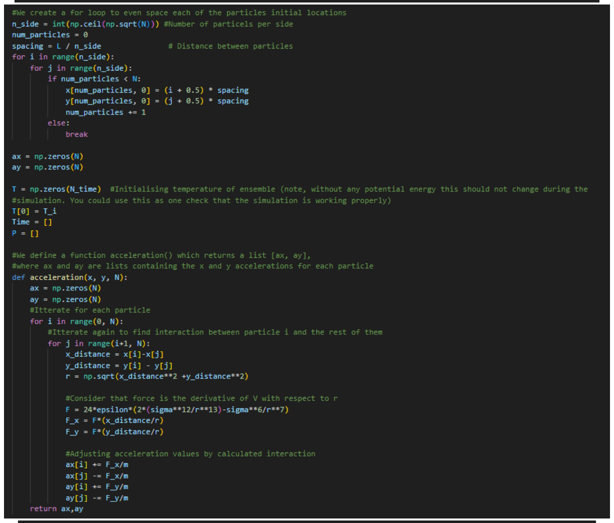
```python
#Basic simulation of trajectories of a single particle in a two-dimensional box.
#Saves position of particle as a function of time, and a movie showing the trajectories.
#Written by Warwick Bowen, February 2021 [translated to Python by Carla Verdi in 2024].
# Can be used as a basis for the computational project of PHYS2020.

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

T_i = 300    #Temperature in Kelvin
k = 1.38e-23 #Boltzmann constant in J/K
m = 1e-26    #Mass of a particle in kg
N = 49    #Number of particles (only 1 particle here)

#We also notice that the definition of Tau must be altered
#to accound for this velocity change, we will take the
#averate thermal velocity given by the rms
average_v = np.sqrt(2 * k * T_i / m)


#Particle position: randomly locating particle
#Note: np.random.rand() pulls a random number from uniform distribution in range 0 to 1
#Note 2: np.random.rand() can be used to create a vector or matrix of random numbers using the syntax np.random.rand(m,n)
# (where m,n are the size of the matrix). You could use this to generate initial positions for many particles in a vector
def simulation(L):
    #Multiply initial positions by L to ensure initial position is inside the box
    x_i = L * np.random.rand()
    y_i = L * np.random.rand()

    vx = np.zeros(N)
    vy = np.zeros(N)

    #To initialise the velocities of all N particles, we
    #draw values randomly from a normal distibution with
    #standard deviation given by the equipartition for
    #thermal equilibrium
    Vstd = np.sqrt(k * T_i / m)
    #Create lists for horizontal and vertical velocities
    #of N particles
    vx = np.random.randn(N) * Vstd
    vy = np.random.randn(N) * Vstd

    Tau = L/average_v  #time for particle to traverse box. Note: when you start pulling velocities from a thermal distribution,
    #you may instead want to use the average thermal velocity here
    tmax = Tau*20  #tot simulation time: choosing a sim. time for which the particle collides many times with the walls
    tres = Tau/500  #time step: choosing a time resolution small compared to the time between collisions with walls
    N_time = round(tmax/tres)  #number of time steps in simulation

    x = np.zeros((N,N_time))  #Initializing positions in loop. Here, we store the positions sequentially in time in a matrix,
    #with one dimension being the number of particles and the other being time. Of course here we have only 1 particle, but it will be easy
    # #for you to extend this to simulate multiple particles. Note that initialising the variables like this, rather than increasing the size
    # #at each step in the loop, makes the computation faster.
    y = np.zeros((N,N_time))
    x[:,0] = x_i  #initial position x
    y[:,0] = y_i  #initial position y

    T = []  #Initialising temperature of ensemble (note, without any potential energy this should not change during the simulation.
    #You could use this as one check that the simulation is working properly)

    #Additionally, we employ a variable J_total to keep track of the
    #total impulse on the right wall
    J_total = 0

    for ring in range(1, N_time):  #Using a "for" loop to loop through time, propagating the trajectory of the particle. If you store the positions
        #and times of your multiple particles in a matrix, this same for loop should work for you.
        x[:,ring] = x[:,ring-1] + vx*tres  #propagating the position: i.e., position 2 is equal to position 1 plus velocity times time step
        y[:,ring] = y[:,ring-1] + vy*tres

        #Checking if particle has moved outside the box and reversing the relevant velocity component if it has.
        #Here, x[ring,:]>L retrieves the indices of x values at the time "ring" for which the particle is outside of the box.
        #Since there is only one particle here, it is somewhat redundant to find the indices. An alternative and equivalent code
        #would be: if x[ring]>L; vx=-vx; end
        #and similar for the other three walls. We use the indices because it should be easier for you to extend this to
        #multiple particles - applied to a vector of particle positions it will simultaneously identify each particle
        #that is outside the bounds of the box
        ind = x[:,ring] > L
        vx[ind] = -vx[ind]

        J_total += 2*m*np.sum(abs(vx[ind])) #Add the momentum change of each particle in
                                            #contact with the wall to the total wall impulse

        ind = x[:,ring] < 0
        vx[ind] = -vx[ind]

        ind = y[:,ring] > L
        vy[ind] = -vy[ind]

        ind = y[:,ring] < 0
        vy[ind] = -vy[ind]
    return J_total


Pressure_sim = []
Volume_sim = []
Pressure_uncert = []
L = [0.0001, 0.0002, 0.0003] #Box side lengths to trial
num_trials = 4
for j in L:
    tmax = (j/average_v) * 20
    results = []

    i = 0
    print(f"{j}:{i}") #Prints in order to see program is running
    while i < num_trials:
        #Calculate the pressure for one trial (P = <J_x>/(L*tmax))
        results.append((simulation(j) / (j * tmax)))
        i += 1
    #Append mean pressure and volume to lists
    P_mean = np.mean(results)
    Pressure_sim.append(P_mean)
    Volume_sim.append(j*j)
    #Define uncertainty
    P_uncert = 0
    #Itterate through results of simulations to find
    #standard deviation
    for i in range(0, num_trials):
        P_uncert += (results[i] - P_mean)**2
    #Append standard deviation to list
    Pressure_uncert.append(np.sqrt(P_uncert / num_trials))

plt.figure()
V = np.linspace(min(Volume_sim), max(Volume_sim), 500)
NA = 6.022*(10**23)  # Avogadro's number
n_moles = N / NA
P_ideal = (n_moles * 8.3141 * T_i) / V
plt.plot(V, P_ideal, label='Ideal Gas Law', color='orange')
plt.errorbar(Volume_sim, Pressure_sim, yerr=Pressure_uncert, fmt='o', capsize=5, capthick=1, elinewidth=1.2, label='Simulated data')
#Title axis and apply grid
plt.xlabel('Volume (m^2)')
plt.ylabel('Pressure (Pa)')
plt.title('PV diagram for particles in a box')
plt.grid(True)
plt.tight_layout()
#Show the graph
plt.show()
```

FIG. 14. Simulation code for Q1b and Q1c.

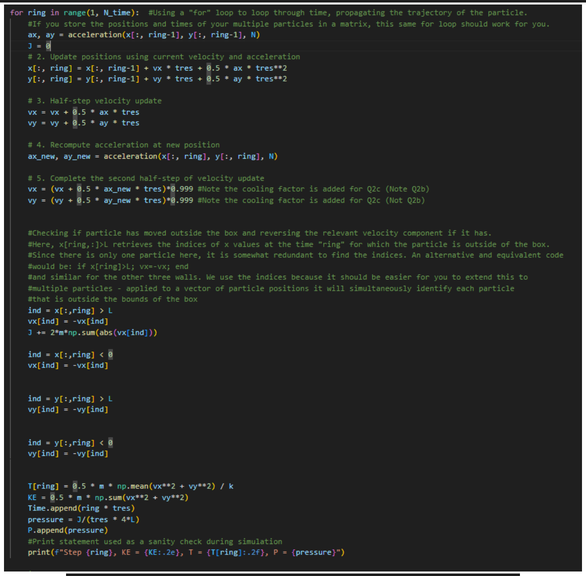
3. Appendix 3: Interacting gas simulation code Q2a, b and c.



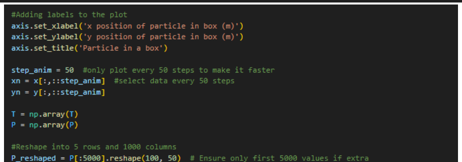
```python
#Basic simulation of trajectories of a single particle in a two-dimensional box.
#Saves position of particle as a function of time, and a movie showing the trajectories.
#Written by Warwick Bowen, February 2021 [translated to Python by Carla Verdi in 2024].
# Can be used as a basis for the computational project of PHYS2020.

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import pandas as pd

T_i = 0.5 #Temperature in Kelvin
k = 1.38e-23 #Boltzmann constant in J/K
m = 6.65*10**(-27)    #Mass of helium in kg
L = 1*10**(-8)       #Length of cube side in meters
N = 100   #Number of particles (only 1 particle here)
#Epsilon and sigma values for helium
epsilon = 1.5*10**(-23)
sigma = 0.264*10**(-9)

#Particle position: randomly locating particle
#Note: np.random.rand() pulls a random number from uniform distribution in range 0 to 1
#Note 2: np.random.rand() can be used to create a vector or matrix of random numbers using the syntax np.random.rand(m,n)
#  (where m,n are the size of the matrix). You could use this to generate initial positions for many particles in a vector

x_i = np.random.rand()
y_i = np.random.rand()

vx = np.zeros(N)  #Initialising velocities in loop. No need to store the velocities for each time step (but you can)
vy = np.zeros(N)

#To initialise the velocities of all N particles, we
#draw values randomly from a normal distibution with
#standard deviation given by the equipartition for
#thermal equilibrium
Vstd = np.sqrt(k * T_i / m)
#Create lists for horizontal and vertical velocities
#of N particles
vx = np.random.randn(N) * Vstd
vy = np.random.randn(N) * Vstd


tres = 0.5*10**(-13)  #time step: choosing a time resolution small compared to the time between collisions with walls
N_time = 5000  #number of time steps in simulation

#Plotting the initial position of the particle
fig = plt.figure(1,figsize=(5,5))
axis = plt.axes(xlim=(0,L), ylim=(0,L))
particle = axis.plot(x_i, y_i, '.', markersize=15)[0]  #for creating the movie later

x = np.zeros((N,N_time))  #Initializing positions in loop. Here, we store the positions sequentially in time in a matrix,
#with one dimension being the number of particles and the other being time. Of course here we have only 1 particle, but it
#  will be easy for you to extend this to simulate multiple particles. Note that initialising the variables like this, rather
# than increasing the size at each step in the loop, makes the computation faster.
y = np.zeros((N,N_time))
```



```python
#We create a for loop to even space each of the particles initial locations
n_side = int(np.ceil(np.sqrt(N))) #Number of particels per side
num_particles = 0
spacing = L / n_side               # Distance between particles
for i in range(n_side):
    for j in range(n_side):
        if num_particles < N:
            x[num_particles, 0] = (i + 0.5) * spacing
            y[num_particles, 0] = (j + 0.5) * spacing
            num_particles += 1
        else:
            break

ax = np.zeros(N)
ay = np.zeros(N)

T = np.zeros(N_time)  #Initialising temperature of ensemble (note, without any potential energy this should not change during the
#simulation. You could use this as one check that the simulation is working properly)
T[0] = T_i
Time = []
P = []

#We define a function acceleration() which returns a list [ax, ay],
#where ax and ay are lists containing the x and y accelerations for each particle
def acceleration(x, y, N):
    ax = np.zeros(N)
    ay = np.zeros(N)
    #Itterate for each particle
    for i in range(0, N):
        #Itterate again to find interaction between particle i and the rest of them
        for j in range(i+1, N):
            x_distance = x[i]-x[j]
            y_distance = y[i] - y[j]
            r = np.sqrt(x_distance**2 +y_distance**2)

            #Consider that force is the derivative of V with respect to r
            F = 24*epsilon*(2*(sigma**12/r**13)-sigma**6/r**7)
            F_x = F*(x_distance/r)
            F_y = F*(y_distance/r)

            #Adjusting acceleration values by calculated interaction
            ax[i] += F_x/m
            ax[j] -= F_x/m
            ay[i] += F_y/m
            ay[j] -= F_y/m
    return ax,ay
```



```python
for ring in range(1, N_time):  #Using a "for" loop to loop through time, propagating the trajectory of the particle.
    #If you store the positions and times of your multiple particles in a matrix, this same for loop should work for you.
    ax, ay = acceleration(x[:, ring-1], y[:, ring-1], N)
    J = 0
    # 2. Update positions using current velocity and acceleration
    x[:, ring] = x[:, ring-1] + vx * tres + 0.5 * ax * tres**2
    y[:, ring] = y[:, ring-1] + vy * tres + 0.5 * ay * tres**2

    # 3. Half-step velocity update
    vx = vx + 0.5 * ax * tres
    vy = vy + 0.5 * ay * tres

    # 4. Recompute acceleration at new position
    ax_new, ay_new = acceleration(x[:, ring], y[:, ring], N)

    # 5. Complete the second half-step of velocity update
    vx = (vx + 0.5 * ax_new * tres)*0.999 #Note the cooling factor is added for Q2c (Note Q2b)
    vy = (vy + 0.5 * ay_new * tres)*0.999 #Note the cooling factor is added for Q2c (Not Q2b)


    #Checking if particle has moved outside the box and reversing the relevant velocity component if it has.
    #Here, x[ring,:]>L retrieves the indices of x values at the time "ring" for which the particle is outside of the box.
    #Since there is only one particle here, it is somewhat redundant to find the indices. An alternative and equivalent code
    #would be: if x[ring]>L; vx=-vx; end
    #and similar for the other three walls. We use the indices because it should be easier for you to extend this to
    #multiple particles - applied to a vector of particle positions it will simultaneously identify each particle
    #that is outside the bounds of the box
    ind = x[:,ring] > L
    vx[ind] = -vx[ind]
    J += 2*m*np.sum(abs(vx[ind]))

    ind = x[:,ring] < 0
    vx[ind] = -vx[ind]


    ind = y[:,ring] > L
    vy[ind] = -vy[ind]


    ind = y[:,ring] < 0
    vy[ind] = -vy[ind]


    T[ring] = 0.5 * m * np.mean(vx**2 + vy**2) / k
    KE = 0.5 * m * np.sum(vx**2 + vy**2)
    Time.append(ring * tres)
    pressure = J/(tres * 4*L)
    P.append(pressure)
    #Print statement used as a sanity check during simulation
    print(f"Step {ring}, KE = {KE:.2e}, T = {T[ring]:.2f}, P = {pressure}")
```



```python
#Adding labels to the plot
axis.set_xlabel('x position of particle in box (m)')
axis.set_ylabel('y position of particle in box (m)')
axis.set_title('Particle in a box')

step_anim = 50  #only plot every 50 steps to make it faster
xn = x[:,::step_anim]  #select data every 50 steps
yn = y[:,::step_anim]

T = np.array(T)
P = np.array(P)

#Reshape into 5 rows and 1000 columns
P_reshaped = P[:5000].reshape(100, 50)  # Ensure only first 5000 values if extra
```