LIÈGE université
Sciences Appliquées

Master in applied sciences
Computer science and engineering
Year 2018-2019

LIÈGE université

Introduction to Complex problem solving

Assignement 2

INFO8003-1
Damien Ernst

Olivier Moitroux
s152122

Liège

Février 2019

2 Implementation of the domain

Organisation of the code

Several files are included inside the submission. `main.py` is the main script that defines the method associated to each question of the assignment. `toolbox.py` is where the logic is coded. It defines a class `Domain`, a class `Simulator` to test policies, build random episodes, ... as well as some useful methods (policies, ...) by interacting with the *domain* instance. All the unit tests are available in a dedicated file, namely `unit_test.py`. Finally, `plot.py` contains the code used to build plots.

Unit test

The most interesting unit test for the domain class is the one that tests wether the euler integration works as expected. The following plot proves visually the correctness of the implementation.

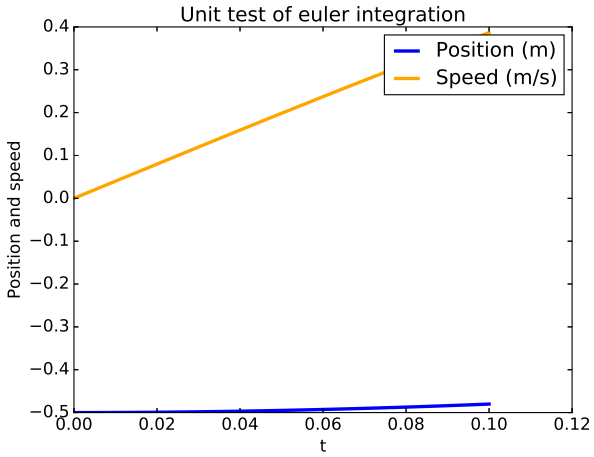
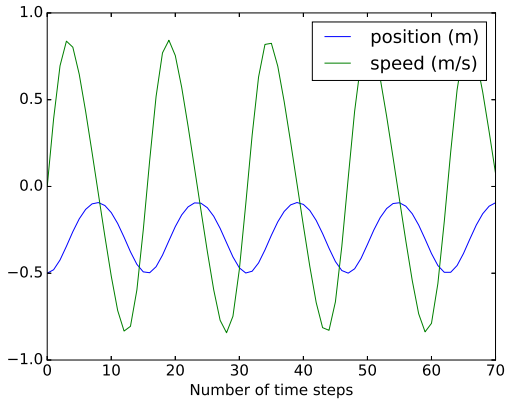


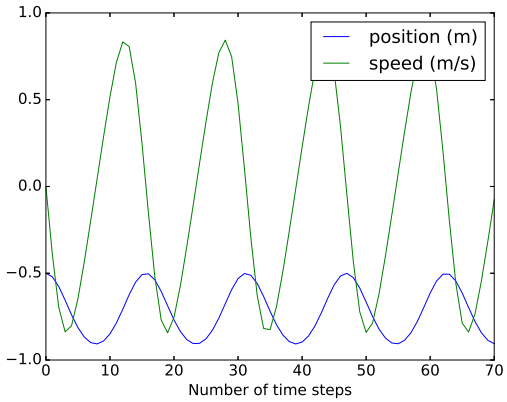
Figure 1 – Evolution of the car position and speed starting with $[p_i = -0.5,\ s_i = 0]$ during one time step and an acceleration of $+4m/s^2$.

Simulation of a simple policy

Figure 2 show the result of a simulation of 70 timesteps starting in -0.5m.



(a) Always accelerate



(b) Always decelerate

Figure 2 – Evolution of the position and speed starting from $[p_i,\ s_i] = [-0.5,\ 0]$. A timestep correspond to 0.1 second.

It can be noted that by generating (only once) 1000 episodes following a random policy, 61288 4-tuples were generated and 12 of them lead to a final immediate reward of 1. Like such, taking into account the discount fact, the average return of these 19 success was equal to 0.0558 and the variance of these results equal to 0.0034.

3 Expected return of a policy in continuous domain

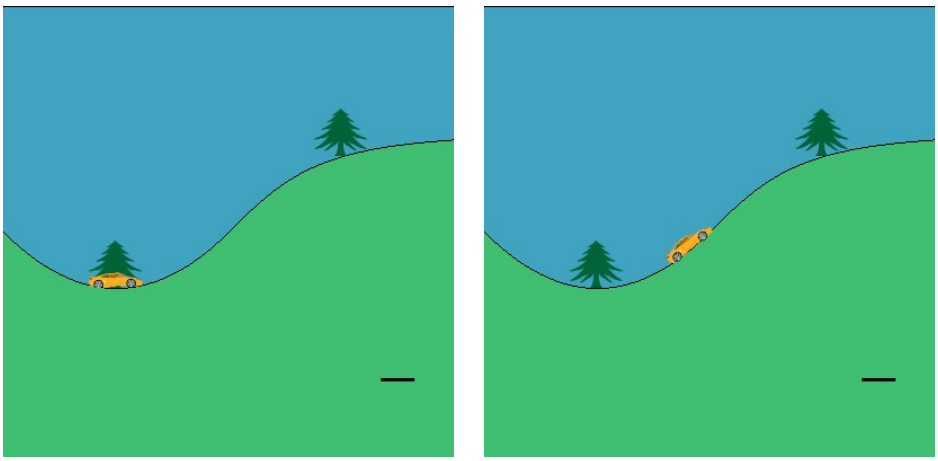
When considering a constant initial position, with a simple policy that always does the same action, the expected return is directly given by doing one simulation. Indeed, the setting is perfectly determined by the starting position in such a case. Alternatively, one can also consider sampling the initial state randomly. The routine that computes the expected return of a policy with the Monte Carlo principle can be explained by the following pseudo code:

```
def Monte_Carlo(n_iter):
        init:
                policy
                rewards = []

        for i in range(n_iter):
                s = random_non_terminal_initial_state()
                episode = GenerateEpisode(s, policy)
                rewards.append(CumulativeReward(episode))

        return mean(rewards)
```

By taking care not to start in a terminal state (f.e. by removing the epsilon machine to the bounds), the cumulative expected reward, averaged over 1000 simulations, was equal to 0.1194. Each simulation was allowed to run for 100 time steps maximum as some configurations may lead to infinite execution (cfr. Figure 2 for instance).

4 Visualization

By calling the method `makeVideo()` defined in *plot.py* and using a policy that always accelerate, we obtain a .avi video. The video actually offer another visualisation of what has already been shown in figure 2. When the car starts at -0.5m without any speed, the car climbs at almost half the concave slope before falling back to its initial position:



(a) Minimum position (-0.5m) (b) Maximum position (-0.0925m)

Figure 3 – Illustration of the two extreme positions by always accelerating, starting from -0.5m without any speed.