Exercise 1, TFY4235 Computational physics

Martin K. Johnsrud

Introduction

The goal of this exercise is to simulate particles as flat, hard disks in a square, 2D container. This is done with a event-driven simulation, as described in the exercise [1]. This is implemented in Python, using the built-in library heapq. The simulation is used to first tested with scenarios we know the outcome of, then used to demonstrate the Maxwell-Boltzmann distribution and to investigate the effect of a large, heavy disk hitting a large number of small, inert particles.

Implementation

The main engine of the code is the function `run_loop()` in `utillities.py`. It follows the algorithm, as laid out in [1], using the objects:

- `particles`, a numpy array with the position and velocity of all the particles.

- `t`, a list where `t[i]` is the time of collision number `i`.

- `collisions`, a priority queue containing a list for each collision. The list has the time of the collision, the index of the particle(s) involved, and the type of collision it is. The list is sorted by the time of the collision.

- `last_collided`, a list of when each particle was involved in a collision.

The function takes the parameters for the simulation, (`N, T, radii, masses, xi`), which is resp. the number of particles, number of time steps to be executed, a list with the radii of the particles, a list of the masses, and the restitution coefficient ξ. It also takes a function `init`, which is used to set the initial distribution of particles.

When the particles are initiated, a while loop executes the algorithm as described in [1]. The next collision is found by the `heappop` method of `collisions`. The list `last_collided` is used to check if the next collision is valid. The function `exectue_collision` translates the particles forward in time, finds the new velocities for the particles involved in the collision, as shown in [1], before finally finding the new collisions, given the updatet positions and velocities. The `run_loop` function can be given the argument `TC=True`. Then, it runs a TC-model, as described in [2]. If the function is passed the argument `condition=func`, it will check the function `func` at regular intervals. This makes it possible to exit the loop early. It is used to run the simulation until 10% of the energy is remaining, as described later in the report.

The `init` function depends on the situation that is being simulated. To place out particles randomly, in either the whole box or in a smaller box, the `random_dist` function, located in `particle_init.py` is used. This function contains a loop that places a particle randomly within the desired bounds, and gives them velocity with a given magnitude, but uniformly distributed direction. The if the particle overlaps with any of the other particles, already placed it is rejected. There is a "emergency break", if too many particles in a row are rejected, the codes throws an error. As this function only returns an numpy array, the just-in-time compliation library numba is compatible with it. With 2000 particles, of radius 0.008, the `@njit()`-decorator gives a speedup from 15.1 seconds to 0.8 seconds.

`profile.ipynb` shows the profiling of `run_loop`, and the subroutines that takes the most time. This shows that it is the loop that pushes the next collisions to the priority queue that is the bottle neck. A large (HOW LARGE?) speedup was found by rewriting the function that finds the collisions. The notebook

`profile_old.ipynb` shows the time used by the old version. The old version of the functions took the index `i` of the particle in question, then found if and when it was to collide with all other collisions, and returned this time as a list. The new version utilizes the fact that everything is contained in numpy-arrays. It does the same operations as the old function, only on arrays instead of single elements. This is done by using masks. An array of booleans can serve as indecies, so `lst[np.arange(N) != i]` gives an array with all the elements of `lst[i]`. The profiling show that while most of the time went to the calculation of the next collision, it now goes to pushing to the `collisions`-heap.

To investigate the creation of craters by a projectile, a function that measures the size of crater is needed. This is done by laying a grid with with a spacing Δx, looping through each cell to check if there is a particle inside it. The process for checking if a disk is inside each square cell is illustrated in figure 1. Then, the size of the crater is given by $m\Delta x^2$, where m is the number of unoccupied cells. This method relies on choosing a cell size Δx large enough so that only cells within the crater are marked as empty. This is done by inspecting the result, and comparing it to a plot of the crater as shown in 2
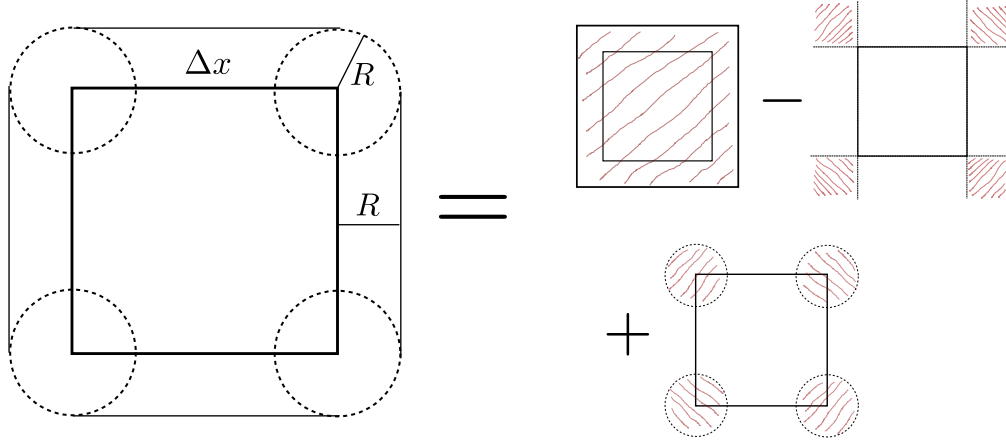

Figure 1: Checking if a disk of radius R is inside of a square of side length Δx, is equivalent to checking if the center of the disk is inside the shape on the left side. Thus, the task is reduced to checking if the center is inside the larger square of side lengths $R + 2\Delta x$, but not at the corners, or inside one of the circles of radius R centered at the corners of the square.
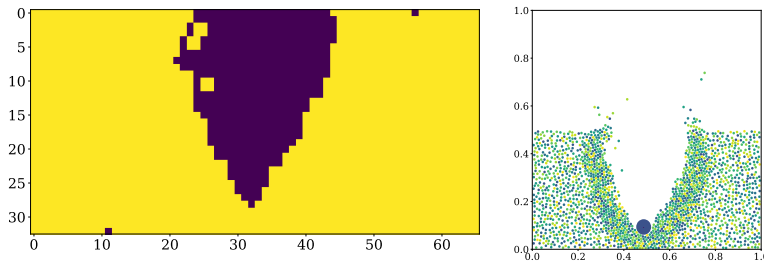

Figure 2: The grid cells with a particle inside is marked yellow, while the ones without a particle inside, i.e. those counted as making up the grid, are purple. The plot to the right shows the crater.

To avoid inelastic collapse, the simulation was implemented as at TC-model as described in [2]. This is only needed when $\xi < 0$, and there are extremely many collisions in a short timestep. inelastic collapse were observed when the projectile is aproaching the bottom of the box- Good results were found with $t_c = 10^{-8}$.

(a) One ball being simulated. after 10000 steps, it still follows a regular pattern.
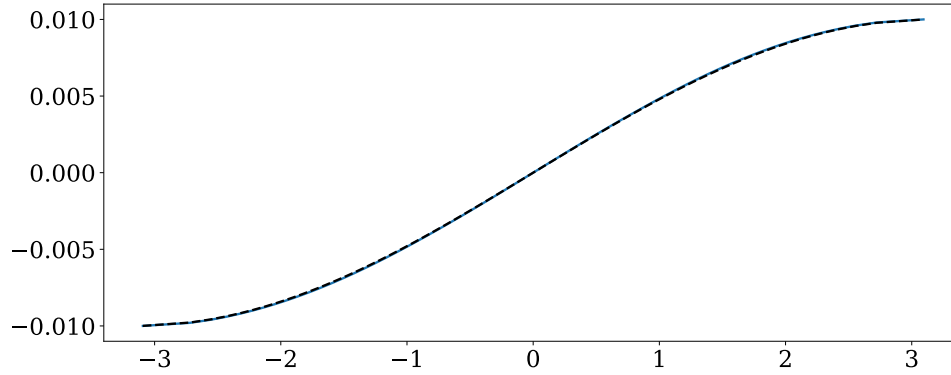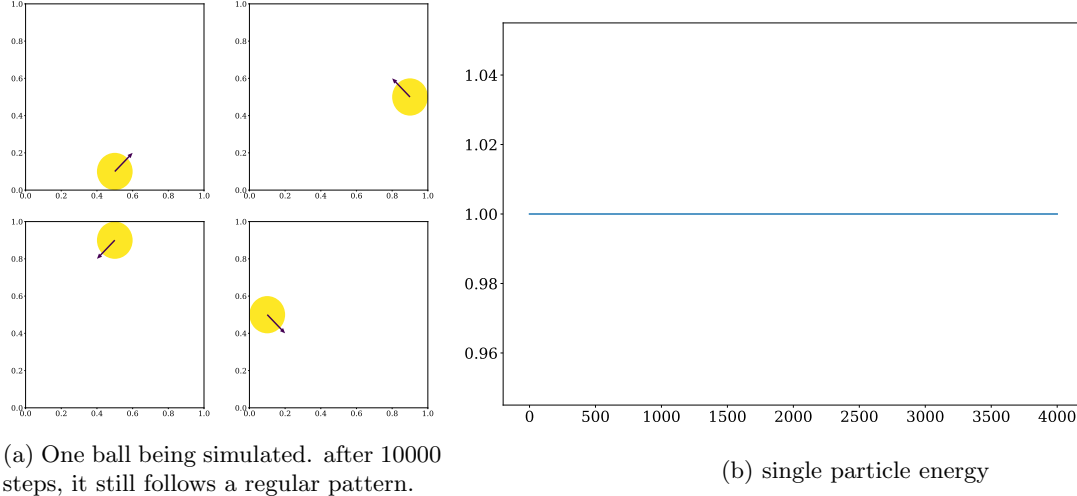
(b) single particle energy

Figure 4: The

Tests

Several functions were developed to test the accuracy of the simulation. First, one particle, starting at in the middle of the box all the way to the left with a velocity with at $45°$ to the x-axis, should move in a titled rectangle. With $\xi = 1$ it should also conserve energy. Figure 3a shows that this is still the case after $10,000$ events. Figure 3b shows that the energy of the system is conserved.

To test the validity of the particle collision, one small, light particle is sent towards a single, large and heavy particle, with varying impact parameter. The relationship between the impact parameter is is (SKJEKK DETTE; REFERER TIL GOLDSTEIN) $\frac{\mathrm{d}s}{\mathrm{d}\theta} = a/2\sin(\theta/2)$. The result is shown i Figure 4, and is in good agreement with the theory. Lastly, the energy from a simulation of 2000 particles over 20,000 events is shown in figure 5, together with a snapshot of the particles. The energy is conserved, aside from negligible fluctuations.

Results

When the system is first initiated, all particles have the same magnitude of velocity. As the system equilibrates, it should reach the Maxwell-Boltzmann distribution, which in 2D is

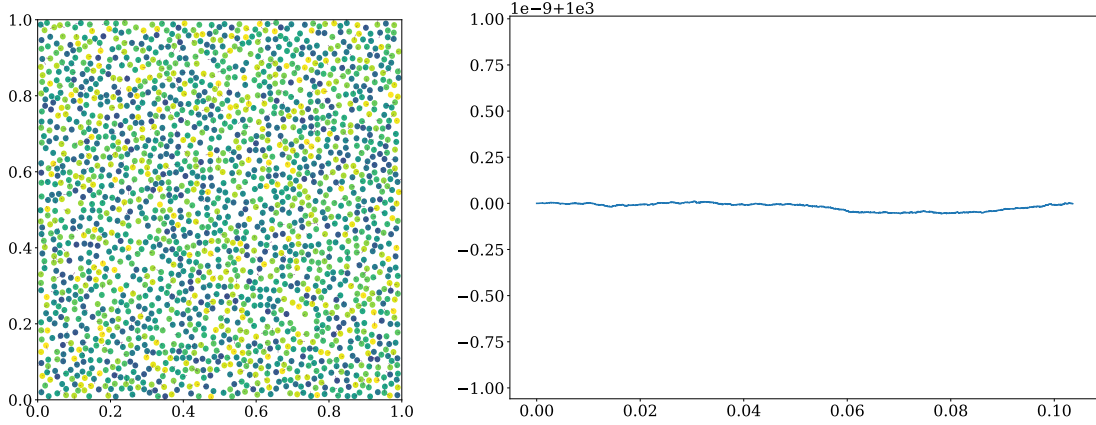$$f(v) = \frac{mv}{T}\exp\left(-\frac{mv^2}{2T}\right),$$

3

Figure 5: On the left is a snapshot of the particles. The arrows represent the velocities. On the right, the energy is plotted as a function of events. The energy loss can be seen to be negligible.
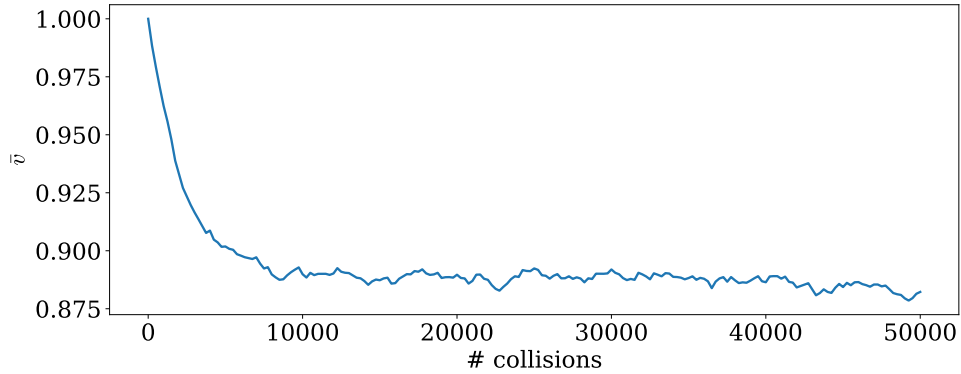

Figure 6: Average velocity, as a function of collisions. The distribution reaches equilibrium around 6000 collisions, or $3N$

when using units in which $k_b = 1$. The equipartition theorem gives the temperature $T = E$ in 2D. Figure 6 shows the average velocity as a function of time. It is a indication of when the system has reached equilibrium, and was used to find a good point to start sampling. After that, the simulation is sampled every N event, where N is the number of particles. This ensures somewhat independent samples. Figure 7 shows the velocity distribution, compared to the Maxwell Boltzmann distribution.

Next, two different types of particles are simulated, one with a mass of 1, the other with a mass of 4. As the particles have the same magnitude of velocity, but different masses, they will have different energies, and thus have different energies (although temperature is in fact only defined in equilibrium). However, after the system equilibrates, the both sets of particles should reach the Maxwell-Boltzmann distribution with a common temperature. The peak will nonetheless be at different velocities.

This is, however, only the case when the restitution coefficient ξ is set to 1. For values less than one, the two sub systems approach each other in temperature, but does not come into equilibrium. This is shown in Figure 7.

To simulate the impact of a projectile, one disk is placed at $\vec{x} = (0.5, 0.75)$, and given a velocity of $v = (0, -20)$. The lower half of the square box is filled with small, light particles. The impact from the large disk into the densely packed disks will create a crater. To investigate the effect of the mass and radius on the size of the crater, the size of the crater must be measured. T To investigate the effect of the size of projectile on the crater, the simulation is run 10 times, using projectiles with different radius. The mass of
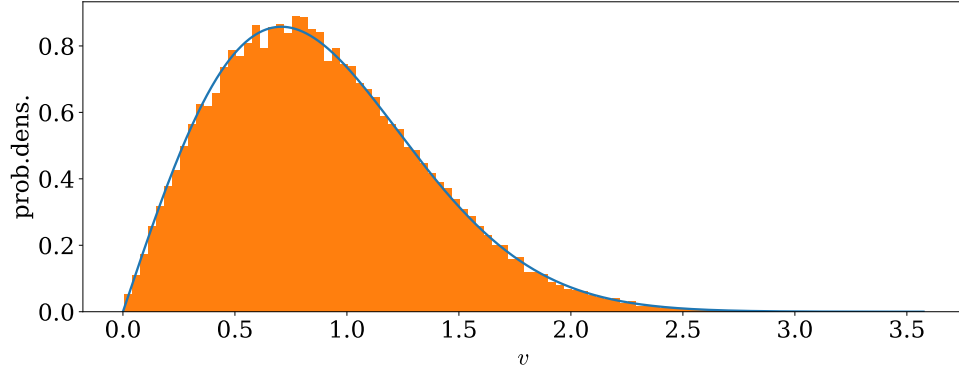
4

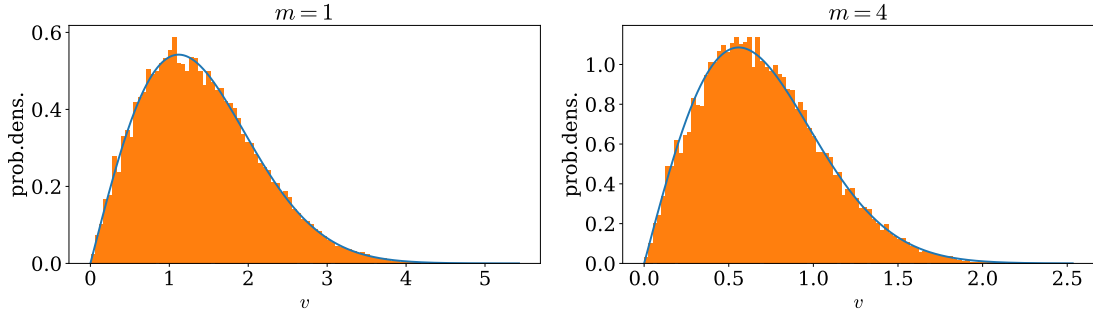Figure 7: The velocity distribution is a good fit with the Maxwell-Boltzmann distribution, shown as a blue line.

Figure 8: The velocity distribution of the particles with $m = 1$ and $m = 4$ is show left and right, resp. and compared to the Maxwell-Boltzmann distribution.
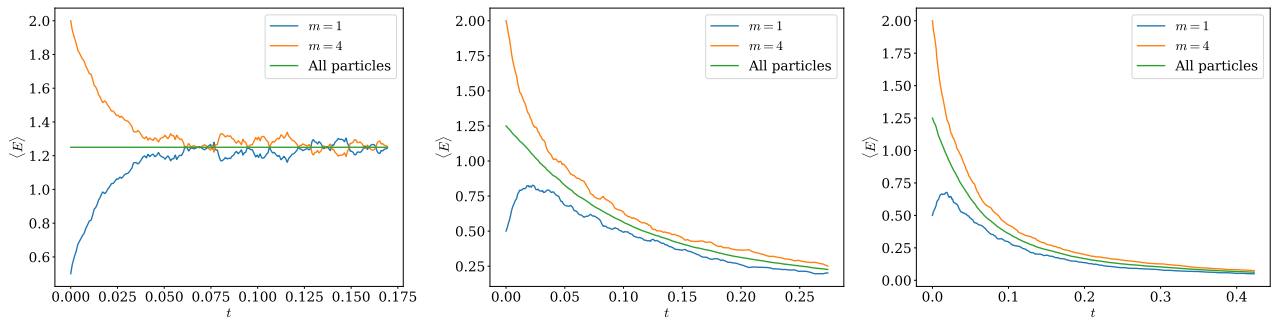

Figure 9: The average energy, as a function of time, of resp. $\xi = 1$, $\xi = 0.9$ and $\xi = 0.8$.
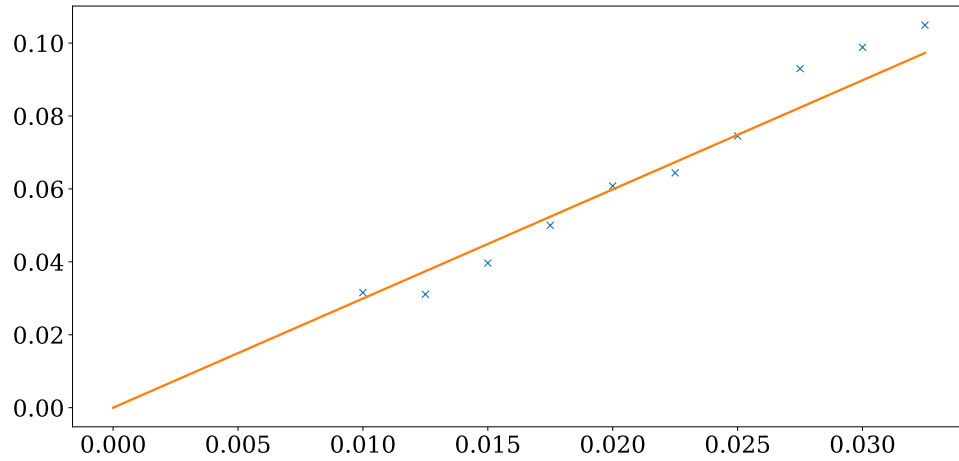
Figure 10: The size of the crater, as a function of the radius of the incoming projectile.

the projectile is proportional to R^2, so that the density is constant. Figure 10 shows the size of the crater,

References

[1] TFY4235 Computational Physics Exercise 1, 2021. Institutt for fysikk.

[2] Stefan Luding. How to handle the inelastic collapse of a dissipative hard-sphere gas with the tc model. *Granular Matter*, 1:113 –128, 1998.