SIMULATING THE EVOLUTION OF THE VELOCITY DISTRIBUTION IN AN IDEAL GAS

Damon Falck

January 8, 2017

1 INTRODUCTION

In an ideal gas, particles collide elastically with one another and the walls of the container, exchanging kinetic energy and momentum but not interacting in any other way. Any attractions or forces between the molecules (such as van der Waals forces) are ignored, and indeed can change the behaviour of the particles dramatically.

Because of the constant collisions between particles, the probability distribution of particle speeds varies over time, tending towards and reaching equilibrium at the Maxwell-Boltzmann speed distribution,

$$f(v) = 4\pi \left(\frac{m}{2\pi k_B T} \right)^{\frac{3}{2}} v^2 \mathrm{e}^{-\frac{mv^2}{2k_B T}}$$

where f is the probability of a given particle having speed v.

While the speeds of the eventually stabilised system can be described by this probability density function, it is the object of this simulation to numerically model and examine the evolution over time of this distribution of velocities.

To do so requires an accurate simulation of the particle interactions in an ideal gas.

2 IMPLEMENTATION

The simulation was written in Matlab R2015b. There are three main parts to the code:

1. Setup of the initial conditions, including modelling the container and particle properties.
2. Detecting collisions with the walls of the container and changing the particle velocities accordingly.
3. Detecting collisions between particles and changing the particle velocities accordingly.

After every iteration of parts 2 and 3 the position of each particle is updated, and the code loops. Finally when the specified number of iterations has been completed, the program halts and plots can be made from the data.

2.1 INITIAL CONDITIONS

For simplicity, the simulation runs within a cube of volume specified by the ideal gas equation. Every particle is given the same initial speed and a random direction.

2.1.1 PARAMETERS

The following physical parameters are specified by the user:

- Pressure P
- Thermodynamic temperature T
- Particle mass m
- Number of particles N

In addition, the following parameters controlling the nature of the simulation are also specified:

- Particle radius R
- Observation time t
- Number of iterations n_{iter}

Note that this is only a way of setting the approximate initial conditions desired; once the simulation runs pressure and temperature are not fixed and particle mass becomes irrelevant.

2.1.2 PARTICLE VELOCITIES

At the start of the simulation, we must determine the speed of every particle.

Firstly we generate a random unit vector to describe the direction of motion of each particle. To ensure that the vectors are uniformly distributed on the unit sphere, we use spherical coordinates. Two random angles θ and ϕ from the uniform distribution on $[-\pi, \pi]$ are generated and the unit vector is defined as

$$\begin{pmatrix} \cos\theta \sin\phi \\ \sin\theta \sin\phi \\ \cos\phi \end{pmatrix},$$

a standard conversion from spherical to Cartesian coordinates. Figure 1 shows 1500 such unit vectors.

Rather than directly specifying the speed of the particles (all particles begin with equal speed), we use the kinetic energy equation

$$\frac{1}{2}m\langle v^2\rangle = \frac{3}{2}k_B T$$

to find the speed, where k_B is Boltzmann's constant and T and m are specified. Hence, the initial velocity of each particle is

$$\overrightarrow{v_{\text{init}}} = \sqrt{\frac{3k_B T}{m}} \begin{pmatrix} \cos\theta \sin\phi \\ \sin\theta \sin\phi \\ \cos\phi \end{pmatrix}.$$

(The v_{init} calculated here is the root-mean-square particle velocity of an ideal gas at thermodynamic equilibrium with the temperature and particle mass specified. Its purpose here is only for an approximately correct starting velocity.)

2.1.3 PARTICLE POSITIONS

The position of the particles also needs to be determined. Given pressure P and number of particles N, the volume of the gas to be examined is given by the ideal gas law

$$PV = Nk_B T$$

and so the side length ℓ of the cube we are considering is

$$\ell = \sqrt[3]{\frac{Nk_B T}{P}}.$$

The positions of each particle in the x, y and z axes are drawn randomly from the uniform distribution on $[0, \ell]$. The randomly generated initial positions of 1500 particles are shown in fig. 2.

2.2 MANAGING PARTICLE COLLISIONS

To correctly model oblique collisions between particles, we cannot assume they are point masses; we must give them a radius R. The distribution of possible collision angles does not depend on the radius chosen, however, so the choice of radius will not affect the final velocity distribution, though it will affect the collision frequency: a larger radius will mean that it takes less time for the system to reach equilibrium.

2.2.1 COLLISION DETECTION

We can say that two particles i and j have collided if the distance between their midpoints is less than the sum of their radii. That is, if

$$\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2} - 2R \leq 0.$$

(Note that if R is too small relative to the speed of the particles, they may pass through each other without a collision being registered, due

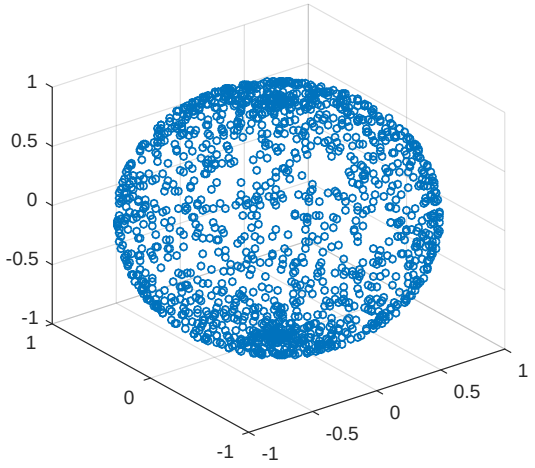

Figure 1: *A 3-dimensional scatter plot showing 1500 unit vectors generated by the method described.*

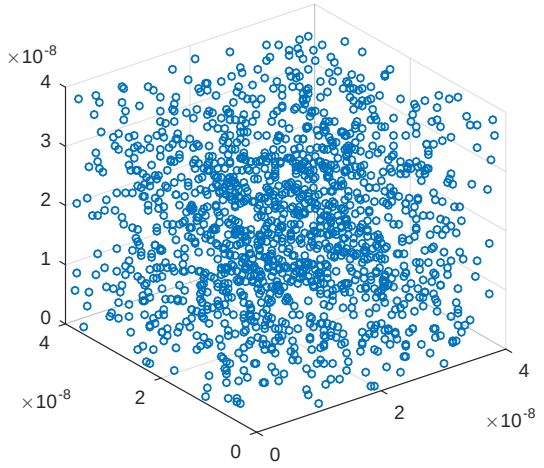

Figure 2: *A 3-dimensional scatter plot showing the initial randomly generated x, y and z positions (in metres) of 1500 particles.*

to the discrete nature of the simulation. For this reason a good choice of R is crucial.)

To implement this, two nested for loops are used and the presence of a collision is only evaluated when $j > i$:

```
for i = 1:N
    for j = (i+1):N
        % Detect collision
        % Act on collision
    end
end
```

This ensures that each collision is only detected once.

2.2.2 POST-COLLISION VELOCITIES

If a collision is detected between two particles, we must change the velocities of these two particles accordingly. When two particles of equal mass collide elastically in 3-dimensional space, their component velocities in the direction of collision (the vector between their two midpoints) are swapped. The other components of their velocities are unaffected.

This is because in a 1-dimensional elastic collision, momentum is conserved so

$$m_1u_1 + m_2u_2 = m_1v_1 + m_2v_2$$

and kinetic energy is conserved so

$$\frac{1}{2}m_1u_1^2 + \frac{1}{2}m_2u_2^2 = \frac{1}{2}m_1v_1^2 + \frac{1}{2}m_2v_2^2.$$

By collecting terms of mass, taking the difference of two squares and dividing the equations, we come to

$$u_1 + v_1 = u_2 + v_2.$$

Hence if the particles have equal mass then $v_1 = u_2$ and $v_2 = u_1$. This applies along the collision axis in an oblique 3-dimensional collision.

To model this in the program, we first find the unit vector $\hat{r}$ that denotes the direction from the midpoint of particle i to the midpoint of particle j. The vector from i to j is

$$\vec{s}_j - \vec{s}_i$$

where s_i is the 3-dimensional position vector of particle i. Therefore,

$$\hat{r} = \frac{\vec{s}_j - \vec{s}_i}{||\vec{s}_j - \vec{s}_i||}.$$

The vector component $\vec{v}_c$ of each particle's velocity in the direction of the collision can be

found using the dot product:

$$\vec{v}_{ci} = (\hat{r} \cdot \vec{v}_i)\hat{r},$$
$$\vec{v}_{cj} = (\hat{r} \cdot \vec{v}_j)\hat{r}.$$

Hence, to swap these components of the particles' velocities, we update them as follows:

$$\vec{v}_i \coloneqq \vec{v}_i - \vec{v}_{ci} + \vec{v}_{cj},$$
$$\vec{v}_j \coloneqq \vec{v}_j - \vec{v}_{cj} + \vec{v}_{ci}.$$

2.3 MANAGING WALL COLLISIONS

Fortunately, dealing with the walls of the container is much simpler. If either

$$x_i < R$$

or

$$\ell - x_i < R,$$

then particle i is in collision with one of the walls at the ends of the x-axis. Then all we need to do is flip the sign of the x-component of i's velocity:

$$v_{xi} \coloneqq -v_{xi}.$$

The same is true for the y and z axes.

2.4 ITERATING THE PROGRAM

The collision checks are run n_{iter} times, with an interval of $\frac{t}{n_{\text{iter}}}$ between each iteration. After every iteration, the positions of every particle are updated according to their determined velocity:

$$\vec{s}_i \coloneqq \vec{s}_i + \vec{v}_i \left(\frac{t}{n_{\text{iter}}}\right).$$

In addition, the velocity and position of each particle is saved after every iteration to 3-dimensional position and velocity matrices.

Once the loop has terminated, we can use the data in these matrices to plot our results.

3 RUNNING THE SIMULATION

The final simulation was run at approximately atmospheric pressure ($P = 101\,325\,\text{Pa}$) and room temperature ($T = 293\,\text{K}$) with particles of mass $m = 4.65 \times 10^{-26}\,\text{kg}$, the mass of an N_2 molecule.

As a compromise between fidelity and computing time, the number of particles was set to be 1500 and the simulation took 200 iterations.

Deciding on the radius of the molecules and the total observation time was rather more difficult. After initially using the van der Waals radius of nitrogen, 155 pm, it became apparent that the majority of collisions were going undetected, because the average distance Δs moved by each particle every iteration was much larger than the particle's radius (so particles would pass through each other and the walls without a collision being detected).

To remedy this, the average total distance travelled by a particle was set at a reasonable 1.5ℓ (which was adjusted to elicit the best rate of evolution of the velocity distribution) and the total time t was then calculated as

$$t = \frac{1.5\ell}{v_{\text{init}}}.$$

So that all collisions are detected, the radius must be larger than Δs, and so

$$R = \Delta s_{\max} = v_{\max}\left(\frac{t}{n_{\text{iter}}}\right)$$

where $v_{\max} \approx 2 \cdot v_{\text{init}}$.

This method worked remarkably well, and the final values used were

$$t = 1.2 \times 10^{-10}\,\text{s},$$
$$R = 5.9 \times 10^{-10}\,\text{m}.$$

4 RESULTS

Figure 3 shows a histogram of the velocity distribution after the 200^{th} iteration.

For comparison, figs. 5 and 6 show the velocity distributions after 20 and 70 iterations respectively.

At the start, the speeds were all $v_{\text{init}} = 510.7\,\text{m}\,\text{s}^{-1}$. As the simulation progressed,

the distribution approached the Maxwell-Boltzmann distribution, reaching it entirely by the end of the simulation as expected. Matlab R2015b does not have Maxwell distribution fitting function, and so for illustration purposes a Rayleigh distribution was fitted to the velocities after all 200 iterations. This is shown in fig. 4.

The root-mean-square velocity of this final distribution was $510.7\,\mathrm{m\,s^{-1}}$, exactly the same as the initial velocity v_{init}.

In total, there were 22,712 collisions between particles and 27,492 collisions with the walls.

5 CONCLUSIONS

After a very short amount of time, the particles in an ideal gas reach thermodynamic equilibrium at a Maxwell-Boltzmann distribution of velocities. If all of the particles start at the same initial speed, then this speed is also the final root-mean-square velocity of the particles.

It was interesting to note that the choice of particle radius played a large role in determining the success of the simulation, despite the final distribution not depending on it.

In the way of further analysis of the completed simulation, it could also be possible to

a) compare the final distributions at various different temperatures.

b) build a subroutine to test the actual pressure and temperature and compare to the result predicted by the ideal gas law.

c) run statistical tests throughout to evaluate exactly how the speeds approach the Maxwell-Boltzmann distribution over time.

The simulation could also be run for a much larger number of particles, given more computing power, to decrease random fluctuations in the distribution of speeds.

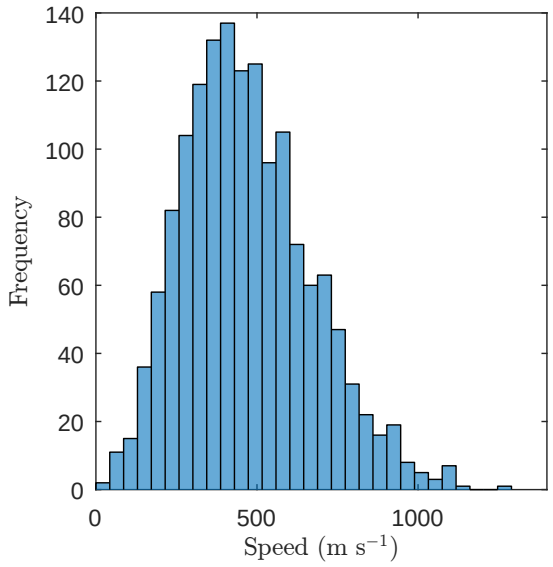

Figure 3: *A 30-bin histogram of the speeds of the particles after the full 200 iterations (*114.9 ps*).*

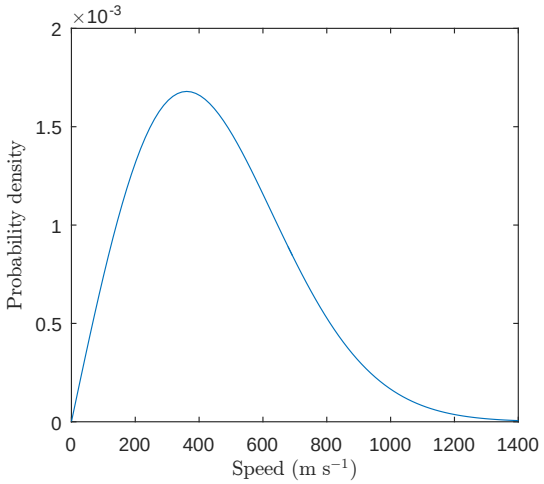

Figure 4: *Probability density function of the final speeds of the particles under a Rayleigh distribution.*

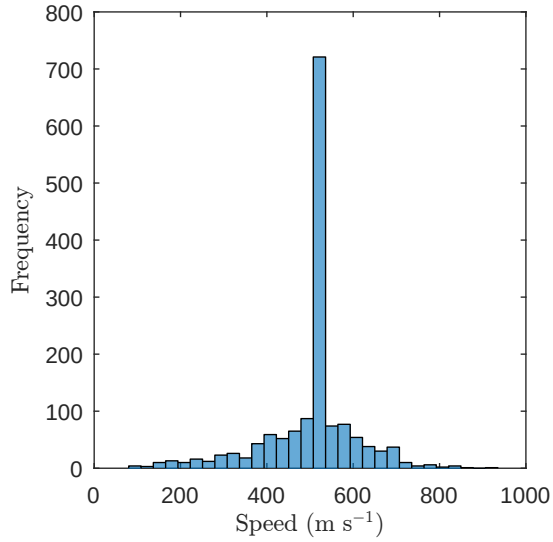

Figure 5: *Speeds of the particles after 20 iterations (*11.49 ps*).*

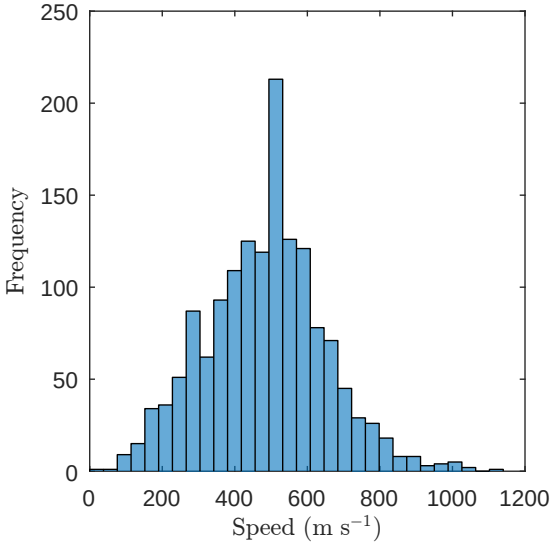

Figure 6: *Speeds of the particles after 70 iterations (*40.22 ps*).*

APPENDIX: SOURCE CODE

(To be run in Matlab or Octave.)

```
clear;
close all;

% Simulation parameters

P = 101325; % Pressure of particles / Pa
T = 293; % Thermodynamic temperature of particles / K
m = 4.65e-26; % Mass of particles / kg
N = 1500; % Number of particles to consider

R = 5.87e-10; % Radius of particles / m
t = 1.15e-10; % Total time to observe for / s
n_iter = 200; % Resolution of the simulation

%----------------------------------------

% Setup

k_B = 1.38e-23; % Boltzmann's constant
v_init = sqrt(3*k_B*T/m); % Initial (uniform) speed of every particle (using
↪  3/2 k_B T = 1/2 m v^2) / m s^-1
V = N*k_B*T/P; % Volume of container / m^3
side = nthroot(V,3); % Side length of the cubic box we're considering / m

dt = t/n_iter; % Time per step / s

s = side*rand(3,N); % Random x,y,z positions for each particle / m

angles = -pi + 2*pi*rand(2,N); % Two random angles per particle (between -pi
↪  and pi) / rad
unit_vect = [cos(angles(1,:)).*sin(angles(2,:)); ...
                sin(angles(1,:)).*sin(angles(2,:)); ...
                cos(angles(2,:))]; % A random uniformly distributed
↪  3-dimensional unit vector per particle
v = v_init*unit_vect; % Random velocities with uniform specified magnitude / m
↪  s^-1



% Data storage

wallcollisions = 0; % Collision counters
particlecollisions = 0;
firsttimeparticlecollisions = 0;

```

```
s_history = zeros(3,N,n_iter+1); % 3D matrices to hold the entire history of
↪  particle positions and velocities
v_history = zeros(3,N,n_iter+1);

s_history(:,:,1) = s; % Fill in the initial values
v_history(:,:,1) = v;

animation = struct('cdata',[],'colormap',[]); % Movie array

% Main loop

for iter = 1:n_iter

    for i = 1:N % For each particle

        % Check for collision with wall

        for dim = 1:3 % Repeat for x,y,z

            if s(dim,i) <= R || side - s(dim,i) <= R % Check distance from
↪  each wall

                v(dim,i) = -v(dim,i); % Reverse velocity
                wallcollisions = wallcollisions + 1; % Increment collision
↪  counter

            end

        end

        % Check for collision with another particle

        for j = i+1:N % Check against all particles with a higher index (to
↪  avoid duplicate collision checks)

            if (s(1,j)-s(1,i))^2 + ...
                    (s(2,j)-s(2,i))^2 + ...
                    (s(3,j)-s(3,i))^2 <= 4*R^2 % Detect collision

                % Unit vector between the two particles' centres
                difference_vector = (s(:,j)-s(:,i));
                difference_magnitude = sqrt((s(1,j)-s(1,i))^2 + ...
                                            (s(2,j)-s(2,i))^2 + ...
                                            (s(3,j)-s(3,i))^2);
                direction = difference_vector/difference_magnitude;

                % Component velocities in the collision direction

                v_i = dot(v(:,i),direction)*direction;
                v_j = dot(v(:,j),direction)*direction;

```

```
                delta_v = v_j - v_i;

                % Swap these velocities in this direction

                v(:,i) = v(:,i) + delta_v;
                v(:,j) = v(:,j) - delta_v;

                particlecollisions = particlecollisions + 1; % Increment collision counter

                if iter == 1

                    firsttimeparticlecollisions = firsttimeparticlecollisions + 1;

                end

            end
        end
    end

    % Update all positions

    s = s + v*dt;

    % Add to history

    s_history(:,:,iter+1) = s;
    v_history(:,:,iter+1) = v;

end

speeds = sqrt((v_history(1,:,:)).^2+(v_history(2,:,:)).^2+(v_history(3,:,:)).^2); % Matrix of history of speeds

% The matrix 'speeds' can then be used to plot histograms etc. of the speed
% distribution at different times.
```