The Ideal Gas Simulator

October 2023

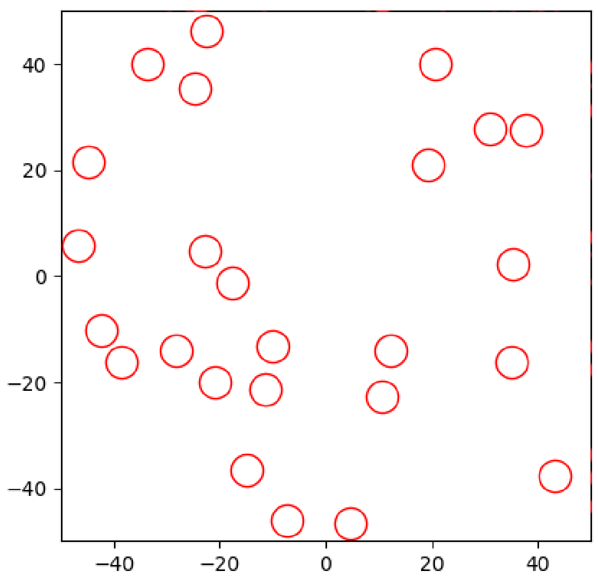

Figure 1: Screenshot of the ideal gas simulation.

Introduction

In this assignment you simulate an ”ideal” gas. You will dynamically visualize the simulation with matplotlib’s animation functions. With a working simulator you can perform all kind of ”experiments”. You will do two experiments with your simulator. In the assignments below you build your simulator step-by-step. It is advised to consult with the supervisors after each step to make sure you are on the right path.

Setting up the animation

Initializing variables. We will simulate N molecules moving in a 2D box of size `box_size = 100`. Each molecule will be represented by a position `r` (a vector composed of the x and y coordinate) and a velocity vector `v`. We take the radii and masses of all molecules equal: `radius` (a scalar) and `mass` (a scalar). Add code to your script that defines the position and velocity of the molecules. Define the mass and radius separately. Use the following data structure: Define a 2D numpy array `r`, in which there are N number of rows (one for each molecule) and 2 columns (for the x- and y-coordinate of the molecule). In a similar fashion define a 2D numpy array `v` that defines the velocity for each molecule. Finally define two scalars `mass` and `radius` that store the mass and radius. Fill the arrays with the following initial data:

We start with defining just three molecules ($N = 3$). Take as a starting position "[0, 0]", "[10, 10]", "[-5, 5]". Take as a starting velocity "[1, 2]", "[-1, 2]" and "[0.5, -1.5]". For the radius you can take a value of 3 and set the mass to 1.

Setting up the figure. For drawing the molecules we will make use of Matplotlib's `ax.plot()` command and using the `o` marker to draw the "molecules". Create the figure and plot the molecules. Adapt the axis limits such that it represents the size of the box. In order to make sure the x-axis and y-axis are also drawn with equal lengths (so it looks like a square on your screen) you properly set the aspect ratio of your plot:

```
fig.gca().set_aspect('equal', 'box')
```

Setting the marker size. As you probably noticed, the size of the marker does not correspond to the real size (value of `radius`) of the molecule in axis-units. The reason is that if you set the marker size to e.g. 3 (e.g. using the keyword argument `ax.plot( , , ms=3)` the size is specified in points. In matplotlib there are 72 points per inch. To actually calculate the required markersize is illustrated in the code snippet below:

```
# get required markersize to draw marker in axis units.
# fig is the reference to the figure.
# box_size is the axis range in axis units
ms = 2 * radius * fig.gca().get_window_extent().height / box_size * 72./fig.dpi

# set the markersize of the plot using the ms keyword argument.
ax.plot( , , ms=ms)
```

When running your script you should now see the three "molecules" as three dots in your plot, with the size corresponding to the radius as specified.

Moving/Animating the molecules

Implement the animation. You will now setup the animation part. For this we make use of the matplotlib `FuncAnimation()` function. You can find a lot of info on how this function works (see e.g. website of matplotlib or other online resources). To get you started example code is show below. In this example a line plot is animated. TIP: try this code and try to understand how it works. Implement the animation in your script such that the "molecules" start moving. Update the positions of the molecules using the Euler method. Use a time step `dt=0.05` to start with. Your molecules should now move when you run the script. However they are not contained in the box. You will implement this in the next step.

```python
"""
example of animation in matplotlib.
loosly based on an example on the website matplotlib.org
"""

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

# create a figure and axes
fig, ax = plt.subplots()

# initialize the data for plotting
t = 0
x = np.arange(0, 10*np.pi, 0.01)
y = np.sin(x - t)

# create the plot and store the reference to a variable
line, = ax.plot(x, y)


def animate(i):
    """this function updates the plot"""
    global t  # this statement is used so the global t is changed

    t = t + 0.1  # increase the time step
    y = np.sin((x - t)) # compute the new function values
    line.set_data(x, y)  # update the data of the line

    return line,

# start the animation
ani = animation.FuncAnimation(fig, animate, frames=1000, repeat=False, interval=10, blit=True)

plt.show()
```

Keeping the molecules in the box.

You will now see that the molecules are leaving the plot window. That is a bit of a pity, so let's try to keep the objects inside the box. A very simple way of avoiding objects leaving the box is to flip the velocity component of the molecule that is leaving the box. For example, when the circle is leaving the box on the right it has a positive x-velocity. To prevent it from leaving the box you have to flip the velocity to negative x-velocity as soon as it touches the wall of the box.

Keeping the molecules in the box. For this assignment you have to prevent the molecules from leaving the box. Make a function called `wall_collisions(r, v, radius, box_size)` which returns the updated velocities of the molecules to make sure that they can never leave the box. Hint: You have two parts to this function, determining which objects are about to leave the box, and the actual flipping of the relevant velocity component.

Increasing the number of molecules in the box. So far you used `N=3` (three molecules). Adapt the script for initialization of the positions and velocities of N molecules using random generators (pick some sensible range for the position and velocity). Make sure all the molecules are located within the box and they are initialized correctly. You can ignore overlapping molecules, as we will fix this later. Test it with e.g. $N = 25$

Speeding it up. You may have noticed that increasing the molecules will slow down the simulation. Increasing the time step in the Euler method can speed it up, but may also result in inaccurate results. As plotting the molecules takes up a lot of time it is also a good idea not to plot the molecules every Euler step. Adapt your script such that in every call to the `animate` function multiple Euler steps are performed.

Adding molecule collisions

Resolve collision between molecules. Resolve collisions between objects using the elastic collision equations below. Implement this in a function `resolve_collisions(r, v, radius)` that updates and returns the velocities of all molecules caused by collisions.

Note: You have to check all possible pairs of molecules, but not *double* (since comparing molecule 1 and molecule 2 is the same as molecule 2 and molecule 1)! First determine if the two objects are supposed to collide, which has two criteria: They must be touching, and they must be moving towards each other (see info below).

The equations to compute the velocities are given:

"In an angle-free representation, the velocities $\vec{v}_1'$ and $\vec{v}_2'$ after collision are computed using the centers $\vec{r}_1$ and $\vec{r}_2$ and velocities $\vec{v}_1$ and $\vec{v}_2$ at the time of contact as" (adapted from Wikipedia) [1]

$$\vec{v}_1' = \vec{v}_1 - \frac{2m_2}{m_1+m_2}\frac{(\vec{v}_1-\vec{v}_2)\cdot(\vec{r}_1-\vec{r}_2)}{\|\vec{r}_1-\vec{r}_2\|^2}(\vec{r}_1-\vec{r}_2)$$
$$\vec{v}_2' = \vec{v}_2 - \frac{2m_1}{m_1+m_2}\frac{(\vec{v}_2-\vec{v}_1)\cdot(\vec{r}_2-\vec{r}_1)}{\|\vec{r}_2-\vec{r}_1\|^2}(\vec{r}_2-\vec{r}_1)$$

These two equations will give the new velocity vectors due to an elastic collision between two molecules. A few remarks on these equations:

- The double enclosing vertical bars $\|\vec{a}\|$ are used to indicate the L_2 norm (or Euclidean length) of the vector.
- Since the masses of our molecules are all identical, the fractions with the masses are equal to 1.
- The $\cdot$ in the equations refers to the dot product of two vectors. In Python you can use `np.dot()` or the special `@` symbol to compute the dot product of two vectors.

Furthermore, you can check if two objects are moving towards each using the following relation. If this relation is true, the molecules are moving towards each other:

$$(\vec{r}_1-\vec{r}_2)\cdot(\vec{v}_1-\vec{v}_2) < 0 \tag{1}$$

[1]The position vectors $\vec{r}$ correspond to the rows in your variable r.

Monitor the average energy of the system. If your script works correctly then the total energy of the system should remain constant. All collisions between molecules and between molecules and the walls are elastic and thus there is no energy lost. This is a behaviour you can easily check. Adapt your script such that it keeps track of the total (kinetic) energy of the system. Does it remain constant?

Speeding it up (optional). As you may have noticed the script can become quite slow with increasing number of molecules. The reason is that you have to check all possible pairs of molecules if they collide. To speed this up you can (if you haven't already) make use of dedicated numpy/scipy functions. Although they perform the same computations as your script does, it uses compiled `C` code which executes much faster. The relevant functions are `pdist()` and `squareform()`, both from the `scipy.spatial.distance` module. You can use these functions to compute a distance table and quickly find all pairs of molecules that are touching. Check out the numpy/scipy documentation on how these functions work. How can you make use of these functions to select only pairs of molecules that are touching?

Experiments

Experiment 1: Diffusion Even without particle-particle interactions you should see the effects of diffusion. You could try to see how two species of molecules "mix" over time if they are initially separated. Setup a system where you have 40 molecules on the left randomly distributed with the same speed (but random direction), and 40 molecules on the right. You can visualize this by using different colors. Determine after how many iterations the system appears to be mixed (qualitatively). If you have time left you can try to calculate how well they are mixed (i.e. count how many red are on the left and on the right etc.). Do the calculations in Python and not by eye!

Experiment 2: Boltzmann distribution With particle-particle interaction enabled you have a system where particles can exchange energy with each other (without loss). According to theory, the distribution of the speed of the molecules should then follow the Maxwell-Boltzmann distribution. This is motivated by the fact that this distribution maximizes the entropy of the system. The Maxwell-Boltzmann distribution $f(s)$ (for 2D) is given by:

$$f(s) = \frac{m}{E_{av}} s \exp\left(-\frac{\frac{1}{2}ms^2}{E_{av}}\right)$$

, with m the mass of the molecule, s the speed, and E_{av} the average kinetic energy of all molecules in the box:

$$E_{av} = \frac{1}{N}\sum_{i=1}^{N} \frac{1}{2} m s_i^2$$

, with s_i the speed of molecule i: $s_i^2 = v_{i,x}^2 + v_{i,y}^2$.

Note: For a 2D gas the average energy is also equal to $E_{av} = k_B T$, where k_B is Boltzmann's constant and T is the thermodynamic temperature of the gas.

Using your simulator, make a histogram of the speeds of the molecules and compare it with the Maxwell-Boltzmann distribution. Try a few different starting distributions (e.g. all molecules the same speed; a uniform distribution of speeds) and check the distributions after some time.