Event driven simulation of a 2-dimensional gas

Sondre Duna Lundemo†
Department of Physics, Norwegian University of Science and Technology, Trondheim
Norway
TFY4235 - Computational physics
(*Last updated on February 17, 2021*)

An event driven approach is used to simulate a gas of hard discs in two dimensions. The framework built is used to study statistical properties of the gas at and towards equilibrium. The numerical results are compared with well known results from classical statistical mechanics. Towards the end, the formation of a crater from a projectile impact on a bed of particles is studied with the gas system.

†sondre@duna.no

Contents

1 Introduction and background **2**

2 Overview of code **2**

3 Results and discussion **5**

3.1 Speed distribution at equilibrium 5

3.2 Mixture of two gases 6

3.3 Mixing in the presence of energy dissipation 6

3.4 Crater formation from projectile impact 7

3.4.1 Distributing the discs randomly in $[0, 1] \times [0, 0.5]$ 7

3.4.2 Illustration of crater formation 8

3.4.3 Dependence on projectile mass 8

1 Introduction and background

An event driven approach is used to simulate a gas of hard discs in two dimensions. The framework built is used to study statistical properties of the gas at and towards equilibrium. The numerical results are compared with well known results from classical statistical mechanics.

The central algorithm for event-driven simulations is the following [1]:

```
Set velocities and positions of all particles in the gas;
Choose a stop criterion;
for each particle in gas do
    Calculate if and when the particle will collide with all the other particles and
     the walls;
    Store all the collision times;
end
while not reached stop criterion do
    Identify the earliest collision;
    if collision is valid then
        Move all particles in straight lines until the earliset collision;
        for each particle involved in collision do
            Calculate if and when the particle will collide with all the other
             particles and the walls;
            Store all the collision times;
        end
    else
        Discard collsion;
    end
end
```

Algorithm 1: Event driven simulation of a gas.

In the above algorithm, a collision is *valid* if the particle(s) involved in the collision has *not* collided since the time the collision time was stored.

2 Overview of code

In this section I briefly present how the algorithm is implemented and which considerations are put into the choices of data-structures. The code is written in `python`.

The main machinery of the code is collected in a class called `Gas` in `events.py`, which in essence is a collection of N `particles` and various methods to manipulate the coordinates of each particle to simulate the gas' time evolution according to the algorithm in section 1 The easiest way to simulate the system is to initialise the gas by calling the constructor with the argument N giving the number of particles, setting the velocities by calling `gas.set_velocities(v)` and subsequently `gas.simulate()`. The particles are represented by their coordinates in the extended configuration space by their coordinates in the extended configuration space, that is their position and velocity. Although the main algorithm presented in the introduction 1 invites to a fully object-oriented approach, I

have chosen to restrain myself somewhat in that respect. For instance, I have chosen not to create an object representing a particle. The most prominent caveat preventing me from doing this is that it might affect the speed of the calculations. By choosing not to create separate object representing each particle, I found it very simple to move the particles at each time step and also accessing them by ordinary slicing and indexing of arrays. Although it is possible to overload operators such that an array of self defined object can be added *like* `numpy`-arrays, I found this to be impractically slow. A quick test of adding the self-made particles compared to simply adding $4 \times N$ arrays establishes this observation quite firmly. The listings below shows the time spent on adding two arrays of 50 000 particles with each of the mentioned methods.

```
%time particles_array_1 += particles_array_2
```

```
CPU times:  user 443 µs, sys:  101 µs, total:  544 µs Wall time:  306 µs
```

```
%time particles_class_1 += particles_class_2
```

```
CPU times:  user 34.1 ms, sys:  87 µs, total:  34.2 ms Wall time:  33.2 ms
```

The issue of finding the earliest event for each timestep is solved using a priority queue from the library `heapq` in `python`. This data structure allows for sorting any objects as long as they can be compared by the "less than"-operator. I have therefore chosen to make a class called `Event` to store the necessary information about each collision, and use these in the queue.

The code itself is well documented and should be easy to undestand on its own. However, I would like to point out some solutions that I found to work particularly well. The part of the code that undoubtedly is the most computationally heavy is the one devoted to calculating if and when the particles will collide with all others. The naive approach would be to iterate over each particle and do the calculation for each of them separately. In `python`, these kind of nested loops will often become impractically slow. I found considerable improvements through vectorising this calculation.

By essentially replacing the piece of calculation in listing 1 by that in 2 I was able to reduce one of the loops over all of the particles.

```
for j in range(self.N):
  if i != j:
    delta_x = self.particles[:2,j] - self.particles[:2,i]
    delta_v = self.particles[2:,j] - self.particles[2:,i]
    R_ij    = self.radii[i] + self.radii[j]
    d       = (delta_x @ delta_v)**2 - (delta_v @ delta_v) * ((delta_x
   @ delta_x) - R_ij**2)

    if delta_v @ delta_x < 0 and d > 0:
      new_t =  - (delta_v @ delta_x + np.sqrt(d))/(delta_v @ delta_v)
      heapq.heappush(self.events, Event(new_t + t ,i,j,"pair",self.
   count[i], self.count[j]))
```

Listing 1: Loop over all particles.

```
T = np.full(self.N - 1, np.inf)
mask = np.arange(self.N-1)
mask[i:] += 1
```

```
4
5  r_ij = self.radii[mask] + self.radii[i]
6
7  delta_x = self.particles[:2,mask] - self.particles[:2,i][:,None]
8  delta_v = self.particles[2:,mask] - self.particles[2:,i][:,None]
9
10 vv = np.einsum('ij,ij->j',delta_v,delta_v)
11 vx = np.einsum('ij,ij->j',delta_v,delta_x)
12 xx = np.einsum('ij,ij->j',delta_x,delta_x)
13
14 d = vx ** 2 - vv * ( xx - r_ij**2 )
15
16 c_mask = (vx < 0 ) * (d > 0)
17
18 T[c_mask] = - ( vx[c_mask] + np.sqrt(d[c_mask]) )/(vv[c_mask])
19
20 T = T[c_mask]
21 J = mask[c_mask]
22
23 for j in range(np.size(T)):
24   heapq.heappush(self.events,Event(T[j] + t ,i,J[j],"pair",self.count[i
     ], self.count[J[j]]))
```

Listing 2: Vectorized calculation.

Putting the central pieces[1] of each of these two calculations into two functions `loop()` and `vect()` and comparing the time spent shows that the latter is approximately 100 times faster than the former, when doing a test on 50 000 particles. It should be noted however that the latter also requires a separate loop for pushing the new events into the queue, whereas the former does not. The speed-up is probably somewhat smaller than shown here.

```
1 %timeit loop()
```

```
52.4 ms ± 260 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
1 %timeit vect()
```

```
429 µs ± 23.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

An explaination is probably appropriate for the somewhat cryptic `einsum`-function from `Numpy`. This function allows for writing sum operations on arrays using Einstein's summation convention. Once I have gotten used to the notation, I find it very readable. In addition it is very fast; the listed code in 2 below produces the following results:

```
55.2 µs ± 364 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
89.6 µs ± 4.78 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
1 u = np.random.random((2,50000))
2 v = np.random.random((2,50000))
3
4 %timeit np.einsum('ij,ij->j',u,v)
5 %timeit np.sum( u*v, axis = 0)
```

[1]The *central* part here is referring to line 1 through 6 in listing 1 and line 1 through 14 in listing 2.

In this example I use the `einsum` function to what is more explicitly stated in the other code-line

3 Results and discussion

In this section, I will present the results from the simulations in the problems. For a more detailed description of the problems consult the exercise sheet [2].

3.1 Speed distribution at equilibrium

The first test is devoted to investigating the distribution of the velocities of the particles after sufficiently long time. From statistical mechanics, we know that the velocities of a 2-dimensional gas at equilibrium will distribute according to Maxwell-Boltzmann's velocity distribution:

$$p(v) = \frac{mv}{k_B T} \exp\left(-\frac{mv^2}{2k_B T}\right), \tag{1}$$

where k_B is Boltzmann's constant, T the absolute temperature, and m the mass of the particles. It is evident that the particles will have to have the same mass for comparing with the known distribution in equation 1. The case of dissimilar masses is considered in section 3.2 and 3.3.

We initialise an system of particles with velocities $\mathbf{v} = [v_0 \cos\theta, v_0 \sin\theta]$ with $\theta \sim \mathcal{U}[0, 2\pi]$. The initial distribution of the velocities is therefore $\delta(v - v_0)$. After simulating the system until equilibrium is reached, i.e. the the average number of particle collision is $\gg 1$, the distribution is as shown in figure 1. For this simulation, the stop criterion used was that the average number of collisions surpassed 50. To get a larger number of independent samples, I have used the speeds at 20 different times in this simulation sampled such that all particles have collided a few times between each sample.

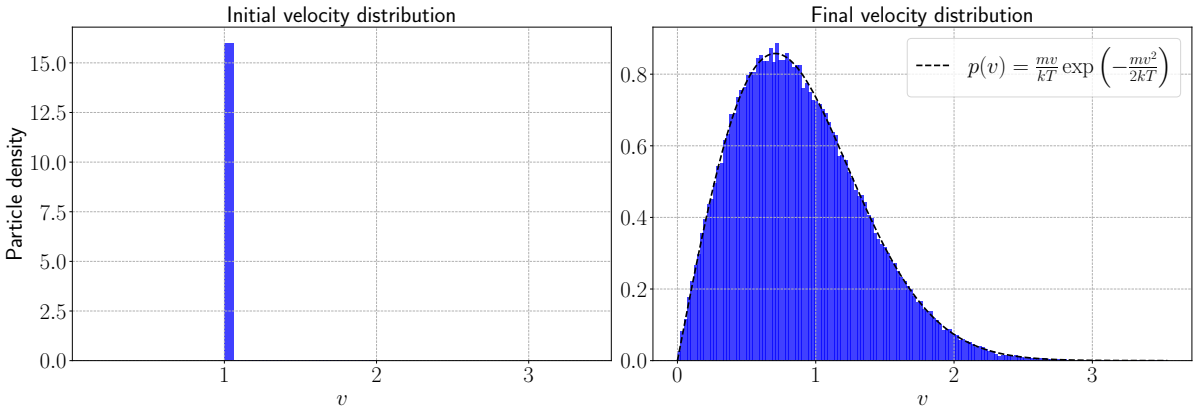

Figure 1: Distribution of velocities in a gas of 50000 particles.

To have a more quantitative measure of how good the fit is, divide the real line into k intervals and consider the normalized ℓ^1-norm of the difference at each interval

$$\texttt{err} = \frac{1}{2} \sum_{j=1}^{k} |d_j - p_j|, \tag{2}$$

where d_j denotes the density of samples in interval I_j and

$$p_j = \int_{I_j} p(v)\,\mathrm{d}v$$

is the probability that an observation falls into interval I_j. It is evident from equation 2 that $\texttt{err} \in [0, 1]$. For the distribution shown in figure 1 $\texttt{err} \approx 0.0078527$. This indicates a good fit with the known distribution.

3.2 Mixture of two gases

We repeat the exact same procedure as in section 3.1, except that we make half of the masses 4 times as big as the rest. The velocity distributions after equilibrium is reached is plotted in 2.

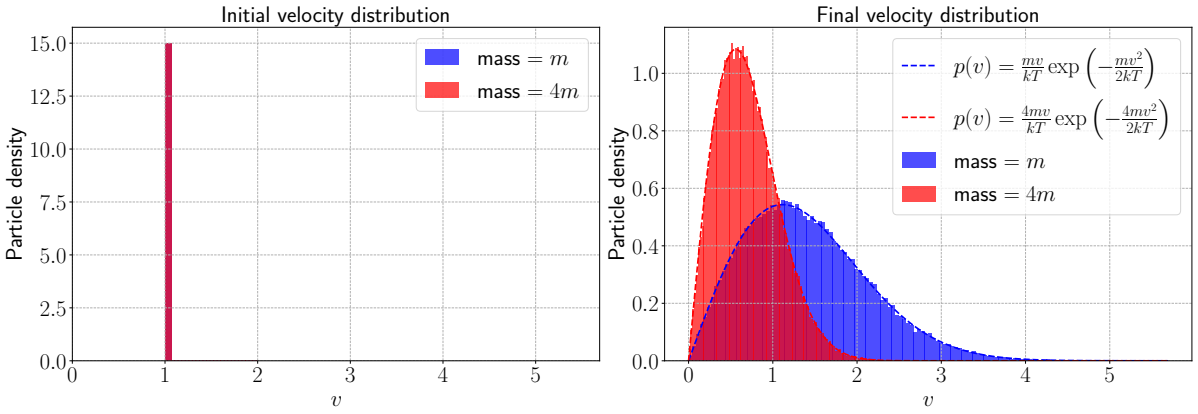

Figure 2: Distribution of velocities in a gas of 50000 particles. The red distributions correspond to the heavy particles, while the blue correspond to the lighter particles.

The average speed and kinetic energy is shown in table 1. These results suggests that although the masses are different, the mixture of the gases eventually reach equilibrium. The fact that the two gases reaches equilibrium is however most easily seen by actually plotting the time evolution of the average kinetic energy. This is shown in section 3.3, where we also simulate situations where this is *not* the case.

Table 1: Average speed and kinetic energy for the light and heavy particles

Mass	Average speed	Average kinetic energy
m	1.400236	12.466584
$4m$	0.702230	12.533416

3.3 Mixing in the presence of energy dissipation

To investigate, at least visually, whether the gas mixture reaches equilibrium we plot the time evolution of the averages of the energy of each species. In addition, we include two similar cases with the restitution parameter ξ set to 0.9 and 0.8. This is shown in figure 3. By the very definition of thermal equilibrium, e.g. the one given in [3, p. 3]

> Thermodynamic equilibrium prevails when the thermodynamic state of the system does not change with time.

it is clear that the system cannot reach equilibrium in the presence of energy dissipation.

The plot in figure 3 shows that the two different particle species reach equilibrium only in the case where the energy is conserved. Furthermore, it is seen that the heavy gas molecules are always at higher temperature in the two latter cases. Although this fact is easily realised from elementary statistical mechanics, demonstrating it through these simulations is a good check that the correct physics is contained within the model made.

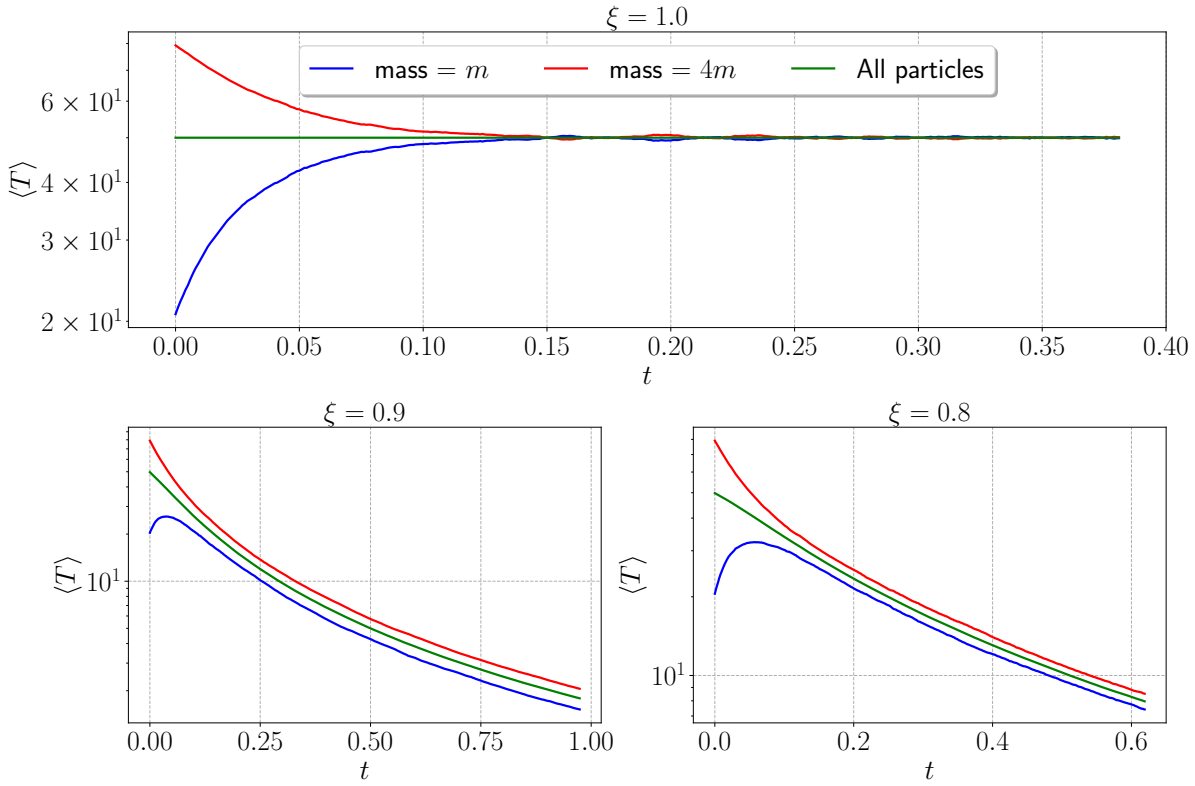

Figure 3: Time evolution of energy averages for restitution parameter $\xi = 1.0, 0.9, 0.8$. The data comes from an ensemble of 8 gases with 4000 particles each.

3.4 Crater formation from projectile impact

3.4.1 Distributing the discs randomly in $[0, 1] \times [0, 0.5]$

To distribute the particles randomly in the box defined by

$$\{(x, y) \in \mathbb{R}^2 : 0 < x < 1, 0 < y < 0.5\}$$

I use the following scheme.

To choose a suitable radius to have the particles fill up the available space with a packing fraction of $\rho \approx 1/2$ I solve for r in

$$N\pi r^2 = A(r)\rho = \frac{1}{2}A(r),$$

where, in order to avoid having particles outside the box, the area depends on r. If we clear a border of r on the boundary of the region facing the walls, we get r from solving

$$N\pi r^2 = \frac{1}{2}\left(1 - \frac{1}{2}r\right)(1-2r) \quad \Rightarrow \quad r = \frac{-1+\sqrt{N\pi}}{2(N\pi-1)}. \tag{3}$$

To avoid having overlapping particles, I do the following

Choose number of particles N;
Find r corresponding to N from (3);
$\mathbf{x} \leftarrow [(0,0),\ldots,(0,0)]$;
Sample $x_i \sim \mathcal{U}_{[r,1-r]}$ and $y_i \sim \mathcal{U}_{[r,0.5]}$;
$\mathbf{x}_1 \leftarrow (x_i, y_i)$;
for $i = 2 \ldots N$ **do**
  Sample $x_i \sim \mathcal{U}_{[r,1-r]}$ and $y_i \sim \mathcal{U}_{[r,0.5]}$;
  $\mathbf{x}_i \leftarrow (x_i, y_i)$;
  while *Particle i does not overlap with particle* $1,\ldots,i-1$ **do**
    Sample $x_i \sim \mathcal{U}_{[r,1-r]}$ and $y_i \sim \mathcal{U}_{[r,0.5]}$;
    $\mathbf{x}_i \leftarrow (x_i, y_i)$;
  end
end

Algorithm 2: Non-overlapping random placement of discs in rectangular region.

3.4.2 Illustration of crater formation

The plots in figures 4 and 5 shows the crater formed when the mass of the projectile is 5 and 25 times the mass of the particles in the bed, respectively. The darker the colour of the particles, the more collisions they have been involved in. Define the *size of the crater*, $\mathcal{S}$, as the number of affected particles by the impact. That is, the number of particles moved during the impact. In the illustrations in figures 4 and 5 the size is the number of non-yellow particles.

3.4.3 Dependence on projectile mass

By simulating crater formation with projectile masses running from 1 to 25 times the mass of the remaining particles, the size of the crater is as shown in figure 6. The size of the crater $\mathcal{S}$ is simply the number of particles involved in the crater formation. The plot clearly shows that a larger projectile mass gives rise to a larger crater, as one intuitively would expect.

Interestingly, the size seems to scale approximately linearly with the projectile mass. A reasonable suspicion to make is that the size of the crater depends on the energy transferred to it. If this is the case, it would explain the linear dependence on the mass. Following this assumption, one should expect the size to scale quadratically with the initial velocity. However, when trying to demonstrate this I found no such relationship.

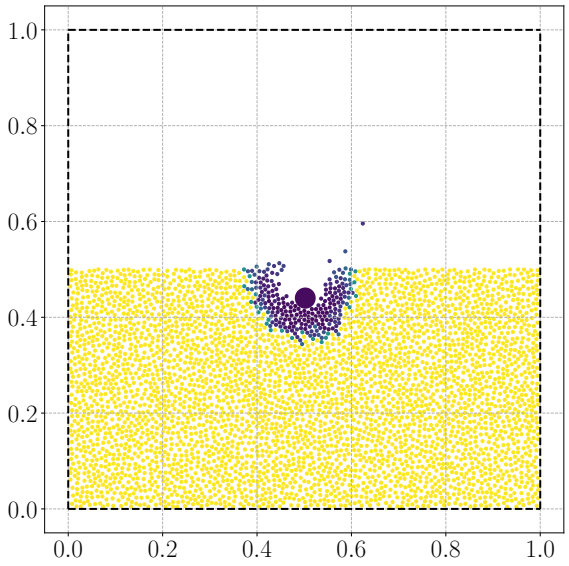

Figure 4: Example of crater formation using a projectile mass 5 times the mass of the remaining particles.

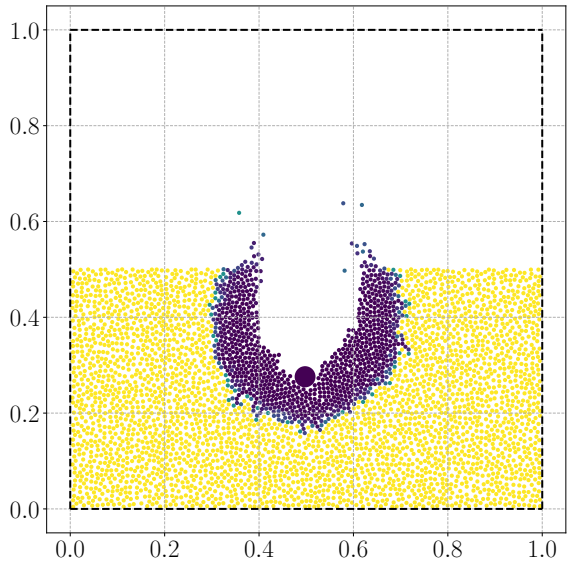

Figure 5: Example of crater formation using a projectile mass 25 times the mass of the remaining particles.

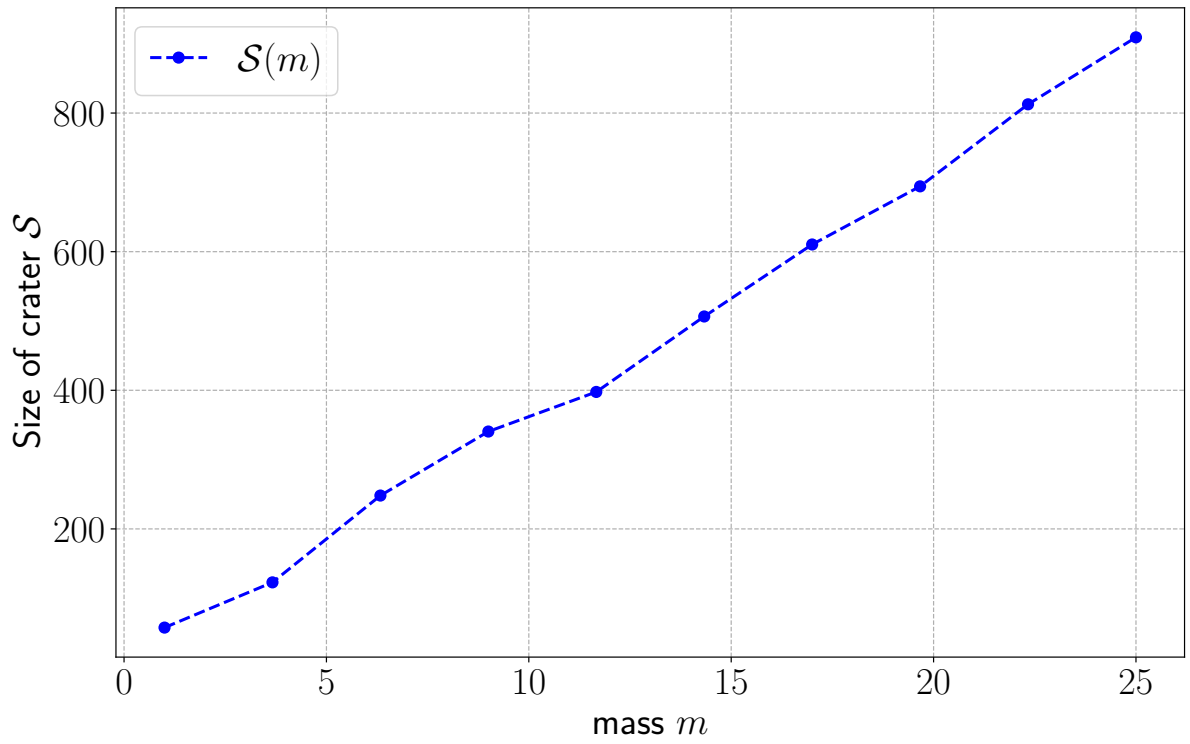

Figure 6: Size of crater as a function of projectile mass. The values are calculated from the mean of simulating projectile impact on 8 ensembles of a bed of 2000 particles.

References

[1] Aleksandar Donev, Salvatore Torquato, and Frank H. Stillinger. Neighbor list collision-driven molecular dynamics simulation for nonspherical hard particles. i. algorithmic details. *Journal of Computational Physics*, 202(2):737 – 764, 2005.

[2] Tor Nordam. Exercise 1 - Computational physics TFY4235, 2021.

[3] Kerson Huang. *Statistical Mechanics*. John Wiley & Sons, 2 edition, 1987.