Python Project - BoxParti

Tomas Pereira de Vasconcelos

1 Introduction

BoxParti is a computer program that allows users to visualise a simulation based on an ideal gas model. The program uses the pre-installed *Python* module `Tkinter` [1] as a *Graphical User Interface (GUI)*. This makes it very easy to display excellent plots and animations while keeping the program user-friendly and interactive. BoxParti can run on Macintosh, Windows and most Unix machines that have Python 2.7 [2] installed. Further installation of the most up-to-date versions of `matplotlib` [3], `scipy` and `numpy` [4] modules may also be required for the script to run with no errors.

In Section 2, we will go over a quick theoretical introduction. Starting from a *microscopic kinetic theory of gases*, we will derive expressions for *pressure* and *kinetic energy* (macroscopic properties) as a function of microscopic parameters of a model. Furthermore, we will see how Maxwell-Boltzmann's speed distribution function develops with changes in temperature and particle mass. In Section 3 we will go over the basic functionalities of the program, how initial conditions (the position and velocity of every particle) are set and how they are updated for every animation *frame*.

2 Theoretical Introduction

To create a simulation of a monoatomic gas, a simple model was derived from a microscopic kinetic theory of gases. From this theory, explicit assumptions and physical characteristics of the system were taken into consideration. The atoms/molecules were considered to be all identical in mass and shape and treated as *hard spheres*. In a hard sphere model, collisions between particles are considered to be *perfectly elastic*, with no loss of kinetic energy. The same assumption was made for collisions of particles with the walls of the *box* they are contained in. The particles move freely inside a container; however, they must follow Newton's laws of motion as well as the principles of conservation of energy and momentum. For simplicity, all intermolecular attractive forces were not taken into consideration. Gravity is also ignored; therefore, all particles are said to follow straight-line paths. With these postulates, we say that, in this system, all internal energy is in the form of kinetic energy. The model could become more and more sophisticated by considering other physical characteristics of the system.

2.1 State Variables

The macroscopic description of this system is provided by three state variables. The relationship between them is given by the ideal gas law in the form of

$$PV = Nk_BT \tag{1}$$

where P is the absolute pressure, V is the volume where the gas is contained, T is the absolute temperature, N is the number of particles and k_B is Boltzmann's constant ($k_B = 1.38 \times 10^{-23} J/K$).

The kinetic theory of gases describes the microscopic behaviour of gases. This theory provides predictions for macroscopic properties, like pressure and temperature, in terms of parameters of a particular model. In the case of a gas inside a container, pressure would arise from the many collisions of the particles with the walls. The velocity vector $\vec{v}$ of a particle is given by $\vec{v} = (v_x, v_y, v_z)$, where v_x, v_y and v_z are the velocity components of the x, y and z directions. For a particle with some mass m colliding with a wall (consider the surface normal along the x-axis), the component of the velocity normal to the wall (v_x) is reversed after the collision. This causes a change in the momentum p in the x-component of $\Delta p_x = 2mv_x$. If we

neglect collisions between particles, the time between collisions with the wall (Δt) is given by $\Delta t = 2L/v_x$, where L is the distance between the two walls normal to the x-axis. Using simple newtonian mechanics, it can be shown that the average force $\overline{F_x}$ of N particles colliding with a wall is given by

$$\overline{F_x} = \frac{\Delta p_x}{\Delta t} = N\frac{m}{L}\langle v_x^2 \rangle \tag{2}$$

where $\langle v_x^2 \rangle$ is the mean of the squared x-component of the velocity. Starting from the definition of pressure ($P = F/A$, where $A = L^2$ for a square), if we consider a cube with sides of length L and volume V, the pressure is then given by

$$P = \frac{\overline{F_x}}{L^2} = N\frac{m}{V}\langle v_x^2 \rangle \tag{3}$$

Assuming that all particles have random speeds in all x, y and z directions, so that

$$\langle v_x \rangle = \langle v_y \rangle = \langle v_z \rangle \tag{4}$$

and

$$\|v\| = \sqrt{v_x + v_y + v_z} \tag{5}$$

the mean squared velocity can be expressed as

$$\langle v^2 \rangle = \langle v_x^2 \rangle + \langle v_y^2 \rangle + \langle v_z^2 \rangle \tag{6}$$

With this relationship, we can express pressure as a function of the mean squared velocity. At last, we have a link between a measurable macroscopic property (pressure) and parameters of a microscopic model. This relationship is given by the equation

$$P = \frac{1}{3}N\frac{m}{V}\langle v^2 \rangle \tag{7}$$

where P is usually referred to as *kinetic pressure*.

2.2 Maxwell-Boltzmann Speed Distribution

The speed distribution of particles of an ideal gas is given by Maxwell-Boltzmann's speed distribution function

$$f(v) = \left(\frac{m}{2\pi k_B T}\right)^{3/2} (4\pi v^2) exp\left(\frac{-mv^2}{2k_B T}\right) \tag{8}$$

The exponential factor $exp(-mv^2/2k_BT)$ is Boltzmann's factor $exp(-\epsilon/k_BT)$, where ϵ is the energy state occupied by the particle. For this model, we consider ϵ to be the kinetic energy ($\epsilon = \frac{1}{2}mv^2$). Particles in a gas undergo random collisions with each other, resulting in an arbitrary transference of kinetic energy. Nevertheless, on average and for a vast number of particles, the ratio of particles which have a certain velocity v is constant.

For any values of temperature and mass, the area under the curve remains constant. Nevertheless, the particles' mass and temperature affects the shape of this distribution. For a specific gas (with constant mass), as temperatures increase, the mean speed also increases and the distribution widens. The same behaviour happens with lighter particles, at a constant temperature. This behaviour is represented in Figure 1, comparing different gasses (with different masses) at two different temperatures.

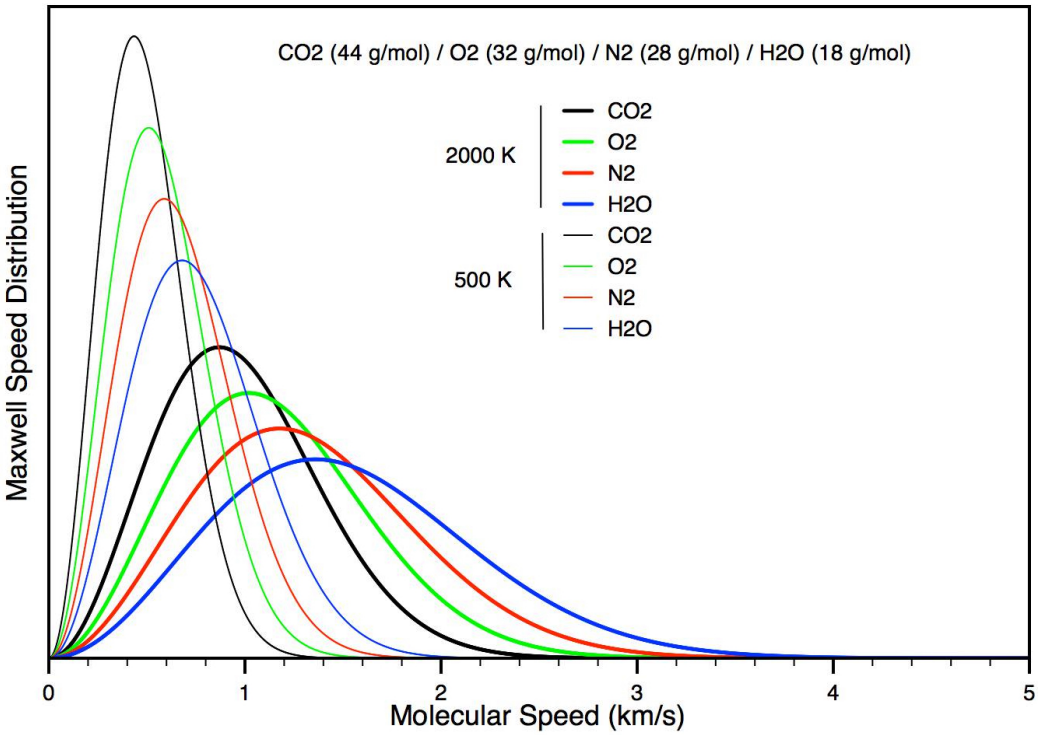
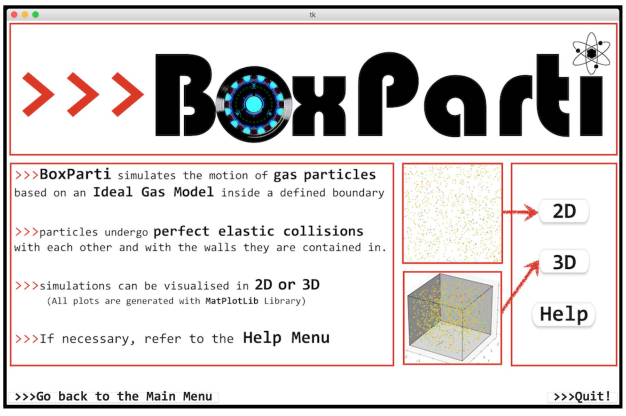


Figure 1: Maxwell-Boltzmann's Speed Distribution for different gases at different temperatures.

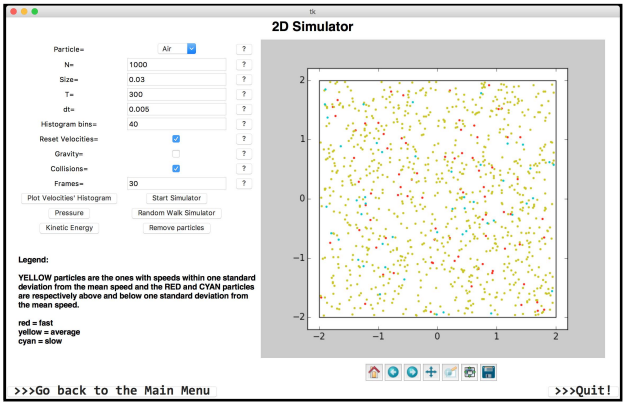
3 The Program

BoxParti greets the user with a *Welcome Window* (see Figure 2[a]), which provides some brief information about the program and allows the user to choose between a two-dimensional (2D) or three-dimensional (3D) simulation. If new to the program or having trouble getting the desired results, the user also has access to a *Help Menu* (see Figure 2[b]). By clicking on the 2D or 3D button, the user is directed to the respective *Simulation Window* (see Figure 2[c,d]). Here, on the left-hand side, the user is presented with a broad range of options that affect the output of the simulation. Each option has a designated *help button*, which opens a pop-up window with a short description of the option, some tips on how to use it and a recommended range of values. After choosing the desired initial values, there are three options: *start the simulator*, *plot the velocities' histogram* or *start random walk simulator*. After one of these is chosen and the animation runs for the selected amount of frames, readings of pressure and translational kinetic energy can also be obtained.

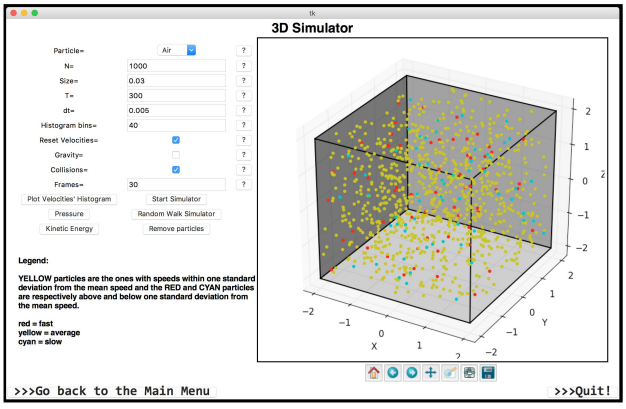


(a) Welcome Window

(b) Help Menu (under development)



(c) 2D Simulator



(d) 3D Simulator

Figure 2: The four different windows the user has access to in BoxParti.

3.1 Initial Conditions

To start the simulation, we first need a set of values with the initial coordinates of the particles and their initial velocities. The initial coordinates of N particles can be assign at random using a `numpy.random.random((N,d))` routine, where d is the number of dimensions (d=2 for two dimensions and d=3 for three dimensions). This returns a *numpy array* with d columns and N rows. Further adjustments might be necessary, depending on the dimensions of the box.

We have previously seen that the speed distribution of gas particles follow a Maxwell-Boltzmann distribution function. A *Monte Carlo* simulation can be set up to return speeds for the x, y and z directions. The most general way of doing this is by using the *Cumulative Distribution Function (CDF)* $C(v)$ given by

$$C(v) = \text{erf}\left(\frac{v}{\sqrt{2k_BT/m}}\right) - \sqrt{\frac{2}{\pi}}\left(\frac{v\exp(-mv^2/2k_BT)}{\sqrt{k_BT/m}}\right) \tag{9}$$

where erf is the error function (here we will use `scipy`'s special function `scipy.special.erf`). We can take a set of random numbers, uniformly distributed between 0 and 1, and use the inverse of the CDF to generate random initial speeds. There is no analytic version of the inverse of the CDF, but we can do it numerically with `scipy`'s interpolation function `scipy.interpolate.interp1d`.

After generating values for the coordinates and velocities of N particles, we store these values in a new numpy array. This saves a lot of computation time afterwards. Here is an example of how the system's initial-state array (`StateArray`) would look like for N particles:

$$\texttt{StateArray} = \begin{bmatrix} x_1, & y_1, & z_1, & v_{x1}, & v_{y_1}, & v_{z1} \\ x_2, & y_2, & z_2, & v_{x2}, & v_{y_2}, & v_{z2} \\ x_3, & y_3, & z_3, & v_{x3}, & v_{y3}, & v_{z3} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_N, & y_N, & z_N, & v_{xN}, & v_{y_N}, & v_{zN} \end{bmatrix} \begin{matrix} \\ \\ \\ \\ \\ \end{matrix} \begin{matrix} \rightarrow \text{particle } 1 \\ \rightarrow \text{particle } 2 \\ \rightarrow \text{particle } 3 \\ \ldots\ldots\ldots\ldots \\ \rightarrow \text{particle } N \end{matrix}$$

where x, y and z are spatial coordinates and v_x, v_y, v_z are the x, y and z components of the velocity vector.

3.2 Updating Positions

The position of each particle is updated for every time-step (dt) using Newton's Laws of motion. Since the particles move in linear trajectories, *Euler's numerical method* to solve differential equations is good enough for this model. Knowing the initial coordinates and velocities of each particle, we can compute the future coordinates $x(t+dt)$, $y(t+dt)$ and $z(t+dt)$ with

$$\begin{aligned} x(t+dt) &= x(t) + v_x(t)dt \\ y(t+dt) &= y(t) + v_y(t)dt \\ z(t+dt) &= z(t) + v_z(t)dt \end{aligned} \tag{10}$$

which, in *Python*, is equivalent to:

```
1 StateArray[:, :3] +=  StateArray[:, 3:] * dt
```

To consider collisions with walls and other particles, these equations alone are not enough to describe the motion of the particles. As an example, lets consider the case for when a particle hits the wall parallel to the surface normal along the x-axis. In this case, the component of the velocity normal to the wall is reversed after the collision. This is given by

$$v_x(t+dt) = -v_x(t) \tag{11}$$

where $v_x(t+dt)$ is the updated velocity after the collision. An identical operation done on a *numpy array* is given by

```
1 #Wall_x_Coordinate is the abscissa coordinate of the wall's boundary
2 test = ( StateArray[:, 0] - ParticleRadius < Wall_x_Coordinate )
3 StateArray[test, 0] = StateArray[0] + ParticleSize
4 StateArray[test, 3] *= -1
```

Here we take into consideration the radius of the particle (`ParticleRadius`). In line 2, for all N particles in the system, we are running a *test* to see if they have passed beyond the wall's boundary. This returns a *list* of *booleans* (`True` or `False`), which are stored in the `test` list. In line 3, for all values where `test` is `True`, particles are relocated at the boundary. And finally, in line 4, the velocity is reversed. BoxParti also has the option of introducing gravity and particle-collisions into the simulation. This can be accomplished with similar routines, by applying Newton's Laws of Motion and *Kinematics*.

3.3 Building the Animation

To animate the simulation, we can use Python's module `matplotlib` to plot the initial positions of N particles. The initial positions are obtained from the `StateArray` array. This would then be the 1^{st} *frame* of the animation. After, using the routines in Section 3.2, we will need to run over some `update()` function to get our updated coordinates and plot them again (2^{nd} frame). If we do this over a *loop* using `matplotlib`'s function `matplotlib.animation.FuncAnimation` we get an animation! To improve the user's experience, BoxParti runs a particular routine before every frame, which allows it to detect and store the speed of each particle. The particles can then be plotted with particular colours, depending on their speed. The following routine exemplifies how this could be achieved

```
# N = Number of Particles
for v in range(0, N):
    if Speed[v] > 1StandardDeviation:
        FastParticles_x.append(StateArray[v, 0])
        FastParticles_y.append(StateArray[v, 1])
        FastParticles_z.append(StateArray[v, 2])
    elif Velocity[v] < 1StandardDeviation::
        SlowParticles_x.append(StateArray[v, 0])
        SlowParticles_y.append(StateArray[v, 1])
        SlowParticles_z.append(StateArray[v, 2])
    else:
        MeanParticles_x.append(StateArray[v, 0])
        MeanParticles_y.append(StateArray[v, 1])
        MeanParticles_z.append(StateArray[v, 2])

# plot = <function matplotlib.pyplot.plot>
plot(FastParticles_x, FastParticles_y, FastParticles_z, 'red',
     MeanParticles_x, MeanParticles_y, MeanParticles_z, 'yellow',
     SlowParticles_x, SlowParticles_y, SlowParticles_z, 'cyan'
     )
```

where the *Mean-Particles* are the ones with speeds within one standard deviation of the mean speed and the *Fast* and *Slow* particles are respectively above and below one standard deviation from the mean speed.

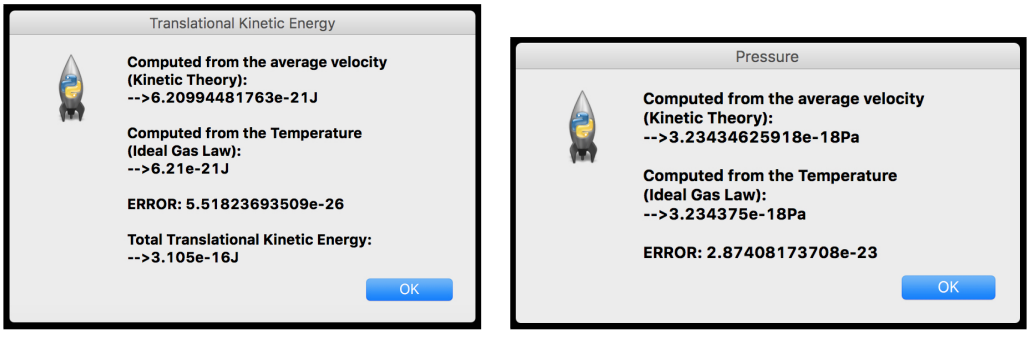
The *random-walk simulator* in BoxParti uses the same functions as the 'normal' simulator but only plots one particle for every time-step. To observe a random walk, the *collisions* option should be turned 'ON'. To produce the 'trace effect' in the random-walk simulator, for every frame, the program saves the position of the particle in a numpy array. By doing this, `matplotlib` can plot the particle's previous positions and connect them with a thin line.

3.4 Making Virtual Measurements

Before making any measurements of pressure and kinetic energy, the user needs to run a simulation at least once for a minimum duration of one frame. This happens because, before the start of the first simulation, the program has not yet generated any values for the coordinates and velocities of the particles. So, when it tries to operate on an empty *list*, Python returns a `NaN` float object, where `NaN` stands for 'Not a Number'. This would always return `NaN` for the value of pressure and kinetic energy. BoxParti opens a Warning! message in the form of a pop-up window if the user does not run the simulation before clicking on the *pressure* or *kinetic energy* buttons.

In Section 2.1 we saw that pressure can be expressed either as a function of temperature (Equation 1) or as a function of the average squared velocity of a particle (Equation 7). A good way of analysing our system and test if it is behaving the way it should is to check if readings of pressure and kinetic energy are the same if they are computed as functions of temperature or particle velocity. BoxParti successfully makes these measurements for two and three dimensions with a very small error. The error, naturally, decreases if we increase the number of particles in our system. At anytime during the simulation the user can click on one of the *pressure* or *kinetic energy* buttons, where the reading for this will show up in the form of a pop-window (see example in Figure 3[a,b]) and without disturbing the animation.

BoxParti also has the capability of plotting a real-time animation of the particles' speed distribution in the form of a histogram. This is easily done using `matplotlib`'s function `matplotlib.pyplot.hist`. If we combine this with `scipy`'s random variable `scipy.stats.maxwell` we can plot the associated continuous distribution on top of the histogram.



(a) Kinetic Energy measurements (b) Pressure measurements

Figure 3: Measurements of pressure and kinetic energy appear in the form of pop-up windows.

4 Conclusion and Future Work

This Python simulation of a gas made very accurate and consistent measurements of macroscopic properties as a function of microscopic parameters. This is a good example where statistical physics gives accurate and steady results for macroscopic manifestations of microscopic behaviour. From this, very precise calculations of pressure, volume and temperature can be made using the ideal gas law. Measurements of the average kinetic energy associated with the atomic motion has its groundwork in a statistical method, the Maxwell-Boltzmann speed distribution function. Again, measurements of temperature and energy can be made very precisely.

BoxParti's source code is organised in a way that minimises computation time and maximises the user's experience. This is done by defining specific functions for specific tasks, avoiding unnecessary calculations being made in the background. Nevertheless, there is always room for improvement and, in the future, I'll be focussing on implementing inelastic collisions. If time allows it, I would like to adapt the current script and use Python's `visual` [5] module, which provides much better 3D animations than `matplotlib`. The `visual` module also provides functionalities like *zooming in* and out of the system or 'move around' in the system. The Help Menu window (see Figure 2[b]) is still under development, but it includes a button that links to this report. A *YouTube* link to a small example video showing the functionalities of the program is also included.

References

[1] Tkinter `https://wiki.python.org/moin/TkInter`

[2] Python 2.7 Release `https://www.python.org/download/releases/2.7/`

[3] matplotlib `http://matplotlib.org/downloads.html`

[4] Obtaining NumPy and SciPy libraries `http://www.scipy.org/scipylib/download.html`

[5] VPython `http://vpython.org`