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 [3]. 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

Though this implementation is largely based on the description in [3], some spesific choices have been made, which this section aims to illuminate. It describes the overarching structure of the code and some of the main datastructures. The most important function in the code is `run_loop()` in `utillities.py`. It instantiates the reuqired objects, and executes a while loop containing the algorithm, as laid out in [3], 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.

`run_loop` takes the parameters for the simulation, `(N, T, radii, masses, xi)`, which is respectivley 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. In the while loop, 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, before finally finding the new collisions, given the updated positions and velocities.

The `run_loop` function can be given the argument `TC=True`. Then, it runs a TC-model, as described in [2]. This is done to avoid inelastic collapse. This is only needed when $\xi < 1$, and there are extremely many collisions in a short time step. Inelastic collapse were observed when the projectile were approaching the bottom of the box, as in Figure 2. Good results were found with $t_c = 10^{-8}$. 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.

As the program runs, the heap `collision` becomes longer and longer, as more than one collision is added each time one is removed. This is often not a problem, however when it becomes too long, it is beneficiary

to discard the `collision` heap, and start from scratch. This is handled by `run_check`. This function is called regularly in the main loop, at the same spot as `func` as described in the paragraph above. If the time between each call is twice as long as it initially was, all collisions are discarded, and `collisions` are instantiated anew. This also decreases the risk of running out of RAM.

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 part of the 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. Then, if the particle overlaps with any of the other particle already in the box, it is rejected. If too many particles in a row are rejected, There is a "emergency break", and the codes throws an error. As this function only returns a 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 speedup of about 10 times, for 1000 particles and 10,000 steps 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 indices, so `lst[np.arange(N) != i]` gives an array with all the elements of `lst`, exept `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 on top of the box, 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 Figure 2.

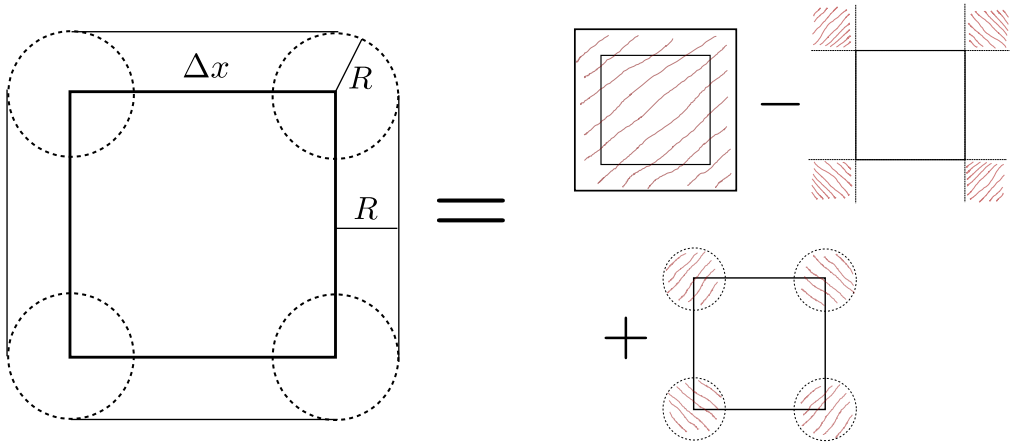


Figure 1: Checking if a disk of radius R is inside of a square of side length Δx, is equivalent to checking if the center of the disk is inside the shape on the left side. Thus, the task is reduced to checking if the center is inside the larger square of side lengths $R + 2\Delta x$, but not at the corners,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

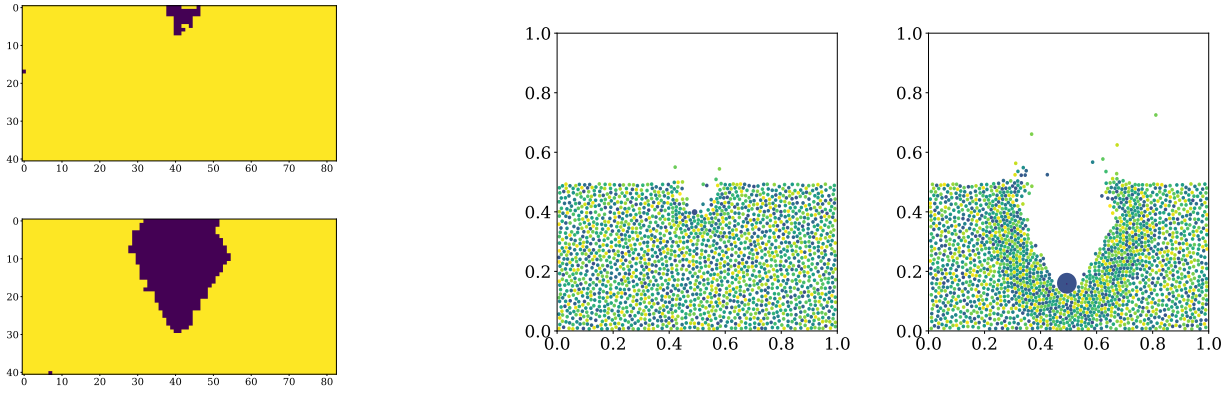
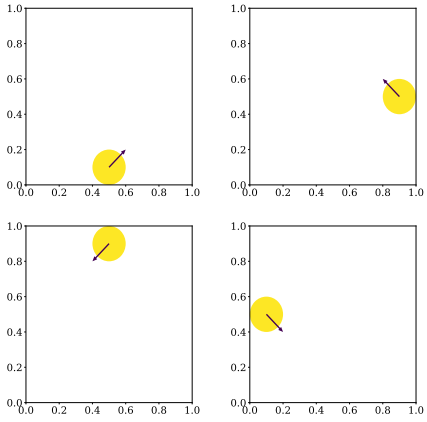


Figure 2: The grid cells with a particle inside is marked yellow, while the ones without a particle inside, i.e. those counted as making up the grid, are purple. The plot to the right shows the crater.

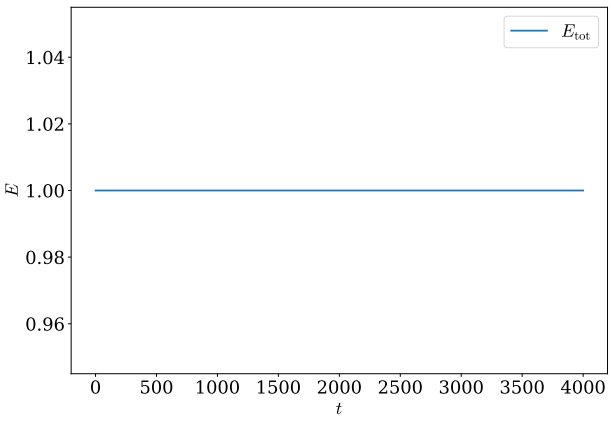
Tests

Several functions were developed to test the implementation for errors and bugs. 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 and the escape angle is $s(\theta) = a\cos(\theta/2)$, where a is the radius of the stationary particle [1]. 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.



(a) After 10000 steps, the disk still follows a regular pattern



(b) The energy of a single particle is constant.

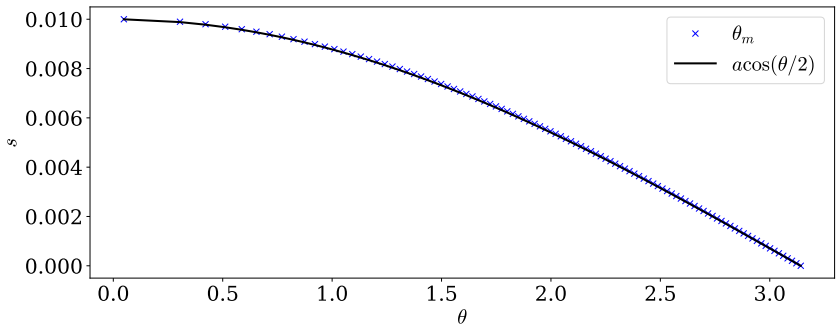


Figure 4: The impact parameter S, as a function of scattering angle. The dashed lines are the theoretical values, θ_m are the values from the simulation.

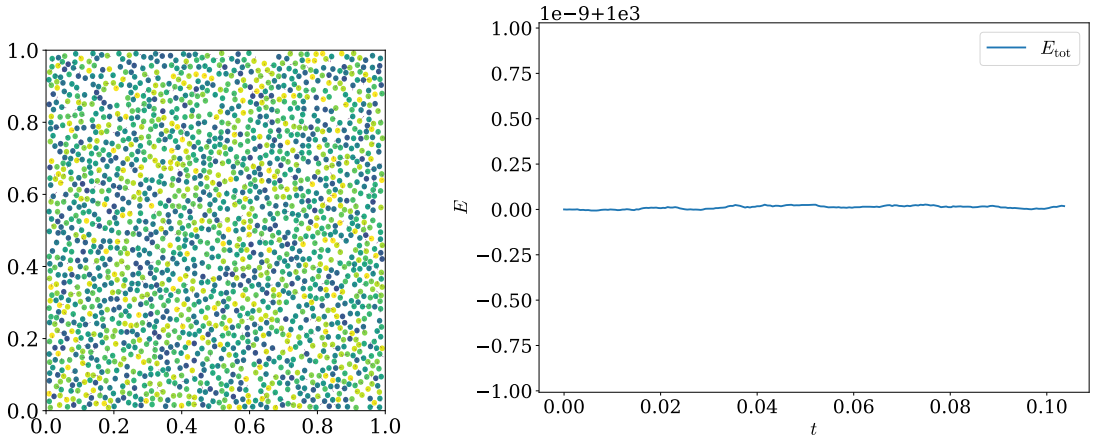


Figure 5: On the left is a snapshot of the particles. The arrows represent the velocities. On the right, the energy is plotted as a function of events. The energy loss can be seen to be negligible.

Results

Velocity distribution

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

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

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.

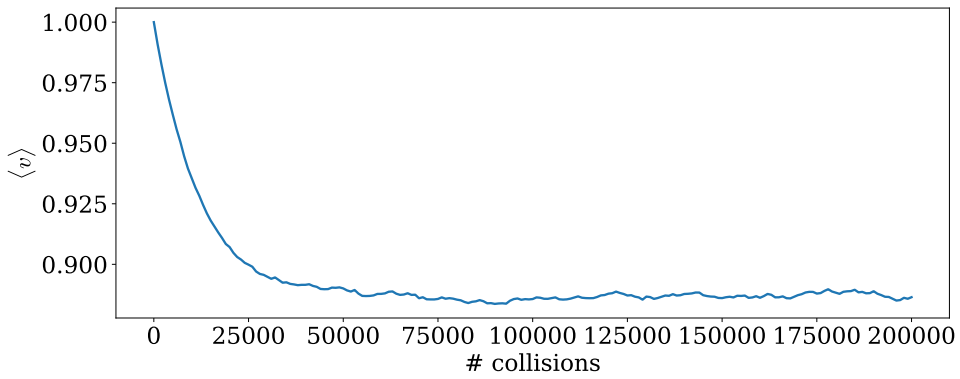


Figure 6: Average velocity, as a function of collisions. The distribution reaches equilibrium around 6000 collisions, or $3N$

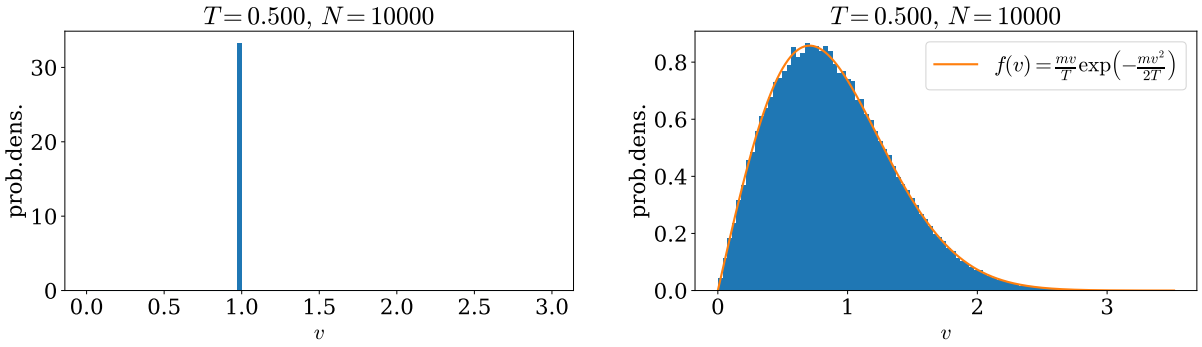


Figure 7: The velocity distribution is a good fit with the Maxwell-Boltzmann distribution, shown as a blue line.

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). Figure 9 shows how the average speed evolves over time. 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, as shown in Figure 10.

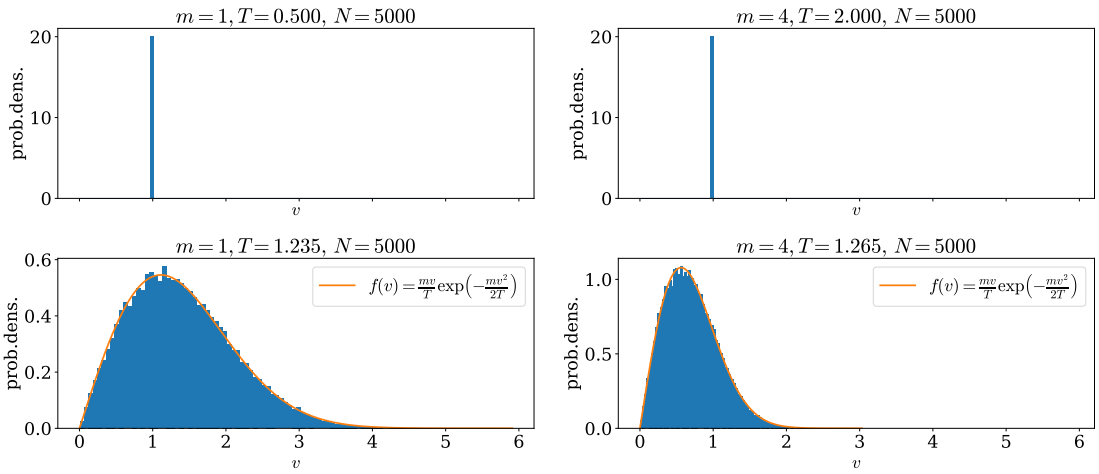


Figure 8: The velocity distribution of the particles with $m = 1$ and $m = 4$ is show left and right, resp. and compared to the Maxwell-Boltzmann distribution.

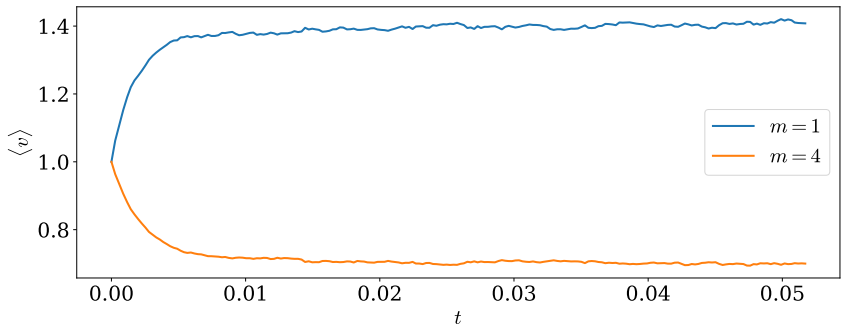


Figure 9: The velocity distribution of the particles with $m = 1$ and $m = 4$ is show left and right, resp. and compared to the Maxwell-Boltzmann distribution.

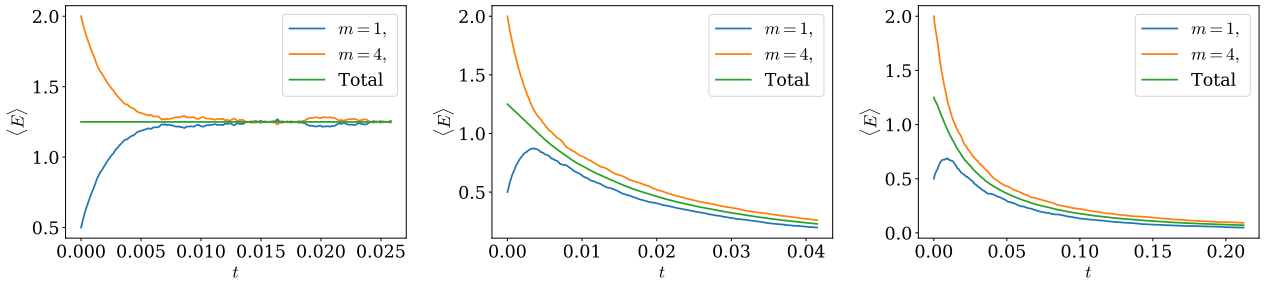


Figure 10: The average energy, as a function of time, of resp. $\xi = 1$, $\xi = 0.9$ and $\xi = 0.8$.

Projectile

To simulate the impact of a projectile, one particle 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 size of projectile on the crater, the simulation is run 10 times, using projectiles with different radius. The crater is measured as described in the implementation section. The mass of the projectile is proportional to R^2, so that the density is constant. The simulation uses the value $\xi = 0.5$, and it is run until the total energy of the system is 10% of the original value. There are placed out 2000 particles, of mass radius $R = 0.0063$ and mass R^2. This means the packing fraction is $2000 \cdot \pi R^2 / 0.5 \approx 0.500$. Figure 11 shows the size of the crater, as a function of the radius of the projectile. The relationship is close to linear.

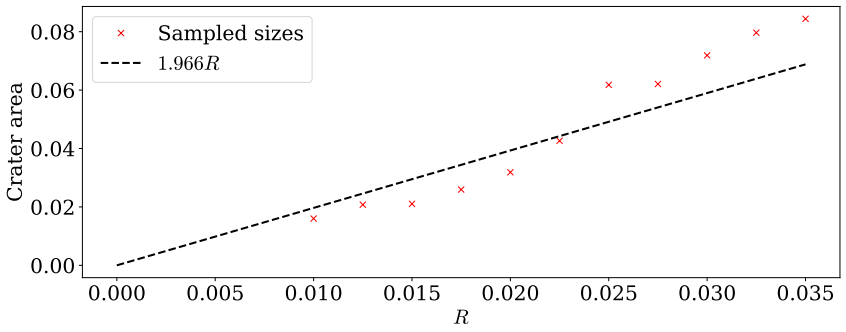


Figure 11: The size of the crater, as a function of the radius of the incoming projectile.

Discussion and conclusion

The simulation implemented yields accurate results. A good indication of this is how well the energy is conserved, as show by Figure 5. The fluctuations here are mostly due to numerical inaccuracies, an negligible for the purposes of these simulations. Discrepancies between the histograms and the Maxwell-Boltzmann distribution comes from the fact that the simulated system is finite, and thus subject to stochastic fluctuations away from perfect equilibrium. The most straight forward way to get better samples would be to simulate more particles, for longer periods of time. At this point, however, the simulation will become increasingly slow, even with the measures taken to speed it up. The larges bottleneck at this time is pushing the collisions to the priority queue. Some measures could be taken to limit this, for example given particle i, only push the next collision with, say j. Then, if j is involved in another collision, find particle i's next collision. This, however, might lead to a large amount of test reducing the possible gain.

References

[1] Jaakko Akola and Martin Johnsrud. *Exercises classical mechanics, exercise 5*. 2020. URL: `https://github.com/martkjoh/exercises_classical_mechanics`.

[2] Stefan Luding and Sean McNamara. "How to handle the inelastic collapse of a dissipative hard-sphere gas with the TC model". In: *Granular Matter* 1 (1998), pp. 113–128. DOI: `https://doi.org/10.1007/s100350050017`.

[3] NTNU, Institutt for Fysikk. *Exercise 1, TFY4235 Computational Physics*. 2021.