

Rapport TER Voitures Autonomes

Kévin Hoarau, Charlène Surault, Martin Raynaud

8 mai 2023

Table des matières

1	Introduction	3
2	La voiture	4
2.1	Partie mécanique	4
2.2	Partie électronique	5
2.2.1	Schéma synoptique et photos	5
2.2.2	Description des différents composants utilisés	7
3	La simulation	8
3.1	Présentation du simulateur : Webots	8
3.2	Le circuit	9
3.3	Modèle de la voiture réelle	10
3.3.1	Paramètres de la voiture TT-02	10
3.3.2	Roues de la voiture	11
3.3.3	Le Lidar	12
4	Partie Martin et Charlène : Méthode des tentacules	13
4.1	Présentation de la méthode	13
4.1.1	Pré-calcul des trajectoires réalisables	13
4.1.2	Choix de la trajectoire optimale	14
4.2	Adaptation et implémentation sur le modèle réduit	15
4.2.1	Adaptation mécanique	15
4.2.2	Pré-calcul	16
4.2.3	Calcul temps réel	18
4.3	Difficultés observées	19
4.3.1	Adaptation des paramètres mécaniques	19
4.3.2	Gestion de la largeur de la voiture	20
4.3.3	Influence de la vitesse	21
4.4	Utilisation d'une caméra afin de mesurer les caractéristiques de la voiture	22
4.4.1	Mise en place	22
4.4.2	Algorithme récupérant les données de la caméra	22
4.4.3	Exemple de mesure effectuée : mesure du rayon de braquage	22
4.5	Conclusion méthode des Tentacules	23
5	Partie Kévin : Pilotage par réseau de neurones	24
5.1	Objectif : "SimToReal"	24
5.1.1	Problèmes rencontrés	24

TABLE DES MATIÈRES

5.1.2	Méthodes pour améliorer l'environnement d'apprentissage	25
5.1.3	Méthodes d'amélioration de l'efficacité et fiabilité de l'apprentissage	25
5.2	Apprentissage par renforcement	27
5.2.1	Principe	27
5.2.2	Application à la voiture autonome	28
5.2.3	Algorithmes d'entraînement	29
5.2.4	Algorithme retenu	29
5.2.5	"Proximal Policy Optimization" (PPO)	30
5.2.6	Modèle de réseau de neurones	31
5.3	Entraînement du réseau de neurone sur le simulateur	31
5.3.1	Environnement d'apprentissage	32
5.3.2	Simulation	38
5.4	Passage à la réalité	42
5.4.1	Implémentation dans la voiture réelle	42
5.4.2	Résultats	43
5.5	Conclusion "SimToReal"	45
5.5.1	Travail réalisé	45
5.5.2	Améliorations possibles	45
6	Conclusion	47
6.1	Améliorations possibles pour le futur	47
7	Remerciements	48
Références		49



Chapitre 1

Introduction

Le TER anticipé que nous avons réalisé tout au long de cette année avait deux objectifs principaux :

1. Faire concourir deux voitures à la course de voitures autonomes de Paris-Saclay (CoVaPSy) qui se déroulait cette année le Samedi 15 Avril 2023 ([Pour plus d'informations sur la course](#)).

Pour cela, il a fallu notamment gérer la partie software de ces 2 voitures afin de les rendre aptes à rouler en autonomie le jour de la course. La partie Hardware (chassis, pièces mécaniques et électroniques) était déjà fournie en quasi-totalité et était fonctionnelle.

2. Implémenter un algorithme différent dans chaque voiture permettant aux voitures d'être autonomes et de faire des tours de piste le plus rapidement possible.

Afin de parvenir à implémenter ces algorithmes, il a fallu mettre en place en parallèle un simulateur permettant de les tester et de les améliorer plus facilement.

Les versions finales de ces 2 algorithmes (celui de Kévin Hoarau et celui de Charlène Surault et Martin Raynaud) sont trouvables sur ce Git qui regroupe également les TER réalisés par tous les étudiants des années précédentes : [Git TER EEA Voitures Autonomes](#)

Ce compte-rendu est décomposé en plusieurs parties regroupant l'ensemble du travail réalisé sur l'année.

D'abord, la partie Voiture détaille les différents composants électroniques utilisés par la voiture et les différents liens entre ces composants.

La partie Simulation décrit les différents paramètres utilisés en simulation pour obtenir une voiture fonctionnelle avec tous ses composants ainsi qu'un circuit permettant à la voiture de s'entraîner en simulation.

Les deux parties suivantes présentent le travail effectué concernant l'implémentation des algorithmes d'automatisation de la voiture (Méthode des tentacules pour Charlène et Martin et pilotage par réseau de neurones pour Kévin)

Enfin vient une conclusion permettant notamment de faire un listing des différentes améliorations qui pourraient être mises en oeuvre dans le futur afin d'améliorer la qualité et les performances des voitures.

Chapitre 2

La voiture

2.1 Partie mécanique

La plupart des pièces mécaniques utilisées étaient fournies dans ce kit montable : Site détaillant les éléments présents dans le kit

Pour plus de détails sur l'utilisation des autres pièces et leur agencement au sein de la voiture : https://ajuton-ens.github.io/CourseVoituresAutonomesSaclay/voiture_type/

A été ajouté à la voiture cette année un kit de drift permettant d'augmenter l'angle de braquage maximal des roues le faisant passer de 18° à 30° environ. Cependant, seule la voiture de Martin et Charlène en a été équipée cette année, pas celle de Kévin. Nous espérons que l'année prochaine toutes les voitures concourants pour la course pourront en être équipées.

2.2 Partie électronique

2.2.1 Schéma synoptique et photos

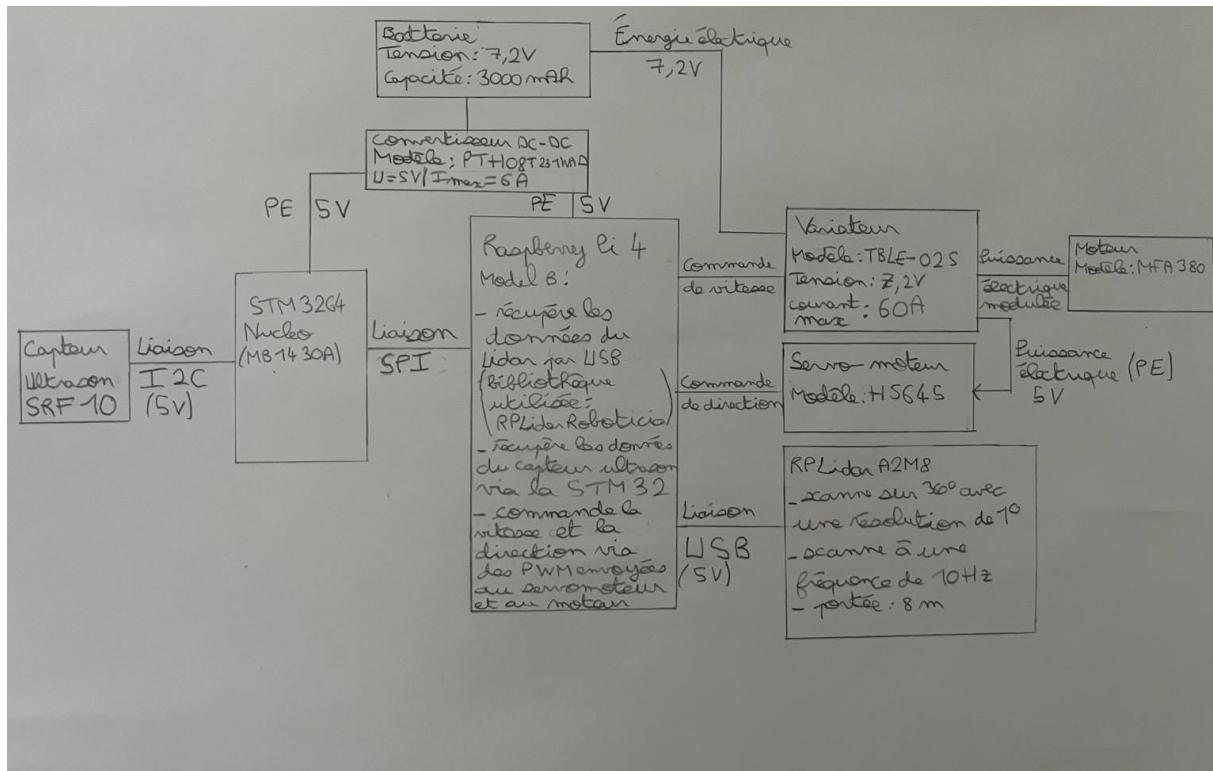


FIGURE 2.1 – Schéma synoptique de la partie électronique de la voiture

Ce schéma présente les différents composants électroniques utilisés par les deux voitures de l'ENS pour la course du 15 Avril 2023 et les liens existant entre ces différents composants permettant de gérer le transport d'énergie et de données dans toute la voiture. D'autres composants sont présents sur la voiture mais n'ont pas été utilisés cette année :

- les (2) télémètres infrarouges présents à l'arrière de la voiture (voir figure 2.3) servants en complément du télémètre à ultrason en permettant d'élargir le champ de vision, à l'arrière, de la voiture.
- une fourche optique présente sur l'arbre de transmission permettant d'obtenir la vitesse du véhicule en temps réel et donc de réaliser un asservissement de vitesse avec l'aide de la STM32

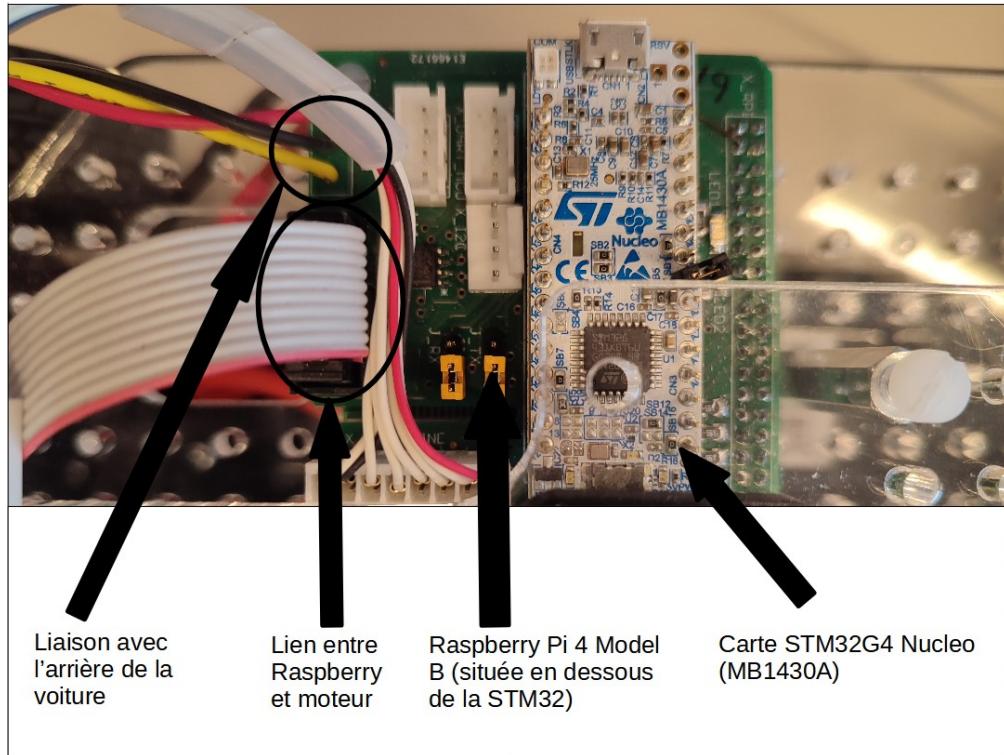


FIGURE 2.2 – Partie électronique de la voiture 1 : STM32 et Raspberry Pi

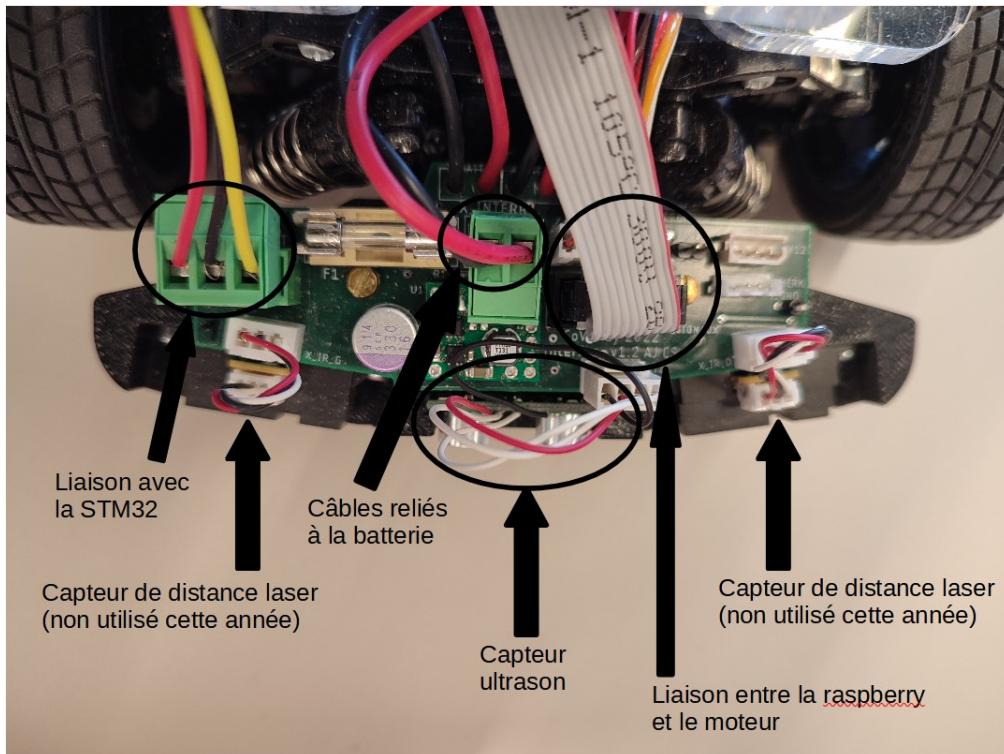


FIGURE 2.3 – Partie électronique de la voiture 2 : Capteurs et liaisons avec les autres composants électroniques

2.2.2 Description des différents composants utilisés

Les différents composants utilisés sont :

1. Partie alimentation électrique :

(a) Batterie :

- tension délivrée : 7,2 V
- capacité : 3000 mAh

(b) Convertisseur DC-DC :

- tension d'entrée : 7,2 V
- tension de sortie : 5 V (pour alimenter la raspberry et la stm32)

[Fiche technique du composant](#)

2. Partie contrôle-commande :

(a) Raspberry Pi 4 B :

- processeur : Quad-core A72
- mémoire : 8 Go

[Datasheet de la carte](#)

(b) Micro-contrôleur STM32G4 (MB1430A) :

- processeur : ARM Cortex M4
- fréquence CPU : 170 MHz
- mémoire flash : 128 Ko
- mémoire RAM : 32 Ko

[User Manual de la STM32](#)

3. Partie châssis et actionneurs :

(a) Le moteur :

- tension : 7,2 V
- vitesse max : 26000 tr/min

[Pour plus d'informations](#)

(b) Le variateur :

- tension : 7,2 V
- courant max : 60 A

[Pour plus d'informations](#)

(c) Le servo-moteur :

- couple max $7,7 \text{ kg.cm}^{-1}$

[Pour plus d'informations](#)

4. Télémètre ultrason :

- portée : de 4 à 30 cm
- ouverture : 45°

[Datasheet du module](#)

Chapitre 3

La simulation

3.1 Présentation du simulateur : Webots

Le simulateur que nous avons utilisé tout au long du TER est Webots dans sa version R2022b. Le simulateur est proposé par Cyberbotics et est un logiciel totalement gratuit permettant de créer un environnement dans le but de simuler des machines robotiques notamment, dans notre cas, la voiture à taille réduite. Webots possède toutes les bibliothèques nécessaires pour la création d'une voiture fidèle au modèle de la TT-02. Le programme pilotant la voiture est fait sous Python même s'il est possible de programmer en Java, C++ ou Matlab.

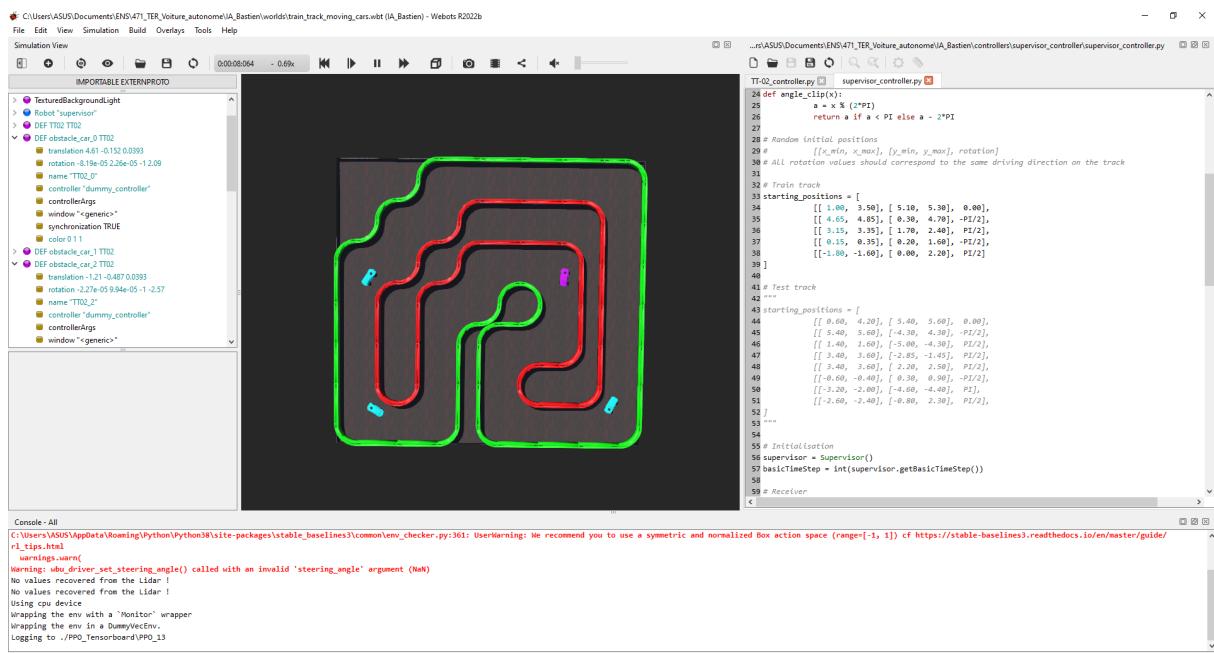


FIGURE 3.1 – Interface du simulateur Webots

Sur la figure 3.1 à droite, on peut apercevoir l’arborescence des éléments présents dans l’environnement. A gauche, un éditeur pour pouvoir modifier les programmes. A noter, qu’il est possible de lancer le programme pilotant la voiture depuis un IDE externe tel que Spyder.

3.2 Le circuit

Le circuit visible sur la figure est constitué :

- Bloc de ligne droite de 1m de longueur
- Bloc de ligne droite de 0.5m de longueur
- Bloc de virage de rayon 0.5m

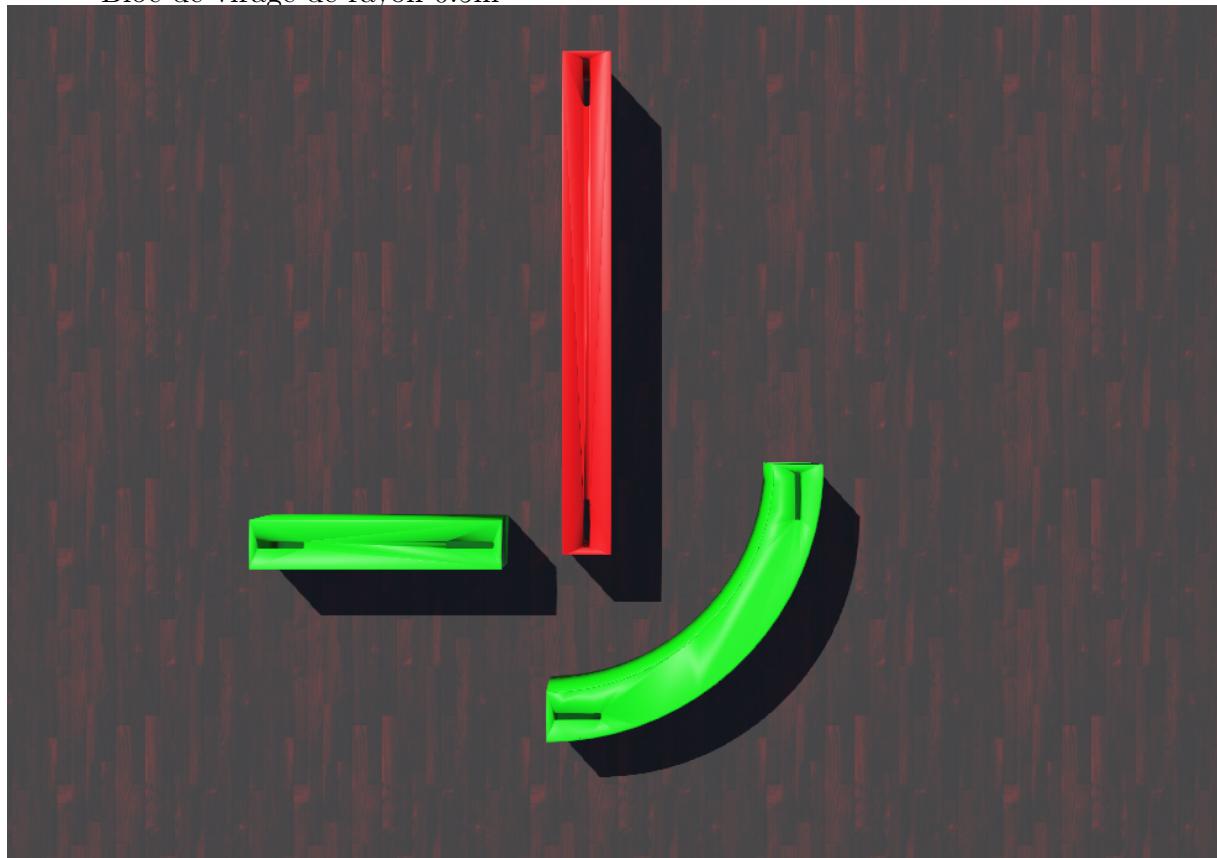


FIGURE 3.2 – Blocs des éléments pour la construction de la piste

Ces éléments sont les modèles 3D des véritables blocs servant le jour de la course et fabriqués par l’IUT de Cachan. Le choix des couleurs pour les bordures intérieures et extérieures est en concordance avec les vrais couleurs utilisées le jour de la course.

Les éléments de la piste sont disponibles sur le dépôt [GitHub public](#) de la course. Une boîte de collision est rattachée à chaque ensemble de blocs de la piste afin de détecter la voiture lorsqu’elle dévie dans le décor. Cet ajout, bien que nécessaire dans notre cas, complexifie la simulation et les calculs sont alors plus lourds pour l’ordinateur. Il est donc nécessaire, voire indispensable, d’utiliser une machine assez puissante pour faire tourner la simulation.

3.3 Modèle de la voiture réelle



FIGURE 3.3 – Modèle de la voiture sur Webots

Afin de créer un modèle réaliste de la voiture TT-02, nous nous sommes basés sur une voiture directement disponible dans Webots : l'Altino. L'Altino avait pendant un moment servi comme voiture pour les simulations avant la création d'une TT-02. Webots permet une grande liberté dans la modification des modèles déjà présents, ce qui nous a permis d'adapter les paramètres pour coller au mieux à la voiture réelle. C'est ainsi que nous avons pu reproduire la voiture réelle dans Webots, sous forme d'un "Proto", un format de fichier du logiciel pour les voitures et les robots. Le "Proto" de la TT-02 est lui aussi disponible sur le [GitHub public](#).

3.3.1 Paramètres de la voiture TT-02

Les paramètres de la voiture ont été trouvés en partie dans le manuel utilisateur. Les autres manquants ont été déterminés avec des tests réalisés au cours de l'année.

Paramètres de la voiture TT-02 :

- trackFront : 0.15
- trackRear : 0.15
- wheelbase : 0.257
- minSteeringAngle : -0.314
- maxSteeringAngle : 0.314
- suspensionFrontSpringConstant : 100000
- suspensionFrontDampingConstant : 4000
- suspensionRearSpringConstant : 100000
- suspensionRearDampingConstant : 4000
- wheelsDampingConstant : 5
- physics Physics density -1 mass 2.331
- radarCrossSection 100
- wheelFrontRight TT02Wheel name "front right wheel"
- wheelFrontLeft TT02Wheel name "front left wheel"
- wheelRearRight TT02Wheel name "rear right wheel"
- wheelRearLeft TT02Wheel name "rear left wheel"
- type "4x4"
- engineType "electric"
- engineSound ""
- brakeCoefficient 700
- time0To100 10
- engineMaxTorque 0.0234
- engineMaxPower 32
- engineMinRPM 1
- engineMaxRPM 13800
- gearRatio [-0.62 0.62]
- maxVelocity 36

3.3.2 Roues de la voiture

Les roues de la voiture sont basées sur les roues du modèle de véhicule Ackermann disponibles directement sur Webots. Ce modèle de roue a été adapté pour correspondre aux roues de la TT-02. Peu d'informations sont disponibles dans la documentation de la TT-02. Par conséquent, des mesures ont été réalisées au cours de l'année afin d'affiner le modèle.

Paramètre d'une roue :

- thickness 0.035
- subdivision 32
- curvatureFactor 0.1
- edgeSubdivision 2
- rimRadius 0.025
- rimBeamNumber 10
- rimBeamWidth 0.01

- centralInnerRadius 0.0070
- centralOuterRadius 0.0130
- rimBeamThickness 0.010
- rimBeamOffset 0.0096

3.3.3 Le Lidar

Le Lidar est aussi disponible dans les données de base de Webots. On a paramétré le Lidar pour qu'il soit le plus proche possible de celui qui est utilisé. Ainsi le Lidar est configuré pour renvoyer des données entre -100 et 100° avec une résolution de 1°. Les données du Lidar sont limitées à 8m comme pour le Lidar réel.

Chapitre 4

Partie Martin et Charlène : Méthode des tentacules

4.1 Présentation de la méthode

Cette partie du TER se concentre sur l'adaptation ainsi que l'implémentation de la méthode dite des tentacules au modèle réduit de voiture mis à notre disposition et présentée au Chapitre 2. La méthode que nous utilisons a été décrite dans l'ouvrage "*Driving with Tentacles - Integral Structures for Sensing and Motion*" écrit par Felix von Hundelshausen, Michael Himmelsbach, Falk Hecker, André Müller et Hans-Joachim Wuensche (voir [Tentacles \[Fel09\]](#)). Cet article présente une méthode de calcul de trajectoire utilisé dans les voitures autonomes. Il s'agit, contrairement aux algorithmes comportementaux, d'une méthode se basant principalement sur les caractéristiques mécaniques du véhicule. Elle se comporte de deux parties majeures. La première consiste en un calcul préalable des différentes trajectoires réalisables par le véhicule tandis que la deuxième correspond au choix en temps réel de la trajectoire optimale.

4.1.1 Pré-calcul des trajectoires réalisables

La première étape de la méthode des tentacules repose sur un algorithme de pré-calcul des trajectoires. Ce calcul repose sur une suite de discrétisations ainsi qu'à la mécanique associée.

La première discrétisation est celle en vitesse. Nous allons discrétiser la vitesse actuelle de la voiture sur un ensemble de 15 vitesses possibles. A chaque rang de vitesse ainsi obtenu correspondra différentes propriétés mécaniques de la voiture comme le rayon de braquage ou encore la distance parcourue en un temps donné. Cette discrétisation permet d'adapter au maximum les paramètres mécaniques associés à la voiture réelle à ceux du modèle.

La deuxième discrétisation effectuée est celle en angle de courbure. En effet, nous

étudions un ensemble d'angles de courbures allant de l'angle de braquage maximum à gauche à l'angle de braquage maximum à droite afin de déterminer le plus grand nombre de trajectoires réalisables par le véhicule possible. L'article sur lequel la méthode se base fait appel à 81 angles discrétisés. Un rayon de courbure est associé à chaque angle de courbure en fonction des propriétés mécaniques de la voiture. On en déduit des trajectoires nommées "tentacules".

Enfin, la dernière discréétisation à faire est celle en distance de parcours de la voiture. Chaque tentacle est divisée en segments dont le nombre est constant. Cette discréétisation permet par la suite de déterminer la tentacle la plus longue qui correspond à la trajectoire optimale.

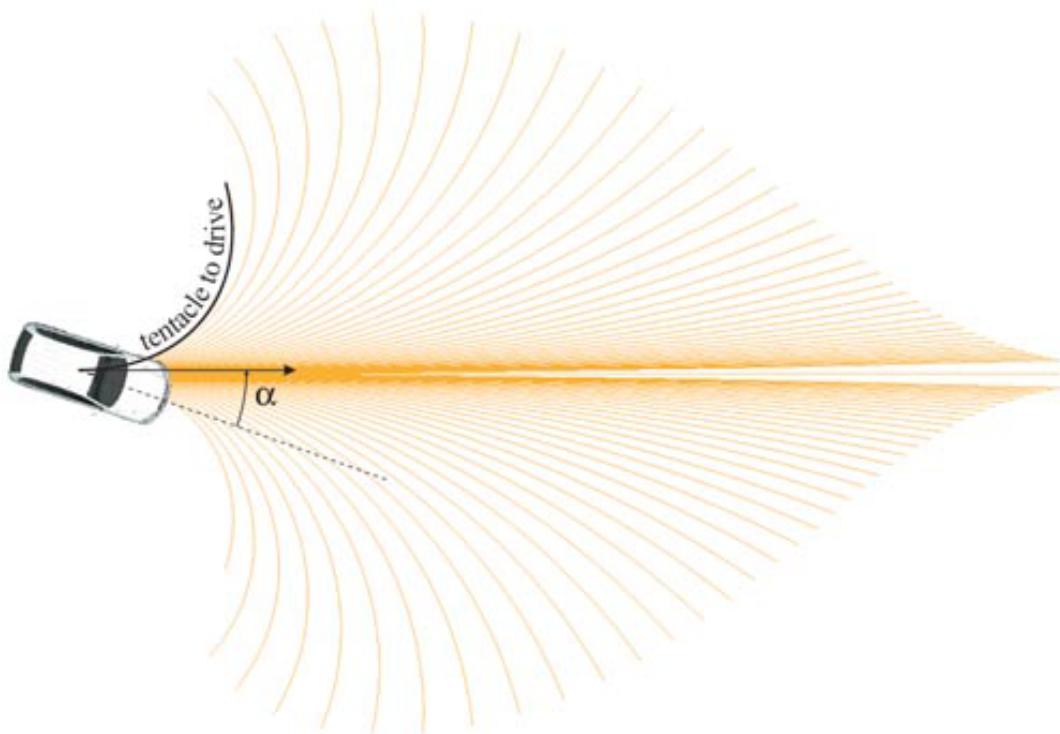


FIGURE 4.1 – Exemple d'une figure de tentacules à une vitesse donné présenté dans l'article étudié

Une fois les calculs effectués sur un ordinateur externe au véhicule, les trajectoires sont sauvegardées puis transmises à l'ordinateur de la voiture.

4.1.2 Choix de la trajectoire optimale

La deuxième étape de la méthode des tentacules repose sur un algorithme en temps réel codé sur le processeur de commande de la voiture. Une fois les trajectoires possibles transmises à la voiture, le choix d'une trajectoire optimale se fait à chaque pas de temps du processeur.

Une fois le rang de discréétisation de la vitesse actuelle obtenue, les données du Lidar sont superposées à la figure de tentacules correspondant au rang de vitesse. Cela permet de tronquer les tentacules au niveau des obstacles afin de ne garder que les points réellement atteignables. Il est alors possible de déterminer le tentacule le plus long, puis d'en déduire l'angle de courbure de commande à appliquer au véhicule.

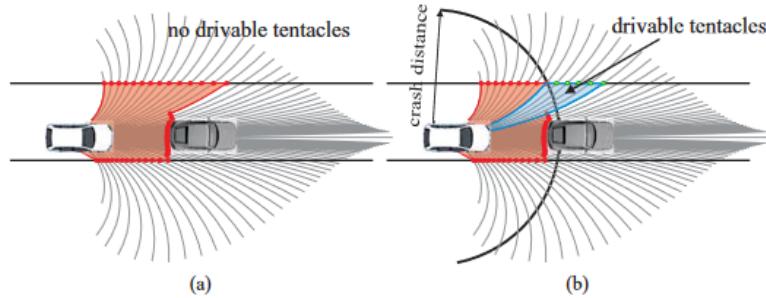


Figure 11. The case in which a car is blocking the right lane of a road and only a narrow passage is left to pass the car. The red points mark the locations along the tentacles where the vehicle would hit either the car or the road border. As can be seen, no tentacle is free of obstacles. Hence, by neglecting the distance to an obstacle, all tentacles would be classified undrivable (a). In contrast, panel b shows the concept of classifying tentacles as undrivable only in case of being occupied within a speed-dependent crash distance (see main text). In this case, some drivable tentacles remain, allowing a pass of the car.

FIGURE 4.2 – Schéma explicatif du choix de tentacules présenté dans l’article. Le tentacule choisi est le plus long parmi ceux possibles présentés en bleu.

4.2 Adaptation et implémentation sur le modèle réduit

4.2.1 Adaptation mécanique

La voiture que nous utilisons est un modèle réduit à 1/10ème. L’étude sur laquelle nous nous basons concerne des voitures réelles. Nous devons donc adapter les échelles de longueur et de vitesse au modèle réduit. On applique un facteur d’échelle de 1/10 aux paramètres mécaniques correspondant de l’article. On suppose dans un premier temps que notre modèle réduit reste fidèle aux voitures réelles, au facteur d’échelle près.

Un autre paramètre important que nous devons prendre en compte est le rayon de braquage maximale de notre voiture. Comme ce dernier n’apparaît pas directement dans les équations, nous mesurons le rayon de courbure maximal de la voiture. A faible vitesse, nous imposons un angle de braquage maximal à la voiture. Une caméra placée au dessus de la voiture et un traqueur posé sur la voiture permettent de tracer la position de la voiture sur quelques tours. On en déduit ensuite un rayon de courbure de l’ordre de 70cm. Cette valeur va permettre par la suite d’adapter les différents paramètres du modèle à notre voiture.

4.2.2 Pré-calculation

Nous utilisons pour le pré-calculation les équations données dans l'article étudié. Les paramètres utilisés sont les suivants :

- n : nombre de vitesses discrétisées
- j : rang de la vitesse de travail
- K : nombre de tentacules
- k : rang du tentacle de travail
- ρ : facteur exponentiel
- q : rang normalisé de la vitesse $\in [0; 1]$
- $\Delta\phi$: angle maximal pour la plus faible vitesse
- R_j : rayon initial pour la vitesse de rang j
- r_k : rayon du tentacle de rang k (pour une vitesse de rang j)
- l : longueur du plus petit tentacle pour la vitesse de rang j
- l_k : longueur du tentacle de rang k (pour une vitesse de rang j)
- a : paramètre non détaillé pour le calcul de l
- b : paramètre non détaillé pour le calcul de l
- c : paramètre non détaillé pour le calcul de l
- d : paramètre non détaillé pour le calcul de R_j
- e : paramètre non détaillé pour le calcul de l_k

Les paramètres a à e sont utilisés dans l'article pour les calculs des rayons de courbure et de longueur des tentacules mais pas détaillés. Nous avons dû manuellement ajuster ces paramètres de sorte à ce que les tentacules pré-calculés collent au maximum aux trajectoires réellement réalisées par la voiture.

Les équations suivantes sont celles qui relient ces paramètres mécaniques :

$$q = \frac{j}{n - 1} \quad (4.1)$$

$$R_j = \frac{l}{\Delta\phi(1 - q^d)} \quad (4.2)$$

$$r_k = \begin{cases} \rho^k R_j & \text{si } k < \lfloor \frac{K}{2} \rfloor \\ \infty & \text{si } k = \lfloor \frac{K}{2} \rfloor \\ \rho^{K-k-1} R_j & \text{si } k > \lfloor \frac{K}{2} \rfloor \end{cases} \quad (4.3)$$

$$l = a + b \cdot q^c \quad (4.4)$$

$$l_k = \begin{cases} l + e\sqrt{\frac{k}{\lfloor \frac{K}{2} \rfloor}} & \text{si } k \leq \lfloor \frac{K}{2} \rfloor \\ l + e\sqrt{\frac{K-k-1}{\lfloor \frac{K}{2} \rfloor}} & \text{si } k > \lfloor \frac{K}{2} \rfloor \end{cases} \quad (4.5)$$

Ces équations permettent d'obtenir le rayon ainsi que la longueur de chaque tentacule. Nous implémentons ces équations sur un programme *Python*. Le tracé de ces tentacules pour un profil de vitesse donne le résultat suivant :

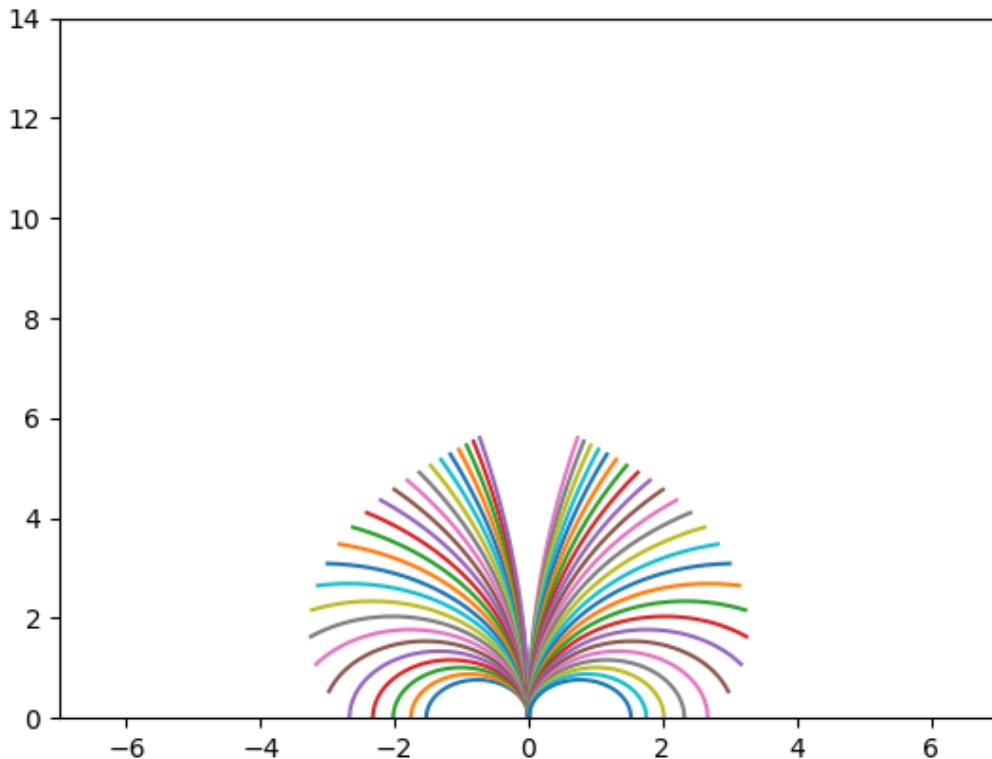


FIGURE 4.3 – Figure de tentacules obtenu pour une vitesse donnée grâce à l’application des calculs ci-dessus. Les distances sont données en mètres. Note : le tentacule du milieu est manquant car il est impossible d’afficher un cercle de rayon infini avec la méthode plt.Circle sur Python ; il est néanmoins bien pris en compte pour le reste du calcul.

Une fois le rayon et la longueur de chaque tentacule obtenu, nous pouvons passer à la discréétisation de chaque tentacule en distance. On discrétise dans un premier temps les tentacules dans un repère Cartésien pour le tracé de ces dernières puis dans un repère polaire qui sera utile pour la comparaison avec les données du Lidar. Chaque tentacule est ici divisé en 20 points repérés par leurs valeurs (X, Y) et (R, θ).

Les données (R, θ) sont classées dans un tableau à 4 dimensions :

- rang vitesse
- rang tentacule

- rang pas
- coordonnées

Elles sont ensuite stockées sous formes de tableau dans un fichier *json* que l'on charge dans la Raspberry de la voiture.

4.2.3 Calcul temps réel

La partie calcul temps réel repose sur un algorithme en *Python* implémenté dans la Raspberry de la voiture. Les données du Lidar sont récupérées en temps réel sous forme d'une liste de 360 cases, chacune correspondant à un pas d'angle d'un degré, et dans lesquelles se trouvent la distance de l'objet le plus proche à cet angle. Comme le Lidar est posé sur l'avant du véhicule, seule la plage de -100° à 100° est visible. Les cases 100 à 260 de notre liste sont donc à 0.

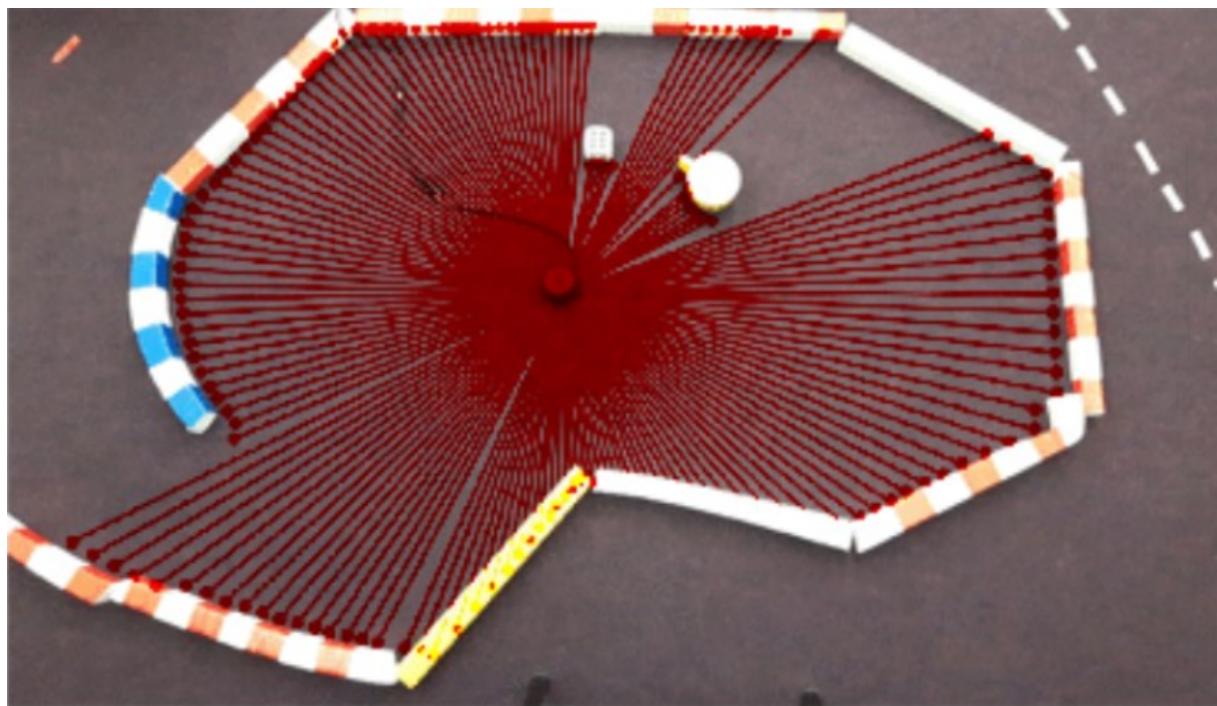


FIGURE 4.4 – Schéma explicatif de l'acquisition Lidar sur 360°

Le traitement se fait chaque 100 ms, ce qui correspond au temps d'acquisition du Liadar. Pour chaque point de chaque tentacule, nous récupérons sa coordonnée θ que nous faisons correspondre à un angle du Lidar. Si la coordonnée R du point est inférieure à la distance donnée par le Lidar à cet angle, cela signifie que le point est accessible, sinon le point est en dehors de la piste. Lorsqu'un point d'un tentacule se trouve en dehors de la piste, nous tronquons le tentacule afin d'obtenir uniquement les points accessibles par la voiture. Les tentacules tronqués ont ainsi une taille variable que nous pouvons calculer à l'aide du nombre de points restant ainsi que de la taille originelle du tentacule. Nous choisissons alors le tentacule le plus long qui correspond à la trajectoire optimale de la

voiture. Nous faisons correspondre l'indice du tentacule choisi à une commande moteur de direction des roues.

La vitesse de la voiture est choisie en fonction de la distance Lidar à l'angle 0 (en face de la voiture). Si la distance augmente par rapport à l'itération précédente, on augmente la vitesse et inversement pour une diminution de la distance. Cela permet d'accélérer dans les lignes droites et de ralentir dans les virages.

Une fonctionnalité de marche arrière a également été implémentée afin de se défaire de situations de blocage du véhicule. Ces situations peuvent arriver si la voiture se retrouve face à un mur ou à un autre véhicule par exemple. Deux capteurs ultrason se trouvent à l'arrière du véhicule et permettent d'avoir une distance approximative des obstacles à l'arrière. Ainsi, lorsqu'un obstacle est détecté par le Lidar à l'avant proche du véhicule et qu'aucun obstacle ne se trouve dans une distance de sécurité à l'arrière du véhicule, on inverse le sens de rotation des roues pendant une à deux secondes, de sorte à ce qu'un tentacule soit disponible pour se défaire de l'obstacle.

4.3 Difficultés observées

L'adaptation de la méthode des tentacules au modèle réduit 1/10ème a posé quelques difficultés que nous avons dû résoudre.

4.3.1 Adaptation des paramètres mécaniques

Comme nous l'avons expliqué dans les parties 4.1 et 4.2.1, la méthode des tentacules [Fel09] a été développée pour des véhicules de taille réelle. Comme les paramètres de cette méthode sont liés à la mécanique de la voiture, un grand nombre de ces derniers changent non seulement en fonction de l'échelle de la voiture mais également de son modèle (vitesses atteignables, longueur, largeur, rayon de braquage, inertie). Cependant, ces facteurs ne se retrouvent pas directement dans les calculs de la partie 4.2.2. Ils influent sur les paramètres en jeu selon des relations qui ne sont pas explicitées. Il serait possible, dans le cadre d'un travail de recherche plus long, de déterminer entièrement la mécanique du modèle réduit pour expliciter les relations liant la voiture aux paramètres. N'ayant pas eu l'opportunité de mener cette recherche, nous avons pris une approche plus expérimentale.

Dans un premier temps, nous avons utilisé directement les valeurs déterminées dans l'article pour l'algorithme de pré-calcul. Ces valeurs nous ont données des tentacules de forme adaptées à une voiture de taille réelle avec des longueurs de tentacules de l'ordre de 10 à 30 mètres et un rayon de courbure minimal de 5 mètres environ. Ces valeurs ne collant pas avec les valeurs déterminées pour notre modèle réduit, nous avons donc dû ajuster manuellement certains paramètres.

L'angle de braquage à la vitesse minimale $\Delta\phi$ a été modifié pour correspondre à celui

donné par le constructeur du kit de drift utilisé sur le modèle réduit. Les paramètres non déterminés, donnés en mètres dans l'article (a , b et e), ont également été multipliés par le facteur d'échelle de 1/10 pour correspondre à la taille du véhicule. Les résultats se rapprochaient de ceux attendus mais les trajectoires estimées ne correspondaient toujours pas totalement à celles effectuées par la voiture. En effet, un modèle réduit n'est pas exactement similaire à la voiture réelle. La répartition du poids n'est pas identique et la puissance moteur ainsi que sa vitesse ne sont pas proportionnelles à celle d'une voiture classique. Tous ces facteurs influencent grandement sur la dynamique de la voiture.

Finalement, au fil des courses sur circuit réalisées par la voiture, nous avons modifié progressivement les paramètres donnés dans l'article jusqu'à ce que les trajectoires pré-déterminées soient quasiment identiques à celles prises par la voiture. Nous avons ainsi obtenu les valeurs suivantes :

$$a : 8 \rightarrow 3.2$$

$$b : 33.5 \rightarrow 6.7$$

$$c : 1.2 \rightarrow 1.2$$

$$d : 0.9 \rightarrow 0.9$$

$$e : 20 \rightarrow 2.7$$

Ces valeurs nous ont permis d'avoir des trajectoires bien optimisées au long de la course.

4.3.2 Gestion de la largeur de la voiture

La méthode utilisée crée des tentacules fines : elles ne sont constituées que d'une ligne curviligne. Cependant la voiture n'est pas un point et possède une largeur qu'il est important de prendre en compte. En effet, utilisés comme tels, les algorithmes peuvent déterminer une trajectoire optimale qu'il serait impossible pour la voiture d'emprunter en raison de sa largeur. Ce cas peut arriver lorsque deux obstacles sont rapprochés de moins de la largeur de la voiture et que le Lidar peut apercevoir entre les deux.

Dans l'article [Fel09], la méthode utilisée consiste en la mise en place d'un espace de sûreté correspondant à la largeur de la voiture autour des tentacules générés. La figure 4.5 détaille le procédé utilisé.

Cette méthode reposant sur l'utilisation d'une plus grande quantité de données à traiter en temps réel, nous avons préféré utiliser une méthode plus simple. Au lieu de directement choisir le tentacle le plus long, nous observons les trois tentacules sur la droite et sur la gauche du plus long. Si la différence de longueur entre le plus long et les adjacents ne dépasse pas un seuil, cela est synonyme d'une absence d'obstacle autour du tentacle le plus long ; cette trajectoire est donc valide. Dans le cas contraire, une forte diminution de la longueur des tentacules adjacents au plus long signifie la présence d'un obstacle autour du tentacle choisi initialement. Dans ce cas, nous choisissons le tentacle du côté opposé à l'obstacle et nous recommençons la dernière étape jusqu'à avoir un chemin dégagé. Cette méthode n'est pas optimale, surtout lors de la présence d'obstacles très proches de la voiture, mais s'est révélée suffisamment efficace pour éviter

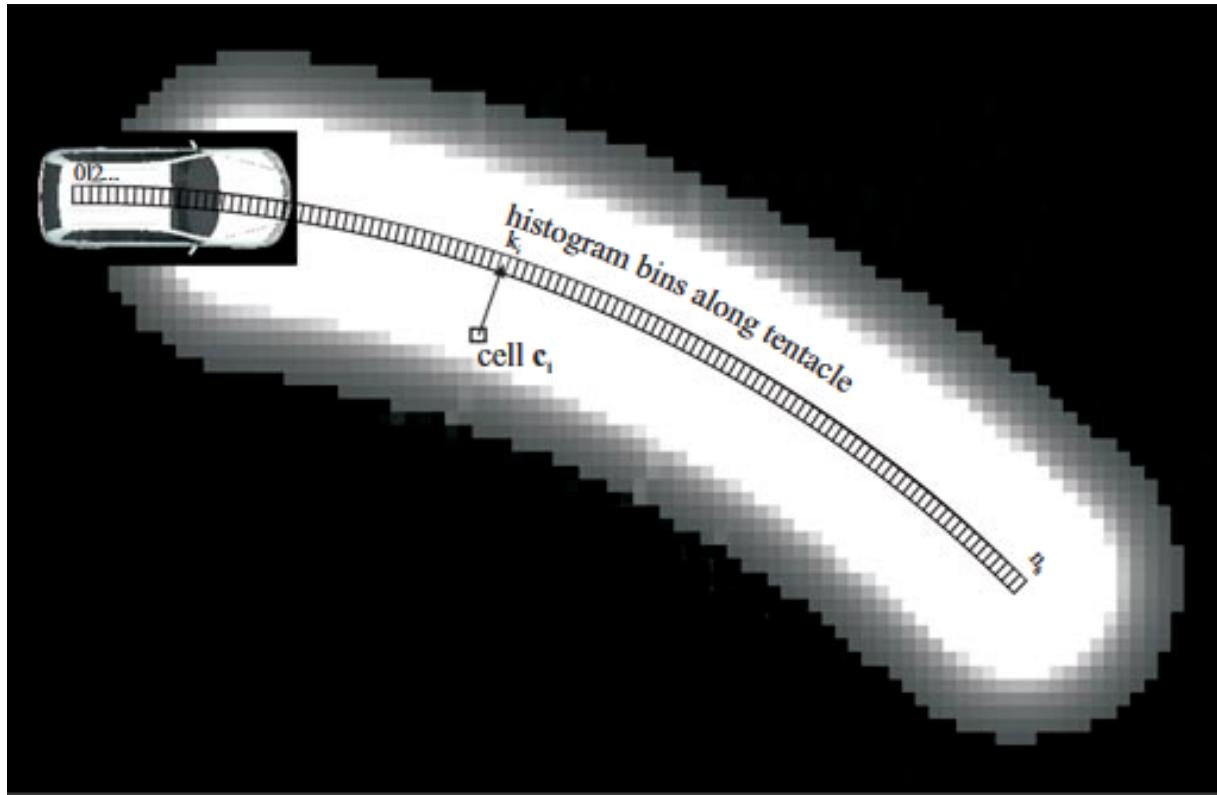


FIGURE 4.5 – Schéma explicatif de la méthode utilisée dans l'article. source : [Fel09]

la grande majorité des obstacles et gagner la course.

4.3.3 Influence de la vitesse

Un dernier problème que nous avons pu rencontrer concerne l'influence de la vitesse sur les figures de tentacules. En effet, le modèle proposé tient compte de la vitesse actuelle du véhicule pour la longueur et le rayon des tentacules. Nous sommes donc partis sur cette base pour les premiers essais. Néanmoins, après plusieurs essais, nous nous sommes rendus compte que ce modèle n'était pas fiable. Nous avons donc re-mesuré les rayons de courbures pour différentes vitesses correspondant à la plage utilisée pendant les courses. Nous avons pu observer que ce rayon restait inchangé pour toutes les vitesses testées. Comme la longueur absolue des tentacules n'entre en compte que pour la distance de freinage que nous n'étudions pas, nous n'avons alors aucun intérêt à conserver des figures de tentacules pour chaque pas de vitesse. Ainsi, pour les dernières courses, nous avons décidé de ne pas changer de figure de tentacules en fonction de la vitesse et de ne rester que sur celui de la vitesse initiale. Cette modification s'est révélé efficace lors des dernières courses.

4.4 Utilisation d'une caméra afin de mesurer les caractéristiques de la voiture

4.4.1 Mise en place

On accroche une caméra au plafond au-dessus de la piste afin que le champ de vision de la caméra couvre l'ensemble de la piste. C'est pour cela que la caméra utilisée initialement (la D435) a été remplacée par la D455 afin de couvrir l'ensemble de la piste.

On imprime par la suite des tags qu'on place sur la voiture et dans les coins de la piste. Les tags utilisés sont des AprilTags qui sont facilement exploitables car une bibliothèque python open source ([pupil-apriltags](#)) permet de réaliser facilement la reconnaissance de tags sur une image.

Afin d'être plus facilement identifiables sur l'image de profondeur obtenue par la caméra, les tags choisis sont des tags 16h5 ([Git regroupant les images de l'ensemble des AprilTags](#)). On choisit par exemple le tag 0 que l'on imprime en 14x14 cm et que l'on place sur la voiture afin de vérifier que l'algorithme présenté par la suite fonctionne correctement.

4.4.2 Algorithme récupérant les données de la caméra

Description succincte de l'algorithme :

- on récupère les données de la caméra
- on construit une image en profondeur de la piste à partir des données de la caméra
- à l'aide de la bibliothèque april-tags présentée précédemment, on détecte le tag sur chaque image en profondeur et notamment sa position sur la piste dans le plan (Ox,Oy) (par rapport à la position de la caméra considérée comme étant l'origine)

Les détails de la fonction et de la classe utilisées pour trouver la position du tag sont présentés ici : <https://pupil-apriltags.readthedocs.io/en/stable/api.html>
La dernière version du code sera déposée sur ce git : [Git TER EEA Voitures Autonomes](#)

L'algorithme obtenu cette année permet seulement une utilisation basique des tags et de la caméra. Il convient donc de le modifier et de l'améliorer si l'on souhaite faire des mesures plus complexes des caractéristiques mécaniques de la voiture.

4.4.3 Exemple de mesure effectuée : mesure du rayon de braquage

Dans le cadre de l'implémentation de la méthode des tentacules, il était nécessaire d'avoir une mesure du rayon de braquage minimal de la voiture (correspondant à un angle des roues de 30°) afin de générer des trajectoires adaptées à la voiture.

Protocole pour la mesure du rayon de braquage :

- on lance l'algorithme présenté précédemment
- on fait faire plusieurs tours sur elle-même à la voiture en la faisant braquer au maximum et on mesure la position (x,y) de la voiture à chaque tour de boucle de l'algorithme
- l'ensemble de ces points est stocké dans un tableau puis dessiné pour former un cercle en prenant soin d'enlever les valeurs aberrantes. On mesure alors le rayon du cercle ainsi obtenu.

Après application de ce protocole, le rayon de braquage mesuré était de 1,65 m environ. Hors en mesurant de façon approximative avec des bouts de piste mesurant 1 m chacun, on obtient 1,40 m environ.

Les résultats obtenus à l'aide des tags et de la caméra ont donc été considérés comme aberrants et n'ont donc pas été utilisées par la suite pour générer les "tentacules" de la voiture.

4.5 Conclusion méthode des Tentacules

L'article [Fel09] mettant en oeuvre la méthode des tentacules pour l'optimisation des trajectoires sur une voiture réelle, nombreux changements ont dû être effectués pour l'adapter à notre modèle réduit 1/10ème. Cependant diverses améliorations peuvent être effectuées.

L'étude approfondie de la mécanique de la voiture permettra d'optimiser les paramètres utilisés dans le calcul des tentacules. Les tentacules pourront ainsi mieux coller aux trajectoires prises et le modèle sera plus robuste aux changements de la dynamique de la voiture : niveau de la batterie, modification d'un des composants de la voiture, changement du poids par ajout ou retrait de capteurs, influence éventuelle de la carrosserie.

La méthode de prise en compte de la largeur de la voiture peut également être améliorée. Nous nous sommes rendue compte que la voiture frôlait de très près ou fonçait dans des obstacles qui lui arrivaient de profil. Nous pensons que cela est dû à une mauvaise prise en compte de la largeur de la voiture. Des travaux futurs pourraient être menés pour étudier la méthode de prise en compte de la largeur de la voiture décrite dans l'article.

Enfin, une amélioration de l'algorithme de marche arrière pourrait optimiser le temps de recul qui est à présent constant lorsqu'un obstacle est détecté. Cela pourrait permettre de faire une longue marche arrière lorsque cela est nécessaire plutôt que plusieurs reculées et avancées successives qui font perdre du temps.

Chapitre 5

Partie Kévin : Pilotage par réseau de neurones

5.1 Objectif : "SimToReal"

L'objectif de cette partie du TER est de pouvoir piloter la voiture de manière complètement autonome avec un réseau de neurones. Ce que l'on souhaite cependant, c'est que l'entraînement du réseau de neurones soit fait entièrement sur le simulateur et que le réseau de neurones soit implanté dans la voiture sans modification. C'est cela qu'on appelle l'objectif "SimToReal".

L'article [Wei21] fait un état de l'art de ce qui a déjà été réalisé et documenté dans la recherche concernant cet objectif.

5.1.1 Problèmes rencontrés

L'article fait état de plusieurs problèmes concernant le passage de la simulation à la réalité. En effet, il est en général difficile de créer un modèle qui représente parfaitement le robot (ou dans notre cas la voiture) avec une physique réaliste et qui modélise toutes les interactions avec l'environnement réel. Certains phénomènes physiques ne sont pas modélisables de manière certaine et dans le cas où cela est possible, les modèles peuvent être non-linéaires. Tous ces aspects font que le simulateur ne rend pas compte, de manière totalement fidèle, du système réel.

Un autre problème d'ordre pratique dans notre cas est que, pour apprendre, la voiture doit se prendre de manière inévitable des obstacles. Cela conduirait à un endommagement du matériel au fil du temps, ce qui n'est pas souhaitable. Il est donc nécessaire de réaliser la phase d'entraînement sur le simulateur.

L'article détaille plusieurs méthodes utilisées pour rendre le passage de la simulation

à la réalité le plus efficace possible.

5.1.2 Méthodes pour améliorer l'environnement d'apprentissage

Adaptation du simulateur

Pour pouvoir se rapprocher au maximum du comportement du système réel, il est nécessaire d'adapter le simulateur. Dans certains cas, un simulateur spécifique est développé pour affiner au mieux les interactions avec l'environnement réel. Il est par contre compliqué de créer depuis zéro un simulateur pour un seul système mais cela permet un passage plus efficace.

Identification paramétrique du modèle

Dans le cas où le comportement du système réel n'est pas facilement modélisable, on peut modéliser le comportement de chacun des actionneurs avec un réseau de neurones. Cela permet de ne pas avoir à connaître les équations qui régissent le comportement des actionneurs (dans le cas où elles existent) et de ne pas avoir à identifier tous les paramètres physiques. L'entraînement de ce réseau de neurones se fait par apprentissage supervisé. Cette manière de modéliser le système réel permet de prendre en compte les coefficients et caractéristiques de certains actionneurs que ne sont pas accessibles. En revanche, il n'y a aucune signification physique pour le réseau de neurones.

Ajout d'aléatoire

En raison du manque d'information sur certaines parties du système réel, notamment les capteurs, il est nécessaire d'ajouter une composante aléatoire sur les grandeurs venant de ses composants. En effet, l'ajout d'aléatoire (ou de bruit dans le cas des capteurs) dans les modèles du simulateur permet de rendre compte de l'aspect aléatoire du réel ou du bruit présent sur les capteurs.

5.1.3 Méthodes d'amélioration de l'efficacité et fiabilité de l'apprentissage

Le but de ces méthodes est de pouvoir réduire le temps d'apprentissage et d'améliorer les performances du réseau de neurones.

Modele of State Transition

L'utilisation de modèle cinématique et dynamique permet de décrire de manière analytique la description de tous types de tâches. En revanche, pour des tâches plus spécifiques, il est plus efficace d'utiliser un modèle local pour chacune d'entre elles. Ce modèle permet de créer de bonnes situations pour l'apprentissage, ce qui permet de réduire le nombre d'épisodes.

Ce modèle local peut être sous forme de réseau de neurones. Lors de la phase d'apprentissage par renforcement, ce réseaux se met à jour avec les situations amenant aux meilleures récompenses.

Modele of motor primitives

Par rapport à la méthode précédente, ici on décrit de manière paramétrique le mouvement que doit effectuer le robot. Il y a donc deux types de mouvement qui peuvent être décrits : les mouvements périodiques et ceux qui ne le sont pas. Les mouvements non-périodiques sont utiles pour décrire des situations où l'on cherche à atteindre un point. Les mouvements périodiques sont utilisés pour les mouvements rythmés.

Hierarchical controllers

Pour cette méthode, on décompose le problème suivant plusieurs niveaux afin de simplifier l'apprentissage. On peut par exemple avoir deux niveaux de commandes, où le réseau de neurones principale est à un haut niveau pour par exemple juste donner une consigne pour les actionneurs qui sont gérés par un autre algorithme pour la commande.

Distributed Controllers

Dans cette méthode, il est toujours question de rendre une tâche complexe en plusieurs tâches plus simples. En revanche ici, le réseau de neurones donne une commande à un groupe d'actionneur, réduisant la complexité du réseau de neurones. On peut par exemple regrouper des actionneurs dont les commandes sont symétriques pour qu'au niveau du réseau de neurones, il y ait moins d'éléments dans les espaces d'observation et d'action.

Teaching

Des essais manuels sont réalisés par l'humain pour donner de bons exemples comme base de l'apprentissage. En effet, on se base sur un comportement voulu au tout début de l'apprentissage pour éviter une longue période pendant laquelle les commandes sur le simulateur sont choisies aléatoirement.

Curriculum

Ici on commence par faire un apprentissage d'une situation simple. Une fois cette tâche réalisée, on complexifie la tâche et on attend que le réseau de neurones soit capable d'effectuer cette nouvelle étape. On continue ce processus de complexification pour arriver vers la tâche finale. Cela réduit encore une fois le temps de convergence de l'algorithme.

5.2 Apprentissage par renforcement

Dans cette partie, je me suis inspiré du rapport de stage de Bastien LHOPITALIER, étudiant en ARIA, qui a réalisé son stage sur le même sujet.

5.2.1 Principe

L'apprentissage par renforcement ("Reinforcement Learning" en anglais) est une catégorie de machine learning dans laquelle un agent interagit avec un environnement par un processus de décision. Une action réalisée par l'agent peut entraîner une modification de l'environnement. Chaque action réalisée dans un certain état de l'agent modifie ce dernier et une récompense est attribuée dont la valeur dépend de si l'action est positif ou négatif pour l'agent.

Lors de chaque étape, l'agent reçoit une observation de l'environnement dans lequel il évolue. Suivant cette observation, l'agent prend une décision. La décision est prise dans un ensemble d'action appelé espace des actions. Cet espace peut dépendre de l'état.

Un exemple simple est celui d'un jeu d'échec dans lequel l'observation correspond à la position de chacune des pièces de l'échiquier et l'espace des actions est l'ensemble des déplacements possibles des pièces (un fou ne peut pas être déplacé au lancement de partie par exemple). Naturellement, on souhaite que l'agent réalise la meilleure action possible suivant l'observation reçue. L'agent, pour atteindre ce but, applique une politique d'action (notée par la suite π) qu'il utilise pour sa prise de décision. A chaque récompense obtenue, cette politique est mise à jour. On espère ainsi atteindre une politique optimale.

Pour entraîner un agent, il y a plusieurs types de méthodes. Ces méthodes estiment la somme des récompenses qui devraient être obtenues dans le futur. Ces récompenses sont coefficientées afin de favoriser les récompenses obtensibles à court-terme. La politique obtenue est souvent modélisée par un réseau de neurones, dont l'actualisation modifie les poids du réseau.

- Les méthodes "value-based" : Méthodes qui se concentrent à estimer au mieux la récompense cumulative. Elles cherchent à obtenir une récompense cumulative optimale.
- Les méthodes "policy-based" : Méthodes qui se concentrent sur la politique optimale. Les valeurs de récompense peuvent ne pas être calculées.

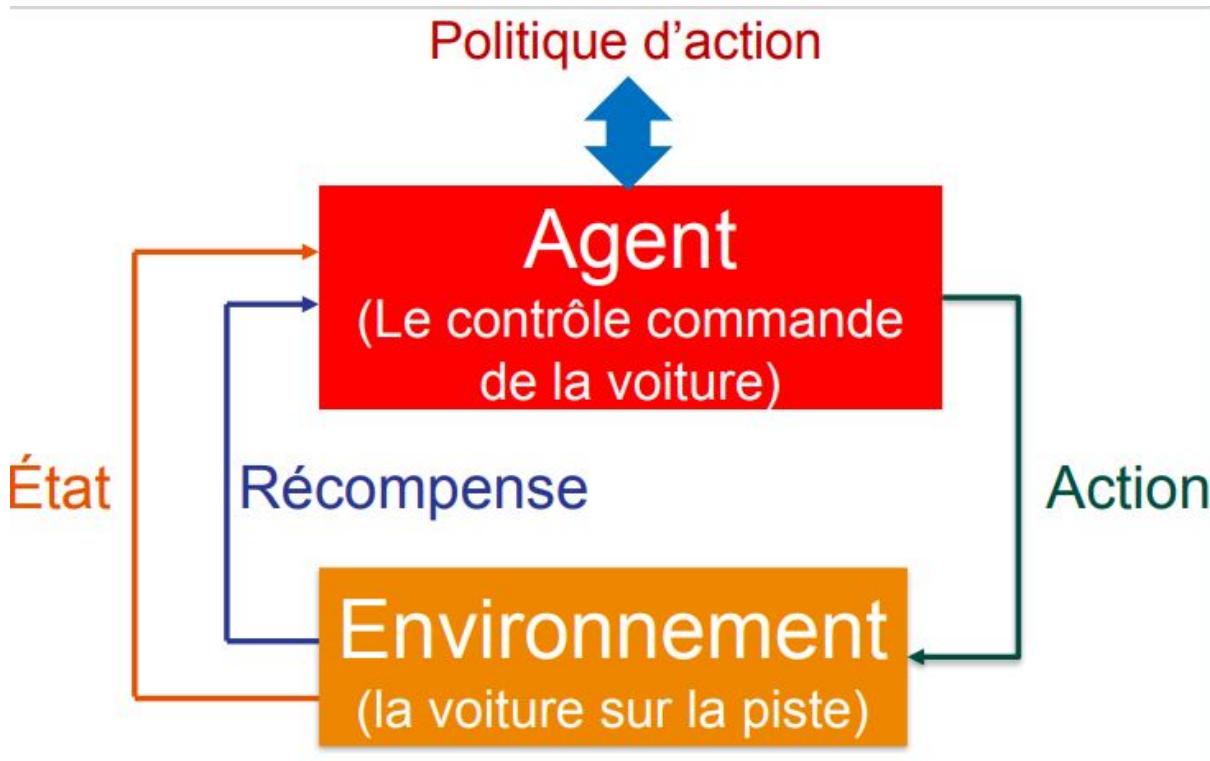


FIGURE 5.1 – Schéma explicatif de l'apprentissage par renforcement

- Les méthodes "actor-critic" : Méthodes qui emploient 2 réseaux de neurones. Le premier choisit l'action tandis que le second juge l'action choisie par le premier et la compare avec l'action qui était prévue.

5.2.2 Application à la voiture autonome

Dans le cas de la voiture autonome, il est nécessaire de définir un espace d'observation cohérent avec la réalité car toutes les grandeurs ne sont pas mesurables sur la voiture. Ainsi, pour l'espace d'observation, nous pouvons nous baser sur le Lidar qui permet de donner la distance des différents obstacles. De plus, si l'on avait mis en place l'asservissement de vitesse, alors la vitesse réelle pouvait être un élément de cet espace.

Dans le cas de l'espace d'action, seuls deux éléments sont possibles. En effet, les actions commandent les actionneurs de la voiture, à savoir le servomoteur pour la direction ainsi que le moteur à courant continu pour la propulsion. Il a donc été décidé que le réseau de neurones donnera un incrément d'angle pour la direction et un incrément de vitesse pour la propulsion.

TABLEAU 5.1 – *Récompense moyenne obtenue après apprentissage*

Algorithmme	Récompense moyenne
PPO	3.8601
DQN	2.6487
SAC	0.9715

5.2.3 Algorithmes d'entraînement

Quelques algorithmes d'entraînement de réseau de neurones sont présents dans la library Python Stable-Baselines3 [Ant21](Détaillée par la suite). Ces algorithmes sont les suivants :

- "Proximal Policy Optimization" (PPO) : Algorithme de type "policy-based"
- "Deep Q-Network" (DQN) : Algorithme de type "valued-based"
- "Soft Actor-Critic" (SAC) : Algorithme de type "actor-critic"

5.2.4 Algorithme retenu

Plus tôt dans l'année, un des objectifs de Bastien LHOPITALLIER était de déterminer lequel de ces algorithmes mentionnés précédemment était le plus adapté à notre problème d'entraînement de réseau de neurones. Les résultats présentés ci-après sont ceux obtenus par ce dernier et détaillés dans son rapport de stage.

Pour comparer les algorithmes, Bastien a entraîné 3 réseaux de neurones avec les 3 algorithmes différents. Les réseaux sont entraînés avec un learning rate de $5 \cdot 10^{-4}$ pendant 100 000 étapes et 10 essais. Les résultats de récompense moyenne sont résumés dans le tableau 5.1

On peut donc observer que l'algorithme PPO est l'algorithme qui donne le maximum de récompense pour des conditions d'apprentissage similaires. Bastien note aussi que l'agent entraîné par DQN conduit systématiquement vers le mur.

De plus, il a pu observer que pour l'algorithme SAC, la voiture ralentit de manière à éviter les murs. Il s'agit du seul algorithme qui présente ce comportement. En revanche, il arrive parfois que la voiture oscille entre l'avant et l'arrière devant un mur.

Concernant l'algorithme PPO, il s'agit de celui qui converge le plus rapidement.

De toutes ces observations, on a pu conclure que l'algorithme le plus adapté à notre projet est l'algorithme PPO.

Dans le paragraphe suivant, l'algorithme sera détaillé plus précisément.

5.2.5 "Proximal Policy Optimization" (PPO)

L'algorithme PPO [Joh17] a pour but d'être facile à implémenter, d'être facile à paramétrer et d'avoir une bonne efficacité au niveau des échantillons.

Cet algorithme est un algorithme de type "policy-based" avec lequel on peut utiliser un ensemble d'action discret ou continu.

On cherche à calculer la récompense cumulative G_t avec pour expression :

$$G_t = \sum_{k=t}^T \gamma^{k-t} r_k \quad (5.1)$$

Ici r_t est la récompense obtenue à l'instant t et le facteur d'actualisation $\gamma \in [0, 1]$. La convergence est assurer car $\gamma < 1$. On indique ici que les récompenses obtenues dans longtemps soient moins impactantes.

PPO possède une partie qui a pour but de choisir l'action suivant l'état obtenu en entrée. Une autre partie est chargée de déterminer G_t à l'instant t . On définit alors l'estimation d'avantage A_t qui est la différence entre la valeur de G_t réelle calculée après la prise de décision et la valeur estimée par le réseau de neurones. Ainsi on cherche à savoir si l'action a été meilleure ou pire par rapport à ce qui était attendu.

On définit la perte comme étant $\mathbb{E}_t[\log(\mathbb{P}_\pi(a_t|s_t))A_t]$, où $\mathbb{P}_\pi(a_t|s_t)$ est la probabilité que l'action a_t est choisie dans l'état s_t par le réseau de neurones suivant la politique π .

L'algorithme PPO utilise le principe de "Trust Region Policy Optimization". Ce principe s'assure que la politique π_{old} ne soit pas trop de la politique actualisée π .

On considère deux distributions de probabilités P et Q sur le même espace χ . On définit alors la divergence Lullback-Leibler (divergence KL) de Q à P comme :

$$KL[P, Q] = \mathbb{E}_{x \sim P}[\log \frac{P(X)}{Q(X)}] \quad (5.2)$$

Cela permet de mesurer la distance de la distribution Q par rapport à la véritable distribution P . L'objectif de TRPO est alors de maximiser la quantité $\mathbb{E}_t[\frac{\mathbb{P}_\pi(a_t|s_t)}{\mathbb{P}_{\pi_{old}}(a_t|s_t)} A_t]$ étant donné un paramètre δ tel que :

$$\mathbb{E}_t[KL[\mathbb{P}_{\pi_{old}}(.|s_t), \mathbb{P}_\pi(.|s_t)]] \leq \delta \quad (5.3)$$

On a donc pour objectif de ne pas trop s'éloigner des anciennes politiques et de trouver la meilleure politique π .

Pour ce qui de l'algorithme PPO, l'objectif devient le suivant : Maximiser la fonction $L^{CLIP}(\pi)$ avec

$$L^{CLIP}(\pi) = \mathbb{E}_t[\min(r_t(\pi)A_t, \text{clip}(r_t(\pi), 1 - \epsilon, 1 + \epsilon)A_t)] \quad (5.4)$$

Ici $r_t(\pi) = \frac{\mathbb{P}_\pi(a_t|s_t)}{\mathbb{P}_{\pi_{old}}(a_t|s_t)}$ et *clip* la fonction qui borne la fonction $r_t(\pi)$ entre $1 - \epsilon$ et $1 + \epsilon$.

5.2.6 Modèle de réseau de neurones

Le choix du modèle de réseau de neurones a été fait progressivement tout au long de l'année. Le réseau de neurones que m'a fourni Bastien LHOPITALLIER était composé de deux sous-couches ("Hidden layers" en anglais). Les neurones d'entrée ("Input layer") sont au nombre de 360, qui correspondent aux 360 valeurs données par le Lidar. Les neurones de sortie sont au nombre de deux correspondant à la commande de vitesse et la commande de direction. Le modèle de réseau de neurones par défaut de l'algorithme PPO implémenté dans la librairie Stable-baselines3 possède déjà deux sous-couches. Il s'agit d'un élément qui n'a pas été modifié au cours du projet.

Le premier modèle de réseau de neurones ne prenait en compte que les observations du Lidar. Je détaillerai les résultats ultérieurement concernant ce modèle.

Dans le but d'améliorer le comportement de la voiture, d'autres données ont été rajoutées dans l'espace d'observation. Dans un premier temps, j'ai rajouté les données du Lidar à "l'instant précédent". Cela donne une continuité dans l'observation des obstacles. De plus, j'ai aussi ajouté la vitesse de la voiture avant une nouvelle commande donnée par le réseau de neurones. Il en est de même pour la direction.

Le schéma ci-dessus illustre le réseau de neurones utilisé. Ce réseau de neurones est celui qui a fourni les meilleurs résultats par ceux qui sont détaillés dans cette partie, notamment au niveau de la voiture réelle.

J'ai essayé par la suite de modifier une dernière fois le modèle du réseau. Pour cela, j'ai rajouté la vitesse et la direction à "l'instant précédent" pour donner une accélération dans le cas de la propulsion ainsi que la vitesse angulaire de la voiture dans le cas de la direction. Ces grandeurs ne sont donc pas mesurées directement. Les résultats concernant ce modèle n'ont pas pu être exploités totalement par manque de temps.

5.3 Entraînement du réseau de neurone sur le simulateur

Dans cette section, je me suis basé une fois de plus sur le travail de Bastien LHOPITALLIER qui m'a transmis son travail à la fin de son stage. La partie qui va suivre

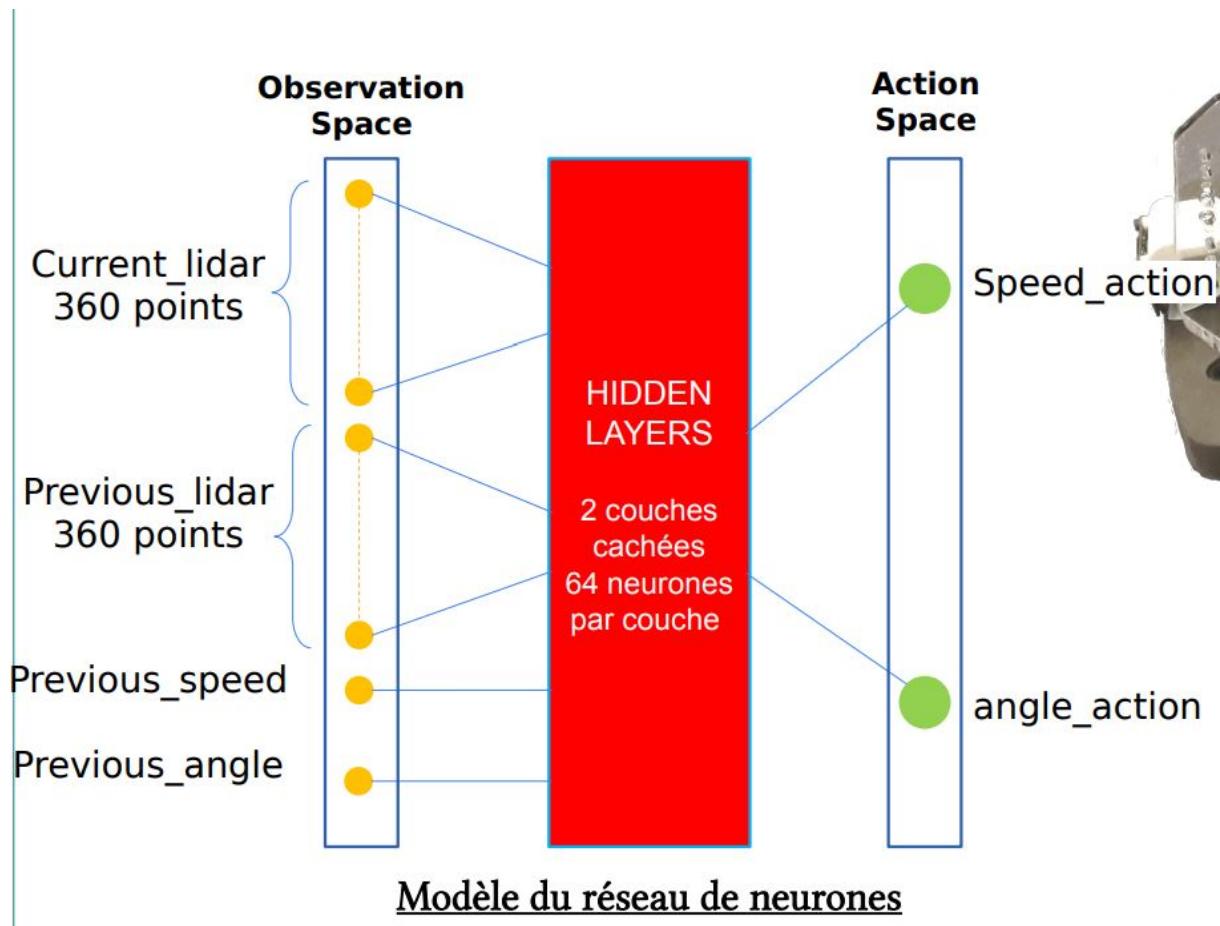


FIGURE 5.2 – Modèle de réseau de neurone final utilisé

détaille le travail réalisé sur le simulateur Webots.

5.3.1 Environnement d'apprentissage

Pour pouvoir entraîner efficacement le réseau de neurones dans le cas de la voiture autonome sur le simulateur, il est nécessaire de disposer de plusieurs éléments qui seront détaillés par la suite. Ces éléments sont les suivants :

- Une bibliothèque qui gère le réseau de neurones avec l'algorithme PPO
- Une bibliothèque qui gère la partie apprentissage
- Les pistes nécessaires pour un apprentissage efficace

— Une fonction de récompense adaptée pour un bon comportement de la voiture

Tous ces éléments composent l'environnement d'apprentissage permettant un apprentissage complet du réseau de neurones.

Pistes utilisées

Pour un apprentissage par renforcement, il est nécessaire de disposer d'un maximum de situations différentes. Dans notre cas, il faut une piste avec de longues lignes droites, des virages dans chaque sens, des virages à 180°,etc. On propose donc la piste suivante comme étant la piste d'entraînement :

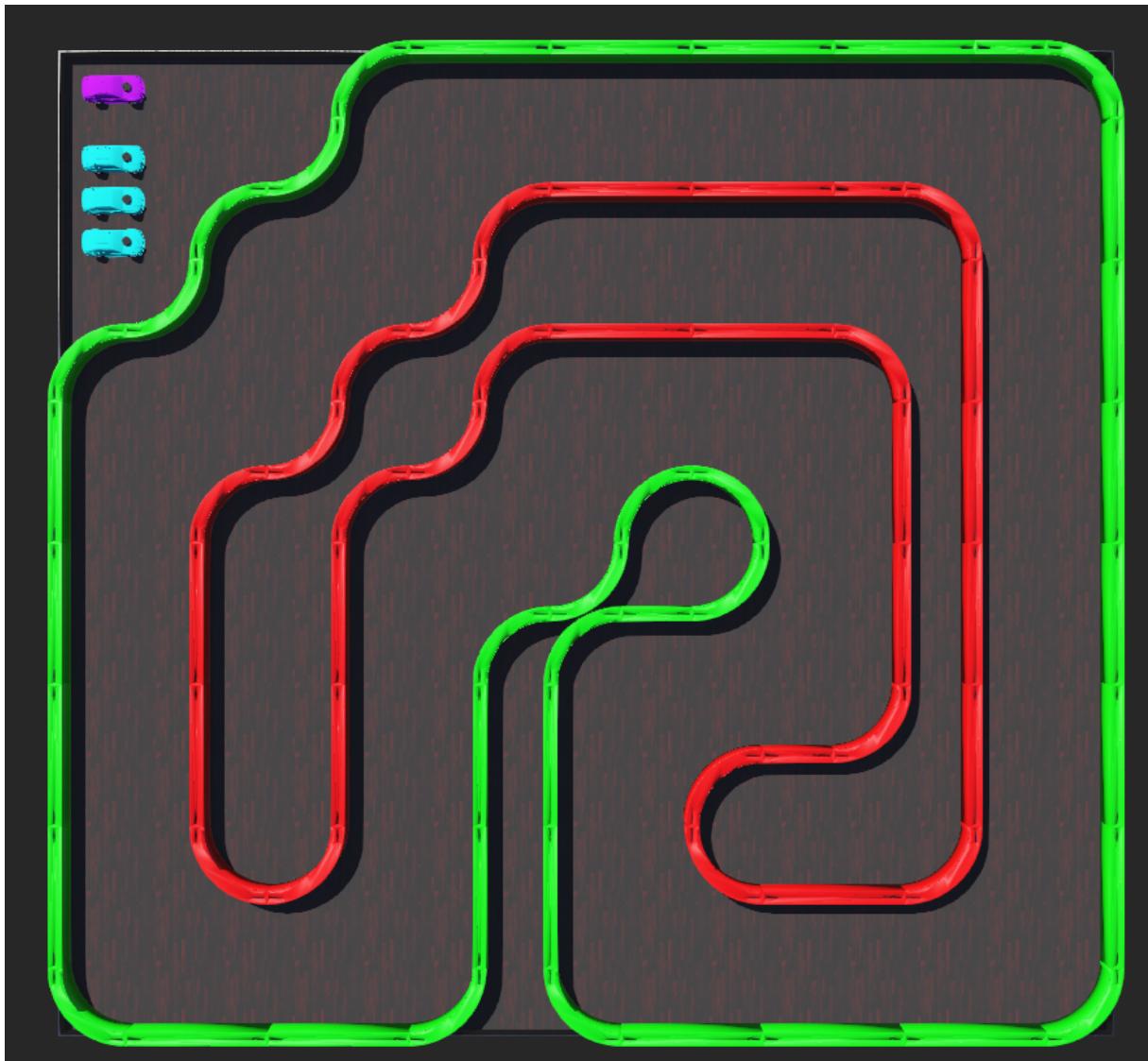


FIGURE 5.3 – Piste d'entraînement pour l'apprentissage par renforcement

Sur cette piste, on rajoute trois voitures ayant pour modèle la TT-02 et ayant un controller simple basé sur l'évitement d'obstacle. Cela donne une situation de course, une situation pour laquelle on s'entraîne.

Dans un processus d'apprentissage, il est nécessaire de valider ce qui a été appris sur la piste d'entraînement. Pour cela, on a créé une piste de validation avec cinq voitures en plus.

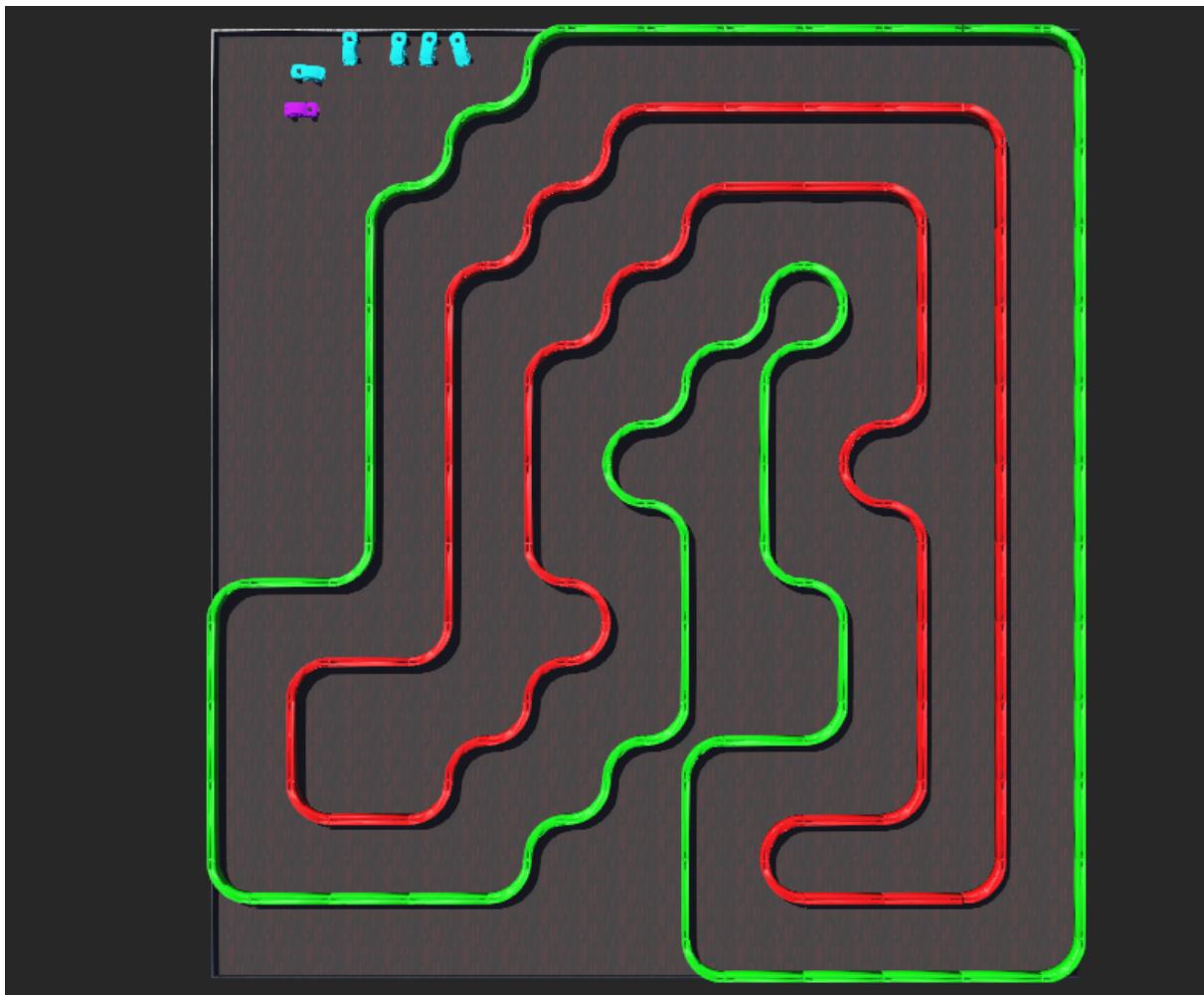


FIGURE 5.4 – Piste de validation pour l'apprentissage par renforcement

Pour chacune des pistes, il est nécessaire de gérer l'ensemble des voitures lors d'une phase de réinitialisation. Pour cela on ajoute un robot "superviseur" qui permet de repositionner toutes les voitures. Ce superviseur reçoit l'information de *reset* de la part de la voiture principale. Il possède aussi un tableau de position pour affecter une position aléatoire présente dans le tableau à chaque véhicule. On donne aussi une orientation choisie aléatoirement entre deux sens de circulation et naturellement la même pour chaque voiture. Le superviseur indique à la voiture principale qu'un repositionnement a été effectué. Cette communication se fait avec des émetteurs et des récepteurs présents dans Webots.

Environnement Gym

La library Gym est une bibliothèque implantée sous Python qui permet de gérer la partie apprentissage par renforcement d'un réseau de neurones. Gym propose par défaut plusieurs projets d'apprentissage par renforcement, notamment un des problèmes les plus connus dans ce domaine : le pendule inversé.

Dans le cas de notre projet, Gym ne propose pas d'environnement adapté. Il a donc été nécessaire de créer un tout nouvel environnement. Gym exige qu'un environnement contienne les fonctions suivantes :

- *get_observation()* : fonction renvoyant les observations de l'environnement
- *get_reward()* : fonction donnant la récompense selon l'action effectuée par l'agent
- *reset()* : fonction donnant la démarche pour repartir au début d'un épisode
- *step()* : fonction faisant évoluer l'environnement

Toutes ces fonctions sont rassemblées dans une classe que nous avons nommée "WebotsGymEnvironment". Dans cette classe nous avons rajouté quatre fonctions propres à la voiture :

- *get_lidar_mm()* : Fonction qui renvoie un tableau de Lidar dans le bon format avec des valeurs cohérentes en mm.
- *drive()* : Fonction qui récupère les actions données par le réseau de neurones et qui calcule la consigne des actionneurs.
- *set_vitesse_m_s()* : Fonction qui prend en argument une vitesse en m/s et qui la convertit en km/h avant de l'envoyer à la voiture.
- *set_direction_degre()* : Fonction qui prend un angle en degré et le convertit en radian avant de l'envoyer à la voiture.

Nous allons détailler par la suite chacune des fonctions.

Dans la fonction *get_observation()*, on appelle la fonction *get_lidar_mm* pour récupérer les observations du Lidar. On peut noter une petite boucle en début de fonction à cause de problèmes liés à la vérification de l'environnement par Gym. J'y reviendrai en détail dans une autre partie. Ce tableau sera le tableau donné en entrée du réseau pour les observations du Lidar à "l'instant présent". Pour ce qui est du tableau à "l'instant précédent", on stocke à chaque appel de la fonction le tableau d'observation dans une variable interne de la classe. Si la fonction est appelée depuis la fonction *reset()*, on donne le même tableau pour les deux parties de l'espace d'observation concernant le Lidar puisque la voiture a été repositionnée. Pour la vitesse et la direction, Webots nous permet de mesurer la vitesse et la direction de la voiture dans le simulateur. Ce sont ces données que l'on donne au réseau de neurones. De même dans le cas où l'observation est demandée par la fonction *reset()*, on indique que ces deux grandeurs sont nulles. Il est à noter que les valeurs données à l'espace d'observation sont normalisées.

Dans la fonction *get_reward()*, on donne la récompense associée à l'état dans lequel se trouve la voiture. On distingue deux états possibles pour la voiture. Le premier état est celui d'une situation de collision. On regarde la plus petite valeur du tableau de Lidar actuel et si celui-ci est inférieur à 120 mm, on considère qu'il y a collision. Dans le cas de la collision, on donne un malus de -300 "points". Le deuxième état regroupe toutes les situations autres que celle de collision. On donne comme récompense ici une valeur dépendant de la vitesse actuelle de la voiture ainsi que la distance minimale donnée par le tableau de Lidar. Les fonctions de récompenses seront détaillées plus tard. On indique aussi ici si l'on a terminé l'épisode via la variable *done*.

Dans la fonction *reset()*, on indique la démarche à suivre lorsque l'on veut retourner au début d'un épisode. Le reset se fait dans le cas d'un crash de la voiture ou si le nombre d'actions autorisées par épisode est atteint. Au moment du reset, on donne des consignes

de vitesse et d'angle nuls et on envoie une indication de reset au robot "Superviseur". A la fin de la routine de réinitialisation, on renvoie une observation. Dans cette fonction, nous avons dû faire appelle plusieurs fois à la fonction *super().step()* qui est a priori la fonction qui fait faire un pas au simulateur. Plusieurs problèmes ont été relevés et lien avec cette fonction *reset()*.

Dans la fonction *step()*, on indique que l'on fait un pas dans le processus d'apprentissage. Dans cette fonction on fait avancer la voiture avec les actions récupérées depuis le réseau de neurones. Ensuite, on récupère une observation depuis le Lidar et on fait faire un pas au processus d'apprentissage. On calcule la récompense avant de retourner toutes les informations obtenues.

Library Stable-Baselines3

La librairie Stable-Baselines3 permet de créer un réseau de neurones qui est géré par l'un des algorithmes que l'on a décrit plus haut. Dans notre cas, nous utiliserons comme précisé plus haut l'algorithme PPO. La librairie propose un large choix de paramètres pour l'apprentissage. Voici un descriptif de ces paramètres ainsi que la valeur qui nous servira de programme de référence :

- *policy* : Type de politique utilisée. Dans notre cas nous sommes avec de multiples entrées ('MultiInputPolicy')
- *env* : Environnement d'apprentissage Gym
- *learning_rate* : Facteur d'apprentissage. Il est possible de renseigner une fonction (3.10^{-4})
- *n_steps* : nombre d'actions autorisées par épisode (2048)
- *batch_size* : Taille du batch (64)
- *n_epochs* : Nombre d'epoch (10)
- *gamma* : Facteur d'actualisation (0.99)
- *gae_lambda* : Facteur permettant de choisir entre le biais et la variance de l'estimateur de récompense (0.95)
- *clip_range* : équivalent à la valeur ϵ décrite dans le paragraphe PPO. Une fonction peut être renseignée (0.02)
- *clip_rang_vf* : La même que précédemment mais spécifique à OpenAI. (None)
- *normalize_advantage* : Si l'on normalise l'avantage ou pas (True)
- *ent_coef* : Coefficient d'entropie pour la fonction de perte (0)
- *vf_coef* : Coefficient de la fonction de perte (0.5)
- *max_grad_norm* : Valeur maximale du gradient de la fonction clipping (0.5)
- *use_sde* : Utilisation de la fonction gSDE au lieu de l'exploration aléatoire d'action (False)
- *sde_sample_freq* : Fréquence d'échantillonnage de matrice de bruit lorsque la fonction gSDE est utilisée (-1)
- *target_kl* : Valeur limite de la divergence KL entre deux mise à jour. Il n'y a pas de limite par défaut. (None)
- *state_window_size* : Taille de la fenêtre affichant la longueur moyenne d'épisode et la récompense moyenne

- *tensorboard_log* : Emplacement pour enregistrer les données de Tensorboard (None)
- *policy_kwarg*s : Arguments additionnels lors de la création de la politique (None)
- *verbose* : Affichage d'informations (1)
- *seed* : Je n'ai pas beaucoup d'informations sur ce paramètre (None)
- *device* : Composant sur lequel faire tourner l'algorithme ("cpu")
- *_init_setup_model* : Construction du réseau à la création de l'instance. (True)

Il est nécessaire de décrire les espaces d'action et d'observation dans la classe de l'environnement Gym mentionné plus haut. Dans notre cas, nous optons pour l'espace d'action :

- un scalaire compris entre 0 et 1 pour l'incrément en vitesse
- un scalaire compris entre -1 et 1 pour l'incrément en angle

Les valeurs sont normalisées car Stable-Baselines3 préfère des valeurs normalisées. Ce n'est que par la suite que l'on multiplie par 0.5 m/s pour la vitesse et 9° pour l'angle.

Concernant l'espace d'observation, nous avons choisi le suivant :

- Un tableau de scalaire compris entre 0 et 1 pour les données actuelles du Lidar
- Un tableau de scalaire compris entre 0 et 1 pour les données précédentes du Lidar
- Un scalaire compris entre 0 et 1 pour la vitesse actuelle de la voiture
- Un scalaire compris entre -1 et 1 pour la direction actuelle de la voiture

Là encore on normalise les valeurs en divisant par leur valeurs maximales possibles respectifs. Concernant la vitesse et l'angle, on utilise les fonctions de Webots qui permettent d'accéder à ces grandeurs.

La librairie permet de lancer l'apprentissage grâce à la fonction *learn()*. Cette fonction prend en paramètre le nombre d'épisodes que va comporter la phase d'apprentissage. Une fois l'apprentissage terminé, il est possible de sauvegarder le réseau de neurones ce qui est particulièrement intéressant pour notre projet dans le but de charger ce réseau dans la voiture réelle. La fonction *load()* permet notamment de charger un réseau de neurones.

Fonction de récompense

La fonction de récompense est un des hyperparamètres les plus importants dans un processus d'apprentissage par renforcement. Elle est aussi un des plus problématiques de par son côté aléatoire. En effet, c'est la récompense qui influe sur comment l'apprentissage s'effectue. Par exemple, si l'on ne sanctionne pas suffisamment le crash de la voiture alors celle-ci pourrait avoir tendance à foncer dans le mur. Autre cas : si l'on ne favorise pas la vitesse alors la voiture aura tendance à rouler lentement voire être complètement à l'arrêt. Il est donc nécessaire de trouver une bonne fonction de récompense pour avoir le meilleur comportement possible.

Dans notre cas, on a choisi de donner un malus de -300 lors d'un crash et dans toutes les autres situations la fonction suivante :

$$reward = 12 * mini + 3 * vitesse \quad (5.5)$$

Ici *mini* est la distance la plus courte du tableau de Lidar. Ici on cherche à fortement pénaliser en cas de collision avec le décor ou une autre voiture. De plus, on veut que la voiture aille le plus vite possible et que celle-ci s'éloigne le plus possible des bords. Nous verrons par la suite dans les résultats si cette fonction de récompense a été concluante.

5.3.2 Simulation

Dans cette partie, je détaillerai la simulation faite sous Webots ainsi que la présentation des résultats.

Mise en place de l'entraînement

Dans un premier temps, on déclare un environnement Gym, qui est celui détaillé dans la partie précédente. On utilise la fonction *check()* de Gym pour vérifier si la classe d'environnement est conforme à ce qu'attend la bibliothèque d'un environnement créé. Une fois cette étape passée, on déclare le modèle PPO avec de multiples grandeurs d'entrée ainsi que spécifier l'environnement instancié précédemment.

On lance un apprentissage avec 1000000 d'épisodes. Ce nombre peut paraître énorme, mais il est nécessaire de donner un grand nombre d'épisode pour que la voiture puisse avoir le temps d'apprendre.

On indique aussi que le modèle doit être enregistré sous le nom "*PPO_results*".

Problème de simulation

Durant la phase d'apprentissage, nous nous sommes heurté à quelques problèmes de simulation. Je vais ici détailler les problèmes rencontrés ainsi que les solutions mises en place.

Dans un premier temps, nous avons eu un problème lié à la classe de l'environnement Gym. Une erreur était renvoyée parce que l'observation renvoyée par la fonction *reset()* n'était pas conforme à ce qui était attendu par l'espace d'observation. En effet, il se trouve que dans la simulation, le Lidar ne renvoie pas de valeur pendant un court laps de temps. Dans notre code nous avons fait le choix de renvoyer un message d'erreur lorsque le Lidar ne renvoie pas de bonnes valeurs. Ainsi nous avons rajouté une boucle dans la fonction *get_observation* pour temporiser, laissant le temps au Lidar de s'activer correctement.

Un autre problème a été particulièrement gênant pour la simulation. Ponctuellement, la voiture ne se repositionnait pas correctement et traversait le sol. Il s'agit d'un problème que Bastien LHOPITALLIER avait déjà rencontré. De plus, puisque la scène est reset par une collision de la voiture, il arrive parfois que celle-ci se retrouve hors de la piste. L'apprentissage continuant de tourner, la voiture apprend donc n'importe quoi. A ce jour,

nous ne savons toujours pas pourquoi ce problème survient mais nous avons pu observer que ce problème est courant lorsque la vitesse de la voiture est élevée. Pour pallier ce problème, on autorise au robot "superviseur" de repositionner lui-même la voiture dans un espace de la piste qui entraînera immédiatement une situation de crash.

Résultats

Dans cette section, je vais rassembler les résultats suivants les différents réseaux de neurones entraînés. Les comparaisons les plus poussées ont été réalisés sur le réseau de neurones le plus concluant, à savoir celui tenant compte de la vitesse et la direction actuelle de la voiture.

Dans un premier temps, le tableau 5.2 donne les différents réseaux de neurones qui ont été entraînés avec leurs particularités et les observations que j'ai pu faire une fois l'apprentissage terminé.

Pour comparer les performances, j'ai mesuré le temps de passage au tour pour chaque réseau de neurones sur la piste de validation (5.3). De plus, je vais ajouter les commentaires de mes observations suite à une démonstration sur cette même piste.

Conclusion

Au vu des résultats, on peut conclure que le nouveau modèle de réseau de neurones présente un comportement plus satisfaisant qu'avec l'ancien modèle. On peut aussi conclure que la modification des paramètres influe sur les résultats de l'apprentissage.

Concernant la comparaison des réseaux de neurones entraînés avec le nouveau modèle, le numéro 1 est celui donnant les meilleures performances. En revanche, pour le passage à la réalité, j'ai choisi de prendre le réseau N°5. En effet, c'est celui qui semblait avoir le meilleur comportement sur le simulateur. Je voulais privilégier le fait que la voiture ne se crash pas au lieu de la vitesse.

TABLEAU 5.2 – *Observation après apprentissage*

N°	Particularité	Observation sur la piste d'entraînement (avec obstacle)
Ancien modèle du réseau de neurones (seulement les données Lidar en observation)		
1	gamma = 0.95	-S'approche assez près des bords lors de virages complexes - Ne fonce pas souvent dans les voitures bleues
2	Lors d'un crash : reward = -300 - numéro du crash	- Assez lente dans les lignes droites - A réussi assez bien à éviter les obstacles
3	gae_lambda=0.90	- S'approche trop près du bord - Est lente dans les lignes droites - A réussi à dépasser une voiture bleue
4	ent_coef=0.02	- Fonce quelques fois dans les obstacles - A réussi à faire un tour complet - S'approche trop près des bords
5	vf_coef=2	- A doublé une voiture bleue - A réussi à faire un tour complet -S'approche trop près des bords
6	gamma=0.95 gae_lambda=0.90	- Va plutôt vite en ligne droite - Se rapproche un peu moins des bords -N'a pas fait un tour complet
Nouveau modèle de réseau de neurones (vitesse, direction et données Lidar précédente en observation)		
1	Configuration de base	- Evite les obstacles - Percute parfois les autres voitures - Dépasse les voitures même dans les lignes droites
2	Lors d'un crash : reward = -300 - numéro du crash	- Capable de faire plusieurs tours sans se crasher - Peut éviter les autres voitures et les dépasser - Ralenti pas mal dans les virages
3	500 000 épisodes pour l'apprentissage	- Pas de différence notable avec les autres réseau de neurones
4	gae_lambda=0.90	- Mouvement un peu hasardeux -Ne tourne pas certaines fois
5	ent_coef=0.02	-Fait plusieurs tour sans crash - Roule un peu moins vite que les autres réseau des neurones
6	ent_coef=0.02 angle entre -2° et 2°	- A réussi à faire un tour complet - Lente dans les virages - Assez réactive dans les lignes droites

TABLEAU 5.3 – *Contre-la-montre*

Réseau N°	Particularité	Temps de passage au tour	Observations sur la piste de validation (sans obsacle)
1	Modèle de base	1 :03.74	- Ne s'est pas crashée - Ralenti dans les virages
2	Lors d'un crash : reward= -300 - numéro de crash	1 :31.26	- S'est crashée une fois - A réussi mieux dans un sens - Ralenti dans les virages
3	500 000 épisodes pour l'apprentissage	2 :05.93	- S'est crashée une fois - A réussi mieux dans un sens - Ralenti dans les virages
4	gae_lambda = 0.90	Aucun	- S'est crashée plusieurs fois - N'arrive pas à gérer les virages après les lignes droites - Voiture la plus rapide mais incapable de faire un tour de piste
5	ent_coeff=0.02	02 :25.27	- Ralenti beaucoup dans les lignes droites - Réagi bien dans les virages
6	ent_coeff=0.02 angle entre -2° et 2°	2 :17.49	- N'avance pas très vite - Oscille moins dans les lignes droites

5.4 Passage à la réalité

Il s'agit ici de la dernière étape du projet, celui où l'on veut faire tourner le réseau de neurones sur la voiture réelle. Cette étape intervient après que le réseau de neurones ait été entraîné sur le simulateur.

5.4.1 Implémentation dans la voiture réelle

Pour s'assurer que les commandes données par le réseau de neurones soit compréhensible par le programme de la voiture. On a donc développé deux fonctions similaires à celles présentes dans le simulateur : *set_vitesse_m_s()* et *set_direction_degre()*. Cette fois, il s'agit de passer des commandes en m/s et en degrés en commandes de PWM. De plus, pour ce qui est du Lidar, on s'assure de ne pas laisser de valeur à 0 dans la partie "utile" du Lidar (les valeurs des indices 0 à 99 et celles de 260 à 359). Pour cela, si une valeur 0 apparaît entre deux valeurs non nulles, on la considère comme étant la moyenne de ces deux dernières. Dans le cas où il y a une série de 0, on copie la valeur non nulle précédente pour compenser. On se retrouve ainsi dans les mêmes dispositions que la simulation au niveau du Lidar.

Concernant le réseau de neurones, il a pu être charger grâce la fonction *load()* de Stable-Baselines3. Il est possible d'utiliser le réseau grâce à la fonction *predict()* qui prend en argument les observations et qui renvoie les actions. Les observations sont stockées sous forme de dictionnaire. Cependant, contrairement à la simulation, l'accès direct à la vitesse et à la direction n'est pas possible car il n'y a pas d'asservissement. Nous ne pouvons donner que les consignes en vitesse et en angle à l'instant précédent en tant qu'observation. Cela marque la principale différence entre la simulation et les conditions réelles.

Pour utiliser le Lidar, j'ai créé une classe *Lidar()* permettant de l'initialiser, le démarrer ainsi qu'acquérir les valeurs. Au démarrage du programme, on lance un thread *thread_scan_lidar* qui permet de faire l'acquisition des données du Lidar en continu. On active et désactive un drapeau dans le code pour indiquer lorsque l'on veut récupérer ces données acquises. Le traitement de ces données comme décrit plus haut se fait en dehors de ce thread.

En parallèle de ce thread, on lance un second thread *thread_conduite_autonome* qui est le programme de pilotage de la voiture. Dans la fonction *conduite_autonome()*, on commence par récupérer les données du Lidar puis on procède au traitement de ces données. On récupère les valeurs traitées tous les 20 degrés pour faciliter l'analyse des obstacles. Dans le cas où l'on considère qu'il y a collision (valeur minimale inférieure à un certain seuil), on recule dans la direction opposée à l'obstacle détectée (indice de cette valeur minimale). Si ce n'est pas le cas, on laisse la main au réseau de neurones pour le pilotage.

Il se passe un phénomène qui n'est pas voulu après le recule où, au démarrage, la voiture

n'avance plus. En effet, le réseau de neurones envoie des incrémentés négatifs de vitesse comme consigne. Cependant, nous n'autorisons pas le réseau de neurones à effectuer la marche arrière. On se retrouve donc bloquer car, les observations restant les mêmes, rien ne permet au réseau de neurones de faire augmenter la vitesse. Il a donc été décidé de mettre une valeur minimale de 0.1 m/s pour éviter cette situation de blocage.

5.4.2 Résultats

Les résultats présentés ici sont ceux de la voiture réelle avec le réseau de neurones avec $ent_coef = 0.02$. En effet, il s'agit de celui qui a montré le meilleur comportement en réel. De plus, je n'ai pas eu le temps ni les moyens techniques de pouvoir tester tous les réseaux de neurones sur une vraie piste.

Les résultats sont ceux réalisés le jour de la course de voitures autonomes. Une première phase concerne une piste sans obstacle où le temps comptabilisé est le temps que prend la voiture pour réaliser deux tours.



FIGURE 5.5 – 1^{ere} piste de qualification

J'ai pu observer que sans aucun obstacle, la voiture réussit très bien à se diriger et ne prend aucun mur pendant son tour. L'apprentissage est donc concluant sur ce point

TABLEAU 5.4 – *Résultat de la première phase de qualification*

	1er passage	2eme passage
Temps de passage (2 tours de piste)	31s	25s

 TABLEAU 5.5 – *Résultat de la deuxième phase de qualification*

	1er passage	2eme passage
Temps de passage (2 tours de piste)	32s	34s

puisque la conduite est très satisfaisante et conforme à ce qui est attendu. En revanche pour atteindre ce résultat, j'ai dû réduire par un coefficient la consigne de vitesse. En effet, sans cette correction, la voiture allait trop vite et se prenait quelques fois les murs dans les tests que j'effectuais. Le problème de cette correction est que sans asservissement de vitesse, celle-ci dépend de l'état de charge de la batterie. Je n'ai pas pu faire une étude de la vitesse maximale en fonction de la tension de la batterie. Donc ce coefficient est empirique et n'est pas constant au fil des essais. Le constat général que je peux dresser est que la voiture réagit bien mais est assez lente en comparaison avec les autres voitures. Aussi, on peut observer que dans les lignes droites, la voiture a tendance à vaciller contrairement à la simulation.

La deuxième phase se fait avec une nouvelle piste et des obstacles fixes.


 FIGURE 5.6 – 2^{eme} piste de qualification

En présence d'obstacles fixes, la voiture se comporte un peu moins bien. En effet, celle-ci aura tendance à ne pas suffisamment tourner pour éviter l'obstacle. En revanche, cela est une bonne situation pour tester la marche arrière couplée au réseau de neurones. La

voiture a pu finir ses deux tours malgré des collisions avec les obstacles. On peut conclure que la phase d'entraînement sur simulateur n'a pas suffisamment d'obstacle fixe.

Enfin, la dernière partie de l'évènement était la course en elle-même. Là encore, il y a eu deux courses. Ici pas de chrono à présenter mais quelques observations sont possibles. Lors de la première course, la voiture n'a pas pu finir la course suite à un carambolage et une conduite à contre-sens. Cela permet de conclure qu'il pourrait être utile d'inclure ce genre de situation dans le simulateur. Durant la deuxième course cependant, la voiture a pu terminer la course en 2e position après un duel final avec la voiture de Charlène et Martin.

5.5 Conclusion "SimToReal"

5.5.1 Travail réalisé

Par rapport à l'article [Wei21], certaines des méthodes décrites ont été mises en oeuvre durant ce projet.

Dans un premier temps, nous avons tenté d'adapter au maximum le simulateur pour que celui-ci rend compte au mieux l'interaction de la voiture et de son environnement. En revanche, nous n'avons pas les outils nécessaires pour développer de zéro le simulateur spécialement pour la voiture autonome.

Nous avons ensuite tenté d'adapter au maximum le modèle de la voiture. Pour cela, nous avons identifié les paramètres qui nous avaient été possible de mesurer directement depuis la voiture réelle. Il y a cependant encore des disparités entre le modèle et le système réel.

Concernant l'ajout d'aléatoire dans le modèle du simulateur, nous avons ajouté du bruit sur les données du Lidar. En revanche, nous avons dû adapter aussi les données du Lidar réel pour concorder avec celui du simulateur.

5.5.2 Améliorations possibles

Il y a encore plusieurs pistes de travail pour ce projet, évoqués dans l'article mais non implémentés.

Il est possible pour la suite de modéliser le comportement de la voiture réelle avec un réseau de neurones et ainsi se rapprocher de la dynamique réelle.

Pour l'ajout d'aléatoire, on peut mettre aussi du bruit sur les actionneurs (jeu au niveau de la direction par exemple). Il peut aussi être intéressant de donner des valeurs

nulles à l'espace d'observation au niveau du Lidar pour se rapprocher des erreurs de mesures quelques fois perçues par le Lidar dans sa partie utile.

Il reste aussi à implémenter des méthodes pour réduire le temps d'apprentissage. En effet, pendant ce projet, les phases d'apprentissage, même en simulation accélérée, prend presque quatre heures pour obtenir des résultats concluants. Il est donc important de réduire ce temps pour pouvoir obtenir plus de réseaux de neurones entraînés.

Chapitre 6

Conclusion

6.1 Améliorations possibles pour le futur

Pour les années à venir, il est possible d'améliorer les performances propres de la voiture, sans prendre en compte nécessairement l'algorithme de pilotage.

La voiture est pourvue de télémètres Laser à l'arrière pour capter la présence des autres voitures lors de la marche arrière. Nous n'avons pas pu les utilisés cette année.

Il serait nécessaire de mettre aussi en place l'asservissement de vitesse avec le microcontrôleur STM32. En effet, la vitesse dépendant de la tension de la batterie, on ne peut pas être sûr de la vitesse réelle de la voiture.

Le modèle de Lidar utilisé peut être remplacé pour des modèles plus performants. En revanche, cela augmenterait considérablement le coût de la voiture.

On pourrait affiner le modèle de la voiture sur le simulateur en réalisant des tests avec la caméra de la salle 1Z83.

Chapitre 7

Remerciements

Nous tenons à remercier Sergio Rodriguez et Anthony Juton qui nous ont proposé les sujets et accompagné tout au long de notre projet. Nous remercions tout particulièrement M. Juton pour l'organisation de la Course de Voitures Autonomes de Paris-Saclay à laquelle nous avons eu l'honneur de participer (*et de gagner*). Nous remercions également Pascal Varoqui qui, bien que n'étant pas responsable de notre TER, était présent pour nous aider lorsque nous en avions besoin et pour faire de magnifiques photographies de la course.

Enfin, nous remercions les étudiants qui ont travaillé sur les voitures autonomes les années précédentes, sans qui notre travail aurait été beaucoup plus difficile.

Références

- [Fel09] FELIX VON HUNDELSHAUSEN, MICHAEL HIMMELSBACH, FALK HECKER, ANDRÉ MÜLLER, HANS-JOACHIM WUENSCHE. “Driving with Tentacles - Integral Structures for Sensing and Motion”. In : *Conference : The DARPA Urban Challenge : Autonomous Vehicles in City Traffic, George Air Force Base, Victorville, California, USA* (jan. 2009).
- [Joh17] JOHN SCHLMAN, FILIP WOLSKI, PRAFULLA DHARIWAL, ALEC RADFORD, OLEG KLIMOV. “Proximal Policy Optimization Algorithms”. In : (2017).
- [Ant21] ANTONIN RAFFIN, ASHLEY HILL, ADAM GLEAVE, ANSSI KANERVISTO, MAXIMILIAN ERNESTUS, NOAH DORMANN. “Stable-Baselines3 : Reliable Reinforcement Learning Implementations”. In : *Journal of Machine Learning Research 22 (2021)* (2021), p. 1-8.
- [Wei21] WEI ZHU, XIAN GUO, DAI OWAKI, KYO KUTSUZAWA, MITSUHIRO HAYASHIBE. “A Survey of Sim-to-Real Transfer Techniques Applied to Reinforcement Learning for Bioinspired Robots”. In : *IEEE Transactions on Neural Networks and Learning Systems* (sept. 2021).