Reinforcement Learning Applied to Autonomous Vehicles

Polytech Nancy 5A project report

GUYOT Antonin

2022-2023

Contents

ACKNOWLEDGEMENTS

I would thank Polytech Nancy for giving me the opportunity to make this project that made me learn a lot of very interesting concepts and for providing the really advanced tools I got to use during this project.

A special thank goes to Mayank Shekhar JHA, that supervised me during all this project and was there to answer to my questions.

I also thank Floriane COLLIN and Hugues GARNIER, who gave me this opportunity and assisted to my presentation, showing a lot of interest and curiosity for these aspects.

INTRODUCTION

I - Context

Reinforcement Learning (RL) approaches have recently emerged as the most efficient control solutions for complex systems. Indeed, these model-free methods provide a way of learning control laws based on real-life data. These approaches are also robust to dynamic changes and to uncertainty.

The goal of this project is to apply RL to autonomous driving vehicles. These systems are indeed pretty simple to control, and can provide a lot of information on their environment, thanks to numerous sensors.

II - The project

The purpose of this project is to apply RL algorithm to a miniaturized version of a self-driving car.

The algorithms are provided by the Reinforcement Learning Toolbox[1] of MATLAB/Simulink. The miniaturized car is the QCAR[2], made by Quanser and provided with the QUARC Real-Time Control Software[3], used to interface the QCAR in MATLAB/Simulink.

The goal of this project is to implement RL algorithms provided by MATLAB for the control of the QCAR, to achieve simple goals (obstacle avoidance and point-to-point driving).

All the work presented in this project is available on GitHub[4].

[1] RL Toolbox documentation: https://fr.mathworks.com/help/reinforcement-learning/
[2] QCAR presentation: https://www.quanser.com/products/qcar/
[3] QUARC software: https://www.quanser.com/products/quarc-real-time-control-software/
[4] GitHub repository: https://github.com/Guyak/ProjectAntonin_QCAR

REINFORCEMENT LEARNING INTRODUCTION

I - Concept

I.a. - RL agent purpose

Reinforcement learning is an approach to machine learning in which an agent learns to make decisions by interacting with an environment and receiving rewards or penalties for its actions. The agent's goal is to maximize by trial and error the overall reward it receives, by learning to take actions that lead to beneficial states.

The environment is everything which is not included by the controller, such as the plant, the reference signal, or the measurements. The goal of the agent is to learn the best policy with respect to this environment to maximize its reward. The policy is represented by the following function:

$$\pi(s) = a$$

Equation 1: Policy representation

With π the policy function, s the current state perceived by the agent (determined by the observations of the agent), and a the action to take.

I.b. - RL compared to traditional control

In general, a control system aims to identify the appropriate actions to control a system and produce the desired behaviour. Feedback control systems utilize measured output to enhance performance and correct any unexpected disruptions or errors. Engineers use this feedback and a model of the system and its surroundings to develop a controller that meets the system's requirements.

Figure 1 shows the difference between traditional controls and RL. We can see that these two controls work almost in the same way, but an RL agent tends to constantly improve its policy by considering the feedback of the environment and update its policy accordingly.

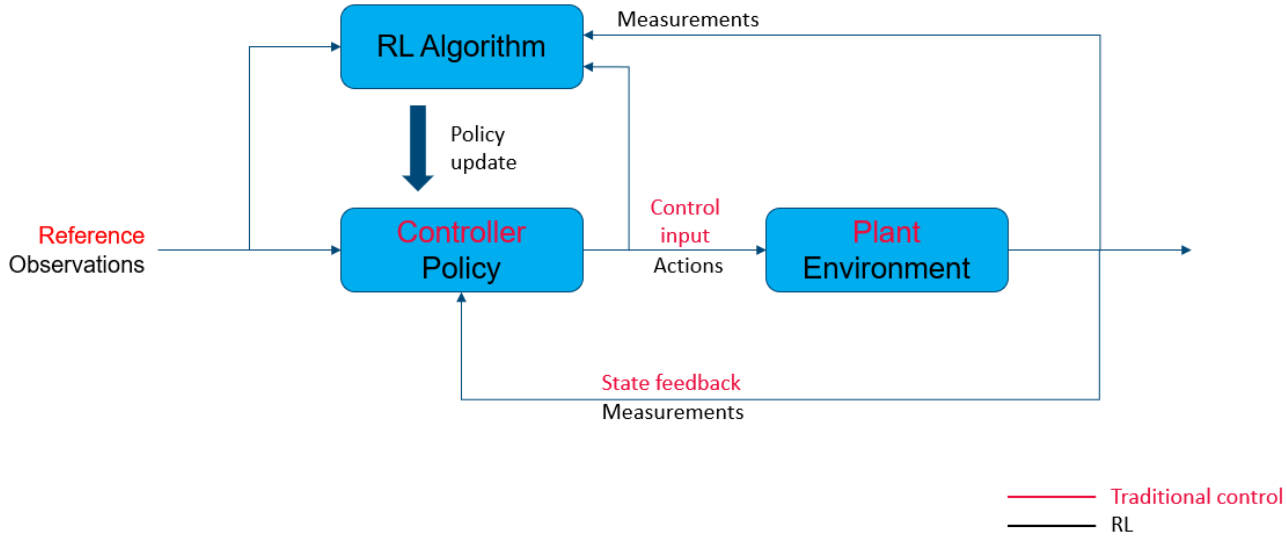


Figure 1 : RL vs Traditional control

II - DDPG algorithm

II.a. - Motivation

RL can work with discrete or continuous states and time. The goal of the project is to make an agent work in an HIL configuration, and sensors used such as LIDAR return continuous values. These requirements have led to the choice of Deep Deterministic Policy Gradient (DDPG) algorithm, which is a RL algorithm inspired by Deep Learning and working with two neural networks.

II.b. - Architecture

DDPG works with two neural networks in the form of an actor/critic architecture.

The actor network applies the policy learnt by the agent, depending to the observations. The critic network tunes the policy by looking at the associated reward and trying to maximise it. The critic generates a Q-value, determined with the reward of the iteration and the Q-value of the next iteration (Equation 2). This value serves to make sure that the action taken at a moment will have a positive aspect at long term.

$$Q = reward + \gamma \cdot Q_{next}$$

Equation 2 : Q-value calculation

γ is a discount value. It determines how much we take in consideration the long-term benefits of an action.

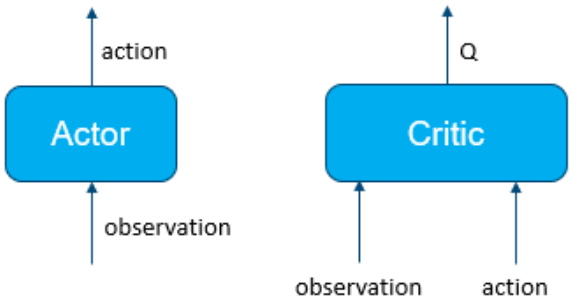


Figure 2 : Actor-critic architecture

II.c. - Training

A training episode is composed of a list of replay buffers created for each iteration. A replay buffer is the tuple displayed in equation 3.

$$[\langle obs, act, rew, obs_{next} \rangle]$$

Equation 3 : Replay buffer

With :

- obs the observation made at the beginning of the iteration
- act the action taken accordingly to obs
- rew the reward associated to the action
- obs_{next} the observation made after the action is taken

For each replay buffer, the networks are trained one after the other.

Actor is trained by maximising Q with the network shown in figure 3:

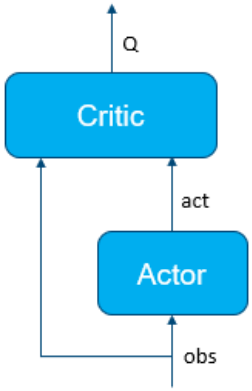


Figure 3 : Actor training

Then, critic is train by minimizing the cost function $Q - (reward + \gamma \cdot Q_{next})$ based on the double network shown in figure 4:

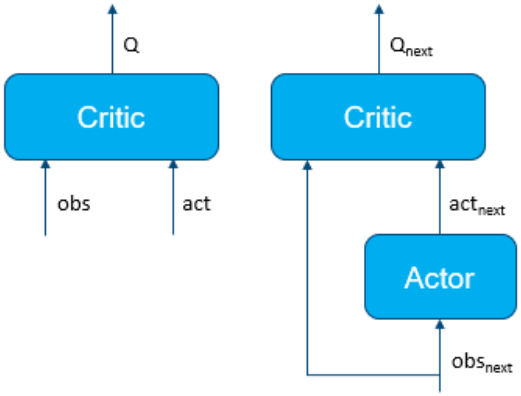


Figure 4 : Critic training

SIMULATION WITH MATLAB/SIMULINK

I - Bicycle model

In simulation, any 4-wheels vehicle with to steering wheels can be represented as a bicycle model. This model (figure 5) is composed of two wheels spaced by a certain length L and only the front wheel can steer. The model is resented by a pose $\langle x, y, \theta \rangle$ (2D position and orientation) and is controlled with a velocity v and a steering angle δ.

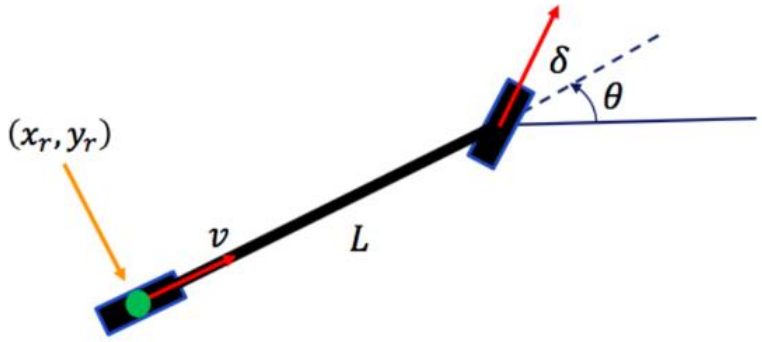


Figure 5 : Bicycle model

Trigonometric formulas can be used to calculate the dynamic of the system:

$$\dot{x}_r = v \cdot \cos(\theta)$$

Equation 4 : x-speed formula

$$\dot{y}_r = v \cdot \sin(\theta)$$

Equation 5 : y-speed formula

$$\dot{\theta} = \frac{v}{L} \cdot \tan(\gamma)$$

Equation 6 : angular velocity formula

Work of Peter Corke[5] has been used as a good basis to study this model. It provides a lot of examples to use bicycle model in Simulink to complete certain tasks, such a pose reaching or trajectory following. The problem used for this project is the point-to-point driving.

I.a. - Point-to-point driving

To reach a certain point, the model computes the distance and orientation difference from the goal point and adapts its velocity and steering angle accordingly. The problem can then be seen as a proportional correction:

[5] Robotics, Vision and Control – Fundamental Algorithms in MATLAB, Peter Corke (2011)

$$v = K_v \cdot \sqrt{(x_g - x_r)^2 + (y_g - y_r)^2}$$

Equation 7 : Velocity control

$$\gamma = K_h \cdot (tan^{-1}\left(\frac{y_g - y_r}{x_g - x_r}\right) - \theta)$$

Equation 8 : Steering control

With:

- $K_v$and $K_h$the gains of the controller
- $(x_g,\ y_g)$ the coordinates of the goal point
- $(x_r,\ y_r, \theta)$ the pose of the vehicle

I.b. - Simulink integration

Bicycle model can be represented in Simulink as shown in figure 6. The inputs are the velocity and steering angle, and it outputs the pose. Initial conditions must be given in the integrator block.

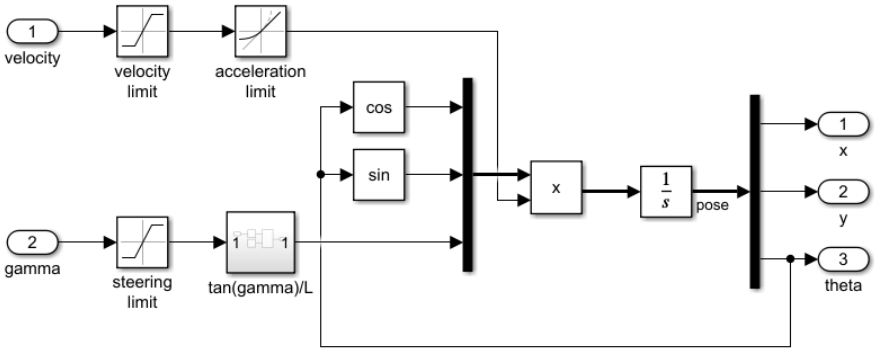


Figure 6 : Bicycle model

Peter Corke's custom MATLAB library also provides Simulink models adapted to every problem seen in its work. These models have been re-worked to be used in new versions of MATLAB.

Figure 7 shows the Simulink Model used to integrate point-to-point driving. Goal position is compared to actual position, and velocity and steering are computed form this comparison.

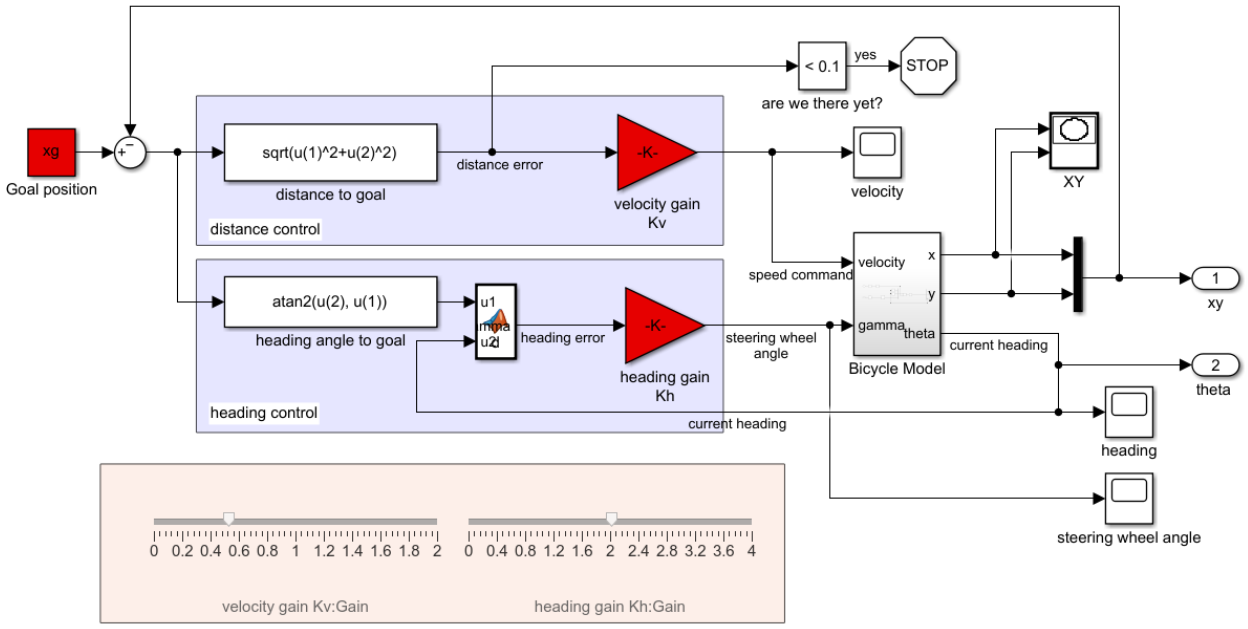


Figure 7 : Point-to-point driving model

II - Sensor and environment modelling

The goal of the project is to control the vehicle to avoid obstacles (walls) and know it position to reach a goal point. The choice has been made to used the LIDAT of the QCAR to detects the walls. QCAR has enough sensors to consider it able to know its position as accurately as the pose given by the bicycle model.

II.a. - Environments modelling

The environments of the simulation have been created using the binaryOccupancyMap function, available in the Robotics system Toolbox[6] of MATLAB. This function uses a booleans 2D-matrix and considers a true as a wall, and a false as an empty space.

This function has been used to create two environments, and empty square, and an environment filled with obstacles.

These maps are useful to simulate an environment, being provided with functions used to detect the distance to walls, collisions and so on.

[6] RS Toolbox documentation: https://fr.mathworks.com/help/robotics/index.html?s_tid=CRUX_lftnav

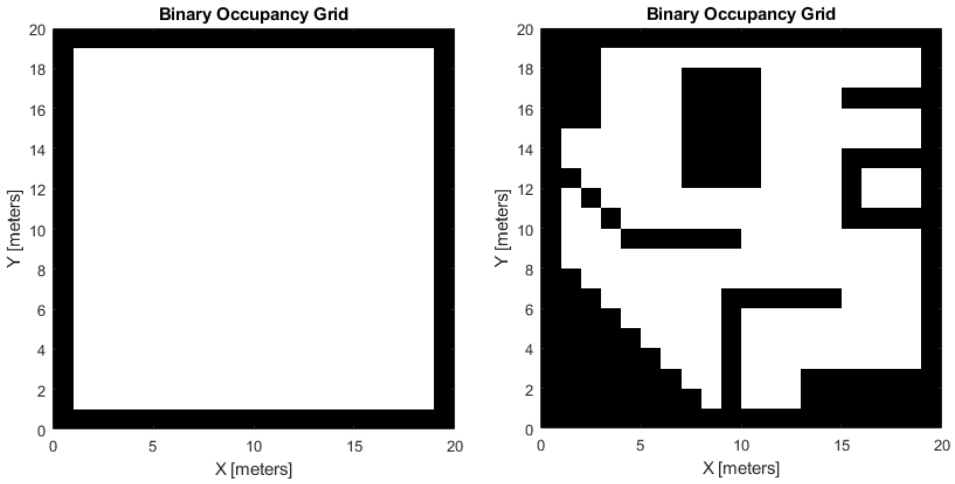


Figure 8 : Environments created for the purpose of the project

II.b. - LIDAR modelling

LIDAR has been simulated using the actual pose of the vehicle and the rayIntersection MATLAB function. This function also comes from the RS Toolbox, and give the distance between the vehicle and the nearest wall following a specific angle (angle 0 gives the distance of a wall in front of the vehicle).

The LIDAR of the QCAR has a precision of 0,5°. A For loop can thus simulate this sensor by looking at the rayIntersection value returned for an angle going from -180° to 180° with a step of 0,5°. The smallest distance and its index can then be extracted to return the position and distance from the closest wall in the environment.

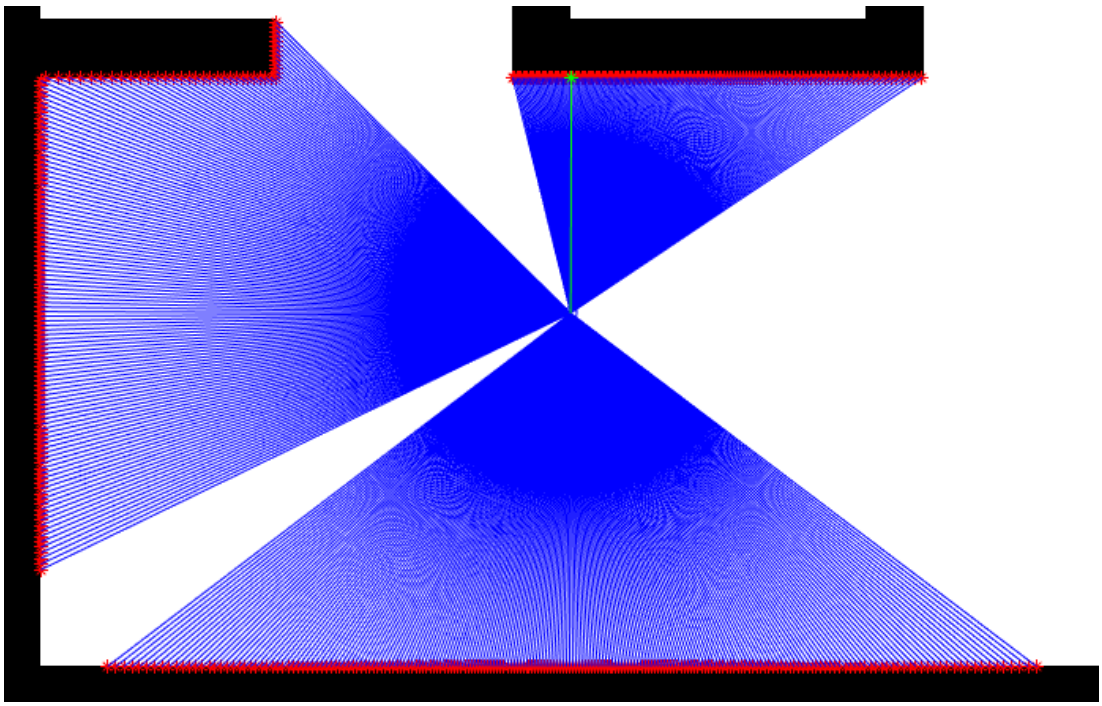


Figure 9 : LIDAR simulation, the green ray represents the smallest distance detected

III - RL implementation and results

With these components integrated, RL Toolbox has been used to start creating the agent and train it. The first example provided by MATLAB has been used to get the hang on the toolbox, then two MATLAB/Simulink simulations were created for the project.

III.a. - MATLAB example: the water tank model

The example provided by MATLAB is based on a water tank. The agent gets a goal of a water level to maintain, and can interact with the environment by adding or removing water in the tank.

It provides the way of creating the neural architecture of DDPG in MATLAB. It also shows the way of creating a Simulink model to implement an RL agent.

III.a.i. - RL agent creation

In MATLAB, the RL agent is created by giving the observations it has access to, and the actions it can make. Actor and critic networks are created by connecting the different layers between each other. Finally, all these elements are used in the rlDDPGAgent function to create a RL agent with these elements.

The program also shows how to define a training episode. For this example, initial and goal heights are set to a random value at each episode. This random setup helps avoiding overlearning.

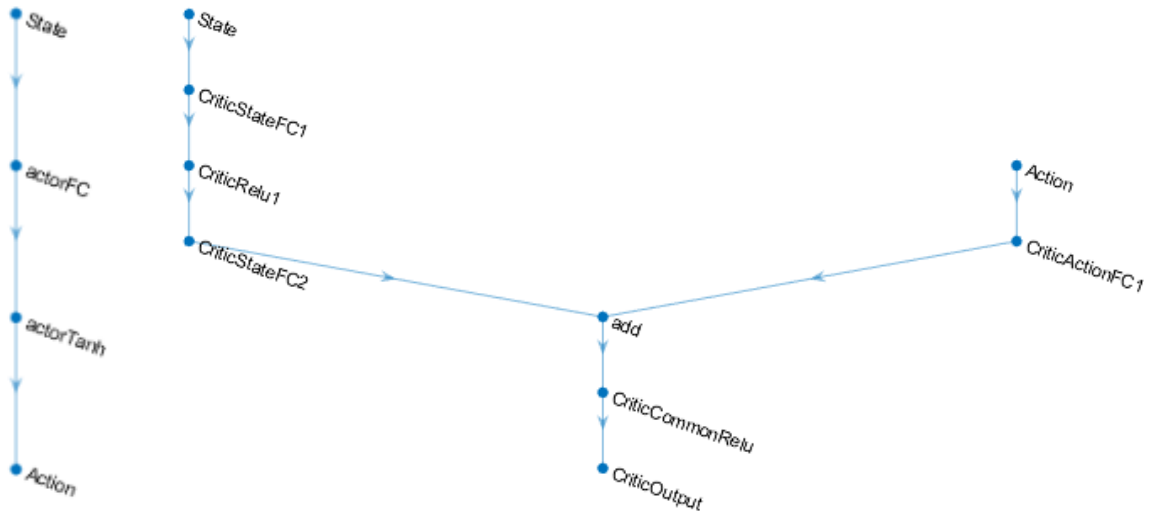


Figure 10 : Actor (left) and critic (right) networks architecture

III.a.ii. - Simulink integration

This example also shows the way of using the "RL Agent" block inside of a Simulink model. This block is connected to observation, reward and stop condition as an input, and gives the action made by the agent as an output. Figure 11 shows the basic architecture of a Simulink RL model.

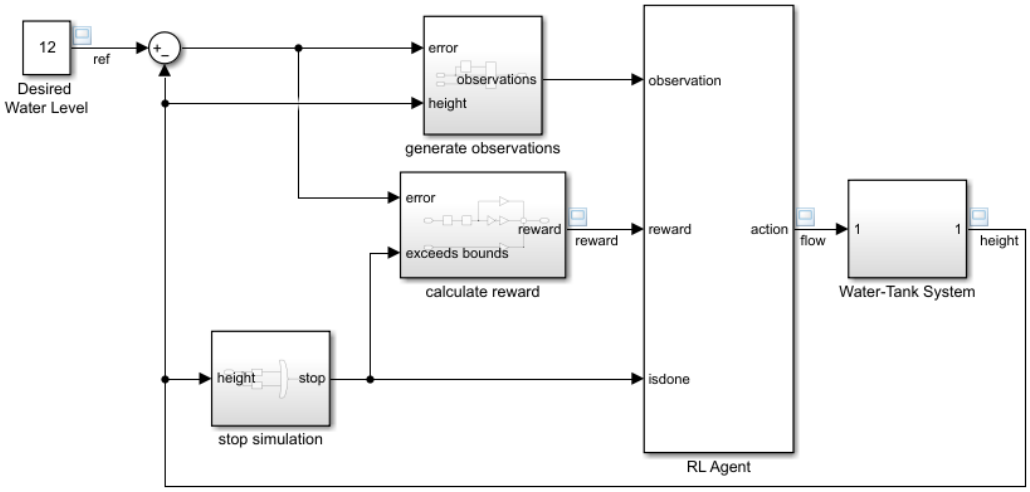


Figure 11 : Simulink water tank model

This model shows that the agent block is the main part of the model, creating a link between the environment (to the left) and the plant (to the right).

III.b. - Obstacle avoidance model

The first goal for this project has been to make the agent control the QCAR inside an environment by moving without a specific goal while avoiding obstacles. The agent has then been setup as follows:

- Observations: distance and position of the nearest wall
- Actions: velocity and steering control
- Reward: see pseudocode below

ALGORITHM 1: Reward "obstacle avoidance" computing

```
dist ← smallest distance observed
vel ← actual velocity of the vehicle
IF (dist < 0.1) // Is there a collision?
    reward = -10 // Punishment
ELSE IF (dist < 0.5) // Is the vehicle near a wall?
    reward = dist // Try to get as far a possible
ELSE // The vehicle is far from any wall
    reward = dist + vel // Better reward if the vehicle goes fast
END IF
RETURN reward
```

This algorithm gives a better reward if the vehicle goes fast and stays far from any wall.

To help convergence and simplify expression of the reward, observations and actions have been normalized using MinMax normalization. Maximum values are easy to get in this case, being the limit fixed for the LIDAR and the maximum number of rays sent by it.

$$M_{norm} = \frac{M - M_{min}}{M_{max} - M_{min}}$$

Equation 9 : MinMax normalization

For each training episodes, the initial pose of the model is set to a random empty position in the map and an orientation between 0 and 2π rad.

Figures 12 and 13 show the Simulink model used for the training. "Detect obstacle" block simulates the LIDAR and returns the minimal distance and its index as observations for the agent. "calcRew" function implements previous pseudocode to compute the reward of the agent.

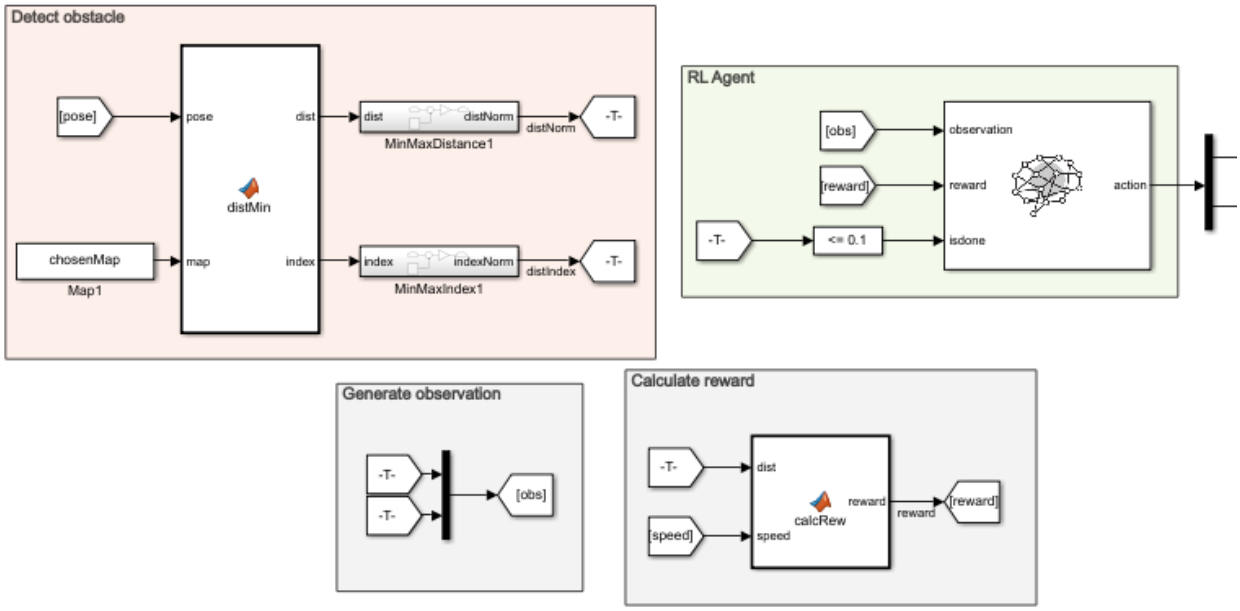


Figure 12 : RL agent modelling

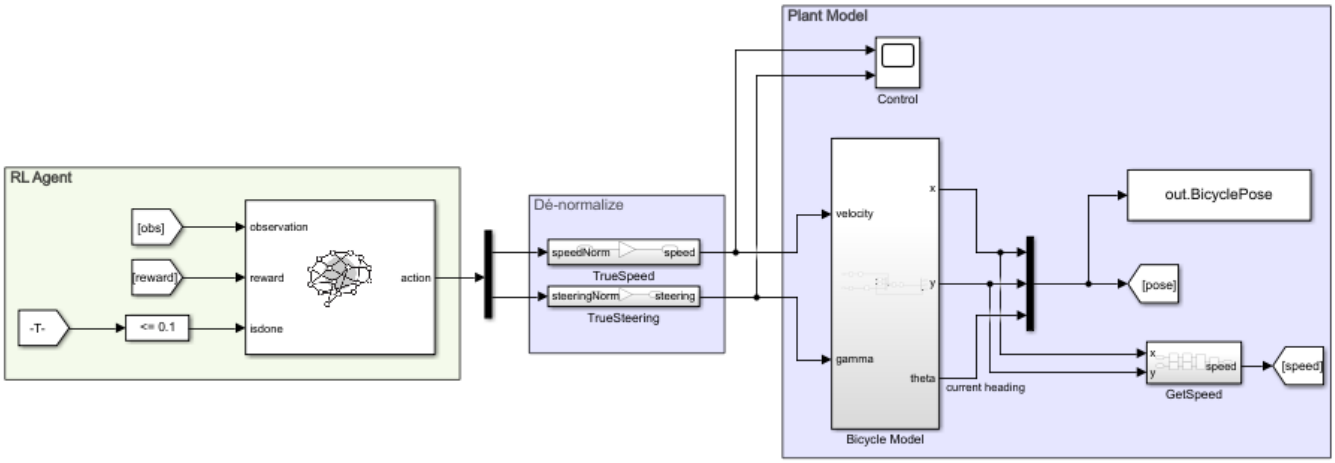


Figure 13 : Plant control

Annexes 1 and 2 show the results of two training sessions made with this model.

First training session has not led to a convergence of the policy. The model used for this experiment did not give the index of the closest wall as an observation. This led to the agent not knowing where the closest wall was, thus not knowing where to go to avoid it. This is why its best policy has been to drive circles to drive as less distance as possible and take the minimum amount of risks.

The second session showed the apparition of a gool learning. The problem was that the simulation stopped to early, the stopping criteria being set too low. Despite this, a policy started to appear, the vehicle avoiding obstacles and trying to reach a point the furthest possible to any obstacles, and then making small back and forth movements.

This result is satisfying to consider the capacity of the model to learn a policy with this implementation and can be improved by adding a better reward if the vehicle keeps getting further away from its starting point, instead of going back and forth after finding a good spot.

III.c. - Point-to-point driving

The second goal for this project has been to make the agent control the QCAR inside an empty environment by reaching a goal point as fast as possible. The agent has then been setup as follows:

- Observations: distance from goal in x and y axis
- Actions: velocity and steering control
- Reward: see pseudocode below

ALGORITHM 2: Reward "point-to-point driving" computing

```
distX ← distance from goal in x axis
distY ← distance from goal in y axis
IF (distX < 0.1 AND distY < 0.1) // Has the vehicle reached the goal?
    reward = 5
ELSE IF (distX < 0.5 OR distY < 0.1) // Is the vehicle aligned to the goal on one axis?
    reward = 2
ELSE // The vehicle is nowhere near the goal
    reward = 0
END IF
RETURN reward
```

The choice of getting the distance in each axis instead of getting distance and orientation from goal as in Peter Corke's work has been made because the agent had issues problems to converge in this configuration. Indeed, is the goal was exactly behind the vehicle, the index value of the orientation would have oscillated between minimum and maximum value (all the way to the left and all the way to the right). The choice has then been made to use observations that don't have this property.

For each training episodes, the initial pose of the model is set to a random position in the map and an orientation between 0 and 2π rad, and the goal position is set to a random position in the map.

Figure 14 shows the Simulink model used for the training. The plant control is the same as for the previous problem. "Away from goal" block computes the relative distances from goal in x and y axis and normalizes it. "distLeft" and "goalOrient" are computed for monitoring purposes.

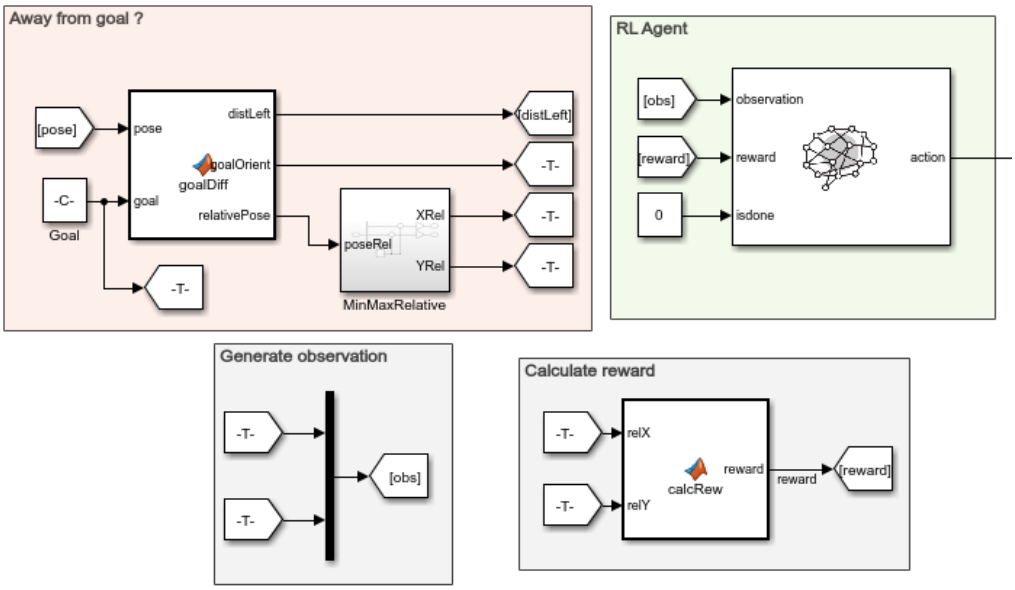


Figure 14 : RL agent modelling

Annex 3 shows the result of the best training session made with this model.

Unfortunately, the program has not been able to converge for this problem. For every initial condition, the vehicle turns around without any other action. A way of improvement could be to change the reward to adapt to the distance in each axis, and not just give a fixed reward when the distance is small enough. This way of improvement has been tried but results are not much better, needing a better tuning.

HIL INTEGRATION

I - Quanser Toolbox

To control the QCAR with a Simulink model, Quanser developed a MATLAB toolbox use to integrate their systems via HiL (Hardware in Loop). These blocks are used to read data sent by the QCAR from its sensors (accelerators, motor count for speed, LIDAR) and to write inputs used to control the QCAR (velocity and steering angle). For this project, the sensors needed to estimate the pose of the QCAR are the motor counts – to get vehicle's speed – and Z-gyroscope – to get its orientation, hence update its position with its speed. LIDAR data will also be extracted to detect obstacles.

I.a. - HIL blocks

Figure 15 shows "HIL Read" and "HIL Write" blocks, used to control and get information from the QCAR.

"HIL Write" block can be modified to have a more complete control of the QCAR. It can, for example, be used to control the lightnings of the QCAR. For this project, the objective is to have a control as close as possible to the bicycle model used to train the agent. The block will then be used the same way as shown in figure 15.

"HIL Read" block can also be modified to get more information. The QCAR gets gyroscopes and accelerometers for the 3 axes. However the motor count as been used to control the speed, to have a more precise information of the speed during the simulation.

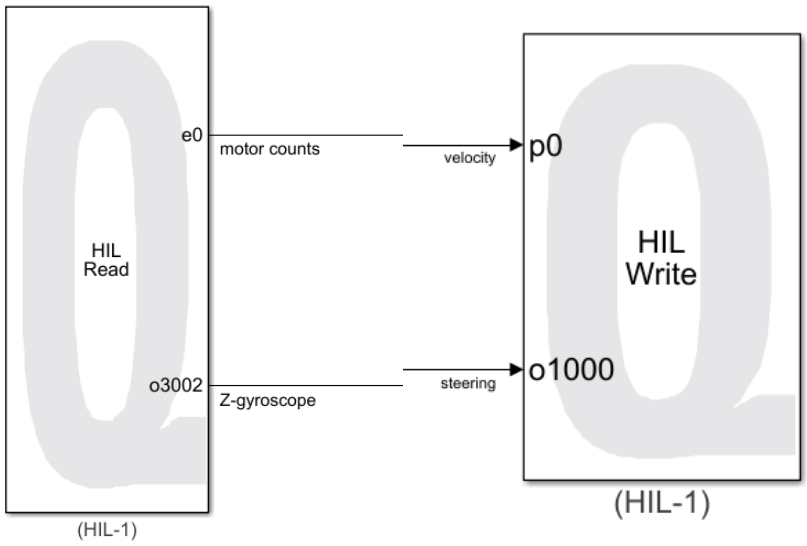


Figure 15 : HiL blocks from Quanser toolbox

The Z-gyroscope was unfortunately poorly calibrated and did not give an acceptable value for determining orientation. The QCAR therefore did not determine the correct pose with respect to reality, which made it difficult to use the HiL for the project.

I.b. - LIDAR data acquisition

For more specific sensors, Quanser toolbox gives access to more specific blocks. Figure 16 shows the Simulink model used to extract information from the LIDAR. First output uses the "Ranging Sensor" block to get the same distance values as simulated in [D.2.b]. Second output gets the index of these distance values, going from 0 to 2π with a step of 0,5°.

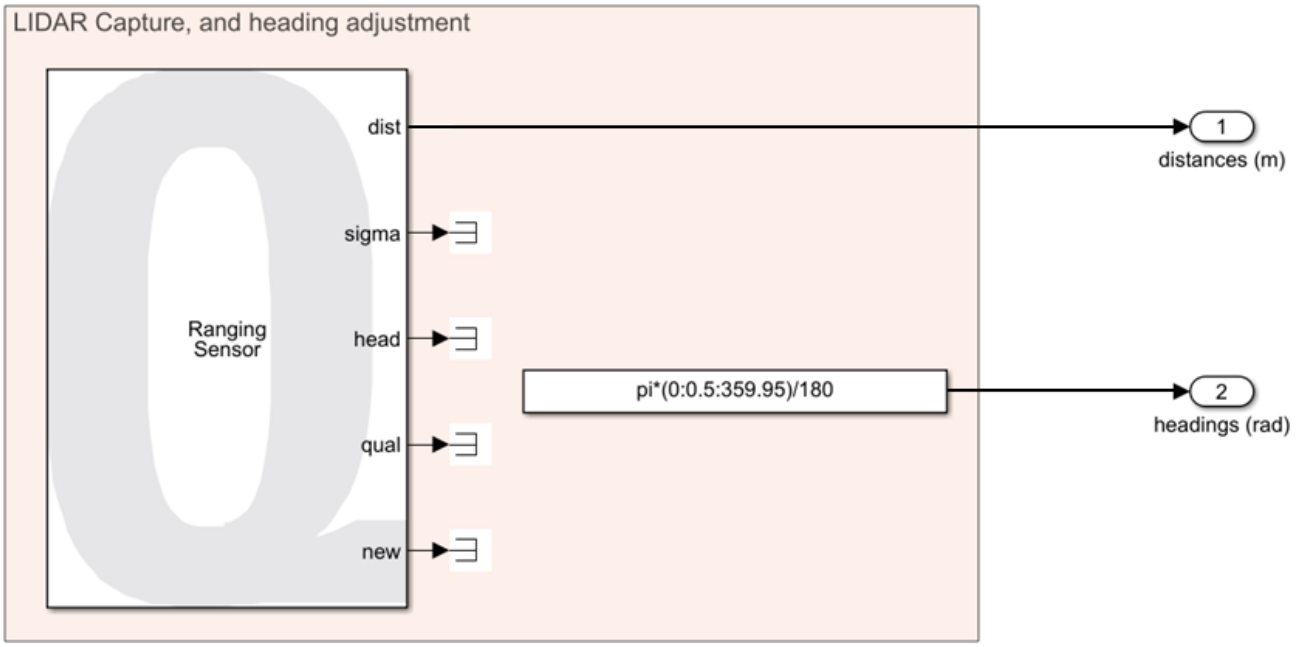


Figure 16 : LIDAR integration

LIDAR was also poorly calibrated during the project, so its data weren't usable. This problem has been corrected so HiL integration should no longer be a problem.

II - HIL Architecture

The bad calibration of sensors unfortunately made it impossible to integrate with the trained RL agent. However, this section will show an integration of the QCAR on Peter Corke's model for point-to-point driving. Annex 4 shows the complete model, and it will be detailed below.

Figure 17 displays the control of the QCAR via the computing made with formulas represented on Corke's work.

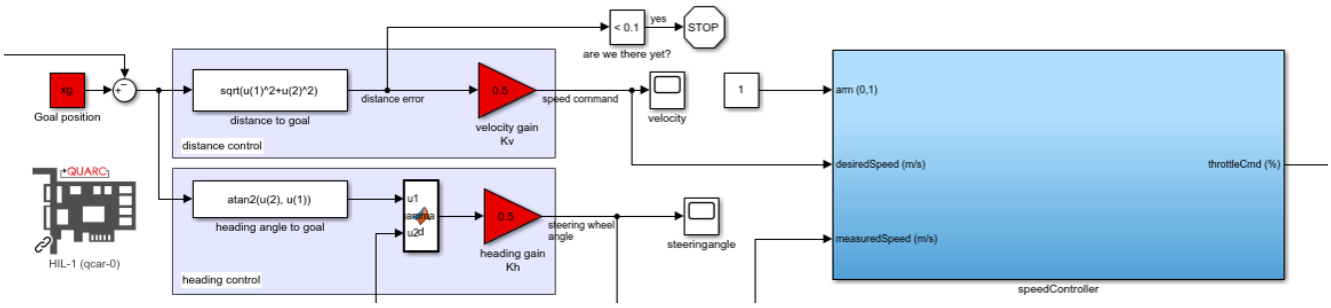


Figure 17 : Simulation model usage

Desired speed is corrected via a PID controller – the "speedController" block – to calibrate the output sent to the QCAR compared to its actual speed. Steering angle is also corrected via a "Saturation" block to avoid exceeding QCAR's limits.

Bicycle model is used to estimate the true pose of the vehicle. The measured true speed of the vehicle and the calculated desired steering angle are used as inputs of the "Bicycle Model" block. It returns the pose used to compute the next desired velocity and steering.

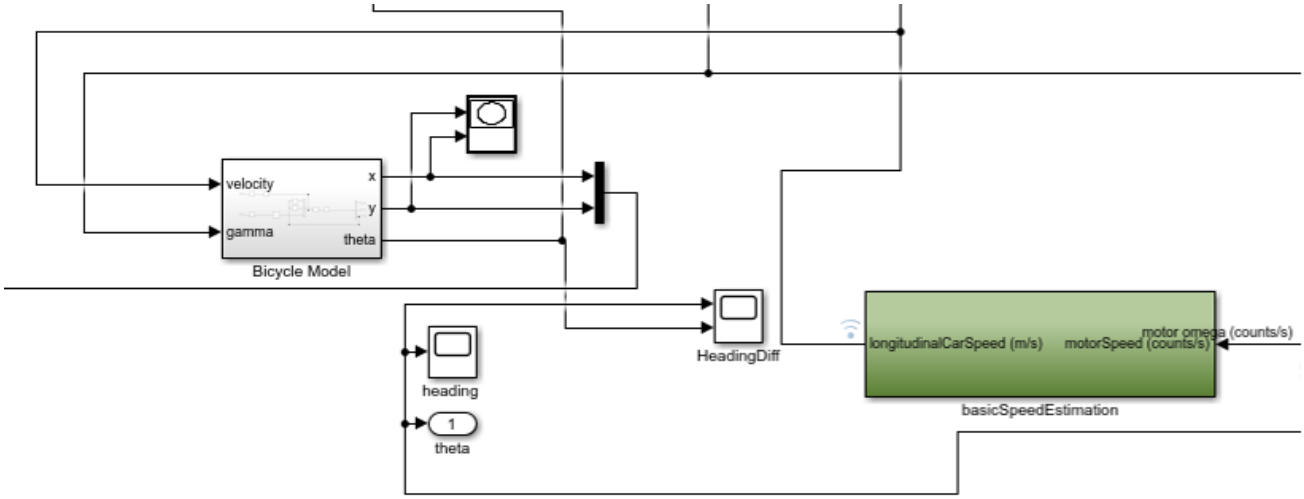


Figure 18 : Pose estimation

True speed of the vehicle is computed with "basicSpeedEstimation" block, getting the motor count – found with the "HIL Read" block – as an input and converting it by knowing the motor specificities.

Bicycle model has been used there as a monitoring tool. Indeed, this model is the one that showed the poor calibration of the sensors. This model was a way to estimate the pose of the vehicle despite the unusable Z-gyroscope data.

III - Results discussion

Quanser Toolbox is a powerful tool for HiL integration. The integration is quite easy and really complete.

The only flaw would be its sensibility to hardware. A bad calibrated sensor makes the model unusable, and the wireless connection can also be seen as a problem when trying to make it work in a noisy environment.

CONCLUSION

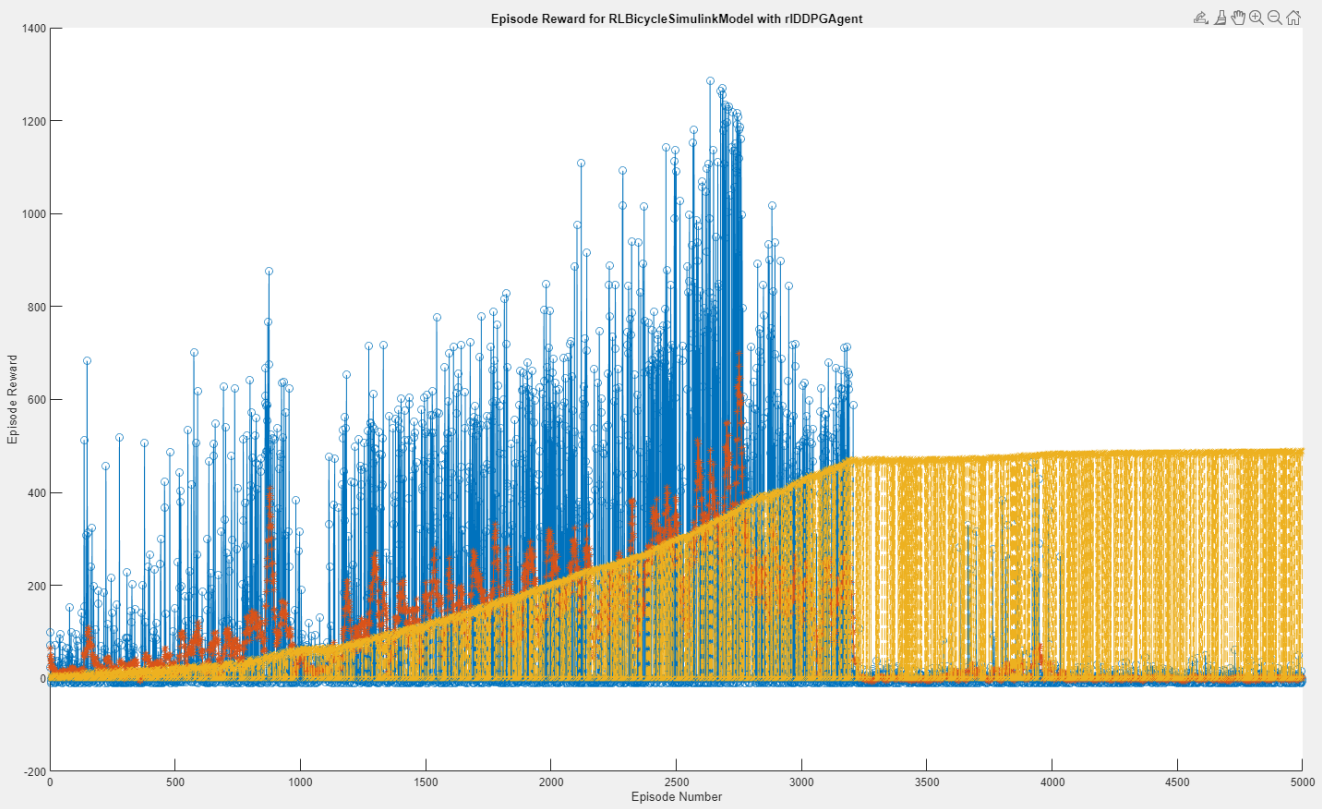
This project has been a great way to get the hang on RL algorithms. Already familiar with this field, I got to increase my knowledge during this year by digging deeper in a lot of aspects. Thus, this project has been a way of bringing reflexions about the correlation between simulation and reality, that is a really important aspect in this field where agent training is made in simulation, but the field of application uses hardware. So, it also asked for a great effort in result analysis. This project also made me appropriate a simplified but very compete model. This was a way of learning how to extract most important information of a system to simplify its expression. Finally, it was great to have the opportunity to work with the QCAR, which is a complex but interesting system, that also was a good introduction to all the problems related to embedded systems to be careful with (connection, calibration, monitoring).

The biggest difficulty of this project has been to fully understand DDPG. This algorithm share the same flaw as every Deep Learning algorithms: it works in a black box way, without a lot of information about the training. So, this aspect also asked for a lot of thinking about the good data to monitor during the training, and the ways to have good feedback to improve it easily.

Now that the sensors have been fixed, the next step of this project would be to fully integrate the HiL model with an RL agent, and start trying to work on more complex tasks, such a road following of interacting with each other for fleet moving.

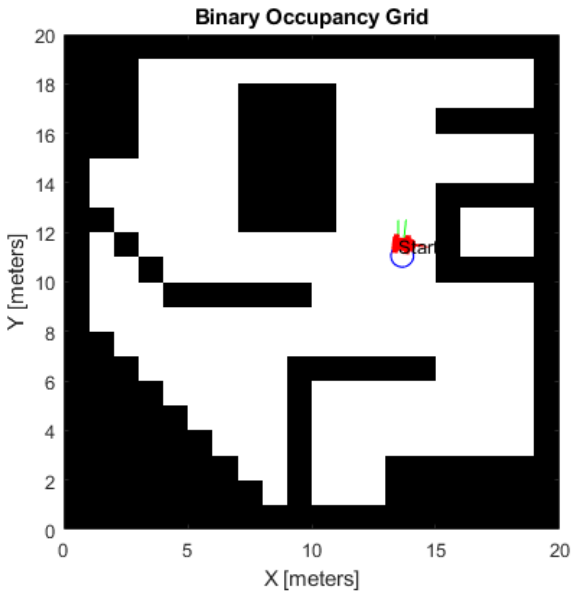
ANNEXES

I - Obstacle avoidance: First training experiment

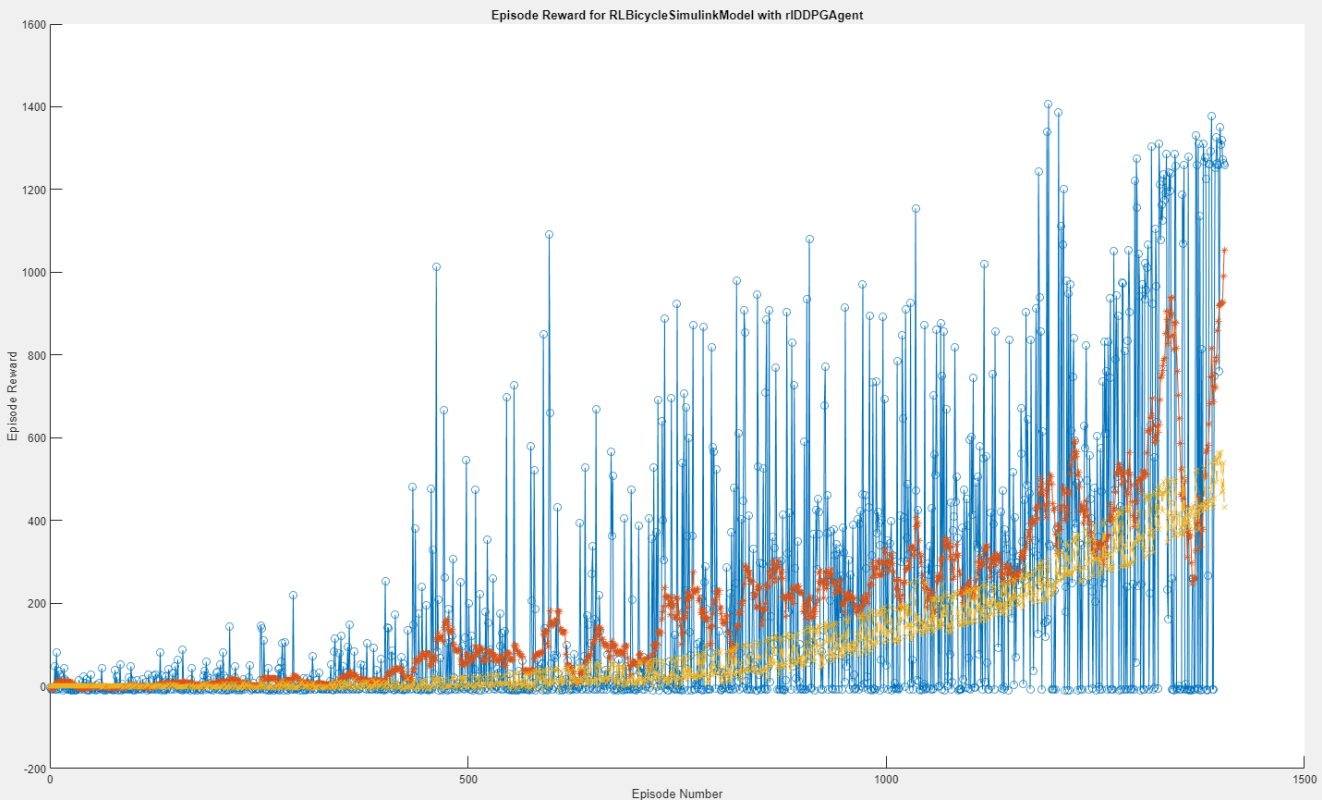


Blue curve: reward for each episode

Red curve: mean reward for last 100 episodes

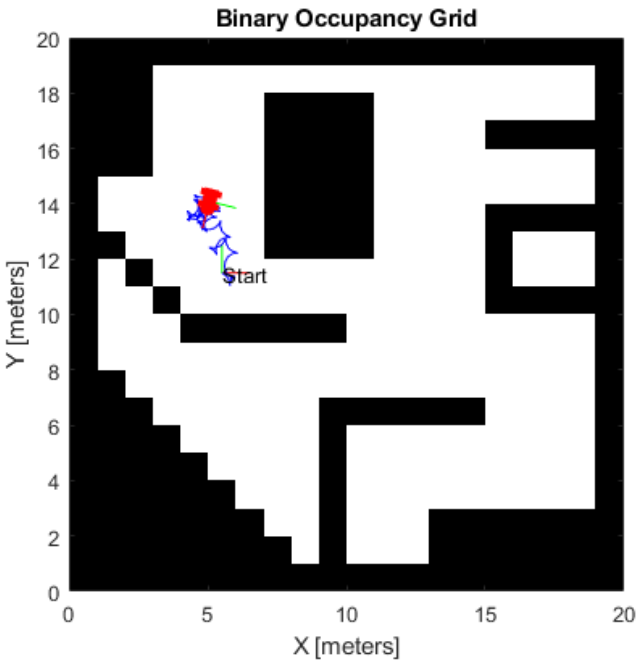


II - Obstacle avoidance: Second training experiment

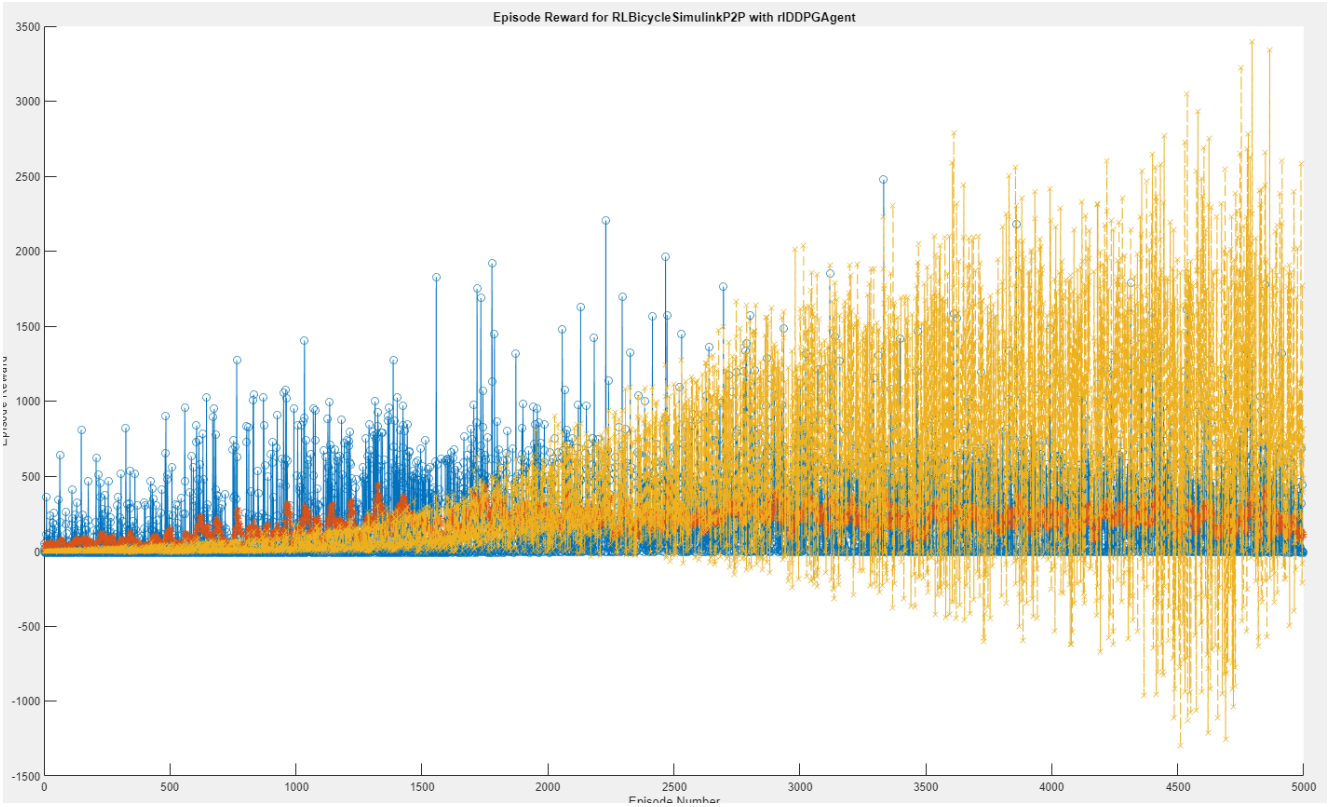


Blue curve: reward for each episode

Red curve: mean reward for last 100 episodes

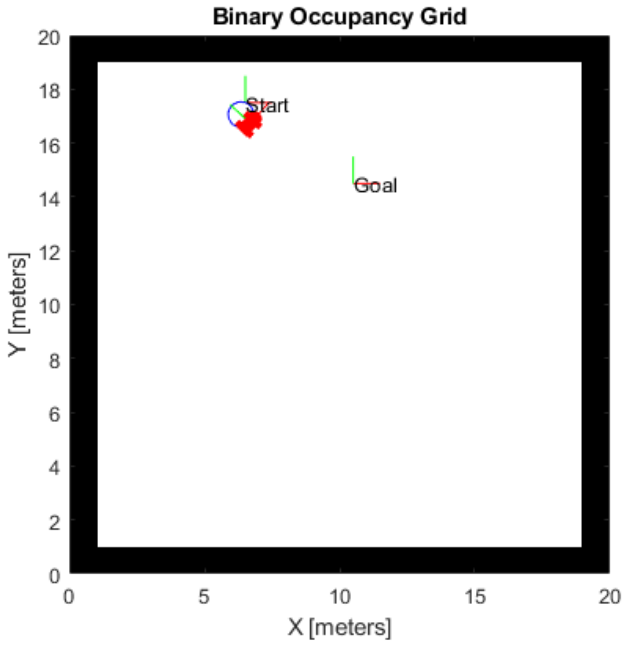


III - Point-to-point driving: training results



Blue curve: reward for each episode

Red curve: mean reward for last 100 episodes



IV - Point-to-point driving: HiL integration