Evaporation cooling simulation: Documentation

Matthew Houtput

25 October 2022

1 Introduction

This documentation accompanies an applet which explains the concept of evaporation cooling. The applet is a clone of a similar applet that was developed in JILA at the University of Boulder, as a part of the Physics-2000 website. This applet was written in Java and is unfortunately no longer available. Because this applet is incredibly instructive in courses on ultracold gases, I remade the applet in today's most popular programming language: Python.

The applet is written using the **Pygame** and **NumPy** packages. Only these two packages and the Python Standard Library are required for running the code. Pygame and NumPy can be installed using pip:

```
> pip install pygame
> pip install numpy
```

or using any package manager you prefer. Should I have used Box2D as well to model the collisions? Probably. Did I? No, and for that I'm very sorry, because this means that if you want to change anything about this applet, you now have to read through my code and this documentation detailing how I implemented everything manually.

The applet can be packaged into a Windows executable using PyInstaller on the provided Evaporation_addfiles.spec file:

```
> pip install PyInstaller
> PyInstaller Evaporation_addfiles.spec
```

This executable can be run on any Windows machine, without requiring to install Python, Pygame, or NumPy on that machine.

Feel free to modify or distribute this applet at will, as long as you do not remove the credit to me or JILA.

Throughout the documentation I will indicate the different elements of the code using different colors: blue for a function_name, green for a ClassName, and red for a variable_name. I will start the documentation by briefly describing the goal and gameplay of the applet. Then I will describe the physics underlying the collisions. Finally, I will discuss how all of this is implemented in Python, and I will discuss the code in more detail.

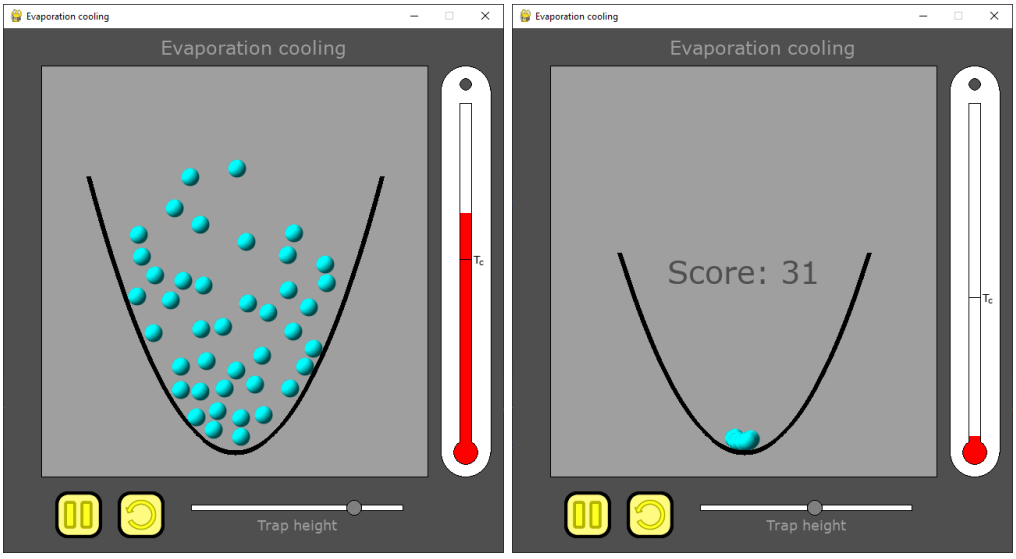

Figure 1: Applet during gameplay (**left**) and after lowering the temperature below the critical temperature (**right**).

2 Goal of the applet

The applet is made to visualize the concept of evaporation cooling, especially in the context of creating a Bose-Einstein condensate. The player starts with a number of atoms in a "harmonic trap", literally represented in the applet by a cup shaped like a parabola (figure 1). When the players starts the applet, the atoms start bouncing around randomly in the cup, similarly to the billiard ball model of a gas. If the atoms fly over the edge of the cup, they can escape the cup. The player can control the height of the trap using a slider, but this is the only control the player has over the gas.

The temperature of this classical gas of atoms is calculated as the average energy of all the atoms *in the cup*:

$$T = \frac{1}{N_{\text{cup}}} \sum_{i=1}^{N_{\text{cup}}} \left(\frac{1}{2} m_i \mathbf{v}_i^2 + m_i g y_i \right) \qquad \text{(arbitrary units)}, \tag{1}$$

where N_{cup} is the number of atoms left in the cup. Note that the usual definition of temperature only includes kinetic energy, but in the applet the gravitational potential energy must be added to ensure that the temperature is constant. The goal of the applet is to create a Bose-Einstein condensate by lowering the temperature below the critical temperature T_c. The player does this by letting the high-energy atoms fly out the top of the cup, while the colder atoms stay at the bottom. However, it is important that the player leaves enough atoms in the cup, since the critical temperature also lowers with the number of atoms, as $T_c \sim N^{2/3}$.

If the player manages to get the temperature below the critical temperature, the player is rewarded with a Bose-Einstein condensate, which is represented very simply (and rather inaccurately) by all the atoms overlapping at the bottom of the cup. The number of atoms left in the cup is the payer's score.

3 Implementation of the physics

3.1 Basic idea

The atoms are implemented as instances of a class Atom, which have four properties[1]: a position $\mathbf{r}_i = (x_i, y_i)$, a velocity $\mathbf{v}_i = (v_{x,i}, v_{y,i})$, a radius R_i, and a mass m_i. Note that this documentation will use the standard Pygame units and coordinate system. Distances will be measured in pixels, and time will be measured in frames (1 frame = 1/FPS seconds = 1/30 seconds), which means velocities are measured in $\frac{\text{pixels}}{\text{frame}}$. The standard Pygame coordinate system is also different from the typical coordinate system used in maths and physics: the origin $(0, 0)$ is in the top left corner, the positive x-direction is right, and the positive y-direction is *down* instead of up. Therefore, gravity points in the *positive* y-direction:

$$\mathbf{g} := g\mathbf{e}_y. \tag{2}$$

This documentation will clearly indicate whenever this distinction between the two coordinate systems is important, but this is rare since most of the theory is implemented using the general $\mathbf{g}$ vector.

If there are no collisions for a time Δt, each atom's position and velocity is updated as follows:

$$\mathbf{r}_i \leftarrow \mathbf{r}_i + \mathbf{v}_i \Delta t + \frac{1}{2}\mathbf{g}\Delta t^2, \tag{3}$$

$$\mathbf{v}_i \leftarrow \mathbf{v}_i + \mathbf{g}\Delta t. \tag{4}$$

This behavior is implemented in the function propagate_time. This is the behavior of all atoms as long as there are no collisions. Therefore, the applet follows the following basic flow:

1. Initialize the atoms randomly in the harmonic trap, with random positions and velocities.
2. Draw all the atoms to the screen.
3. Calculate the new position of the atoms after $t = 1$ frame (move_atoms):
   - Calculate the time Δt until the next collision (collision_time_parabola, collision_time_atoms).
   - Propagate all the atoms freely for a time Δt, without collisions (propagate_time)
   - Perform the collision by updating the velocities of the relevant atom(s) (collide_parabola, collide_atoms).
   - Repeat until the total elapsed time is 1 frame.
4. Remove any atoms outside the screen (remove_outside_atoms).
5. Calculate the new temperature (get_temperature).
6. Repeat steps 2-5 until the temperature is below the critical temperature.

[1] In the current version of the applet, every atom has the same radius and mass, but the implementation allows the masses and sizes of the atoms to be different. One can give the atoms a different radius or mass by modifying the function create_atoms.

The only physics that are left to consider is in the collisions. In order to have the atoms bounce around in the cup, two kinds of collisions must be considered: collisions between two atoms, and collisions between one atom and the parabola. For both kinds, we need to calculate both how the velocities of the atom(s) change after the collision, and we need to be able to calculate how long it will take before this collision occurs given the current configuration of atoms. Each of these will be considered in the following subsections.

3.2 Collisions between the atoms

3.2.1 New velocities

We start by calculating the new velocities of two atoms after they collided with each other, because this is the easiest part. Suppose two atoms with masses m_1, m_2 and velocities $\mathbf{v}_1, \mathbf{v}_2$ collide with each other. If the collision is elastic, there must be conservation of momentum and conservation of energy.

In order to satisfy conservation of momentum, we explicitly write the new velocities in terms of the momentum exchange $\Delta\mathbf{p}$:

$$\mathbf{v}_1' = \mathbf{v}_1 - \frac{\Delta\mathbf{p}}{m_1}, \tag{5}$$

$$\mathbf{v}_2' = \mathbf{v}_2 + \frac{\Delta\mathbf{p}}{m_2}. \tag{6}$$

Conservation of energy can then be written as:

$$\frac{1}{2}m_1\mathbf{v}_1^2 + \frac{1}{2}m_2\mathbf{v}_2^2 = \frac{1}{2}m_1\left(\mathbf{v}_1 - \frac{\Delta\mathbf{p}}{m_1}\right)^2 + \frac{1}{2}m_2\left(\mathbf{v}_2 + \frac{\Delta\mathbf{p}}{m_2}\right)^2, \tag{7}$$

$$\Leftrightarrow \Delta\mathbf{p}\cdot(\mathbf{v}_1 - \mathbf{v}_2) = \frac{m_1 + m_2}{2m_1m_2}(\Delta\mathbf{p})^2. \tag{8}$$

If we write the momentum exchange $\Delta\mathbf{p} = ||\Delta\mathbf{p}||\mathbf{n}$ explicitly in terms of its magnitude $||\Delta\mathbf{p}||$ and direction $\mathbf{n}$, the above equation can be used to calculate the $||\Delta\mathbf{p}||$ if we know $\mathbf{n}$:

$$||\Delta\mathbf{p}|| = \frac{2m_1m_2}{m_1 + m_2}\mathbf{n}\cdot(\mathbf{v}_1 - \mathbf{v}_2). \tag{9}$$

Or, in other words, the new velocities of the atoms are given by:

$$\mathbf{v}_1' = \mathbf{v}_1 - \frac{2m_2}{m_1 + m_2}[\mathbf{n}\cdot(\mathbf{v}_1 - \mathbf{v}_2)]\mathbf{n}, \tag{10}$$

$$\mathbf{v}_2' = \mathbf{v}_2 + \frac{2m_1}{m_1 + m_2}[\mathbf{n}\cdot(\mathbf{v}_1 - \mathbf{v}_2)]\mathbf{n}. \tag{11}$$

These new velocities always satisfy conservation of energy and momentum for any choice of the unit vector $\mathbf{n}$. All that remains is to choose $\mathbf{n}$. If we treat the atoms like billiard balls, they can only exert forces perpendicular to their mutual tangent line, so the momentum exchange should be in this direction as well. Therefore, we should choose:

$$\mathbf{n} = \frac{\mathbf{r}_2 - \mathbf{r}_1}{|\mathbf{r}_2 - \mathbf{r}_1|}. \tag{12}$$

This is how the collisions are implemented in collide_atoms.

3.2.2 Time until collisions

To find the time between the collision of two atoms, we assume the atoms are spheres with radii R_1 and R_2. We trace the trajectories of the two atoms and check the distance between their centers, $|\mathbf{r}_1 - \mathbf{r}_2|$: if this distance is equal to $R_1 + R_2$, the atoms must collide. If the atoms currently have positions $\mathbf{r}_1, \mathbf{r}_2$ and velocities $\mathbf{v}_1, \mathbf{v}_2$, the distance d between the atoms at a time Δt is equal to:

$$d(\Delta t) = \left| (\mathbf{r}_1 + \mathbf{v}_1 \Delta t + \frac{1}{2}\mathbf{g}\Delta t^2) - (\mathbf{r}_2 + \mathbf{v}_2 \Delta t + \frac{1}{2}\mathbf{g}\Delta t^2) \right|, \tag{13}$$

$$= |(\mathbf{r}_1 - \mathbf{r}_2) + (\mathbf{v}_1 - \mathbf{v}_2)\Delta t|. \tag{14}$$

The gravity-dependent term drops out because both atoms accelerate at the same rate. Therefore, we also don't have to worry about Pygame using a different coordinate system. We have to solve for the time when this distance is equal to $R_1 + R_2$:

$$[(\mathbf{r}_1 - \mathbf{r}_2) + (\mathbf{v}_1 - \mathbf{v}_2)\Delta t]^2 = (R_1 + R_2)^2, \tag{15}$$

$$\Leftrightarrow (\mathbf{v}_1 - \mathbf{v}_2)^2 \Delta t^2 + 2(\mathbf{r}_1 - \mathbf{r}_2) \cdot (\mathbf{v}_1 - \mathbf{v}_2)\Delta t + (\mathbf{r}_1 - \mathbf{r}_2)^2 = (R_1 + R_2)^2. \tag{16}$$

This is a simple quadratic equation. It only has solutions when:

$$[(\mathbf{r}_1 - \mathbf{r}_2) \cdot (\mathbf{v}_1 - \mathbf{v}_2)]^2 - (\mathbf{v}_1 - \mathbf{v}_2)^2 \left[(\mathbf{r}_1 - \mathbf{r}_2)^2 - (R_1 + R_2)^2\right] > 0. \tag{17}$$

This is the necessary condition for the atoms to cross paths. If this condition is not satisfied, the atoms do not cross and there can be no collision, so we set $\Delta t = 10^6$ frames as a large placeholder value. If this condition is satisfied, there are two possible solutions:

$$\Delta t_\pm = \frac{-(\mathbf{r}_1 - \mathbf{r}_2) \cdot (\mathbf{v}_1 - \mathbf{v}_2) \pm \sqrt{[(\mathbf{r}_1 - \mathbf{r}_2) \cdot (\mathbf{v}_1 - \mathbf{v}_2)]^2 - (\mathbf{v}_1 - \mathbf{v}_2)^2 \left[(\mathbf{r}_1 - \mathbf{r}_2)^2 - (R_1 + R_2)^2\right]}}{(\mathbf{v}_1 - \mathbf{v}_2)^2}. \tag{18}$$

There are two solutions because if the atoms move in a straight line without colliding, there will be two moments when the distance between the centers is equal to $R_1 + R_2$, but one of those would be after the atoms have crossed. Therefore, we are interested in the smaller of the two times, so we take the minus sign:

$$\Delta t = \frac{-(\mathbf{r}_1 - \mathbf{r}_2) \cdot (\mathbf{v}_1 - \mathbf{v}_2) - \sqrt{[(\mathbf{r}_1 - \mathbf{r}_2) \cdot (\mathbf{v}_1 - \mathbf{v}_2)]^2 - (\mathbf{v}_1 - \mathbf{v}_2)^2 \left[(\mathbf{r}_1 - \mathbf{r}_2)^2 - (R_1 + R_2)^2\right]}}{(\mathbf{v}_1 - \mathbf{v}_2)^2}. \tag{19}$$

This is the time it will take for two atoms to collide. The calculation of this time is implemented in collision_time_atoms.

An important remark is that this collision time can be negative: if the atoms are moving away from each other, the collision can be "in the past". Since this is not what we are interested in, we only accept positive solutions $\Delta t > 0$: if Δt is negative, no collision is possible and we set $\Delta t = 10^6$ frames. There is a nice side effect to this formula. Since we only accept positive times, the following condition must always hold for the collision to be accepted:

$$(\mathbf{r}_1 - \mathbf{r}_2) \cdot (\mathbf{v}_1 - \mathbf{v}_2) < 0, \tag{20}$$

which is equivalent to saying that the atoms are moving towards each other, i.e. the relative velocity of the atoms and their relative position vector must be (roughly) opposite in direction. This means we do not have to worry about any errors in the code when the atoms overlap by accident. In the rare case that any atoms would overlap, they might unphysically collide once, but then they will simply move away from each other without colliding any more. As soon as the atoms no longer overlap, they resume their normal behavior.

We do not have to calculate the collision time Δt between two atoms every frame. Indeed, since the collision time is exact, we can simply take the collision time that we calculated in the previous frame, and subtract one frame. We only have to recalculate the collision time whenever another collision occurs: if an atom collides with the parabola or another atom, we must recalculate the collision time between that atom and all other atoms. However, this is still much faster than recalculating all collision times at every frame.

3.3 Collisions between an atom and the parabola

In order to calculate the collisions between an atom and the parabola, we must be a little more careful with the reference system. Because the y-direction is down in the Pygame coordinate system, the parabola is actually concave. More specifically, if the parabola has curvature a (variable PARABOLA_CURVATURE) and top coordinates $\mathbf{r}_p = (x_p, y_p)$ (variable PARABOLA_XY), the equation of the parabola is:

$$y = y_p - a(x - x_p)^2. \tag{21}$$

For further calculations, we will use coordinates relative to the top of the parabola:

$$\tilde{x} = x - x_p, \tag{22}$$

$$\tilde{y} = y - y_p, \tag{23}$$

$$\tilde{\mathbf{r}}_i = \mathbf{r}_i - \mathbf{r}_p, \tag{24}$$

and in this coordinate system, the equation of the parabola is simply:

$$\tilde{y} = -a\tilde{x}^2. \tag{25}$$

We will now consider atoms bouncing off this parabola.

3.3.1 New velocity of the atoms

Suppose there is an atom at relative position $\tilde{\mathbf{r}}_i = (\tilde{x}_i, \tilde{y}_i)$ that is close enough to the parabola to bounce off it. In order to calculate the new velocity of the atom, we will assume that we bounce the atom off the line that is tangent to the parabola at the point that is closest to the atom. First, we need to find this point in order to find the tangent line: this is the point $(x_0, -ax_0^2)$ in figure 2. If we have this point, it is easy to find the equation for the tangent line: the slope is $-2ax_0$, so:

$$\tilde{y} = -2ax_0(\tilde{x} - x_0) - ax_0^2 \tag{26}$$

If this point is closest to the atom, then the center of the atom must be on the normal line of the parabola through $(x_0, -ax_0^2)$. The equation of this normal line is:

$$\tilde{y} = \frac{1}{2ax_0}(\tilde{x} - x_0) - ax_0^2 \tag{27}$$

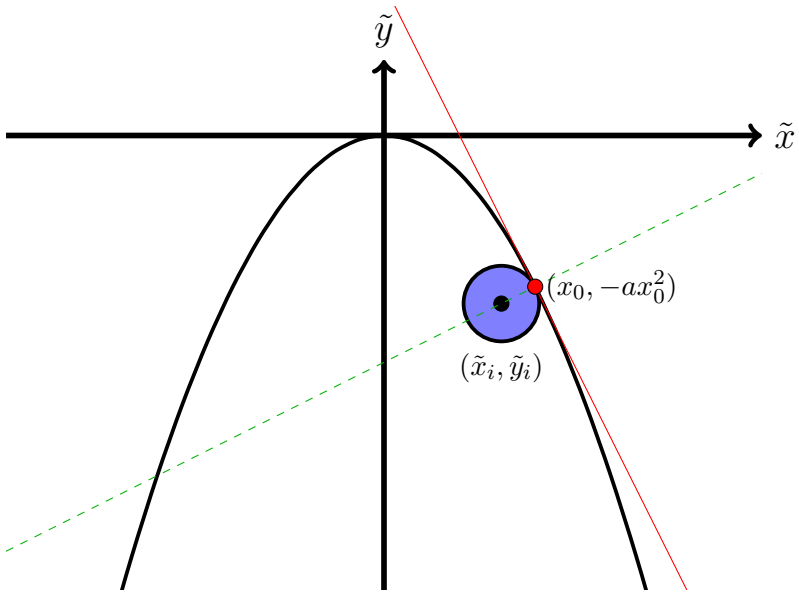

Figure 2: When an atom bounces off of the parabola, its velocity should be reflected along the red tangent line at the point $(x_0, -ax_0^2)$, where the atom and the parabola touch. This figure is drawn in the coordinate system used by Pygame, but the image is flipped around the x-axis to match with the more intuitive Cartesian axes.

Now we want the atom at given coordinates $(\tilde{x}_i, \tilde{y}_i)$ to be on this line, so we obtain the following equation for x_0:

$$\tilde{y}_i = \frac{1}{2ax_0}(\tilde{x}_i - x_0) - ax_0^2, \tag{28}$$

$$\Leftrightarrow 2a^2x_0^3 + (2a\tilde{y}_i + 1)\, x_0 - \tilde{x}_i = 0. \tag{29}$$

This equation must be solved for x_0. It is a depressed cubic equation, so we may use Cardano's formula; however, this will introduce several edge cases to take into account. It is conceptually simpler to use the Newton-Raphson method, since we have a decent starting guess $x_0 \approx \tilde{x}_i$. A small calculation gives the following iteration scheme to solve this equation:

$$x_0 \leftarrow \frac{4a^2x_0^3 + \tilde{x}_i}{6a^2x_0^2 + 2a\tilde{y}_i + 1}. \tag{30}$$

This replacement is repeated 5 times to find a decently converged solution for x_0.

As soon as x_0 is known, we know that the velocity of the atom should be reflected around the tangent line in x_0, which has the following slope:

$$m = -2ax_0. \tag{31}$$

This is a linear transformation whose transformation matrix can be found in the literature:

$$\mathsf{M} = \frac{1}{1+m^2}\begin{pmatrix} 1-m^2 & 2m \\ 2m & m^2-1 \end{pmatrix} = \frac{1}{1+(2ax_0)^2}\begin{pmatrix} 1-(2ax_0)^2 & -4ax_0 \\ -4ax_0 & (2ax_0)^2-1 \end{pmatrix}. \tag{32}$$

Therefore, we update the velocity of the atom as follows:

$$\mathbf{v}_i' = \mathsf{M} \cdot \mathbf{v}_i. \tag{33}$$

Since $\mathsf{M}^T = \mathsf{M}^{-1}$ we have $|\mathbf{v}_i'| = |\mathbf{v}_i|$, so that conservation of energy is satisfied.

This calculation is implemented in the function collide_parabola. Actually, the function is a little bit more flexible than the theory presented above. In particular, there is a variable damping, and when it is nonzero the velocity is updated as follows:

$$\mathbf{v}_i' = (1 - \text{damping})\mathsf{M} \cdot \mathbf{v}_i. \tag{34}$$

By default damping $= 0$, but after the player wins the game, we set damping $= 0.2$ in order to let the atoms converge near the bottom of the parabolic cup.

collide_parabola also allows the user to choose between a regular collision (described above) and an "emergency collision". The emergency collision is used to prevent unexpected behavior when an atom ends up in or below the edge of the parabolic cup. For an emergency collision, the atom is simply bounced away perpendicular to the parabola:

$$\mathbf{v}_i' = \frac{|\mathbf{v}_i|}{\sqrt{1 + (2ax_0)^2}} \begin{pmatrix} -2ax_0 \\ -1 \end{pmatrix}. \tag{35}$$

This collision looks strange but still satisfies conservation of energy, and it ensures that the atom will not collide with the parabola again (see section 3.3.2). Therefore, it can be used to prevent infinite loops, where an atom gets stuck inside the parabola because the time until the next collision is always $\Delta t = 0$. This can cause the applet to freeze on rare occasions. To prevent this, the current implementation of the applet performs an emergency collision whenever the time until the next collision is $\Delta t < 0.001$ frames. This is quite rare, so that emergency collisions are barely noticeable.

3.3.2 Time until parabola collision

It is quite hard to calculate the exact time Δt it will take the atom to collide with the parabola. In theory, it can be done numerically, but this approach is quite slow since we have to calculate Δt for every atom. Instead, we use an approximation that is much faster, but requires the recalculation of Δt every frame.

The idea behind the method is that we can exactly calculate the time before the atom will collide with a straight line through two points $\mathbf{P}$ and $\mathbf{Q}$. Therefore, we calculate the time before the atom collides with either of the two red lines in figure 3. The coordinates of the points $\mathbf{P}$, $\mathbf{Q}_1$ and $\mathbf{Q}_2$ are easy to calculate:

$$\mathbf{P} = (\tilde{x}_i, -a\tilde{x}_i^2), \tag{36}$$

$$\mathbf{Q}_1 = (-\sqrt{|\tilde{y}_i|/a}, \tilde{y}_i), \tag{37}$$

$$\mathbf{Q}_2 = (\sqrt{|\tilde{y}_i|/a}, \tilde{y}_i). \tag{38}$$

Furthermore, as the atom gets closer and closer to the parabola, the relevant red line will become a better and better approximation for the parabola, giving an excellent approximation for Δt.

Let us explicitly calculate the time it takes before the atom collides with the line through two points $\mathbf{P} = (x_P, y_P)$ and $\mathbf{Q} = (x_Q, y_Q)$. In vector form, the distance between a point $\mathbf{r}$ and this line is:

$$d = ||(\mathbf{r} - \mathbf{P}) \times \mathbf{n}_{QP}||, \tag{39}$$

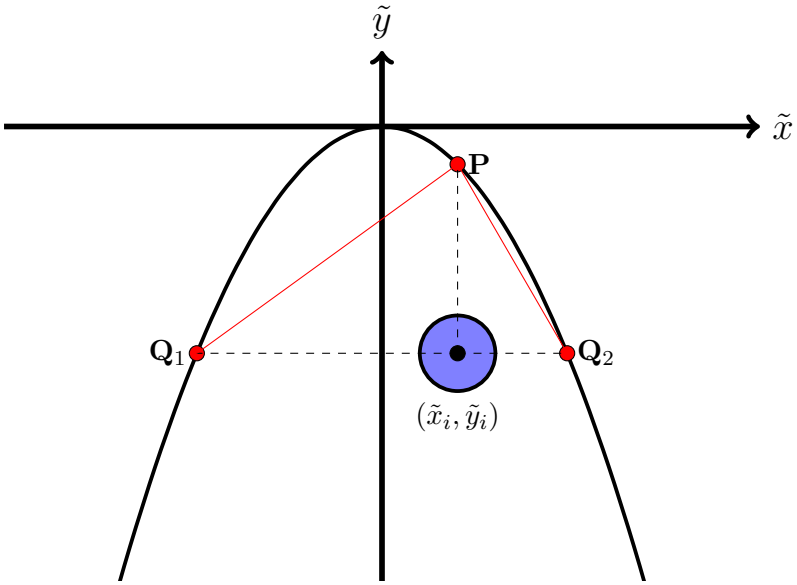

Figure 3: Instead of calculating the time before the atom collides with the parabola, we calculate the time it takes to collide with either of the red lines. This approximation becomes excellent when the atom is close to the parabola.

where $\mathbf{n}_{QP}$ is the unit vector indicating the direction of the line:

$$\mathbf{n}_{QP} := \frac{\mathbf{Q} - \mathbf{P}}{|\mathbf{Q} - \mathbf{P}|}. \tag{40}$$

We are looking for the time Δt where the distance between the center of the atom and the straight line is equal to R_i, so:

$$R_i = \left|\left| \left(\mathbf{r}_i + \mathbf{v}_i \Delta t + \frac{1}{2}\mathbf{g}\Delta t^2 - \mathbf{P} \right) \times \mathbf{n}_{QP} \right|\right|. \tag{41}$$

All the vectors are in the xy-plane, which means that the cross product is certainly in the z-direction. Therefore, it must hold that:

$$\left[\frac{1}{2}(\mathbf{g} \times \mathbf{n}_{QP}) \cdot \mathbf{e}_z \right] \Delta t^2 + \left[(\mathbf{v}_i \times \mathbf{n}_{QP}) \cdot \mathbf{e}_z \right] \Delta t + \left[((\mathbf{r}_i - \mathbf{P}) \times \mathbf{n}_{QP}) \cdot \mathbf{e}_z \right] = \pm R_i. \tag{42}$$

In principle, both the $+$ or $-$ sign give valid solutions. We know that if the atom can go through the line, there is one valid solution just below the line and one just above the line. On one side, the cross product in (41) will be positive, and on the other side it will be negative. In the end, we are interested in the lowest solution for Δt, since that is the collision that will happen first.

Equation (42) is a quadratic equation in Δt which can easily be solved. Of the two solutions, we must choose the one that remains finite if $\mathbf{g} \to \mathbf{0}$, since the other solution represents the time it takes for the atom to cross the line, and accelerate backwards to cross it again. In order to do this, we order the points $\mathbf{P}$ and $\mathbf{Q}$ so that $x_Q \geq x_P$. Essentially, we choose that the x-coordinate of $\mathbf{n}_{QP}$ is positive. In that case, if the atom is moving towards the line, it must hold that:

$$(\mathbf{v}_i \times \mathbf{n}_{QP}) \cdot \mathbf{e}_z < 0, \tag{43}$$

i.e. the linear term in (42) is always negative. If the atom is not moving towards the line we do not expect a collision, so we reject the collision if (43) is not satisfied. That means we can always write the time until the collision as follows:

$$\Delta t = \frac{-(\mathbf{v}_i \times \mathbf{n}_{QP}) \cdot \mathbf{e}_z - \sqrt{[(\mathbf{v}_i \times \mathbf{n}_{QP}) \cdot \mathbf{e}_z]^2 - 2\,[(\mathbf{g} \times \mathbf{n}_{QP}) \cdot \mathbf{e}_z]\,[((\mathbf{r}_i - \mathbf{P}) \times \mathbf{n}_{QP}) \cdot \mathbf{e}_z - R_i]}}{(\mathbf{g} \times \mathbf{n}_{QP}) \cdot \mathbf{e}_z}. \tag{44}$$

We chose the minus sign in front of R_i because if $x_Q \geq x_P$, it can be shown that $((\mathbf{r}_i - \mathbf{P}) \times \mathbf{n}_{QP}) \cdot \mathbf{e}_z \geq 0$ if the atom is in the parabola, and $(\mathbf{g} \times \mathbf{n}_{QP}) \cdot \mathbf{e}_z < 0$ if gravity points in the positive y-direction[2]. Therefore, choosing the minus sign gives the smallest value of Δt. We can use the above formula to calculate the time it will take before the atom will collide with the line going through $\mathbf{P}$ and $\mathbf{Q}$.

In order to get a good estimate for the time until the atom collides with the parabola, we calculate the above time for the two red lines in figure 3, and we take the minimum of the two calculated times. If either of the red lines does not satisfy condition (43), we simply take the other calculated collision time; if neither satisfies the condition (43) no collision with the parabola is possible and we set $\Delta t = 10^6$ frames. This calculation is implemented in the function collision_time_parabola. Note that the built-in NumPy function cross automatically returns only the z-component when the inputs are two-dimensional vectors $\mathbf{A}$ and $\mathbf{B}$, so the combination $(\mathbf{A} \times \mathbf{B}) \cdot \mathbf{e}_z$ is simply implemented as cross(A, B).

This calculation is an approximation which gets more accurate when the atom is closer to the parabola. Unlike the calculated time for the collision between two atoms, which is exact, we must recalculate this collision time for every atom at every frame. However, because this calculation is quite fast and only grows linearly with the number of atoms, this is not really a problem.

For the collision time between atoms, we rejected the collision if $\Delta t < 0$. However, it turns out that the applet actually runs better if we do not do this for the parabola collisions, since the atoms regularly manage to go slightly below the parabola edge. If we would reject the collision, the atom simply falls out the bottom of the cup. Instead, if $\Delta t < 0$, we simply set $\Delta t = 0$ in order to trigger an emergency collision, which bounces the atom back into the cup. No infinite loops are possible since we only accept the collision if (43) is satisfied, which is no longer the case after an emergency collision.

Finally, note that we only accept the collision if the atom is inside the parabola, but also if it is below the edge of the cup, so that the atoms can fly out if it can gain enough height. In practice, we only accept the collision if:

$$-a\tilde{x}_i^2 > \tilde{y}_i > -\text{parabola\_height} - R_i, \tag{45}$$

where parabola_height is the height of the cup, which the player can manually set through the slider.

[2]Remember that this is indeed the case in the Pygame coordinate system.

4 Further details of the implementation

Asides from the physics, there is plenty of other code that is necessary for the applet to run. Most of this code is written using the Pygame syntax: for detailed explanations of how this code works, I'll refer to the Pygame documentation (pygame.org/docs). In this chapter I'll go over the applet's main function, and give a more detailed explanation of what each of the functions do. This chapter is meant to be read in parallel with the code.

4.1 Setup

When the applet is started but before the player presses play, we initialize some variables and the playing field. The variables that are supposed to be universally available constants are initialized before the main function is called, which are:

- The dimensions of the window (PLAY_WIDTH, PLAY_HEIGHT, ...)
- The frame rate (FPS)
- The gravitational acceleration (GRAVITY_VEC)
- The properties of the atoms (ATOM_RADIUS, ATOM_STARTING_NUMBER)
- The curvature and position of the parabola (PARABOLA_CURVATURE, PARABOLA_XY)
- Several colors (WHITE, LIGHT_GRAY, ...).

Then, the main function is called[3], which starts with the initialization of Pygame. Specifically, we:

- create fpsclock, the clock that will take care of the timing of the applet
- create displaysurf, the surface that the whole applet will be drawn to and which is eventually printed on the screen
- set the caption of the screen to "Evaporation cooling"

Then, we initialize some fonts for further use, and initialize the variables that we will use to store the state of the mouse. We also create the buttons and slider that the player will use to control the applet; more information on the buttons and sliders can be found in section 5.

Then, all the physics-related variables are initialized in the function setup. In order, the function setup does the following:

- Set the height of the parabolic cup to 400 pixels
- Set the variable game_paused to True, such that the applet starts paused
- Set flag_restart to False, it will be set to True if the player clicks the restart button to restart the game

[3]More precisely, we define all the functions, and the main function is called at the end of the program.

- Create ATOM_STARTING_NUMBER atoms from class Atom, with positions randomly distributed in the parabolic cup. The exact procedure for the creation of the atoms is in the function create_atoms. It ensures the atoms do not overlap by filling the parabola with circles of radius ATOM_RADIUS + random_radius pixels, and then placing one atom in each circle at random. The velocities are chosen at random in such a way that atoms higher in the cup have a slightly higher velocity, such that they are "hotter". The velocity of the atoms is controlled using vel_base (average velocity at the bottom of the cup) and vel_step (increase in velocity as you go higher). The atoms are stored in a list named atoms: this list contains essentially all the physical information of the simulation and is therefore passed to many functions.
- Calculate the current temperature T of the atoms, which is just the total energy defined by (1)
- Set the initial critical temperature $T_{c,0} = 0.75T$
- Set the maximal temperature that will be drawn on the thermometer equal to $1.2T$
- Calculate all the collision times between any two atoms[4], using the method described in section 3.2.2. If there are N atoms on the screen, the collision times are represented by an $N \times N$ matrix atom_collision_times: the time it takes before atom i and j will collide is given by atom_collision_times$[i, j]$.

Finally, we define how the critical temperature will be calculated as a function of the number of atoms N remaining on the screen:

$$T_c(N) = T_{c,0} \left(\frac{N}{\text{ATOM\_STARTING\_NUMBER}} \right)^{\frac{2}{3}}. \tag{46}$$

4.2 Game loop

At this point, the applet has all the information it needs to start running, and it enters the main game loop. This loop is a simple "while True" loop, which will be exited when the players presses Escape or closes the applet by pressing the red X button. All of the following subsections occur in the main game loop, and will therefore keep occurring until the applet is terminated. Note that everything in this loop is executed once per frame, or 30 times per second. Indeed, the last line in the main game loop is:

fpsclock.tick(FPS),

where FPS = 30. Every time the method fpsclock.tick is called, it halts the program until at least 1/30 seconds have passed since the last time it was called. This ensures that the game will run at 30 frames per second, or slower if the calculations are too intensive.

[4]Originally, the collision times for the atoms and the parabola were also calculated here. However, since these are calculated on-the-fly anyway, it is not necessary to calculate them in the setup phase. More information can be found in section 4.2.5.

4.2.1 Retrieve user input

First, the user input is retrieved using Pygame's built-in events (pygame.org/docs/ref/event.html). Pygame has a queue of events that track all the user's input, such as button presses and mouse movement, and a list of all of these events can be obtained by calling pygame.event.get().

In the applet, these events are used to do two things. Firstly, the applet checks whether the user has pressed Escape or the window's X button: in either case, the applet is terminated. Secondly, the applet tracks the current position of the mouse and the state of the left mouse button: mouse_xy will store the current coordinates of the mouse, mouse_is_down stores whether the left mouse button is down or up, and mouse_is_clicked stores whether the left mouse button was clicked on this exact frame. Since the player controls the applet exclusively via the mouse, no other user input is needed.

4.2.2 Calculate the temperature

Next, the temperature of the atoms is calculated, simply by using the formula (1) for the average energy of the atoms. This computation is straightforwardly implemented in the function get_temperature. We also recalculate the current critical temperature using (46), and check whether the temperature is larger or smaller than the critical temperature. If it's smaller, the player wins.

4.2.3 Draw everything to the screen

Every frame, we have to update the graphics on the screen: this includes redrawing the atoms on their correct positions, but also drawing the thermometer, the text on the screen, the background colors, the parabola, ... The only things that are not drawn in this step, are the buttons and the slider. Pygame draws images as follows: one first draws graphics to a surface, and near the end of the code, the statement pygame.display.update() actually draws this surface to the screen. For this applet, we only have one surface called displaysurf, so we draw all graphics to this surface.

Most of the graphics in this applet are fairly simply drawn using Pygame's built-in functions such as pygame.draw.line, pygame.draw.rect, and pygame.draw.circle, which respectively draw a line, a rectangle, or a circle to a surface. Sprites and text are drawn in a slightly different way, using the method displaysurf.blit: this method is used e.g. in the Atom.draw method to draw the atoms to a surface.

Each of the things to draw is grouped together into several draw functions:

- The atoms are drawn to the screen using the Atom.draw method.
- draw_parabola draws the parabola to the surface, approximating it as a bunch of short straight lines
- draw_borders draws the dark grey borders around the edge of the scren, on top of which the thermometer, buttons, and slider are located
- draw_thermometer draws the thermometer to the screen. It is drawn from scratch in Pygame. The height of the mercury in the thermometer is based on the current temperature of the atoms, and the critical temperature is also indicated by a line.
- Finally, draw_text draws all the remaining text to the screen. This text consists only of the title and the score after the player wins, but this is grouped in a separate function since drawing text in Pygame is rather verbose.

4.2.4 Control the sliders and buttons

Next, using the user input in mouse_state = (mouse_xy, mouse_is_clicked, mouse_is_down), we retrieve the information that is stored in the buttons and the sliders. The buttons and the sliders are also implemented manually near the end of the code, in terms of several classes: Button, ImageButton, ToggleButton, and Slider.

In short, the buttons and sliders are implemented such that they are operated by a single control method. control does everything associated with the button or slider. For a button, it checks whether the mouse is inside it, it updates the state of the button depending on whether it is clicked or not and returns a flag that represents its state (True if the button is down, False if the button is up), and it draws the button to the displaysurf surface. For a slider, it does the same, except instead of returning its state as a True or False value, it returns the value of the quantity associated with the slider. In this case, there is only one slider, and its control method returns the value of parabola_height.

The applet contains two buttons, one that allows the player to pause and one that allows the player to restart. The restart button is of the class ImageButton, which simply means it is a button that has an image associated with it. When the button is clicked (and only on that frame), it returns the flag flag_restart=True, which restarts the game by re-running the setup function. The pause button is of the class ToggleButton: clicking it changes the state from game_paused=True to game_paused=False and vice versa.

The Button and Slider classes can be used in other applets as well: in particular, they are also used in the sister applet on laser cooling. Therefore, their implementations are separate from the rest of the code. For the interested reader, I have detailed the implementation of these classes in chapter 5.

4.2.5 Move all the atoms

Whenever the game is not paused, we move the atoms for a time equal to 1 frame. This is done using the function move_atoms, which follows the theory detailed in sections 3. However, taking care of all the specific cases makes this a rather large function.

The function starts by calculating the collision times between every atom and the parabola using collision_time_parabola. As detailed in 3.3.2, these times need to be recalculated every step. These collision times are stored in the 1D array para_collision_times; remember that the atom collision times are stored in the 2D array atom_collision_times.

move_atoms then proceeds by moving the atoms freely until a collision happens, performing that collision, and then repeating the process until a total time of 1 frame has passed. The behavior of the function is different based on whether the player has won or not. If the player has won, we do not consider the atoms colliding with each other, and set the damping parameter equal to 20%. If the player has not won yet (which is far more likely), we consider both collisions between the atoms and with the parabola.

The minimum Δt of all the relevant collision times is calculated, so that we know which collision happens first. If no more collisions occur until the total elapsed time is 1 frame, Δt is set to the remaining time in the frame. All the atoms are then moved freely for a time $0.99\Delta t$ using the function propagate_time, which simply implements equations (3)-(4) for every atom. We only move the atoms 99% of the way to avoid potential overlap between the atoms and the parabola: this effect is barely visible.

There are now three possibilities:

- There are no more collisions to be performed in this frame, so quit the function.
- The collision is between an atom and the parabola. In this case, we perform the collision collide_parabola, and we recalculate all the collision times for that atom using update_collision_times.
- The collision is between two atoms. In this case, we perform the collision collide_atoms, and we recalculate all the collision times for both atoms using update_collision_times.

After the collision is performed, the atom(s) that took part in the collision have a different velocity, and so their collision times with all other atoms and the parabola will be different. Therefore, we must recalculate their collision times, which is implemented in update_collision_times. It amounts to recalculating two entries in para_collision_times and two rows/columns in atom_collision_times; we do not have to recalculate the other collision times, since the other atoms still retain their original velocity.

This whole procedure is repeated until 1 frame has elapsed, after which this function has updated the positions and velocities of the atoms.

4.2.6 Update the screen

Finally, after the atoms have been moved, there are two lines of code left, which we have already discussed. The function pygame.display.update() will update the screen using everything that was drawn on the surface displaysurf, and the method fpsclock.tick will ensure that the game runs at FPS frames per second. After these lines of code have been executed, the main game loop repeats, and we start back at section 4.2.1.

5 Implementation of the classes

In this section I will go deeper in the implementation of the Atom, Button and Slider classes. The Button and Slider classes are not specific to this applet, and can be used in other Pygame applications. Both the button and slider are controlled in the main game loop by a single call to the control method.

5.1 Atoms

The Atom class is a very basic class. Essentially, every instance of Atom only has some basic properties:

- A position, an array of the form $[x, y]$
- A velocity, an array of the form $[v_x, v_y]$
- A radius
- A mass
- An image, which is a blue sphere by default (images/Sphere.png)

The complex behavior of the atoms is implemented in other functions such as move_atoms, collide_atoms, ... Atom itself only has two methods besides the constructor:

- draw, which draws the image of the atom to a given surface
- set_radius, which allows the user to set the radius of the atom to a different value, automatically scaling the image in the process. This is a convenience function, but it is currently unused in the applet.

Other than this, Atom has no functionality: the purpose of this class is that the other functions in the code can retrieve the various positions and velocities from the different Atom instances in the applet.

5.2 Buttons

5.2.1 The Button class

The Button class is a simple button that does not have any visuals. It is meant as a class that other button classes inherit from: it has all the core mechanics that we need from a button, but we will not make any buttons directly from this class.

The button has two possible states: "active" and "idle". Whenever the control method is called, it calls the action method when it is active and the idle method when it is idle. Both of these methods can be quite general, but the basic behavior is that the action method simply returns True and the idle method returns False.

A basic Button has two core variables:

- surface, the surface that the button lives on. Whenever the draw method is called, the button is drawn to this surface.

- bounding_rectangle: A 4-tuple of the form (x, y, width, height) that represents the bounding box of the button. Following the standard Pygame convention for rectangles, the top left corner of this bounding box is (x, y).

Besides this, it has six methods:

- action, the method that is called when the button is active. The default behavior is that it returns True.
- idle, the method that is called when the button is not active. The default behavior is that it returns False.
- is_active: Determines whether the button is active based on the mouse_state. The default behavior is that it returns True when mouse_clicked is true and when mouse_xy is inside the hitbox defined by check_mouse.
- check_mouse: Determines whether the mouse is inside the hitbox of the button. By default, this hitbox is a rectangle equal to the bounding_rectangle.
- draw: Draw the button to self.surface. By default, it simply draws a black rectangle, but usually this function is overwritten to draw an image based on the mouse_state.
- control: This function combines the functionality of the prevous five methods and is the only method used in practice. It draws the button to the screen, checks whether the button is_active, calls action or idle, and returns that method's result.

The typical usage of this button would be to add the following statement to the main game loop:

flag_pressed = my_button.control(mouse_state)

This statement draws the button, and the variable flag_pressed returns True if the button was clicked and False if it was not. After that, the code can include what needs to happen if flag_pressed is True or False.

5.2.2 The ImageButton class

The ImageButton class inherits from the basic Button and has exactly the same functionality. The difference is only aesthetic: an ImageButton is drawn as an image rather than a black rectangle. In total, an ImageButton can have up to three images associated with it:

- idle_image when the button is idle
- hover_image when the mouse hovers over the button (default: equal to idle_image)
- down_image when the button is pressed (default: equal to hover_image)

These images are loaded in the constructor using the Pygame syntax. All other methods are inherited from Button, except for the draw method. For ImageButton, draw decides which of the images is relevant based on the mouse_state, and draws that image to the surface associated with the button. In the applet, the restart button is the only ImageButton.

5.2.3 The ToggleButton class

The pause button in the applet functions differently than the restart button: the restart button needs to activate only once when it is clicked, but when the pause button is clicked, the game needs to remain paused until the button is clicked again. This functionality is implemented in the ToggleButton class, which inherits from the ImageButton class because it uses the same draw method. It represents a button with two states, "On" and "Off", whose state can be toggled by clicking the button.

ToggleButton has six images associated with it: the idle, hover, and down images when the button is on, and those when the button is off.

- idle_on_image
- idle_off_image
- hover_on_image
- hover_off_image
- down_on_image
- down_off_image

Aside from the variables inherited from ImageButton and Button, the only remaining variable is is_on, a Boolean that determines the state the button is currently in. The action and idle methods are overwritten: idle returns the state of the button (is_on), and action toggles the state from off to on and vice versa. ToggleButton also has two new methods:

- update_images, which sets idle_image, hover_image, and down_image to the correct images based on the value of is_on.
- set_state, which sets the state of the button is_on to on (True) or off (False), and updates the images using update_images.

If the state of the button must be changed, it should be done using set_state, since otherwise the images of the button will not be updated.

5.3 Sliders

A slider plays a similar role to a button in the sense that it allows the player to interact with the applet, and both can be included in the main game loop using a single call to the control method. The main difference is that we use a button to return a Boolean variable, while a slider can return any real number within a given range. Therefore, the implementations of the Slider and Button classes overlap slightly.

An instance of Slider will have the following core attributes:

- surface, the surface that the slider is drawn to
- bounding_rectangle: A 4-tuple of the form (x, y, width, height) that represents the bounding box of the slider

- min_value and max_value, respectively the minimal and maximal values that the slider can return
- activation: The position of the slider will be mapped to an activation $\in [0, 1]$, where activation $= 0$ corresponds to the extreme left of the slider and activation $= 1$ corresponds to the extreme right.
- slider_radius: The radius of the knob of the slider
- sliding: A boolean that indicates whether the knob is being moved
- text_string, text_font, text_color: Two variables associated with the text that is drawn under the slider

The idea behind the slider is that its x-position can be changed by dragging the slider, which will translate to changing the value of activation. In turn, the control method will return a different value based on the value of activation:

$$\text{return } \text{min\_value} + \text{activation} * (\text{max\_value} - \text{min\_value}) \tag{47}$$

The behavior is implemented in the following methods:

- get_slider_activation, which calculates the activation based on the given position x of the slider knob
- get_slider_xy, which calculates the position of the slider knob based on the current value of the activation; this function is essentially the inverse function of get_slider_activation
- get_slider_value, which returns the output value given by (47)
- set_slider_value, which sets the activation in a way that the output value is equal to a desired value
- is_sliding, a customizable function that decides when the knob is sliding based on the input mouse_state. Currently, it is implemented that clicking the hit box of the slider (determined by check_mouse) and holding the mouse button down allows the knob to slide. Sliding is possible as long as the mouse button is held down, even if the mouse exits the hit box.
- check_mouse, which defines the hit box of the slider. It is currently implemented as a rectangle that is as wide as the slider bounding box, and as high as the slider knob.
- draw, which draws the slider to its associated surface. This includes the background rectangle, the knob, and the description text.
- control, which combines the functionality in all of the above methods into a single method. It draws the slider, calculates the slider activation using get_slider_activation if it is_sliding, and finally it returns the slider value using get_slider_value.

The typical usage of a Slider in the main game loop is very similar to the usage of a button:

```
output_value = my_slider.control(mouse_state)
```

with the only difference that it returns a real number rather than a Boolean. In the applet, the player only has access to one slider, which controls the height of the parabolic cup.

6 Closing remarks

My highscore is 31. Can you do better?

But mostly, have fun! I hope you and your students will enjoy this applet as much as I have enjoyed making it. A fun idea is to let multiple students try the applet, and have some competition to see who can leave the most amount of atoms in the condensate.