Compte-Rendu de TP : Event Driven Simulation

AVALLE Domitille
EA Eric

Pr. KLMASER Juliane

Table des matières

1 Introduction

The goal of this project is to simulate a system with various kind of events happening over time, such as particle collisions. An event driven simulation in short. One could argue that moving the simulation from an event directly to the next is enough when the system is perfectly determinist since what happens in-between is already known to the experimenter. However, it is really important to make a simulation in constant time-step to measure average velocities for example.

1.1 Physical system of interest

The system we have simulated can be compared to a perfect gas enclosed by infinite potential walls. That is to say, a set of particles taking all of the available space and interacting with each other through elastic collisions only. This means there is no interaction potential and the energy of the system is given by :

$$E = \sum_{p=0}^{N} \frac{1}{2} m_p v_p^2 = N k_b T \tag{1}$$

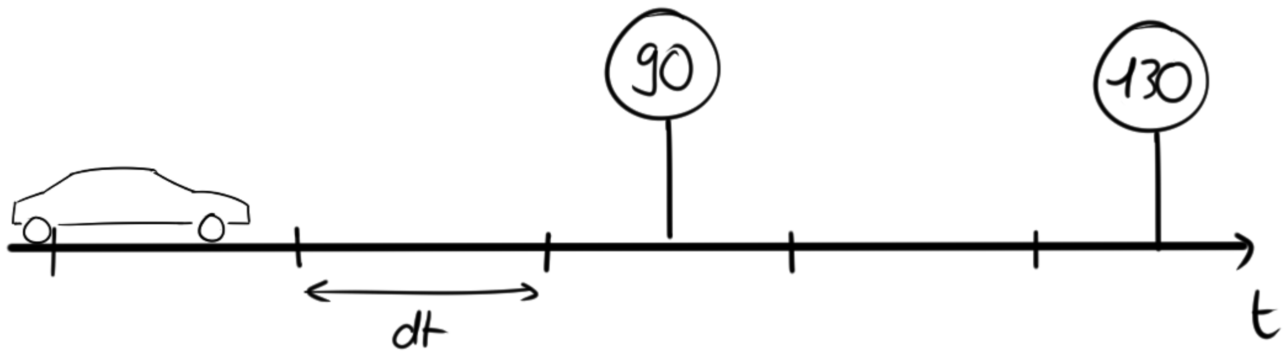
Particles are thus modelled as hard spheres, with two types, monomers and dimers. Monomers are made of a single hard sphere while dimers are made of two hard spheres.

1.2 Challenges

There were a few questions we had to properly answer throughout this project. How to make a simulation in constant time-step ? How to manage inter-particles and wall-particles interactions ? And how to avoid numeric precision related errors ?

2 Making a simulation in constant time-step

We want the simulation to update in constant time-step dt, however events can randomly occur between successive dts. Let us consider for now a simple model, a moving car on a straight road with speed-limits sign randomly placed along the road. The car has to keep moving at a constant speed v_{car} and keep sending a GPS signal every dt, until it encounters a sign and has to update its speed and move to the next sign or the next dt.



This "until" can be translated by a "while" loop. So while the car keeps encountering signs, it has to update its velocity and position. But when it does not, it can simply move from a dt to the next.

```python
import numpy as np

N_GPS_signals = 100 # total number of GPS signals we want to simulate
dt_of_GPS = 5.0
times_list_of_GPS_signals = np.arange(dt_of_GPS, dt_of_GPS*N_GPS_signals, dt_of_GPS)

```

```
np.random.seed(10)# this is for debugging

# we intialize the system at t = 0
current_time = 0.0
car_position = 0.0
car_speed = 1.0
position_next_speed_limit = car_position + np.random.uniform(0,5)

# before entering the event-driven loop, we initialize the loop parameters
dt_till_speed_change = (position_next_speed_limit - car_position) / car_speed
future_time_speed_change = current_time + dt_till_speed_change

for GPS_time in times_list_of_GPS_signals:
  #time at which GPS signal should be transmitted
  future_time_next_GPS_signal = current_time + dt_of_GPS
  while future_time_speed_change < future_time_next_GPS_signal:

    ''' START EVENT-DRIVEN SIMULATION LOOP
    You are not allowed to change the code,
    but you can change the indentation.'''

    #1. move car till next speed limit and update current_time
    car_position += car_speed * dt_till_speed_change
    current_time = future_time_speed_change

    #2. change the speed and find distance to next speed limit
    car_speed = np.random.uniform(0,2)
    position_next_speed_limit = car_position + np.random.uniform(0,5)

    #3. calculate dt to reach next limit and get the time at which car will reach the
    limit
    dt_till_speed_change = (position_next_speed_limit - car_position) / car_speed
    future_time_speed_change = current_time + dt_till_speed_change
    ''' END EVENT-DRIVEN SIMULATION LOOP'''
    print("we send a GPS signal at time \t%.5f\nbut it should be send at \t%.5f\n" % (
    current_time, GPS_time))

  car_position += car_speed*(future_time_next_GPS_signal-current_time)
  dt_till_speed_change -= (future_time_next_GPS_signal-current_time)
  current_time = future_time_next_GPS_signal
```

If we had the generalise the process, first we compute the time from the current time to the next event but also the time where it takes place on the global axis of time.

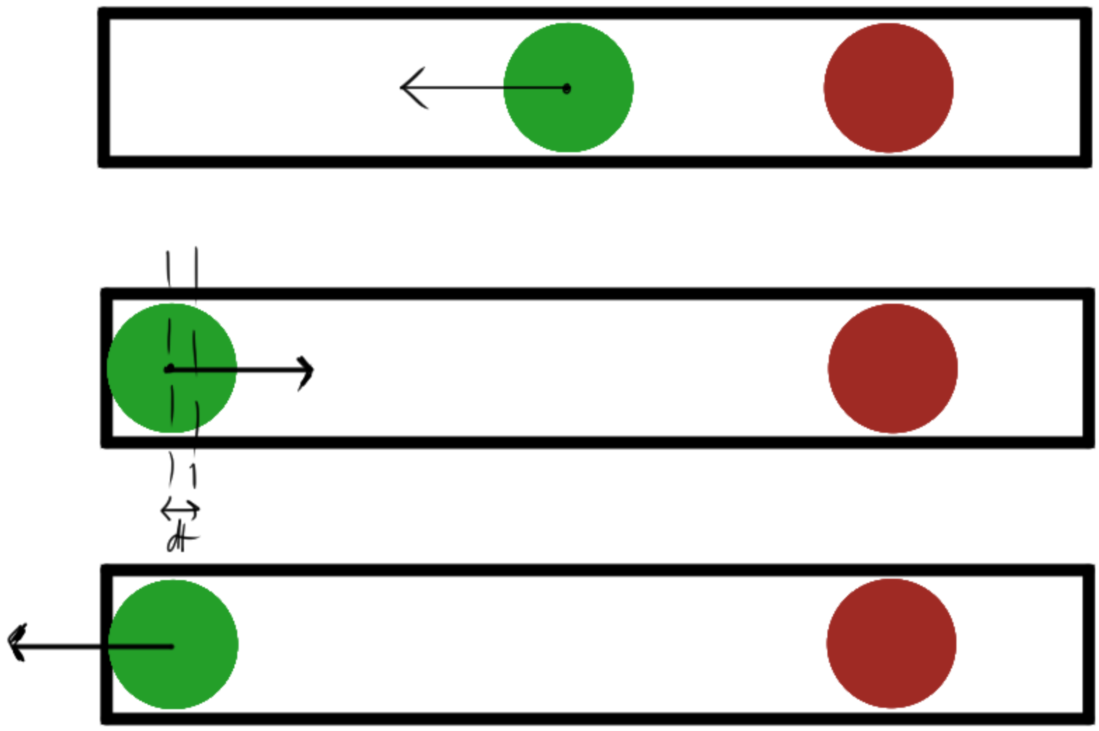
As long as the time to the next event is smaller than the time until the next dt, meaning events are occuring before the next dt, the system has to update its parameters, and we also have to update the current time.

If the time to the next event is greater than dt, the system just moves from dt to dt without updating parameters that are affected by events.

3 Numeric precision related errors

In our case, numeric precision related errors can lead to particles escaping the box when wall collisions are not well managed. Let us take the example of two particles moving along the x axis only in a box, and let us ignore inter-particle collisions.

One way to compute when the next wall collision will happen is to compute for each wall (left and right) a list of collision time for all particles, and select the smallest one in absolute value. Once the collision happens, we would just have to invert the sign of the particle's velocity. That is a purely mathematical way to approach this problem, and this can lead to particles escaping the box. What can happen because of precision errors is the following scenario :



A green particle is moving to the left wall. The simulation is ran in constant time-step until the wall collision event happens. However, because of a precision error, the center of the particle ends up a little bit too far from the position it's supposed to be. The velocity is updated, but since there's a very tiny dt between the same particle and the wall, the next collision will be between the left wall and the green particle again because the code only checks the smallest dt. The velocity of the green particle will then be inverted again, causing it to escape.

So as to avoid particles escaping the box, we have to consider physical possibilities. In our case, a particle with positive horizontal speed cannot collide with a left wall. If the former code took this information into account, even though it would not have prevented a precision error, it would have at least ignored such an impossible collision and kept the particle inside.

4 Managing collisions

4.1 Wall collisions

Wall collisions are not particularly hard to manage once we keep the previous error in mind.

```
Correct_Wall=(self.BoxLimMax - self.rad[:,None]) * (self.vel > 0) + (self.BoxLimMin +
    self.rad[:,None]) * (self.vel < 0)
CollTime = (Correct_Wall-self.pos)/self.vel
```

When a velocity is positive, the code only compute the distance from the particle to right or top wall, and when it is negative, it does the opposite. The distance is divided by the particle's speed to get the collision time. Then, to make the particle bounce off the wall, we will only need to reverse its speed.

4.2 Particle collisions

One way to calculate the collision time is to keep track of all particles distances between each other. When the distance between the centers of two particles are two radii apart, they collide. Thus we need to compute when that collision is happening. The colab notebook gives us the following solution :

$$\Delta t_{\pm} = \frac{1}{2a}\left(-b \pm \sqrt{b^2 - 4ac}\right) \tag{2}$$

With :

$$a = \Delta v_y^2 + \Delta v_x^2 \tag{3}$$

$$b = 2\left(\Delta v_x \Delta x_0 + \Delta v_y \Delta y_0\right) \tag{4}$$

$$c = \Delta x_0^2 + \Delta y_0^2 - (R_1 + R_2)^2 \tag{5}$$

The right $\Delta t_{\pm}$ to choose in our case has to satisfy $\sqrt{b^2 - 4ac} \in \mathbb{R}$ obviously but also $\Delta t_{\pm} > 0$ for external collisions. For internal collisions of monomers belonging to the same dimer, the math is the same except we only need to check $\sqrt{b^2 - 4ac} \in \mathbb{R}$.