Thermodynamic Simulations of Rigid Spherical 'Balls' of Gas in a Fixed Container

including of Inelastic Collisions in a Container, Ideal Gas Relationships, Conservation Laws, the Ideal Gas Laws, the Van der Waals Law, and the Maxwell-Boltzmann Distribution

Martin A. He Sunday 4th December, 2022

Abstract – 2D-thermodynamic simulations were performed for a fixed number of rigid *spherical ball* approximations of atoms, within a circular container of fixed area over a set number of collisions. This initially examined the collisions of a single ball within a circular container, before being extended to encompass multiple balls, and producing various plots of properties of the system and the ball over time. In particular, with relation to the laws governing an ideal gas, conservation of various properties, Van der Waals forces and the Maxwell-Boltzmann distribution. Our simulation was found to be in agreement with all theoretical distributions.

I Introduction

It has long been theorised that the erratic movement of various particles in mediums can be explained through particle collisions, mostly notably in 1827 where Robert Brown observed the movement of pollen grains suspended in water.

From this motion, he deduced that this must be the product of bombardment with water molecules. coined Brownian motion in his honour. These intermolecular interactions are outside the scope of this model, and the collisions which are simulated are purely rigid-body elastic collisions, without any attractive forces.

II Theory, Method & Data Collection

In all, the project was split between three main Python files: *ball.py*, which controls the movements of individual Ball objects, *simulation.py*, the primary file controlling the set-up of the simulation, as well as *plots.py* for the testing and generation of plots. Simulation contains the *Simulation* class, along with the *Plot* and *Event* classes, whilst Ball has the *Ball* and *Event* classes.

In order to optimise the data processing efficiency, the Python libraries Numpy, Matplotlib, Scipy, as well as other modules commonly found on Anaconda[1] were utilised, including: Pandas, Seaborn, Itertools, Heapdict, as well as built-in Python packages. These were chosen to extend the functionality of Python, in particular with the extensive use of Pandas *Dataframes* as opposed to say, Python *Arrays,* likewise with Numpy *Arrays* where appropriate[2]. This resulted in a fast, efficient runtime composed of a mere *16 functions* in the entire Simulation class. A progress bar was incorporated to aid the user experience.

A. *Ball.py* *(import ball as bl)*

More information can be found at martinhe.com/thermo-project.

This module is composed of 5 primary sections, including the: Initialisation, Information, Movement, Attributes and *Container* sections. The Initialisation section controls the configuration of the ball, including its main properties to be parsed, whilst the Information section gives the representation and string formats. Movement governs the functions which control how the ball travels in the *Container*, by calculating the time to the next collision for the ball *(time_to_collision)*, which makes use of the following equation shown below:

$$\left(\mathbf{r}_1 + \mathbf{v}_1\delta t - \mathbf{r}_2 - \mathbf{v}_2\delta t\right)^2 = \left(R_1 \pm R_2\right)^2 \tag{1}$$

Equation 1: The dynamic equation for two colliding idealised balls in an elastic collision, where ***r***, ***v***, and *R* are the position, velocity, and radius of the balls or container respectively. Evidently, there are many solutions from this equation.

Rearranging (1) to form a quadratic expression, we arrive at:

$$\delta t = \frac{-\vec{v}\cdot\vec{r} \pm \sqrt{\left(\vec{v}\cdot\vec{r}\right)^2 - \left(\vec{v}\cdot\vec{v}\right)\left(r\cdot\vec{r} - R^2\right)}}{\vec{v}\cdot\vec{v}} \tag{2}$$

Equation 2: Rearranged form of the dynamic equation, from (1).

We have that (2) evidentally gives four possibilities: no real solution of δt, only negative solutions for δt, a repeated root of 0, or a positive and negative solution. Next, we must further consider the possibility of a collision with a container, as well as with another ball. Evidentally, we can discount the complex and negative solutions to δt since they indicate the next collision is in the past, which reduces to an absurdity. We thus, only need to consider the latter two scenarios: for the repeated root, there is only a logical outcome where $\delta t = -b/2a > 0$. As well as this, we also have the positive and negative solution. In any case, when colliding with another ball, this results in (3):

$$\begin{aligned}\mathbf{v}_1' &= \mathbf{v}_1 - \frac{2m_2}{m_1+m_2}\frac{\langle \mathbf{v}_1 - \mathbf{v}_2, \mathbf{x}_1 - \mathbf{x}_2\rangle}{\left\|\mathbf{x}_1 - \mathbf{x}_2\right\|^2}\left(\mathbf{x}_1 - \mathbf{x}_2\right)\\ \mathbf{v}_2' &= \mathbf{v}_2 - \frac{2m_1}{m_1+m_2}\frac{\langle \mathbf{v}_2 - \mathbf{v}_1, \mathbf{x}_2 - \mathbf{x}_1\rangle}{\left\|\mathbf{x}_2 - \mathbf{x}_1\right\|^2}\left(\mathbf{x}_2 - \mathbf{x}_1\right)\end{aligned} \tag{3}$$

Equation 3: Solution to the dynamic equation[3] applied to masses (m_1, m_2), positions (x_1, x_2) and velocities (v_1, v_2, v_1', v_2') of the balls, between two balls. The solutions between a ball and container can consequently be dervied.

The *Attributes* section simply sets and returns all the ball properties, whereas, the *Container* class simply inherits from the Ball class, whilst assigning a radius, mass, and count to it.

B. *Simulation.py* *(import simulation as sim)*

The Simulation module, the core of the project, contains the *Simulation*, *Event* and *Plot* classes. The *Event* object merely acts as a tuple record for collision events between any pair of balls, whilst the *Plot* class provides methods for plots and histograms. Seaborn was used alongside Matplotlib to add more flexibility to the plots which could be created, including 3D plots. The Simulation class is divided into *13 functions*, which each have many parameters to improve organisation and prevent clutter, with extensive documentation and judicious commenting. Prior to the Simulation class, the *gen_random_uniform* function is declared due to its universal application across classes. The first three functions are self-explanatory, and given by their docstrings, so we shall proceed to the *glossary* function which returns a dictionary of all quantities for plotting. This made use of a *heapdict* due to the fast processing speed, as well as its additional methods of *popitem* to obtain data. The *property* function is used to define and return various properties, whilst *set_vel_ball* defines all the ball velocities. Where appropriate, the *enumerate* function was substituted for *np.ndenumerate* for its faster runtime, though in some cases this was not possible due to its tuple output resulting in *TypeErrors*.

The *next_collision* function is arguably the most important of the entire project, defining the logic to set up and perform it, by finding this *dt*, before moving the system to this point, and performing the collision. Its logic is convoluted, by essentially, it selects from the heapdict, before running through all possible collision pairs, ascertaining the minimum *dt* for collisions with other balls as well as a container, whilst checking for a situation where multiple collisions occur *à la fois*. Finally, the UTC time moves forward, with new plotting, and recording completed.

$$\mathbf{r}' = \mathbf{r} + \mathbf{v} * \mathrm{dt} \quad (2)$$

Equation 4: The simple logic for moving the colliding balls to a new position.

The *randomiser* function provides all randomisation options for position and velocity. The *draw* function first initialises the patches, before proceeding to redraw all ball patches for every active tuple pair helpfully provided by the *itertools* library. The *record_property* function unsurpisingly records properties, including a complete dataset written into a *Pandas DataFrame*. Similarly *record_data_states* enables for a snapshot in time of data to be captured. Finally, the *run* method runs the above functions to perform the collisions and animation, recording data. A *progress* bar allows the user to keep track of the simulation. Finally, the *Event* and *Plot* classes enable for events to be recorded and plots to be generated respectively.

III Results & Analysis

The obtained plots are attached in the *Appendix* section. Overall, we see that the plots are in accordance with the expected graphs for each task, and a further breakdown can be found below.

In **Figure 1**, we obtain an animation of the simulation. This demonstrates fluid Brownian motion for varying numbers of N. In **Figure 2**, we obtain the plot for the absolute distance distribution from the Origin, as well as the relative distances between every ball pairs and their corresponding histograms. For the former, this is as expected since this problem can be reconsidered in two manners: firstly, going out from 0 to r, we find that the number of possible points increases by r^2; secondly, we observe that after collision events with the Container, it traverses a path of similar *r* to before, resulting in a spike at *r=10*. Meanwhile, the probability decreases by a factor of r^{-1}, resulting in an overall linear trend. For the latter, this is also as expected, which indicates a Gaussian-like curve with a left skew, and peak around *8m* separation, between *0* and *20m*. This is as expected as for a large number of colliding balls, by the CLT, it should tend to Gaussian. The skewness can similarly be explained by the same collision logic given before, where it traverses similar distances right after.

In **Figure 3**, have insight into conservation of quantities in the simulation. We can see that the pressure of the system resolves quickly into a steady state value after initial setup. For the momentum of collisions with the container, we see that momentum is similarly conserved, as is systemic kinetic energy. In **Figure 4**, we examine the various ideal gas relations, such as between pressure, temperature, volume, and N. We find linear fits with values corroborating theoretical explanations.

In **Figure 5**, we examine the Law itself through various isotherms of pressure against volume, for varying T and N. This results in decay curves as are expected according to the Theory. In **Figure 6**, we take a closer look at van der Waals' Law, discovering a upwards curve, for the Power Law over an effective area. Values for the results can be found in the Appendix Finally, in **Figure 7**, we look at the Maxwell-Boltzmann distribution which we find to be perfectly followed by the system for varying parameters. To demonstrate this, a 3D plot has been generated varying the number of collisions, and balls. Additional extension figures are shown in **Figure 8** onwards.

IV Conclusion

From our extensive data collection, we have demonstrated our simulation to be in agreement with the theoretical values. As a result, the following simulation can be accurately examine the provided tests and parameters can be freely altered if one wishes.

Yet, many improvements could be made to this experiment if time permitting. 3D plots would be generated for all data specified above, to see how varying the system parameters affects the strength of these distributions, whilst in order to improve computational efficiency, almost everything would be converted to *Numpy Arrays* of defined size. This would vastly

improve upon the current method appending to a *List*. Likewise, more repeats would be run of the simulation, and an average obtained to reduce random uncertainties. Additional plots could be generated for a situation describing the Brownian motion of pollen grains suspended in water grains: the origin of the Theory.

V Appendicies & Figures

1 Anaconda Environment

The Anaconda environment contains all the relevant Python packages as listed above in this report. For more information on how to go about installing these packages in other environments, please see the README file.

2 Numpy vs. Python Efficiency

It is oft said, and assumed that any corresponding Numpy feature will be faster than its Python counterpart, however over the duration of this project, I was surprised to find this to not always be the case, the most notable example being appending to Python lists as opposed to numpy arrays, where there can be up to a staggering 2000× difference in processing time.

3 Solutions to the Dynamic Equation in 2D

These solutions to the Newtonian collision between two rigid disc balls can be dervied, as found on the following website.

VI References

VII Testing Logs

A compilation of testing woes, debugging, and many tears which were shed.

Initial Aims

1. Change uses of lists [] to np.array where possible to improve efficiency. (I'm not very good at using np.append and np.zeros, so this will be quite the experience for me).

2. Remove uses of enumerate function, in favour of np.ndenumerate if possible.

3. Check docstring formatting is consistent, as is the list of functions and properties is consistent (e.g. speed, KE, temperature, pressure, etc.).

4. Create a final minified version of the Python code when testing, to run more repeats.

5. Use np.linspace instead of nested for-loops to increase efficiency.

6. Make sure singular/plural variable names is consistent between Python files.

7. Make sure spacing is appropriate between lines, and commenting is aligned.

8. Commenting should be judicious, in order to avoid cluttering the "flow" of the code and readibility. However, inline commenting is acceptable when spaced correctly. Commenting in every line is acceptable only for the initialisation and run functions.

9. Split the next_collision function into the three steps outline by the lab script, to increase flexibility in the order of which the operations can be performed, as well as clarity (this is because many targeted functions > one large convoluted function).

10. Implement an "event" Class for each collision, to store vital information about ball pairs.

Final Tests

1. Test the working functionality of Ball.py, the ball class, as well as the Container.