Project B: Thermodynamics Snookered

Note: Remember the importance of keeping records of the tasks that you have completed. Not all tasks will produce a plot that you wish to include in your final report. Rather than executing your code directly in the interpreter, where it is not persisted and will require you to input the lines several times, it might be a better idea to write the code that you use for testing or indeed for some intermediary tasks into a script. This has the advantages of being reproducible, persistent between session and directly assessable.

Introduction

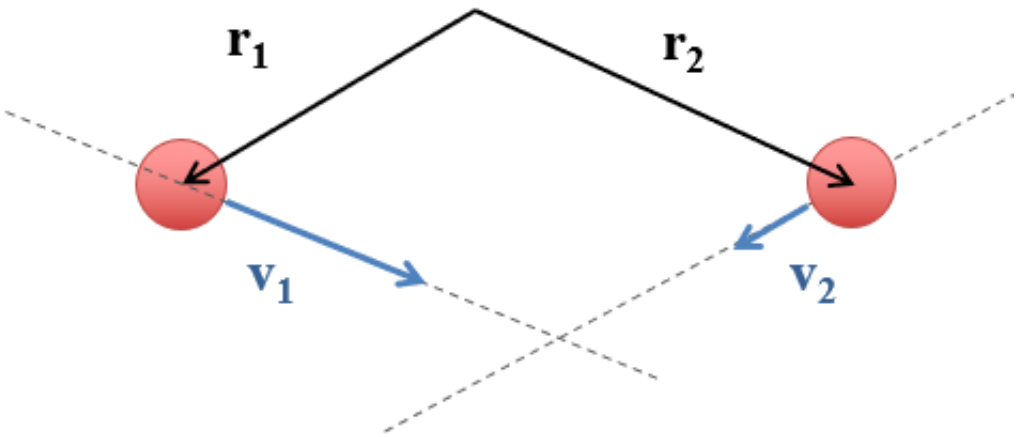
How does the modern description of a gas link to classical thermodynamics? The kinetic energy is linked to the temperature and the force due to atoms bouncing off the container leads to pressure. Can we build a 2-dimensional simulation of a gas and animate it to illustrate and investigate these links?

Project Aims:

- Write a code to describe a single ball bouncing inside a circular container
- Incorporate this into an animation
- Extend the code to include multiple balls
- Calculate the temperature and pressure
- Study the effects of varying the various parameters to deduce the basic laws of thermodynamics

Note that the animation is a useful tool for understanding the physics and, particularly, for debugging your program, but the main objective of the exercise is to write code to study a physical problem. There are no extra marks for unnecessarily fancy graphics.

Hard Spheres



Instead of considering more complicated inter-atomic interactions, such as the Lennard-Jones potential, we can confine ourselves to hard-spheres: balls which only interact when they collide with one another. We consider the collision to be perfectly elastic, so that energy and momentum are fully conserved. This calculation is very similar to what would be required to describe the game of snooker, billiards or pool. In addition we will consider the balls to be contained within a circular container. Hence the most important equation, describing the dynamics is

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

where $\mathbf{r}$, $\mathbf{v}$ and R are the position, velocity and radius of the balls, or the container, respectively. Make sure you understand this equation and what it means when

- (1) has no real solution for δt.
- (1) only has negative solutions.
- (1) has both a positive and a negative solution.
- what is the significance of the $\pm$ on the right-hand-side of (1)?

Task 1: Using this equation, calculate δt, the time until the next collision

Hint, let:

- $\mathbf{r} = \mathbf{r}_1 - \mathbf{r}_2$
- $\mathbf{v} = \mathbf{v}_1 - \mathbf{v}_2$
- $R = R_1 \pm R_2$
- Solve as you would any other quadratic equation

Now that you know the time to the next collision you must understand how to collide the balls. The elastic collisions of the balls were covered in 1st year mechanics, and probably in school.

To compute them, you will have to:

- Split the relative velocity, $\mathbf{v}_1 - \mathbf{v}_2$, into components parallel and perpendicular to the line between the centres of the 2 balls, $\mathbf{r}_1 - \mathbf{r}_2$.
- Solve for the parallel components after the collision just as you would for a collision in 1-dimension.
- The perpendicular component is unchanged by the collision.

Note: Note that the collision with the container is almost the same as that between 2 balls.

Task 2: You should define class `Ball` with attributes for the mass, radius, position

and velocity. I suggest you use *numpy arrays* for the latter two, as that will make it possible to do normal vector arithmetic, including the `dot` method of *numpy*. In addition you should define a patch attribute that will hold the circle primitive for the ball in your animation (see Appendix A: Matplotlib Graphics).

Your `Ball` class should at least have the following methods:

`pos`(*`self`*)
: return the current position of the centre of the ball.

`vel`(*`self`*)
: return the current velocity of the ball.

`move`(*`self, dt`*)
: move the ball to a new position $\mathbf{r}' = \mathbf{r} + \mathbf{v} * \mathrm{dt}$.

`time_to_collision`(*`self, other`*)
: calculate the time until the next collision between this ball and another one,or the container.

`collide`(*`self, other`*)
: make the changes to the velocities of the ball and the *other* one due to a collision.

You might find it useful to hide the position of the ball and to always use a method to access it. That way you can update the centre coordinate of the *patch* at the same time, to guarantee that it stays in sync with the position of the ball.

In addition to these attributes and methods you might want to include the time and the ball for the next collision, and a counter for each ball.

A Single Ball

As a first example we shall consider a single ball moving inside a circular container.

Task 3: Define a class `Simulation` which can be initialised with a container and a ball object.

Your `Simulation` class should have at least the following method:

`next_collision`(*`self`*)
: Sets up and performs the next collision.

The logic of the `next_collision` method should be as follows:

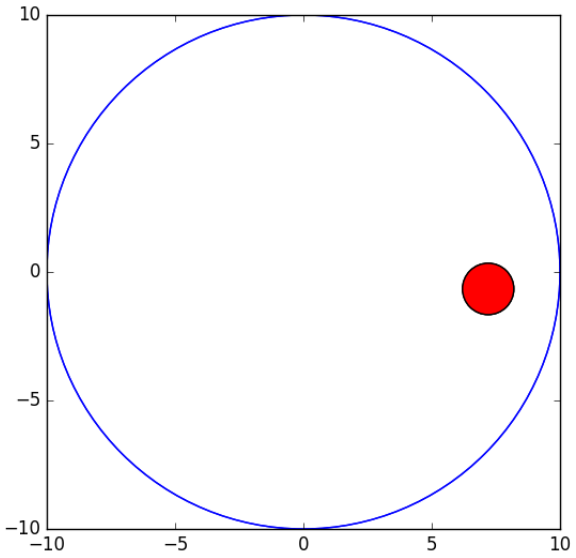
- find the time to the next collision
- move the system to that point in time
- perform the collision

Task 4: Create a `Simulation` object and initialise it with a container of radius 10. and a ball of radius 1. and mass 1. Test your `time_to_collision` and `collide` methods by locating the ball at a position [-5, 0] with velocity [1, 0]. Check that the time to collision and the resultant velocity after collision look correct after you run the `next_collision` method.

If you run the `next_collision` method again do you get what you would expect this time?

What about for a ball with velocity in both the x and y direction?

It can be useful (when debugging for example) to actually see the state of the system in a simple animation. We can set this up by drawing the ball and the container and updating it after each collision. It is not expected that you will run the animation when producing the final plots as it will slow things down. As mentioned before, it should be thought of as a debugging aid.



Note: *Before* attempting the task below, read over Appendix A: Matplotlib Graphics. Make sure you can get the example animation (at the end) to work on its own (i.e. separately from your project code) successfully, before attempting to add animation to your project.

Spyder needs some additional graphics settings in order for animations to play correctly. These are covered in Appendix B: Spyder Graphics Settings.

Task 5: Add the following `run` method to your `Simulation` class and try running an

animation and verify it looks sensible:

```
import pylab as pl

def run(self, num_frames, animate=False):
    if animate:
        f = pl.figure()
        ax = pl.axes(xlim=(-10, 10), ylim=(-10, 10))
        ax.add_artist(self._container.get_patch())
        ax.add_patch(self._ball.get_patch())
    for frame in range(num_frames):
        self.next_collision()
        if animate:
            pl.pause(0.001)
    if animate:
        pl.show()
```

- There is an assumption here that attributes *_container* and *_ball* contain the `Ball` objects representing the container and ball respectively. If you named these attributes differently then you will need to adjust the above code accordingly.
- The axes are hard coded to the radius of the container.

Task 6: Add code to check that the *kinetic energy* is being conserved and to calculate the *pressure* on the container due to the ball bouncing off it. Is there another conservation law obeyed by this system?

A Hard Sphere Gas

For a gas of several balls you will need to modify your `Simulation` class to accommodate not a single `Ball` any more, rather a list(or similar object) of `Ball` instances.

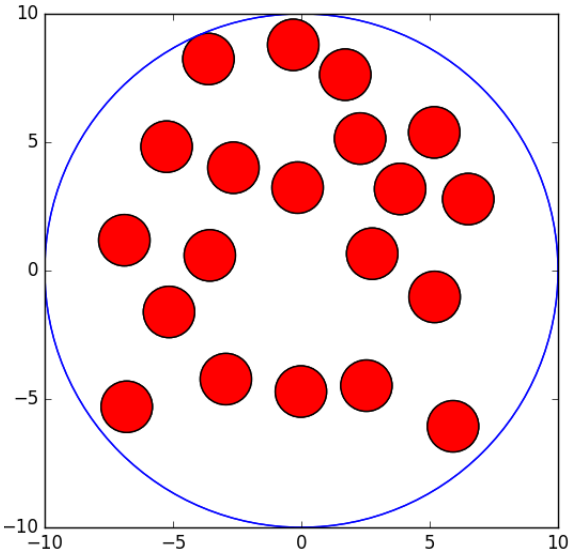
Task 7: Modify your `Simulation` class to handle multiple balls.

- When modifying your `next_collision` method bear in mind that in general you will only have to make changes which involve the 2 balls which have just collided, rather than rechecking every pair of balls. That is what makes the hard sphere calculation more efficient.

Now that we have the ability to handle multiple balls, we need a way of initialising the system with many balls without you having to manually specify each one. It should be obvious to you that this approach would become increasingly more laborious as the number of balls to fill with increases.

Task 8: Modify your `Simulation` class either in the initialisation or via a new meth-

od to provide a systematic way of initialising the system with a group of balls. The easiest way to do this is to arrange the positions of the balls systematically, but with random velocities (with average zero). That way it's easy to avoid starting with overlapping balls.



Try running your animation with many balls in the container. Run for a while and make sure that things look alright and you are not getting overlapping balls or balls escaping the container etc.

Task 9: Let's try and make sure things look right in a slightly more quantitative way. Run your simulation of many balls again, this time without the animation switched on. Make histograms of the distance each ball extends from the central position and of the distance between each pair of balls.

Do these plots look like you would expect them to?

At this stage, feel free to add your own checks to make sure the system is behaving as you expect it to.

Once everything looks good and you have the basic simulation running you should be in a position to do some physics.

Thermodynamics Investigations

Before using your simulation you should think about what can be done analytically:

Using your knowledge of physics

Task 10:

- How does the kinetic energy relate to the temperature? Consider a time other

than when 2 balls collide.

- Are there any conservation laws in addition to conservation of energy?
- If all the velocities are doubled:
    - What happens to the temperature?
    - What about the pressure?
    - How would the animation change?

Using your simulation

Task 11:

- Check the conservation law(s). Not particularly interesting plots, but useful as a test that your code is correct.
- Calculate the pressure, p on the wall of the container.
    - Does this depend on T as expected.?
- How do p and T change when
    - you change the volume (area) of the container.
    - you change the number of balls in the container.

Task 12:

- Compare your results to the ideal gas law $pV = Nk_{\mathrm{B}}T$.
    - How do your results depend on the size of the balls in the simulation?

Task 13: Compare the distribution of velocities to the Maxwell-Boltzmann distribution.

- The variance of the distribution is one useful measure.
- Use the data to build the distribution as a histogram.

Task 14: consider the van der Waals' law

$$\left(P + a\left(\frac{N^2}{V^2}\right)\right)(V - Nb) = Nk_{\mathrm{B}}T?$$

- Can you identify a and b?
- One of these can be deduced analytically.
- The other can be deduced graphically.

Stop 1: At this point if you have finished everything talk to a demonstrator about possible extensions.

List of plots for Project B

Please submit the following plots for assessment, appended to your 2-page summary report.

Fig. No.	Task	Description
1.	T-9	Histogram of ball distance from container centre.
2.	T-9	Histogram of inter-ball separation.
3.	T-11	System kinetic energy versus time
4.	T-11	Momentum versus time.
5.	T-11	Pressure versus temperature.
6.	T-12	A plot to illustrate how the changing ball radius affects the equation of state.
7.	T-13	Histogram of ball speeds & comparison to theoretical prediction.
8.	T-14	Plot of data used to fit to van der Waal's law.

You can submit additional plots at your discretion.

Appendix A: Matplotlib Graphics

From the Matplotlib Artist tutorial:

> There are three layers to the matplotlib API. The matplotlib.backend_bases.FigureCanvas is the area onto which the figure is drawn, the matplotlib.backend_bases.Renderer is the object which knows how to draw on the FigureCanvas, and the matplotlib.artist.Artist is the object that knows how to use a renderer to paint onto the canvas. The FigureCanvas and Renderer handle all the details of talking to user interface toolkits like wxPython or drawing languages like PostScript®, and the Artist handles all the high level constructs like representing and laying out the figure, text, and lines. The typical user will spend 95% of their time working with the Artists.
>
> There are two types of Artists: primitives and containers. The primitives represent the standard graphical objects we want to paint onto our canvas: Line2D, Rectangle, Text, AxesImage, etc., and the containers are places to put them (Axis, Axes and Figure). The standard use is to create a Figure

instance, use the Figure to create one or more Axes or Subplot instances, and use the Axes instance helper methods to create the primitives.

Here is an example of drawing a filled in circle:

```
import pylab as pl

f = pl.figure()
patch1 = pl.Circle([0., 0.], 4, fc='r')
patch2 = pl.Circle([5., 2.], 4, fc='b')
ax = pl.axes(xlim=(-10, 10), ylim=(-10, 10))
ax.add_patch(patch1)
ax.add_patch(patch2)
pl.show()
```

Note: *pl.Circle()* returns a circular patch object. To add the Circle primitive to the Axes we use the add_patch method. These instances are stored in variables Figure.patches and Axes.patches ("Patch" is a name inherited from MATLAB, and is a 2D "patch" of colour on the figure, e.g., rectangles, circles and polygons)

We can see the circle within the axes Artist by looking at this patches attribute:

```
>>> ax.patches
[<matplotlib.patches.Circle object at 0x7ff9bc9e73c8>]
```

For completeness here is an example of drawing a circle outline:

```
import pylab as pl

f = pl.figure()
patch = pl.Circle([0., 0.], 4, ec='b', fill=False, ls='solid')
ax = pl.axes(xlim=(-10, 10), ylim=(-10, 10))
ax.add_patch(patch)
pl.show()
```

We can move the circular patch object by updating it's *center* attribute with new co-ordinates and then re-drawing the canvas. An example of this is given below:

```
import pylab as pl

f = pl.figure()
patch = pl.Circle([-4., -4.], 3, fc='r')
ax = pl.axes(xlim=(-10, 10), ylim=(-10, 10))
ax.add_patch(patch)

pl.pause(1)
patch.center = [4, 4]
pl.pause(1)
pl.show()
```

Note: In the above example *pl.pause()* will redraw the canvas and then pause for

the given length of time (in this case 1 second) before proceeding. *pl.pause()* is necessary for two reasons. Firstly to trigger the redrawing of the canvas but secondly to give you time to see the circle patch in the first position. Using *pl.show()* is not sufficient as it will wait (block) until you close the figure window.

By repeatedly moving and redrawing the patch in a loop, we can create an animation as in the example below:

```
import pylab as pl

f = pl.figure()
patch = pl.Circle([-10., -10.], 1, fc='r')
ax = pl.axes(xlim=(-10, 10), ylim=(-10, 10))
ax.add_patch(patch)

for i in range(-10, 10):
    patch.center = [i, i]
    pl.pause(0.001)
pl.show()
```

Appendix B: Spyder Graphics Settings

For the animation to work properly you will need to use the correct graphics backend. The one you need might depend on the operating system your laptop/computer is using, and even the version of Spyder.

Spyder can render graphics either in '*inline*' in the IDE itself – in the Plot browser (or directly in the IPython console, for older versions) – or in a separate window. Animations need the latter method to play properly.

You can select inline or windowed plots by typing the IPython magic command `%matplotlib <mode>` in the console:

- `%matplotlib inline` – for rendering in the plot browser.
- `%matplotlib auto` – for rendering in a window. This mode enables animations to play.

Use shouldn't need to restart the kernel when changing graphics mode this way. If you are lucky, animations work with `%matplotlib auto`. In some cases it doesn't and you just see the first frame. In this case you will need to change the graphics backend.

Changing the Graphics backend

This needs to be done via Spyder's preferences. From the menu bar

- **Windows 10**: Tools -> Preferences -> IPython console -> graphics -> graphics

backend
- **macOS**: python -> Preferences -> IPython console -> graphics -> graphics backend

Known Settings

- **UG PCs** (Windows 10 + Spyder 3.3.1): use *Tkinter*.
- **Windows 10** + Spyder 4.1.4: use *Automatic* or *Tkinter*.
- **macOS** + Spyder 4.1.4: use *Automatic* or *OSX*.

Then the python kernel needs to be reset, by right clicking on your IPython console tab and selecting restart kernel or using a keyboard shortcut:

- **Windows 10**: Ctrl+.
- **macOS**: ⌘+.

> **Note:** If the suggested settings above do not work in your case, experiment with other settings to see if they do work.

Appendix C: The Maxwell-Boltzmann distribution

The Maxwell-Boltzmann distribution, as you have probably learned it, takes the form

$$p(v) \propto v^2 \exp\left(-\frac{\frac{1}{2}mv^2}{k_\mathrm{B}T}\right) \tag{2}$$

Mathematically this is just the radial part, expressed in spherical polars, of a distribution of the 3 Cartesian components of velocity

$$p(v_x, v_y, v_z) \propto \exp\left(-\frac{\frac{1}{2}m(v_x^2 + v_y^2 + v_z^2)}{k_\mathrm{B}T}\right) \tag{3}$$

In 2D the appropriate form for the Maxwell-Boltzmann distribution is given by the radial part in (cylindrical) polar coordinates, which takes the form

$$p(v) \propto v \exp\left(-\frac{\frac{1}{2}mv^2}{k_\mathrm{B}T}\right) \tag{4}$$

In general, in *d*-dimensions the initial factor take the form v^{d-1}.

Document history

Created: Sep 2016 - A MacKinnon.

Revised: Oct 2020 - R Kingham.