Event Driven Simulation Project : Moving Wall

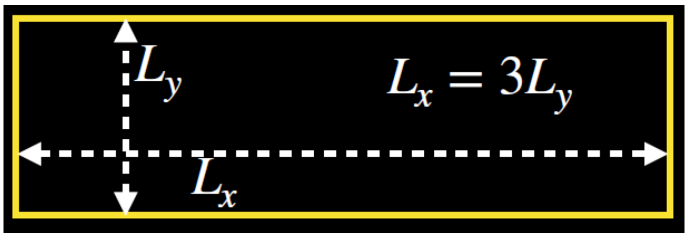
AVALLE Domitille
EA Eric

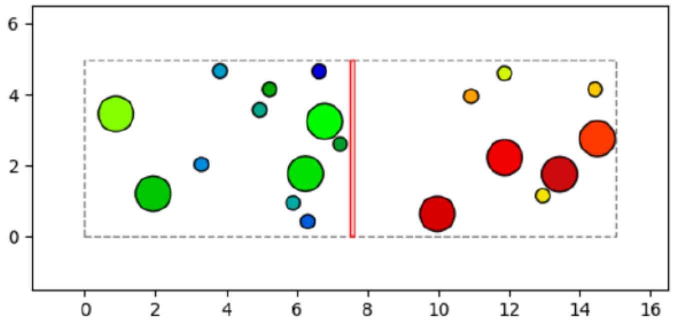
Pr. KLMASER Juliane

Table des matières

1 Description of the physical system

The system is made of particles interacting with each other through elastic collisions. It is defined by a rectangle box and separated in two chambers by a vertical moving wall. Its mass is finite and its speed is null along the y axis. This wall also makes elastic collisions with particles.





The left chamber's initial temperature and particle density is higher than the right chamber's. Density is calculated as the ratio of the total mass of particles in a chamber divided by it surface using the following formula :

$$T_j = \frac{E_j}{N_j k_B} \tag{1}$$

$$T_j = \frac{\sum_{q=1}^{N_j} \frac{m_q}{2} \vec{v}_q^2}{N_j k_B} \tag{2}$$

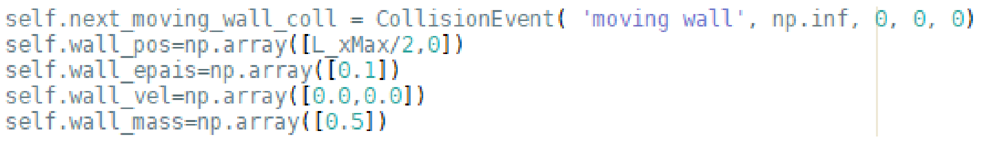
with N_j and T_j the number of particules and temperature in chamber $j \in [1, 2]$.

Our goal is to study the establishment of an equilibrium state with different parameters and its characterisitcs

2 Approach taken

2.1 Creation of a moving wall

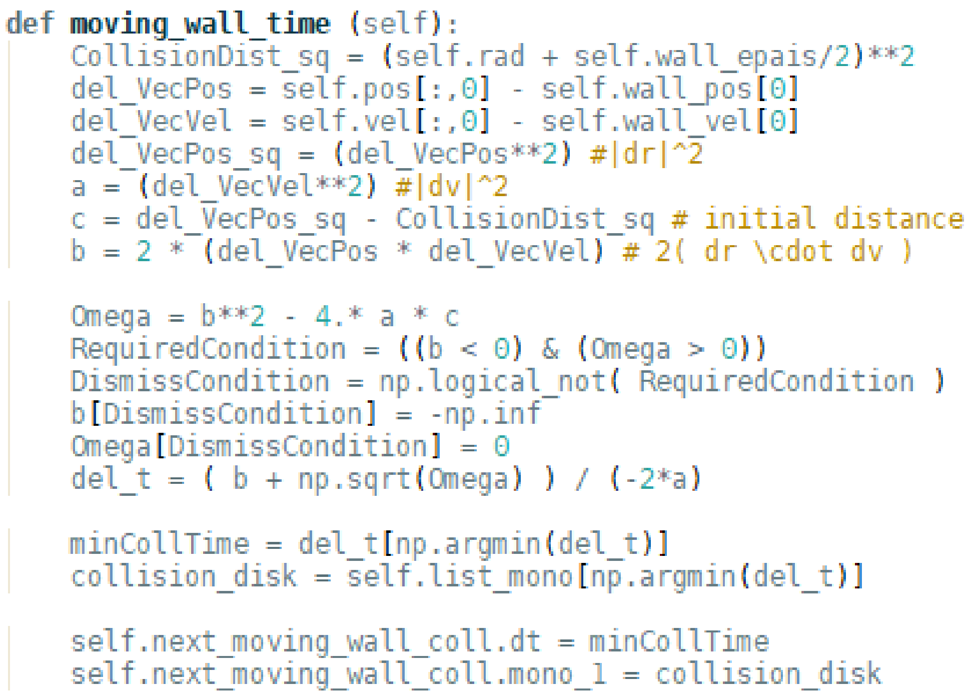
To simulate such a system, we have reexploited the code made for the event-driven simulation of monomers and added a wall with finite mass. It is characterized by its position, mass, thickness, and speed :



```
self.next_moving_wall_coll = CollisionEvent( 'moving wall', np.inf, 0, 0, 0)
self.wall_pos=np.array([L_xMax/2,0])
self.wall_epais=np.array([0.1])
self.wall_vel=np.array([0.0,0.0])
self.wall_mass=np.array([0.5])
```

2.2 Study of the impact between the monomers and the moving wall

In addition to the shocks between monomers and the shocks between the box and the monomers, we must also add the elastic shocks between the particles and the moving wall. For this purpose, we have created a new function that calculates the date of the next collision between a moving wall and the nearest particle :



```python
def moving_wall_time (self):
    CollisionDist_sq = (self.rad + self.wall_epais/2)**2
    del_VecPos = self.pos[:,0] - self.wall_pos[0]
    del_VecVel = self.vel[:,0] - self.wall_vel[0]
    del_VecPos_sq = (del_VecPos**2) #|dr|^2
    a = (del_VecVel**2) #|dv|^2
    c = del_VecPos_sq - CollisionDist_sq # initial distance
    b = 2 * (del_VecPos * del_VecVel) # 2( dr \cdot dv )

    Omega = b**2 - 4.* a * c
    RequiredCondition = ((b < 0) & (Omega > 0))
    DismissCondition = np.logical_not( RequiredCondition )
    b[DismissCondition] = -np.inf
    Omega[DismissCondition] = 0
    del_t = ( b + np.sqrt(Omega) ) / (-2*a)

    minCollTime = del_t[np.argmin(del_t)]
    collision_disk = self.list_mono[np.argmin(del_t)]

    self.next_moving_wall_coll.dt = minCollTime
    self.next_moving_wall_coll.mono_1 = collision_disk
```

During the impact between a monomer and the moving wall, the velocities of the wall and the monomer are changed according to the following conditions :

- The momentum is conserved
- Kinetic energy is conserved
- The moving wall maintains zero velocity along y

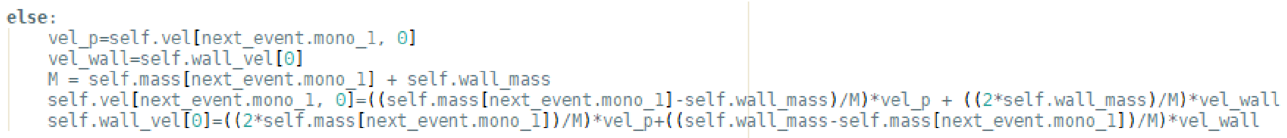
We therefore write the following system :

$$\begin{cases} \frac{1}{2}(m_p v_p^2 + m_w v_w^2) = \frac{1}{2} m_p (v_p^{'2} + m_w v_w^{'2}) \\ m_p \vec{v}_p + m_w \vec{v}_w = m \vec{v}_p^{'} + m \vec{v}_p^{'} \end{cases}$$

m_p, m_w, $\vec{v}_p$, $\vec{v}_w$ are respectively the mass of the monomer, the mass of the wall, the velocity of the monomer, the velocity of the wall before the impact and after the impact.
Thus :

$$\begin{cases} \vec{v}_p^{'} = \frac{m_p - m_w}{m_p + m_w} \vec{v}_p + \frac{2 \times m_w}{m_p + m_w} \vec{v}_w \\ \vec{v}_w^{'} = \frac{m_w - m_p}{m_p + m_w} \vec{v}_w + \frac{2 \times m_p}{m_p + m_w} \vec{v}_p \end{cases}$$

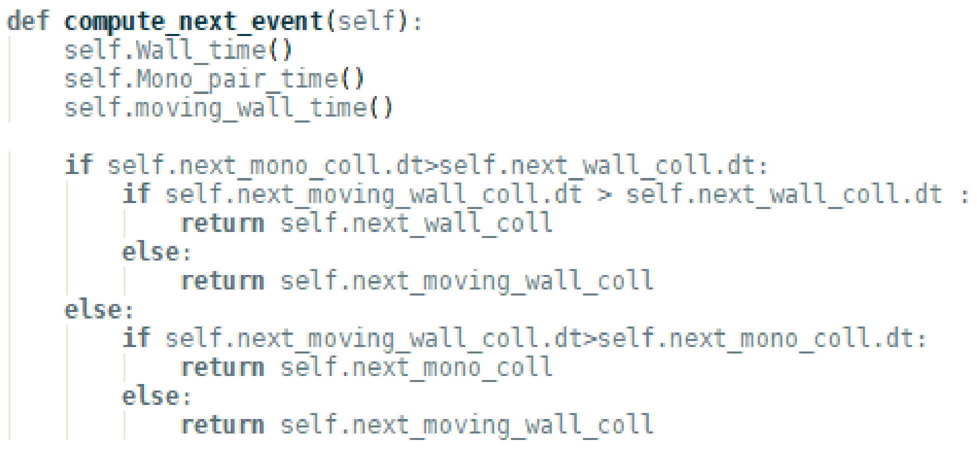
This is written in Python :



```python
else:
    vel_p=self.vel[next_event.mono_1, 0]
    vel_wall=self.wall_vel[0]
    M = self.mass[next_event.mono_1] + self.wall_mass
    self.vel[next_event.mono_1, 0]=((self.mass[next_event.mono_1]-self.wall_mass)/M)*vel_p + ((2*self.wall_mass)/M)*vel_wall
    self.wall_vel[0]=((2*self.mass[next_event.mono_1])/M)*vel_p+((self.wall_mass-self.mass[next_event.mono_1])/M)*vel_wall
```

The "compute next event" function has also been modified to take into account the three possible events :

– The collision between two monomers
– The collision between a monomer and one of the walls of the box
– The collision between a monomer and the moving wall



```
def compute_next_event(self):
    self.Wall_time()
    self.Mono_pair_time()
    self.moving_wall_time()

    if self.next_mono_coll.dt>self.next_wall_coll.dt:
        if self.next_moving_wall_coll.dt > self.next_wall_coll.dt :
            return self.next_wall_coll
        else:
            return self.next_moving_wall_coll
    else:
        if self.next_moving_wall_coll.dt>self.next_mono_coll.dt:
            return self.next_mono_coll
        else:
            return self.next_moving_wall_coll
```

2.3 Initialization of the positions and speeds of the monomers

Here we have reused the monomer initialization function and split it into two parts to intialize the right and the left chamber separately, with the left chamber having a higher density than the right one. (cf. Annex)

2.4 Temperature and density display

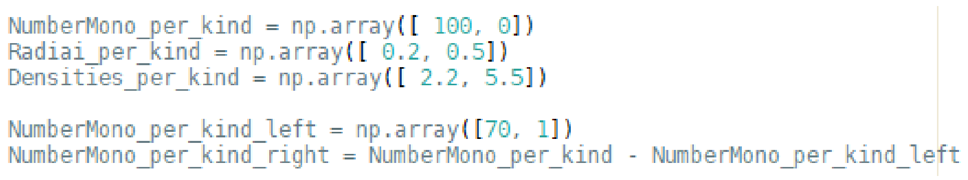
We then displayed the evolution of the wall position, temperatures and densities in the two chambers.

3 Results

We modified the different possible parameters (number, size, mass of the particles, mass of the wall and time of the simulation) to study the different behaviour of the system.

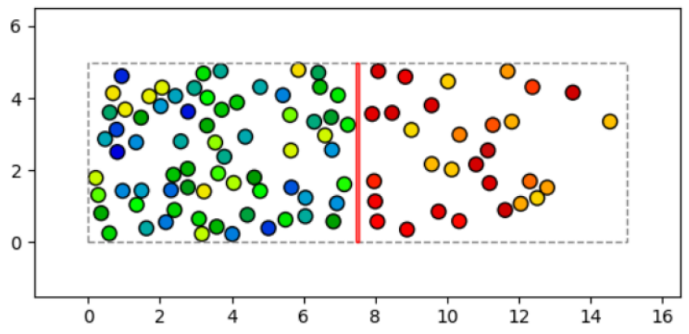
3.1 Position of the moving wall, temperature in both chambers and densities in both chamber

Parameters of the simulation :
k_BT_right = 1
k_BT_left= 5
wall_mass=0.5
NumberOfFrames = 2000

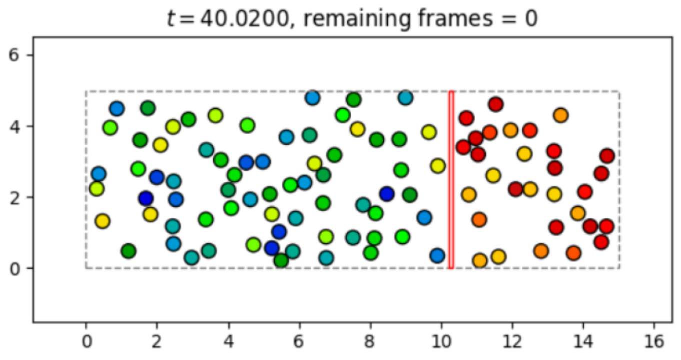


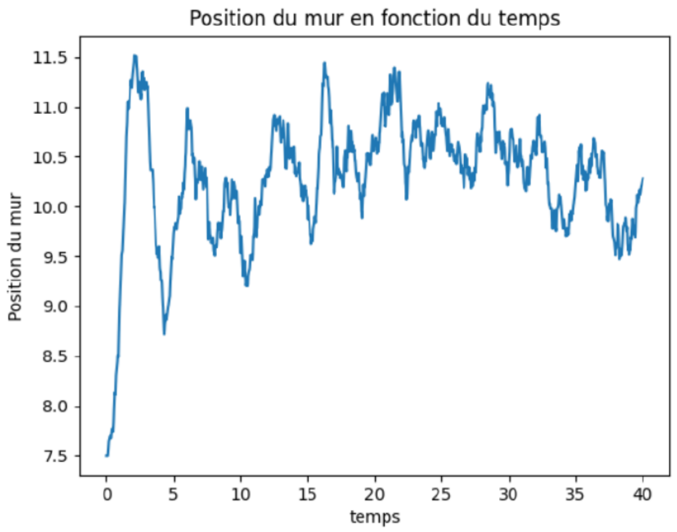
```
NumberMono_per_kind = np.array([ 100, 0])
Radiai_per_kind = np.array([ 0.2, 0.5])
Densities_per_kind = np.array([ 2.2, 5.5])

NumberMono_per_kind_left = np.array([70, 1])
NumberMono_per_kind_right = NumberMono_per_kind - NumberMono_per_kind_left
```

Initial state : the barrier is placed in the centre of the figure.



Final state :



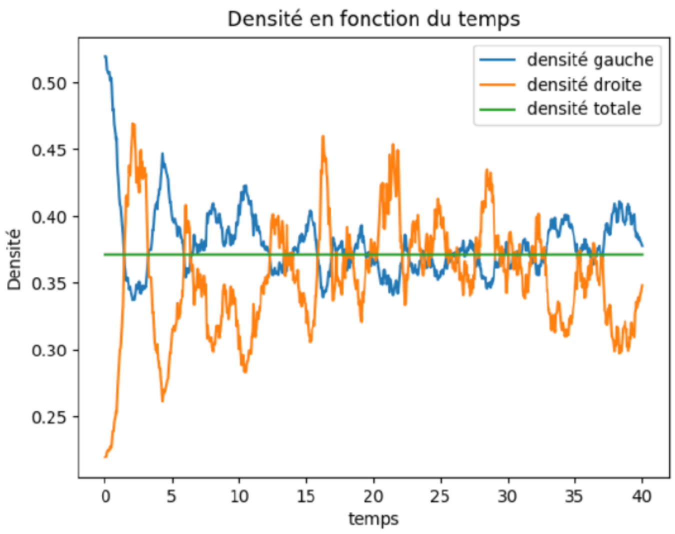


A steady state is observed. During this steady state, the position of the wall oscillates around the position 10.5. For monomers of all sizes, the final wall position corresponds to :

$$pos_{wall}^{final} = L_x \frac{N_{left}}{N_{total}} \tag{3}$$

$$AN \quad : \quad pos_{wall}^{final} = 15 \times \frac{70}{100} = 10.5 \tag{4}$$

with L_x the length of the box, N_{left} the number of monomers in the left chamber and N_{total} the total number of monomers.

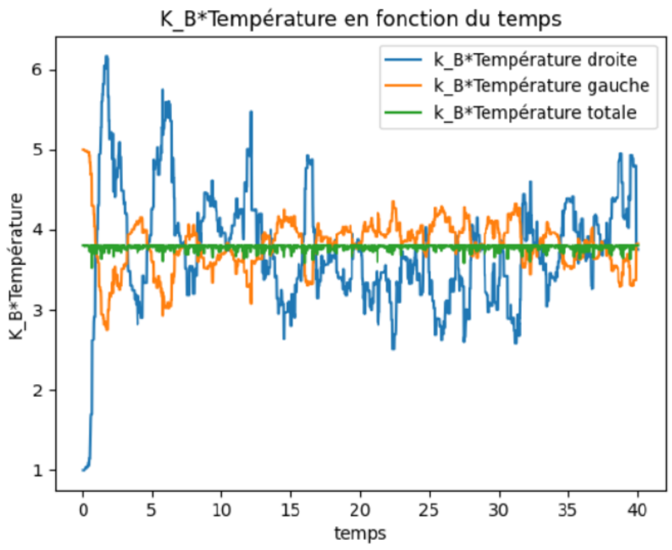


The density of the two chambers converge. It would appear that this value is the total density value of the system calculated as follows :

$$D = \frac{m_{total}}{S} \tag{5}$$

$$D = \frac{N \times m}{S} \tag{6}$$

with D, m_{total}, m, N, and S respectively the density, the total mass, the mass of a monomer, the number of monomers and the surface area of the system.
The wall has been excluded from the calculation here.

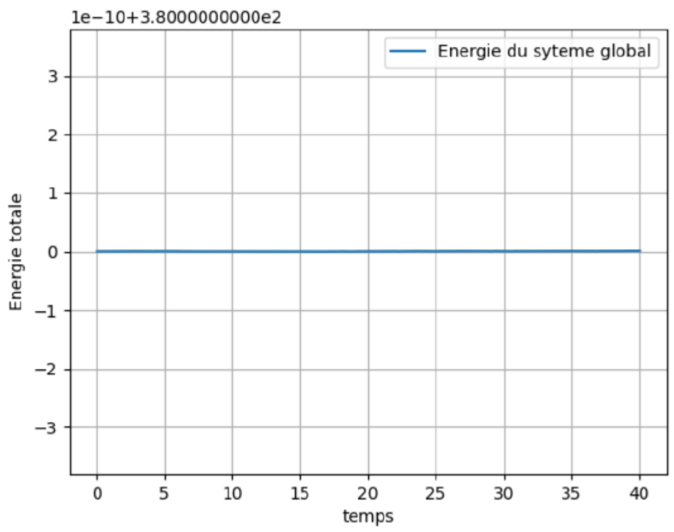


One can notice that the $k_B T$ of the two chambers converge to the same value.

We have computed the temperature of the system with the following formula :

$$T = \frac{\sum_{q=1}^{N} (\frac{m_q}{2} \vec{v_q}^2)}{N_j \times k_B} \tag{7}$$

with N the total number of monomers in the system.



The energy of the system remains unchanged.

3.2 Characterisation of the steady state

The first simulation resulted in the following steady state assumptions :

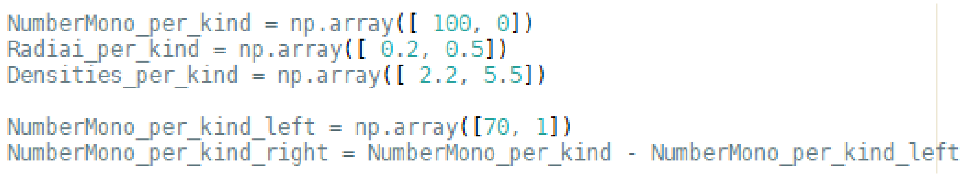
- the temperatures of the two chambers fluctuate around the average temperature of the system.

- the density of the two chambers oscillate around the average density of the system.

We seek to give more credence to these assumptions by running different simulations with different parameters.

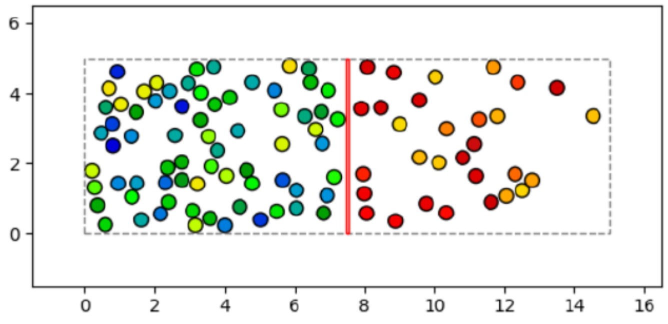
3.3 Influcence of the mass of the moving wall on the time evolution and the steady state

k_BT_right = 5
k_BT_left= 1
wall_mass=1 (previous value : 0.5)
NumberOfFrames = 2000

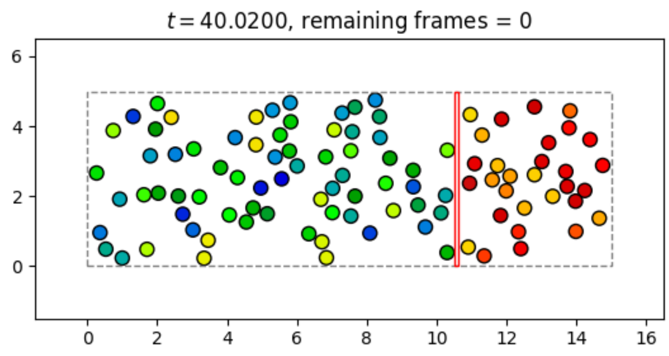


```
NumberMono_per_kind = np.array([ 100, 0])
Radiai_per_kind = np.array([ 0.2, 0.5])
Densities_per_kind = np.array([ 2.2, 5.5])

NumberMono_per_kind_left = np.array([70, 1])
NumberMono_per_kind_right = NumberMono_per_kind - NumberMono_per_kind_left
```

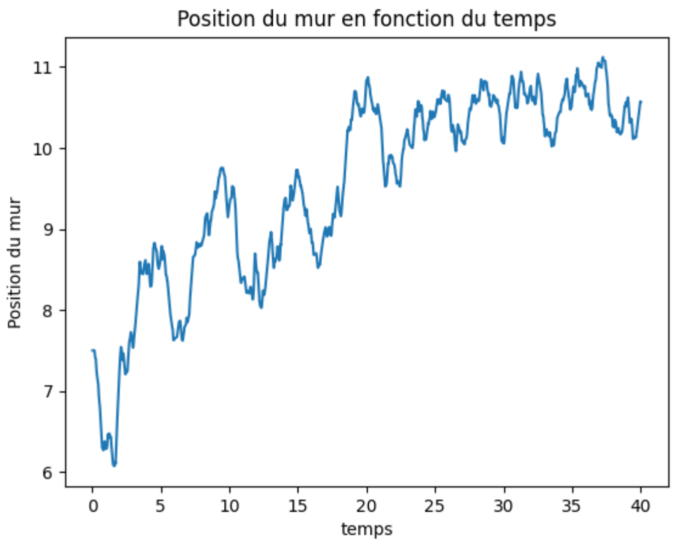
Initial state :



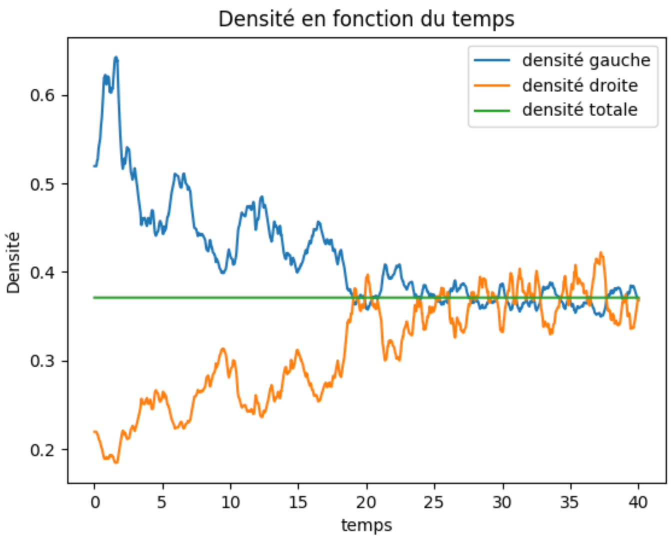
Final state :



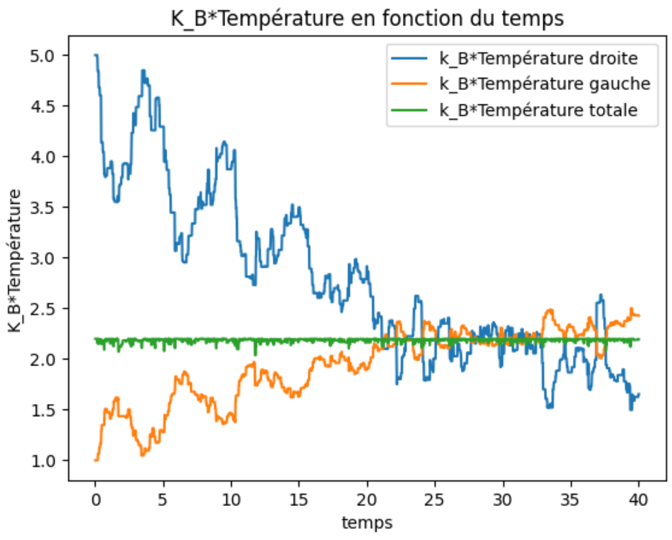
$t = 40.0200$, remaining frames = 0



Position du mur en fonction du temps
Position du mur
temps



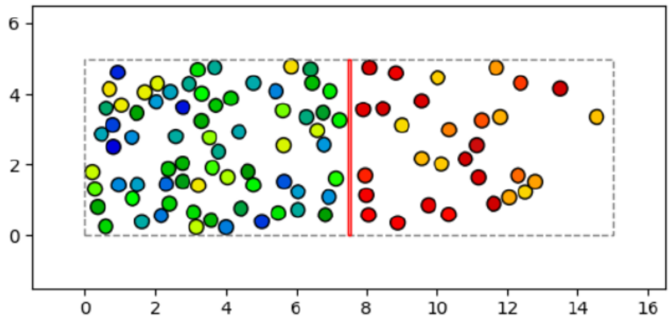
Densité en fonction du temps
densité gauche
densité droite
densité totale
Densité
temps



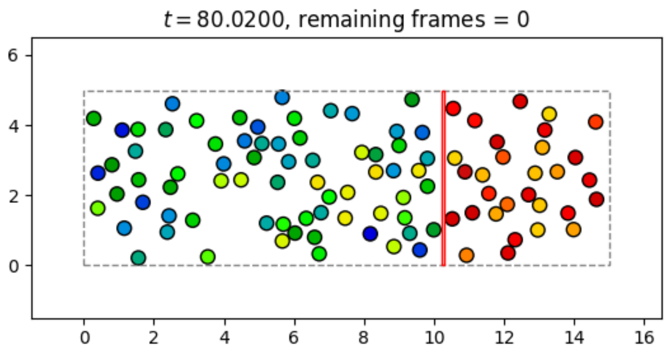
We lauch a second simulation, with an even more important mass.

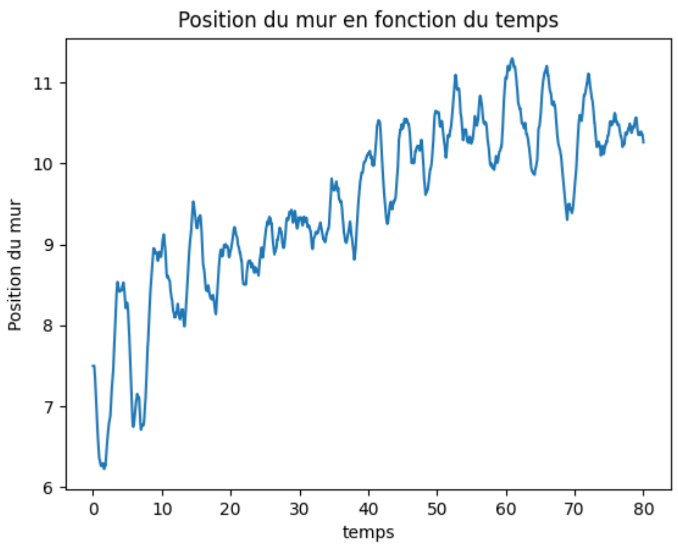
k_BT_right = 5
k_BT_left= 1
wall_mass= 4 (previous value : 1)
NumberOfFrames = 4000

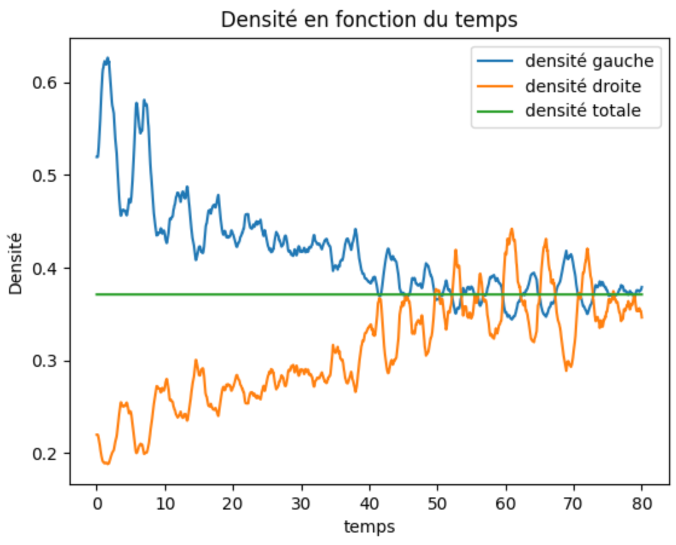
Initial state :

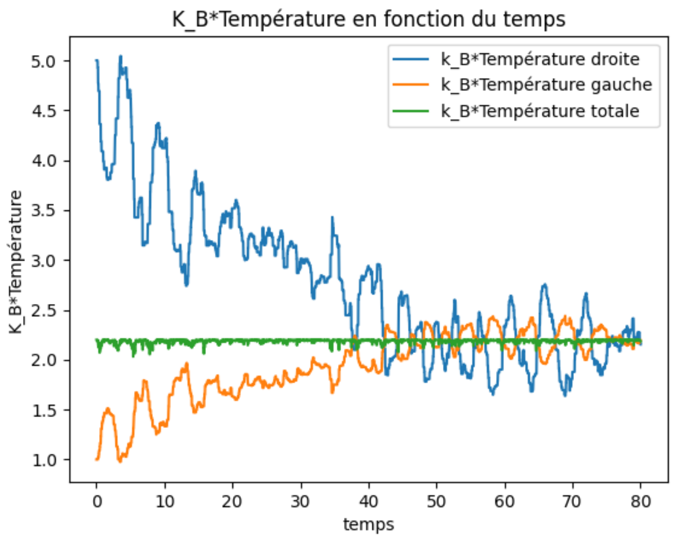


Final state :









It can be seen that the time taken to reach the steady state increases with the mass of the wall. However, the steady state characteristics remain the same with a decrease in the amplitude of the oscillations of the wall position, densities and temperatures.

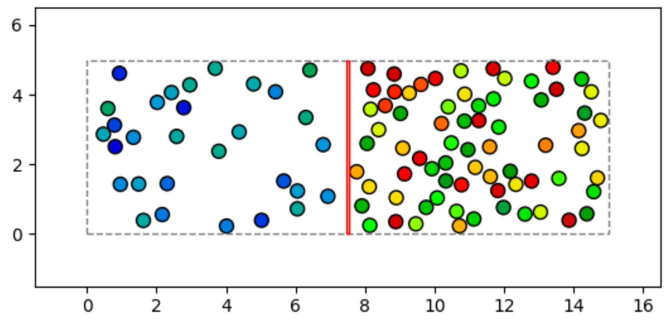
3.4 Do we reach the same steady state if we exchange the two chambers ?

The role of the two chambers is now being exchanged.

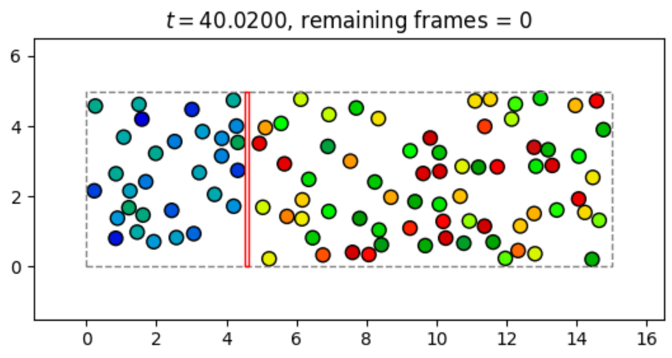
k_BT_right = 5
k_BT_left= 1
wall_mass=0.5
NumberOfFrames = 2000

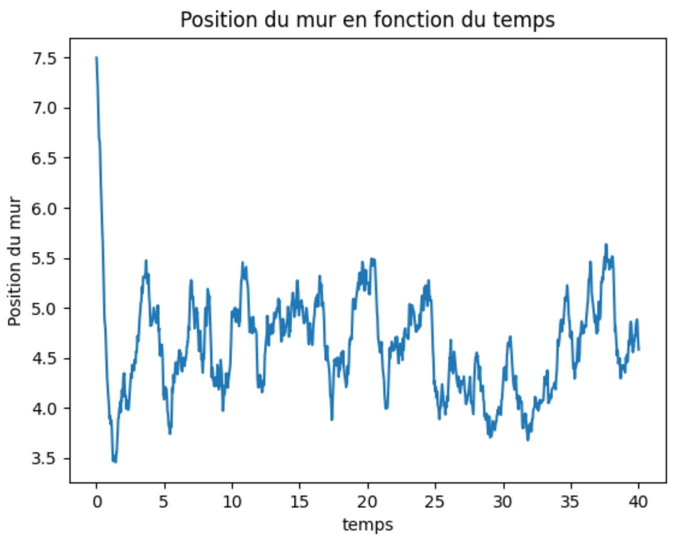
```
NumberMono_per_kind = np.array([ 100, 0])
Radiai_per_kind = np.array([ 0.2, 0.5])
Densities_per_kind = np.array([ 2.2, 5.5])

NumberMono_per_kind_left = np.array([30, 1])
NumberMono_per_kind_right = NumberMono_per_kind - NumberMono_per_kind_left
```

The following results are then obtained, which are identical to those obtained in the first simulation as one would expect.
Initial state :

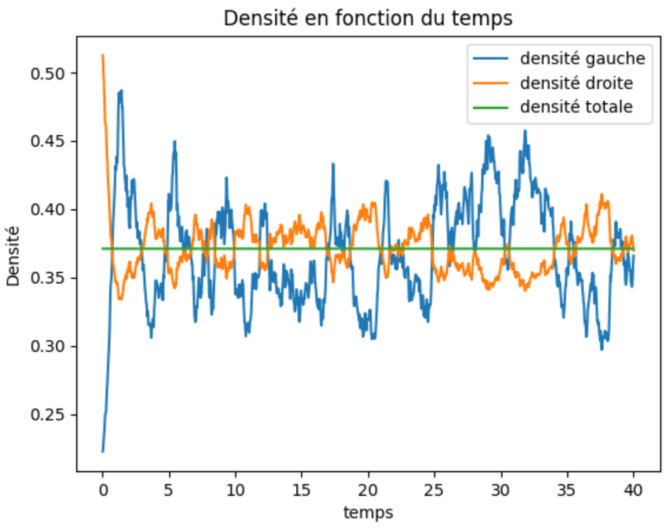


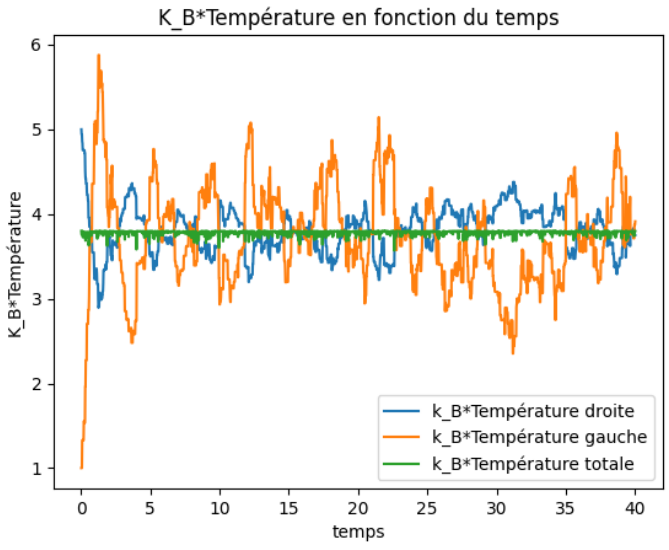
Final state :





The position of the wall this time tends towards 4.5 instead of 15 (the role of the chambers having been reversed).



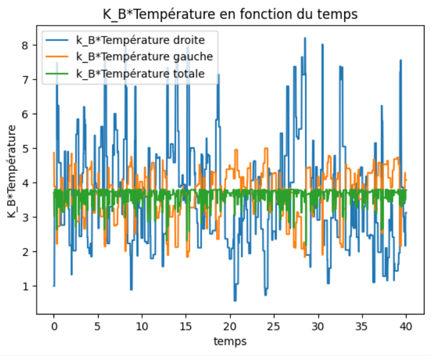


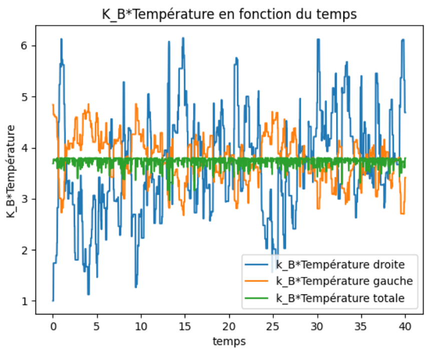
3.5 Influence of the total number of particles on the time evolution and the steady state

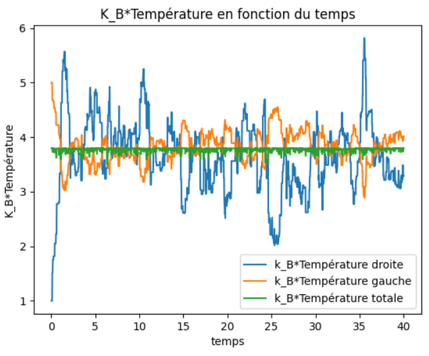
To study the influence of the total number of particles N_{total} on the time evolution and the steady state, we ran 5 simulations with increasing number of particles and kept the density constant by adapting the box's size. Each simulation was ran with 2 000 frames and a wall mass of 0.5.

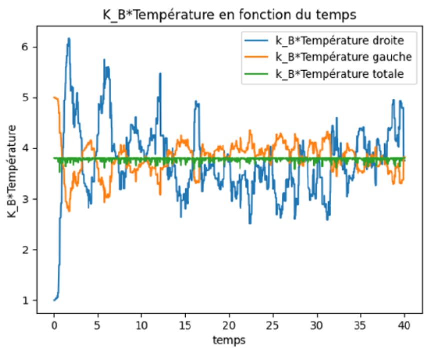
Simulation	N_{total}	$N_{left\ \ chamber}$	$k_B T_{left}$	$L_x Max$	$L_y Max$
1	10	7	5	$\frac{3\sqrt{10}}{2}$	$\frac{\sqrt{10}}{2}$
2	30	21	5	$\frac{3\sqrt{30}}{2}$	$\frac{\sqrt{30}}{2}$
3	70	49	5	$\frac{3\sqrt{70}}{2}$	$\frac{\sqrt{70}}{2}$
4	100	70	5	15	5
5	300	210	5	$15\sqrt{3}$	$5\sqrt{3}$

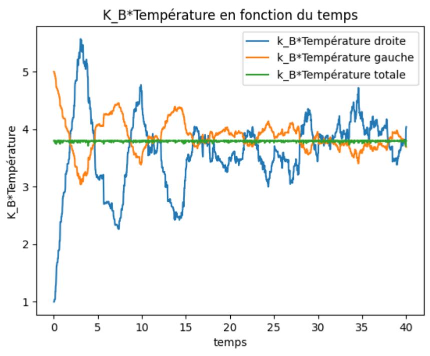
The following graphs were thus obtained :

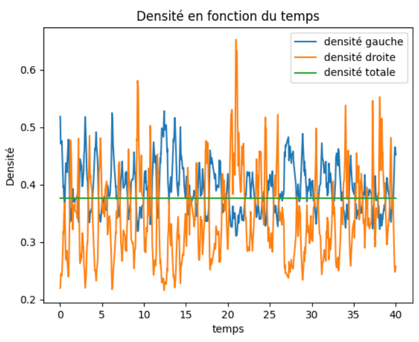


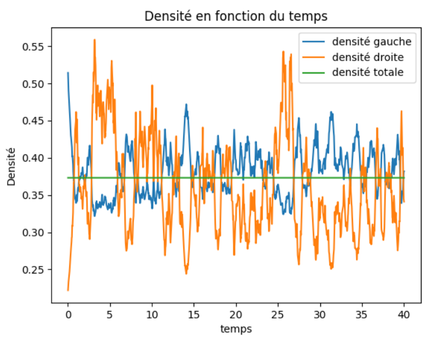


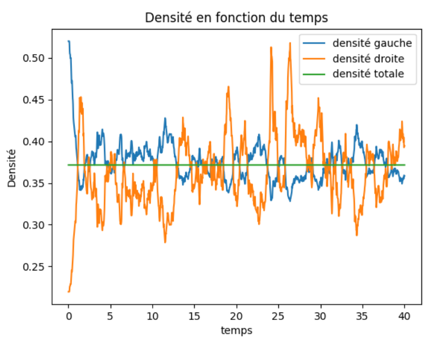


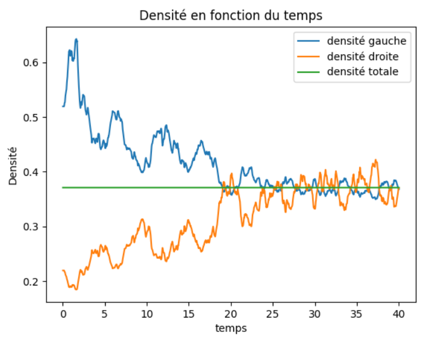


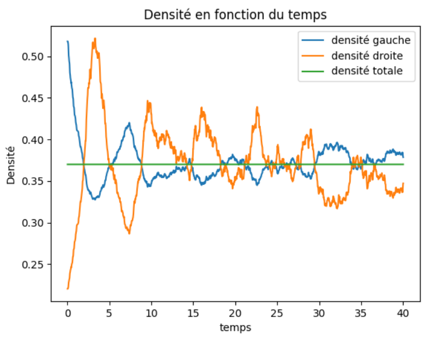












Graphs become smoother as N_{total} increases. It is to be expected since the frequency of collisions increases too. This prevents the wall from moving large distances without colliding with a particle.

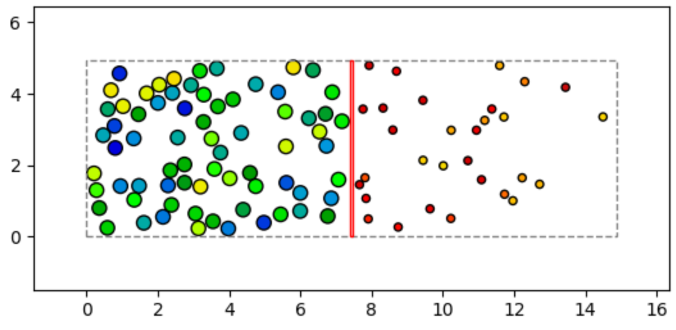
What is striking is the appearance of a steady state at all. For low $N_{total} \leqslant 30$, the evolution of temperatures is very unstable and no steady state is reached for our experiments' duration. However for $N_{total} \geqslant 70$, we can guess a transition period before a steady state.

It also seems that a steady state appears faster when looking at the density. Indeed, for 300 particles, the system is starting to reach a steady state after about 10 seceonds, while it takes 20 seconds approximately for 100 particles.

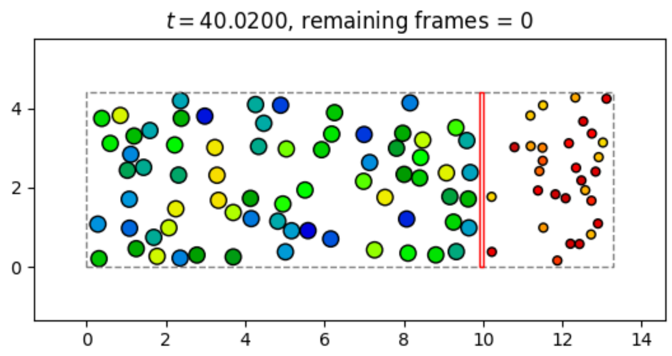
3.6 Smaller and lighter particules in the right chamber

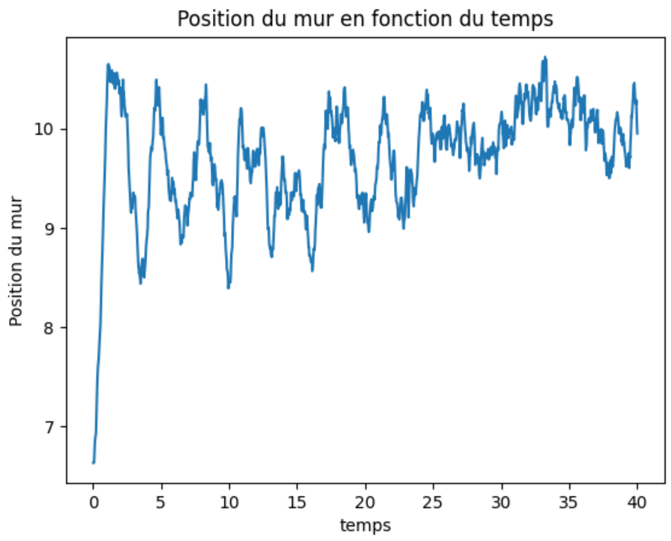
In this final setting, we use the first setting, however the particles' radius and density of the right chamber are decreased to respectively 0.11 and 2. The box's size was decreased to keep a constant global density too ($l = \frac{\sqrt{313}}{4}$). The results are the following :

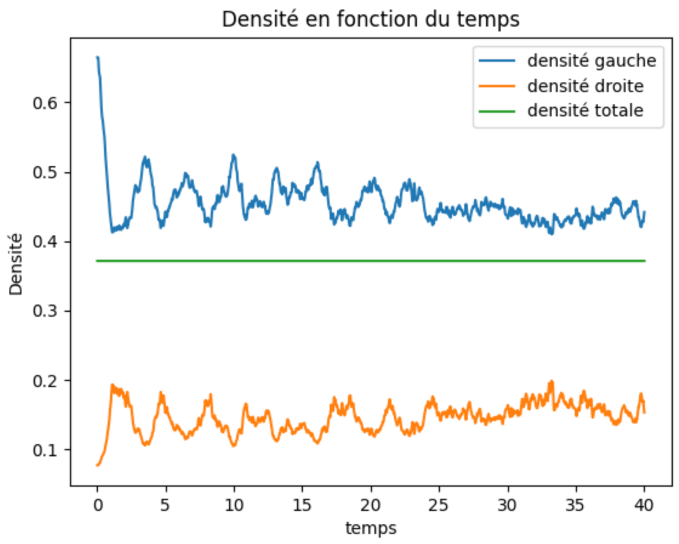
Initial state :

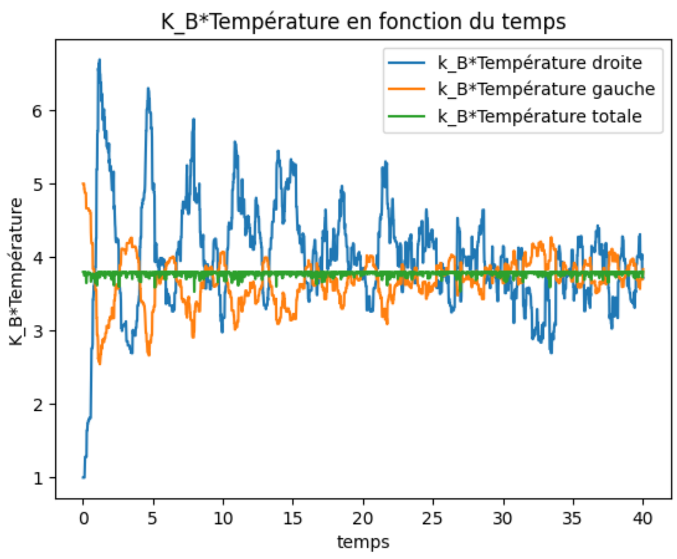


Final state :


$t = 40.0200$, remaining frames = 0
4
2
0
0
2
4
6
8
10
12
14


Position du mur en fonction du temps
Position du mur
10
9
8
7
0
5
10
15
20
25
30
35
40
temps


Densité en fonction du temps
densité gauche
densité droite
densité totale
Densité
0.6
0.5
0.4
0.3
0.2
0.1
0
5
10
15
20
25
30
35
40
temps



While the temperatures reach a common equilibrium state, we observe that the densities do not. The densities converge but to different values. This is not surprising considering we have made particles in the right chamber smaller and lighter, and that density is not a variable of the entropy.

Indeed, let us consider $S = S_1 + S_2$, $U = U_1 + U_2$, $P = P_1 + P_2$, $V = V_1 + V_2$, $T = T_1 + T_2$. Since S is extensive, we can write :

$$dS = dS_1 + dS_2 \tag{8}$$

$$dS = \frac{1}{T_1}dU_1 + \frac{P_1}{T_1}dV_1 + \frac{1}{T_2}dU_2 + \frac{P_2}{T_2}dV_2 \tag{9}$$

Since $U = U_1 + U_2$ and $V = V_1 + V_2$:

$$dU_1 = -dU_2 \quad ; \quad dV_1 = -dV_2 \tag{10}$$

Thus :

$$dS = \left(\frac{1}{T_1} - \frac{1}{T_2}\right) dU_1 + \left(\frac{P_1}{T_1} - \frac{P_2}{T_2}\right) dV_1 \tag{11}$$

The system reaches its steady state for $S = S_{max}$, or put in an another way $dS = 0$:

$$\left(\frac{1}{T_1} - \frac{1}{T_2}\right) = 0 \quad ; \quad \left(\frac{P_1}{T_1} - \frac{P_2}{T_2}\right) = 0 \tag{12}$$

$$\frac{1}{T_1} = \frac{1}{T_2} \quad ; \quad \frac{P_1}{T_1} = \frac{P_2}{T_2} \tag{13}$$

$$T_1 = T_2 \quad ; \quad P_1 = P_2 \tag{14}$$

Thus, reaching a steady means an even temperature and pressure in the system, but does not mean even density. Lighter particles only move faster.

4 Annex

4.1 Initialization of monomers' masses and speeds

```
    def assignRadiaiMassesVelocities(self, NumberMono_per_kind = np.array([4]), Radiai_per_kind = 0.5*np.ones(1), Densities_per_kind =
np.ones(1), k_BT_right = 1, k_BT_left = 1 ):

        assert( sum(NumberMono_per_kind) == self.NM )
        assert( isinstance(Radiai_per_kind,np.ndarray) and (Radiai_per_kind.ndim == 1) )
        assert( (Radiai_per_kind.shape == NumberMono_per_kind.shape) and (Radiai_per_kind.shape == Densities_per_kind.shape))

        NumberMono_per_kind = np.array([ 100, 0])
        Radiai_per_kind = np.array([ 0.2, 0.5])
        Densities_per_kind = np.array([ 2.2, 5.5])

        NumberMono_per_kind_left = np.array([30, 1])
        NumberMono_per_kind_right = NumberMono_per_kind - NumberMono_per_kind_left

        #Initilization of the left chamber
        compteur=0
        for k in range (self.N_left_chamber):
            if (k+1)>sum(NumberMono_per_kind_left[:compteur+1]):
                compteur+=1
            self.rad[k]=Radiai_per_kind[compteur]
            self.mass[k]=Densities_per_kind[compteur]*np.pi*Radiai_per_kind[compteur]**2

        #Initilization of the right chamber
        compteur=0
        for k in range (self.NM - self.N_left_chamber):
            if (k+1)>sum(NumberMono_per_kind_right[:compteur+1]):
                compteur+=1
            self.rad[k + self.N_left_chamber]=Radiai_per_kind[compteur]
            self.mass[k + self.N_left_chamber]=Densities_per_kind[compteur]*np.pi*Radiai_per_kind[compteur]**2

        '''initialize velocities'''
        assert( k_BT_right > 0 )

        vitesse=np.ones((self.NM, self.DIM))-2*np.random.rand(self.NM, self.DIM)
        N_right=self.NM-self.N_left_chamber
        E_right=N_right*self.DIM*k_BT_right/2
        E_left=self.N_left_chamber*self.DIM*k_BT_left/2
        Somme_left=sum((self.mass[:self.N_left_chamber]*(vitesse[:self.N_left_chamber]**2).sum(1))/2)#left chamber
        Somme_right=sum((self.mass[self.N_left_chamber:]*(vitesse[self.N_left_chamber:]**2).sum(1))/2)#right chamber
        masse=self.mass
        Masse=np.transpose(np.array([masse.tolist(),masse.tolist()]))

        self.vel[:self.N_left_chamber]=vitesse[:self.N_left_chamber]*(E_left/Somme_left)**(1/2) # j'ai décommenté cette ligne
        self.vel[self.N_left_chamber:]=vitesse[self.N_left_chamber:]*(E_right/Somme_right)**(1/2)


    def assignRandomMonoPos(self, start_index = 0 ):
        assert ( min(self.rad) > 0 )#otherwise not initialized
        mono_new, infiniteLoopTest = start_index, 0

        Left_Chamber_Min = self.BoxLimMin
        Left_Chamber_Max = np.array([(self.wall_pos[0]-(self.wall_epais[0]/2)), self.BoxLimMax[1]])
        Right_Chamber_Min = np.array([(self.wall_pos[0]+(self.wall_epais[0]/2)), self.BoxLimMin[1]])
        Right_Chamber_Max = self.BoxLimMax

        #left chamber
        self.pos[mono_new] = np.random.uniform(Left_Chamber_Min+self.rad[mono_new, None], Left_Chamber_Max-self.rad[mono_new,None])
        mono_new+=1
        while mono_new < self.N_left_chamber :
            flag = True
            while flag :
                self.pos[mono_new] = np.random.uniform(Left_Chamber_Min+self.rad[mono_new, None], Left_Chamber_Max-
self.rad[mono_new,None])

                delta_r_ij = np.where(self.pos != 0, self.pos - self.pos[mono_new], 0)
                delta_r_ij_sq = (delta_r_ij**2).sum(1)

                min_distance = np.where(delta_r_ij_sq == 0, np.Inf, delta_r_ij_sq)
                min_distance = np.argmin(min_distance)
                if (delta_r_ij_sq[min_distance])**(1/2) > self.rad[min_distance]+self.rad[mono_new] :
                    flag = False
            mono_new += 1

        #right chamber
        self.pos[mono_new] = np.random.uniform(Right_Chamber_Min+self.rad[mono_new, None], Right_Chamber_Max-self.rad[mono_new,None])
        mono_new+=1
        while mono_new < self.NM :
            flag = True
            while flag :
                self.pos[mono_new] = np.random.uniform(Right_Chamber_Min+self.rad[mono_new, None], Right_Chamber_Max-
self.rad[mono_new,None])

                delta_r_ij = np.where(self.pos != 0, self.pos - self.pos[mono_new], 0)
                delta_r_ij_sq = (delta_r_ij**2).sum(1)

                min_distance = np.where(delta_r_ij_sq == 0, np.Inf, delta_r_ij_sq)
                min_distance = np.argmin(min_distance)
                if (delta_r_ij_sq[min_distance])**(1/2) > self.rad[min_distance]+self.rad[mono_new] :
                    flag = False
            mono_new += 1
```