Simulation of a many-particle system using space partitioning

Roald Frederickx
Kasper Meerts

16 october 2010

1 Scientific model

1.1 Speed distribution

The well-known **Maxwell-Boltzmann distribution** describes the molecular speed for a monoatomic classical ideal gas in thermodynamic equilibrium.

$$f(v)\,\mathrm{d}v = \left(\frac{\beta m}{2\pi}\right)^{3/2} 4\pi v^2 e^{-\beta\frac{mv^2}{2}}\,\mathrm{d}v$$

where β is given by $\beta \equiv \frac{1}{k_B T}$, T is the temperature, k_B is the Boltzmann constant and m is the molecular mass of the gas.

In this section, we derive a similar expression for a two-dimensional gas. Consider a gas consisting of a single molecule of mass m in a container in contact with a heat bath of temperature T. The probability of the particle being in a state with energy E is given by the Boltzmann distribution.

$$P(E)\,\mathrm{d}E \propto e^{-\beta E}\,\mathrm{d}E$$

Since the energy of particle is only a function of the velocity $\vec{v}$, $E = \frac{mv^2}{2}$, we can deduce

$$P(\vec{v})\,\mathrm{d}\vec{v} \propto e^{-\beta\frac{m|\vec{v}|^2}{2}}\,\mathrm{d}\vec{v}$$

But we're interested in the speed distribution, not the velocity distribution. For this, we use a substitution to polar coordinates

$$\mathrm{d}\vec{v} = \mathrm{d}v_x\,\mathrm{d}v_y = v\,\mathrm{d}v\,\mathrm{d}\theta$$

where $v = |\vec{v}|$ is the speed. So

$$f(v)\,\mathrm{d}v\,\mathrm{d}\theta \propto v e^{-\beta\frac{mv^2}{2}}\,\mathrm{d}v\,\mathrm{d}\theta$$

Since the distribution is independent of the parameter θ, we can integrate over it and remove it from the equation.

$$f(v)\,\mathrm{d}v \propto v e^{-\beta\frac{mv^2}{2}}\,\mathrm{d}v$$

After normalisation we find the **Rayleigh distribution**

$$f(v)\,\mathrm{d}v = \beta m v e^{-\beta\frac{mv^2}{2}}\,\mathrm{d}v$$

1.2 Collisions

The most basic short-range interaction that can be modeled in a many particle system is the elastic collision between hard spheres or disks. It is important that this interaction is not only addressed correctly, but also efficiently. Therefore, we want to avoid calculating computationally expensive trigonometric functions like sine and cosine and try to keep square roots to a minimum.

Collision simulations function in basically two ways, where the collisions are detected before they happen, *a priori*, or after, *a posteriori*.

In the a posteriori case, we advance the simulation by a small step each iteration and check if collisions have happened. If they did, we roll the simulation back a short while to before the collision happened and restore the situation correctly. In the a priori case, our collision detection algorithm needs to interpolate the positions of various bodies to predict whether a collision will happen or not.

A priori algorithms are more stable and correcter, because the intersections of bodies that need to be fixed in the a posteriori case are unphysical. However, determining ahead of time when two bodies will collide is very difficult and it requires the collision detection algorithm to know about the physics of moving bodies.

Since we detect collisions a posteriori, we need to restore the situation to before the collision. Thus, we need to rewind each pair of intersecting spheres to before the collision. Assuming there is no external forcefield, the particles are in a state of uniform rectilinear motion. The equation we need to solve for the time t is

$$|\vec{r_1}(t) - \vec{r_2}(t)| = R_1 + R_2$$

with $\vec{r_i}$ the position vectors of the bodies and R_i their radii. Since their motion is uniform, this becomes

$$|\vec{r_1}t_0 + \vec{v_1}\Delta t - \vec{r_2}t_0 - \vec{v_2}\Delta t| = |\Delta\vec{r} + \Delta\vec{v}\Delta t| = R_1 + R_2$$

with $\Delta\vec{r}$ and $\Delta\vec{v}$ the particles' relative position and motion and Δt the sought after time difference. Squaring both sides, we find

$$(\Delta\vec{r})^2 + 2\Delta\vec{v}\Delta\vec{r}\Delta t + (\Delta\vec{v})^2(\Delta t)^2 = (R_1 + R_2)^2$$

The solutions of this quadratic equation are

$$\Delta t = \frac{-2\Delta\vec{v}\Delta\vec{r} \pm \sqrt{4(\Delta\vec{v}\Delta\vec{r})^2 - 4(\Delta\vec{v})^2((\Delta\vec{r})^2 - (R_1 + R_2)^2)}}{2(\Delta\vec{v})^2}$$

As $\Delta\vec{v}\Delta\vec{r}$ is typically a negative number and $R_1 + R_2$ is always greater than $|\Delta\vec{r}|$, we take the positive squareroot and simplify

$$\Delta t = \frac{-\Delta\vec{v}\Delta\vec{r} + \sqrt{(\Delta\vec{v}\Delta\vec{r})^2 + (\Delta\vec{v})^2((R_1 + R_2)^2 - (\Delta\vec{r})^2)}}{(\Delta\vec{v})^2}$$

With this value, we can put the particles back in their original position before the collision. Now, we need to handle the collision itself.

1.2.1 One-dimensional elastic collision

Consider two particles denoted by subscripts 1 and 2. Let m_i be the masses, v_c be the the velocity of the center of mass (COM) frame, v_i the velocities before the collision, v_i' the velocities before collision in the COM frame, w_i the velocities after the collision and w_i' the velocities after collision in the COM frame. We assume the speed of both bodies is non-relativistic.

To considerably simplify the equations, we are going to transform to the intertial frame of the center of mass (COM). The COM velocity is

$$v_c = \frac{m_1 v_1 + m_2 v_2}{m_1 + m_2}$$

The resulting velocities in the new frame are

$$v_1' = v_1 - v_c = \frac{m_2}{m_1 + m_2}(v_1 - v_2)$$

and

$$v_2' = v_2 - v_c = \frac{m_1}{m_1 + m_2}(v_2 - v_1)$$

In this frame, the total momentum must be zero before and after the collision. The kinetic energy must also be conserved. Thus

$$p_1 + p_2 = q_1 + q_2 = 0$$

and

$$\frac{p_1^2}{2m_1} + \frac{p_2^2}{2m_2} = \frac{q_1^2}{2m_1} + \frac{q_2^2}{2m_2}$$

where p_i and q_i are the momenta respectively before and after the collision.

From this follows that

$$q_2 = -q_1$$

and

$$p_2 = -p_1$$

which we can fill in in the energy equation

$$p_1^2 = q_1^2$$

and

$$p_2^2 = q_2^2$$

The only possible solutions are $q_i = p_i$, that is no collision happened at all, or $q_i = -p_i$. This means that the velocities of both bodies are flipped.

$$w_i' = -v_i'$$

Thus we can derive the velocities in the original frame of reference

$$w_1 = w_1' + v_c = \frac{v_1(m_1 - m_2) + 2m_2 u_2}{m_1 + m_2}$$

$$w_2 = w_2' + v_c = \frac{v_2(m_2 - m_1) + 2m_1 u_1}{m_1 + m_2}$$

1.2.2 Two- and higher-dimensional elastic collisions

Because the colliding bodies are spheres, the only forces they can exert on each other at the contact point are according to the normal vector of each sphere at the contact point. The tangential component(s) of the momentum of each body do not change while the normal component is transformed according to the equations of an elastic collision. To get the normal vector, we simply normalise the displacement vector between the bodies

$$\vec{d} = \frac{\Delta\vec{r}}{|\Delta\vec{r}|}$$

1.3 Spatial partitioning

Most many-particle simulations incorporate some sort of interaction between the particles dependent on their mutual distance. Of considerable interest are short-range interactions, for example when the particles represent atoms. Since a particle could hypothetically interact with every other particle, each pair needs to be tested separately. Thus $n(n-1)/2$ interactions will be considered, with n the number of particles. Most of these are useless because the particles are too far apart. This $O(n^2)$ complexity is undesirable for performance reasons. An often used technique is called **spatial partitioning**.

By splitting the world into smaller partitions, *boxes*, one only has to check for interactions between particles in nearby partitions. If we keep the particles per box a constant (x), the amount of boxes becomes n/x. The complexity of this problem reduces in this way to $O(x^2 * n/x) = O(n)$, linear in the number of particles.

2 Implementation

2.1 Programming language

For this project, performance was a prime goal and thus the only contemplated programming languages were C and C++. Originally, we decided to implement the project in C++. It seemed to us that the clear distinction between the system, a partition and a particle lent itself well to an object-oriented programming language. Another advantage was the Standard Template Library (STL) which provided us with an efficient doubly linked list.

The nature of the problem, on the other hand, did not lend itself too well to OOP. For this reason, we decided to switch to C. This conversion only took an hour and the imperative paradigm seemed more fit to develop a performant program in a reasonable amount of time.

2.2 Data structures

The project needed a lot of manipulation of vector quantities, like position and velocity. For this we used the following straightforward C structure

```
1 struct Vec3 {
2         float x, y, z;
3 };
```

A particle is represented by its position and velocity. The particles needs to be contained in box so we store them in a circular doubly-linked list.

```
1 struct Particle {
2         Vec3 pos;
3         Vec3 vel;
4         struct Particle *prev, *next;
5 };
```

In this case, a box is little more than a pointer to any particle it contains. For various reasons, we keep track of the amount of particles in each box.

```
1 struct Box {
2         Particle *p;
3         int n;
4 };
```

The parameters for the system are given to the program by commandline arguments. Methods to benchmark the performance of the program are also built in the software. The scripts used to gather the data are in the same repository as the code.

At the start of the program the world is filled with a given amount of particles. The amount of time this procedure takes not added to the benchmark results. Then, either the system is rendered graphically for debugging purposes, or it loops a set amount of iterations. Optionally, the positions and velocities of each particle are saved. This way, we can plot a distribution of the speed or demonstrate Brownian motion, convincing one that our model is simulated physically correct.