Faculty of Applied Sciences
University of Liege
March 6, 2019

-

Optimal decision making for complex problems:

Assignment 2 - Section 1 to 4

s141770
Derroitte Natan

2 Implementing the domain

The first part of the statement asked to implement the domain in which the agent, the car, will evolve during this project.
In order to test this implementation, a simple policy was created. The program therefore allows to easily generate, according to the given argument, 3 very simple policies: always move to the right, always move or take a random action.
Formally, this can be summarized by

$$\mu(x) = \begin{cases} 4 & \forall x \quad OR \\ -4 & \forall x \quad OR \\ \{-4, 4\} & \text{chosen randomly} \quad \forall x \end{cases}$$

Since no initial state or policy is recommended in the statement, it was chosen to take the initial state $x = (-0.99, 0.0)$ and the policy of always moving to the right. The corresponding result is that the car arrives at the top of the hill. These initial values have been chosen arbitrarily as examples and can therefore be modified.

In order to visualise the result, the implementation developed for question 4 was already used. The corresponding video has been jointed in the archive under the name `Q2.avi`.

3 Expected return of a policy in continuous domain

In order to estimate the expected return of a policy in continuous domain, two methods were considered. The first is to take a large number of trajectories from random initial states, following the policy. For each trajectories, it is then necessary to calculate the expected cumulative reward. When the number of

trajectories is high, the average of the expected cumulative reward will tend to average $J(x) \forall x \in X$.
The second consists in not relying on trajectories with a random initial position. Instead, one goes through the states that can take $x \in X$ per small step and creates a trajectory, still following to the policy, for each of these states. From these trajectories, estimating the expected return of the policy is the same as with the first method : calculated the expected cumulative reward for each trajectory and averaged over all trajectories.
The problem with this method is that if the step is too small, many initial states will be considered and the computation time will be too long. On the contrary, if the step is too large, the precision of the technique is too low...
The main advantage of this second method however is that if the step is chosen properly, it will need a lot less trajectories to converge towards the expected return.

In both case, the method used is based on the expected cumulative reward in deterministic case which is :

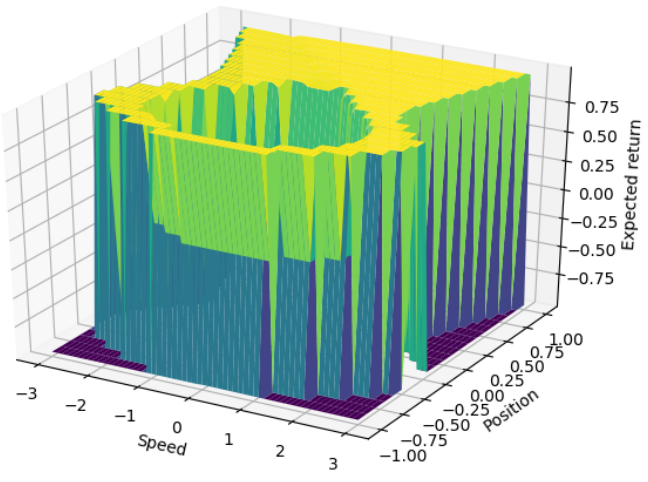
$$J^{\mu}(x) = \lim_{T->\infty} \left[\sum_{t=0}^{\infty} \gamma^t r(x_t, u_t \sim \mu(h_t, ..)) | x_0 = x \right] \tag{1}$$

This formula will be use to compute the expected cumulative reward of the trajectories in our implementation.

The second method was chosen with a step of 0.1 for position and speed for its shorter computation time and good accuracy. The results can be seen in the figures 1 and 2. These are computed for trajectories of sizes 10000. In order to recover the t of formula (1), it is mandatory to consider the integration time step: 0.001.
From then on, it appears that:

$$t = experience\_number * 0.001 \tag{2}$$

For the figures 1, the policy of always moving to the right is considered. The expected return of the policy in this case is 0.2739. This represents the average of $J(x) \forall x$ and can be seen as the average final reward when the policy of always moving to the right is adopted. The figures 1 represent $J(x)$. It can be seen that depending on the speed and initial position, different states can be reached. It appears that the agent has a good chance of reaching the winning state, where the reward is 1, especially when the initial position is high.



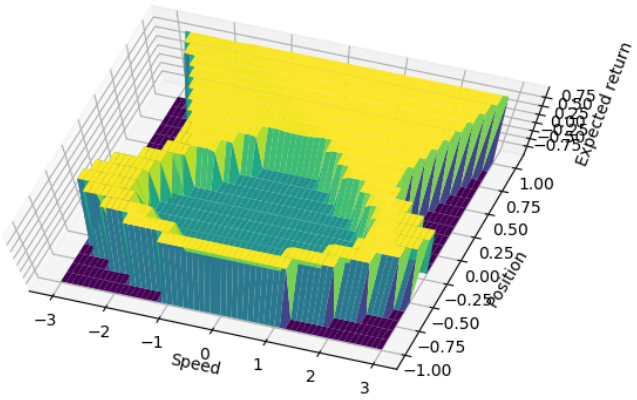
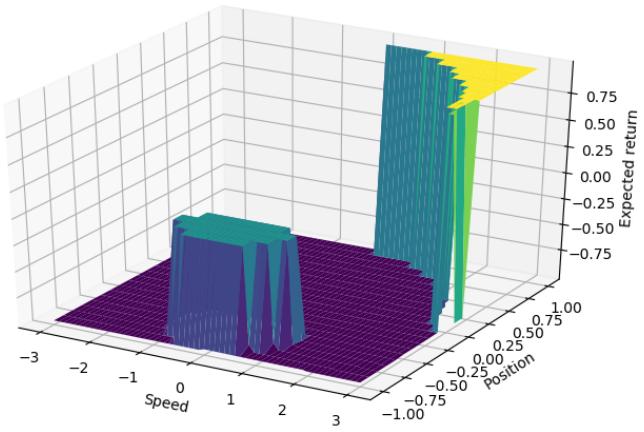


Figure 1: $J^{\mu}(x) \forall x \in X$ when $\mu(x) = 4 \forall x \in X$.

As shown in the figures 2, when the policy is to always move to the right, the chance of reaching a winning final state ($J(x) = 1$) is much lower but the chance of being in a losing final state is higher. Extremely favourable initial conditions are required to reach a winning state.
The expected return of this policy is -0.7360.



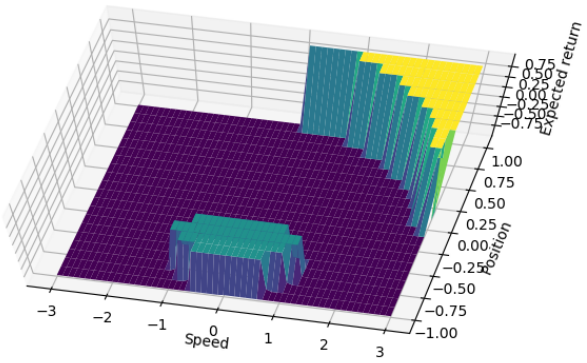


Figure 2: $J^{\mu}(x) \forall x \in X$ when $\mu(x) = -4 \forall x \in X$.

The problem being entirely discrete, the use of the Monte Carlo principle amounts to making an arithmetical sum of a single element. However, the code has been designed in such a way that it is scalable and can therefore be applied to a stochastic case.

4 Visualisation

In order to visualise the calculated results, a method allowing to generate a video from a series of states has been implemented. To do so, the `opencv` package was used.
As mentioned above, an example video has been placed in the archive under the name `Q2.avi`.
Some frames of this video are shown in the figure 4.

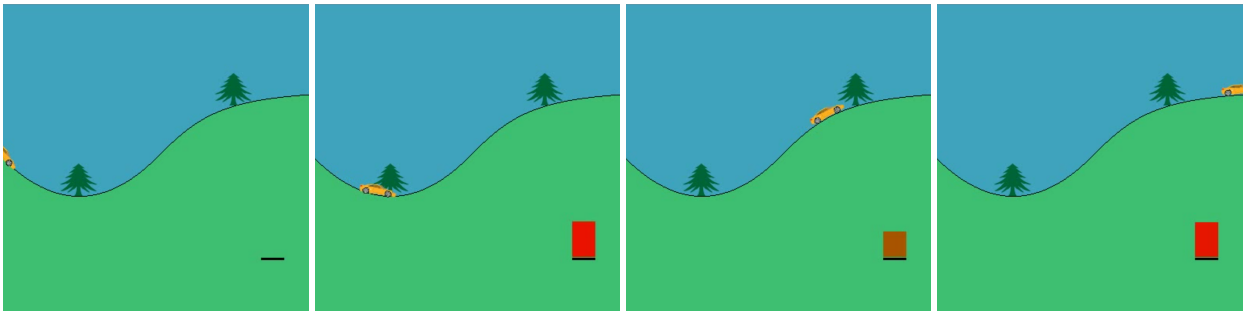


Figure 3: Some frames from the video generated with `opencv`.

The simultaneous use of `opencv` and `pygame` on the *macOS* operating system can cause problems. Beyond 928 frames recorded, the video cannot be recorded and the program will shut down.
The origin of this error is purely due to the fact that these packages were not originally developed for *macOS* and that the versions developed for this system still contain some bugs. Under no circumstances does the segmentation fault originate in our code.
The simulation time should therefore not exceed 36 seconds for *macOS* but is not limited for other operating systems. No problems were detected on *Linux*, no matter how many frames were recorded.