

```
#define HARDY Rémi Lembeye  
#define LAUREL Rémi Bouteiller
```

Rapport de TER EEA | Voiture Autonome

- Rapport de TER EEA | Voiture Autonome
 - Introduction
 - Partie de Laurel
 - Amélioration de la base mécanique
 - Mise en place du contrôle moteur
 - Odométrie et asservissement
 - Les capteurs utilisés
 - La communication au sein de la voiture
 - La communication avec l'extérieur
 - Multithreading
 - Conclusion
 - Partie de Léopold
 - Qu'est-ce que l'apprentissage par renforcement ?
 - Modélisation du problème
 - Framework TensorFlow
 - Implémentation
 - Simulation sur Webots
 - Voiture réelle
 - Conclusion
 - Partie d'Amaury
 - Travail sur la caméra Pixy2
 - Travail sur la caméra v2 de la Raspberry pi 4
 - Méthode K-means:
 - Pourquoi traiter l'image en hsv (Hue, Saturation, Value) :
 - Comment faire pour traiter l'image en continu :
 - Méthode pour savoir de quel côté repartir lorsqu'on rencontre un mur :
 - Conclusion
 - Partie d'Hardy
 - Création d'un simulateur
 - Développement de la stratégie de course
 - Conclusion
 - Conclusion générale
 - Bonus
 - Références

Introduction

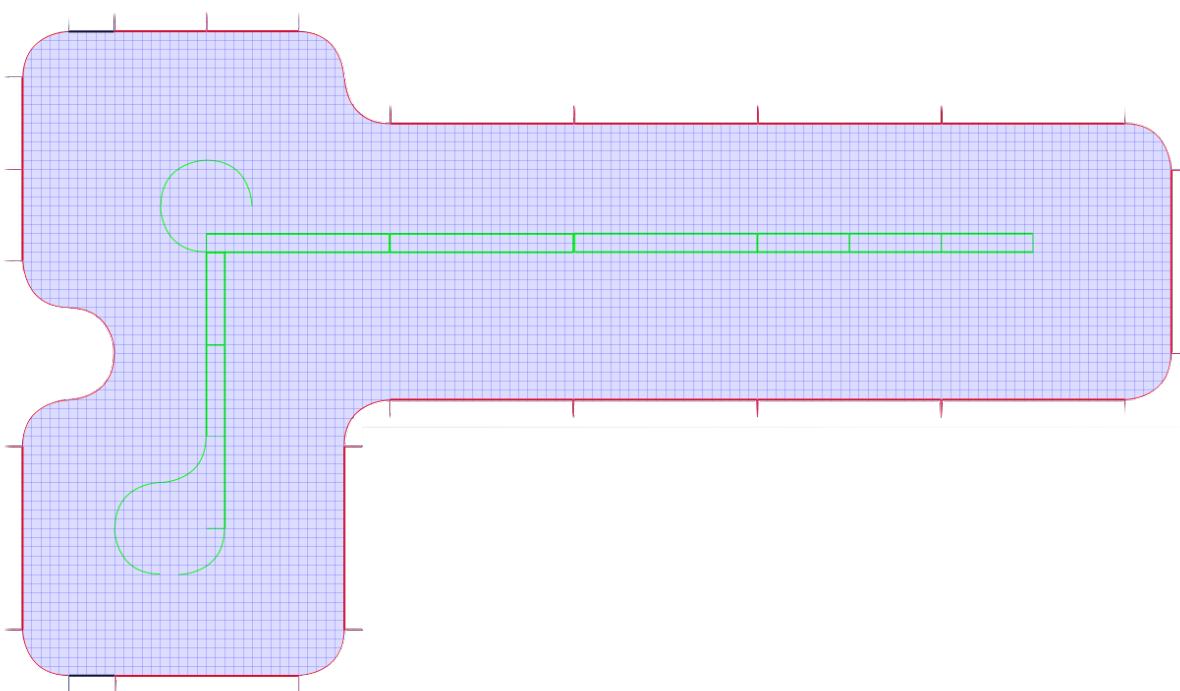
Le but du projet est de gagner la course de voitures autonomes Saclay.

La course met en concurrence des équipes d'étudiants de différentes écoles du plateau de Saclay.

Pour cela, les équipes disposent d'une base de TT02, nous sommes cependant les seuls à l'avoir fiabilisé.  =>  => 

Les équipes sont chargées de réaliser l'électronique et l'informatique de commande de la voiture afin que celle-ci puisse suivre une piste délimitée par des bords hauts de 20cm, rouges à gauche et verts à droite. La voiture doit donc faire des tours de piste le plus rapidement possible sur un tracé inconnu à l'avance et en évitant les obstacles potentiels.

Voici un exemple de piste possible 



L'objectif de ce rapport est de rendre compte du travail accompli dans le cadre de ce projet en décrivant les différentes solutions mise en œuvre. Le but est ainsi de permettre une reprise des codes, CAO, et électroniques utilisés pour les futures équipes de l'ENS Paris-Saclay.

Partie de Laurel

Mon travail s'est porté sur la mise en place d'une architecture mécanique et électronique optimisée ainsi que sur l'implémentation du code bas-niveau. Puis sur la mise en place des stratégies développées par Hardy sur le Raspberry Pi.

En premier lieu, voici les différents éléments qui composent la voiture.

- Un châssis de base TT02

- Un Raspberry Pi4 et sa caméra V2 pour la stratégie
- Un Lidar Hokuyo UBG04-LX
- Un STM32F4 => Nucleo L432KC pour l'asservissement et la récupération des états capteurs
- Un mix entre les kits TT02-S01BU et TATT-S03BU de Yeah Racing
- Les différents supports ont été imprimés en 3D
- Des capteurs Sharp GP2Y0A21YK0F infrarouges pour une mesure de distance en marche arrière
- Un codeur magnétique AS5600 pour la mesure de vitesse

Je vais donc dans ce rapport revenir plus en détail sur l'implémentation de ces différents éléments.

Amélioration de la base mécanique

Toutes les voitures de la course doivent être basées sur un châssis de tamiya TT02. Cependant, cette base mécanique souffre de quelques points faibles. Nous avons repéré et modifié les points suivants :

- Le servomoteur de direction d'origine
- La direction en plastique
- Pas de supports pour l'électronique

Je me suis donc chargé d'implémenter différentes solutions technologiques afin d'obtenir une plus grande fiabilité de la voiture.

Changement du servomoteur

Le premier point que nous avons remarqué en faisant tourner la voiture à la main est que le servomoteur d'origine était trop *mou* pour notre utilisation : il avait du mal à atteindre sa position de commande. De plus, une pièce en plastique entre le pignon de sortie et la direction servait de sauve-servo (cette pièce est flexible et permet donc de ne pas casser le servo si on est hors course) mais induit un jeu dans la direction.

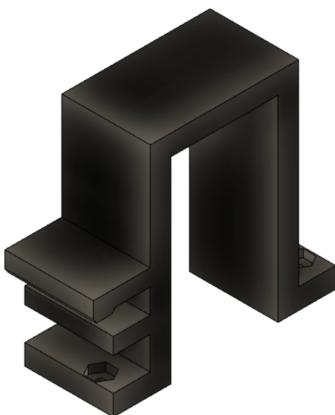
Nous avons donc choisi d'installer un dynamixel AX-12 pour plusieurs raisons :

- Le contrôle de la position s'effectue en UART
- Il s'éteint si le couple demandé est trop important (sauve-servo électronique et donc sans jeu)
- Dispose d'un couple beaucoup plus important, ce qui nous permet d'être certain de la position de la direction.

Cependant, l'AX-12 n'étant pas supporté nativement par le châssis de la TT02, il a

fallu concevoir un support (et pas des moindres).

Après 3 itérations, voici la version finale utilisée le jour de la course.



Support du servomoteur dynamixel

Les points importants sont le passage des câbles sur le côté et les emplacements pour les écrous de fixation.

Après ce changement, la direction est bien plus nerveuse et cela nous permettra donc de mettre en place une loi de commande pour la direction plus agressive.

Changement de la direction

Lors de l'étude préliminaire de la voiture, nous avons pensé qu'il serait intéressant de disposer d'un rayon de braquage plus faible afin de prendre les virages les plus serrés possibles. Nous avons donc acheté le *kit drift de yeah Racing* qui permet d'augmenter significativement l'angle de braquage. Cependant, ce kit porte bien son nom, et après mise en place sur la voiture, il s'est avéré que la voiture était incontrôlable : elle continuait d'aller tout droit quand les roues étaient tournées. En effet, ce kit supprime également la transmission **4x4** vers l'avant. Nous avons donc fait un mix avec un *kit direction alu* qui nous a permis d'obtenir le meilleur de la voiture. La mise en place de pièces en aluminium nous a également permis d'être plus robuste aux chocs répétés qu'a pu subir la voiture pendant les différents essais.

Fixations du lidar et de la Raspberry PI

Afin de se repérer sur la piste, nous utilisons un Lidar (*capteur de mesure de distance rotatif*) et un raspberry PI pour la stratégie.

Afin de fixer ces éléments sur la voiture, j'ai conçu des supports simples imprimés en 3D.

Mise en place du contrôle moteur

Le contrôleur de la voiture est contrôlé par PWM (*MLI, modulation de largeur d'impulsion*). Cependant, c'est la norme servomoteur qui est utilisé. Ainsi, la période du signal doit être de 20ms mais le temps à l'état haut doit être compris entre 1 et 2ms.



Exemples de signaux de commande du contrôleur moteur

Sur notre moteur, une impulsion de 1.5ms correspond à l'état neutre, un temps supérieur correspond à la marche avant et un temps inférieur correspond au frein ou à la marche arrière. La loi de contrôle est uniquement proportionnelle.

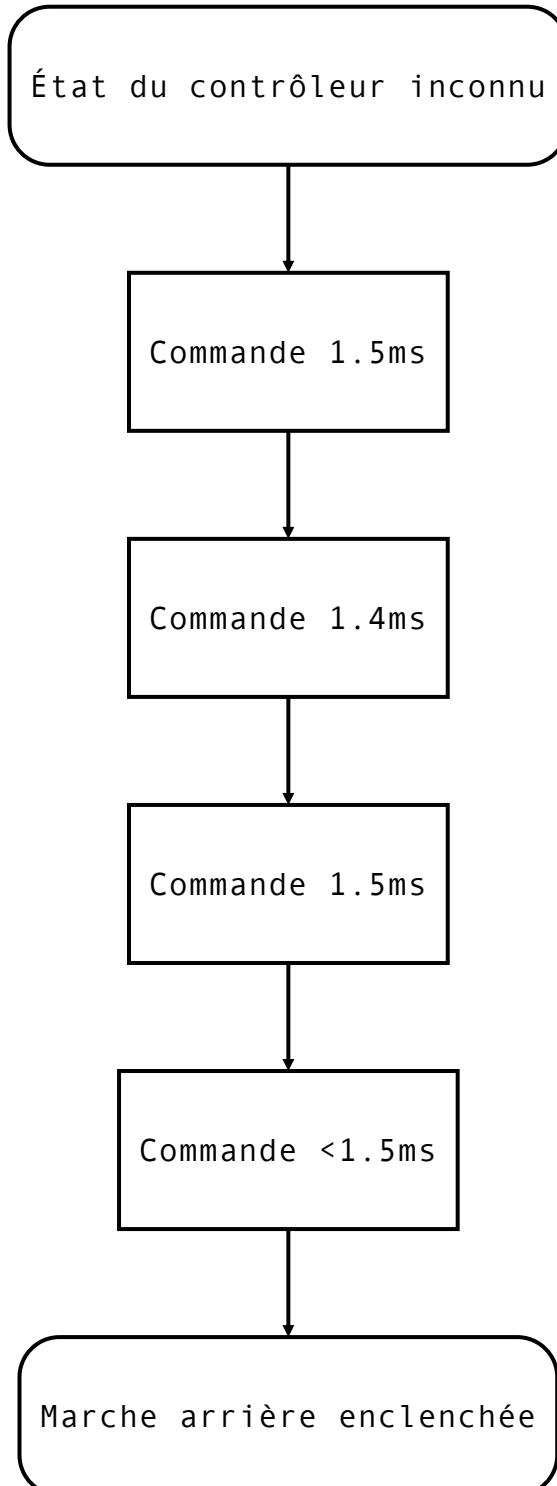
On a donc les tensions suivantes sur le moteur : **1.5ms => 0V et 2ms => Vbatterie**

Frein et Marche arrière

Le contrôleur moteur permet d'utiliser les impulsions inférieures à 1.5ms comme une commande de frein ou comme une commande de marche arrière.

- Si la commande précédente est neutre (=1.5ms), une commande <1.5ms mettra le moteur en marche arrière.
- Si la commande est positive (>1.5ms), une commande négative (<1.5ms) freinera juste le moteur.

Pour être certain de passer en marche arrière, j'ai donc mis en place l'algorithme suivant avec une temporisation de 200ms entre chaque état :



Ces différents protocoles de commande sont implémentés directement sur le microcontrôleur STM32F4 afin que la gestion de la vitesse soit transparente pour le PI qui enverra juste une consigne.

Odométrie et asservissement

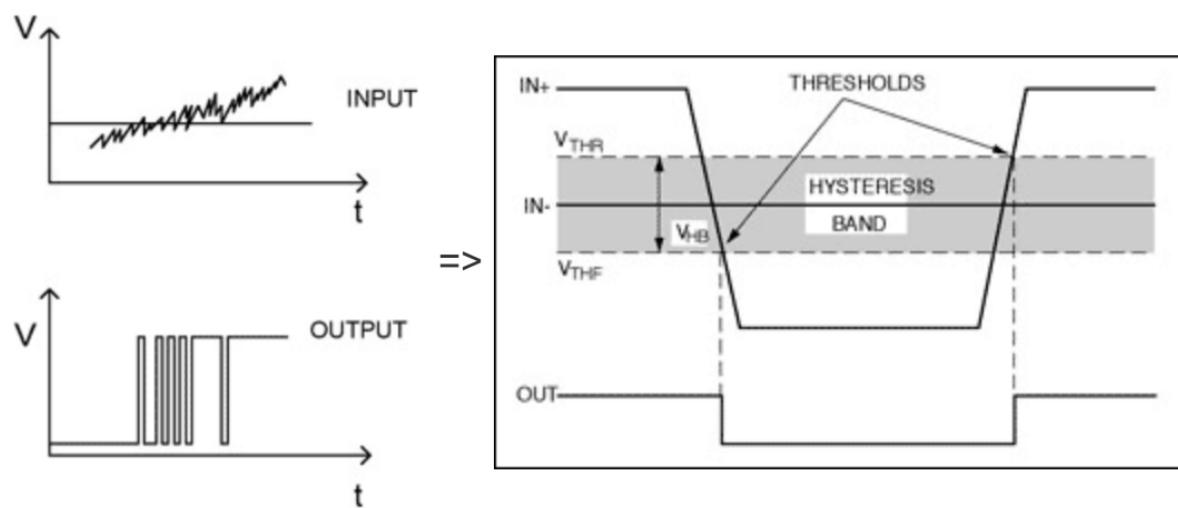
Nous souhaitons pouvoir mettre en œuvre un asservissement de la vitesse sur la voiture. Cela nous permettra d'être relativement insensible à la tension de la batterie et d'obtenir des freinages et accélérations plus vives.

La mesure de la vitesse

Capteur optique

Pour l'asservissement du moteur, nous avons en premier lieu utilisé un capteur optique ainsi qu'un marquage blanc sur l'axe de transmission noir. Ainsi, on obtient une variation de tension sur une PIN digitale du uC lorsque le capteur passe de noir à blanc et de blanc à noir. Une interruption permet alors d'incrémenter la distance parcourue à chaque front.

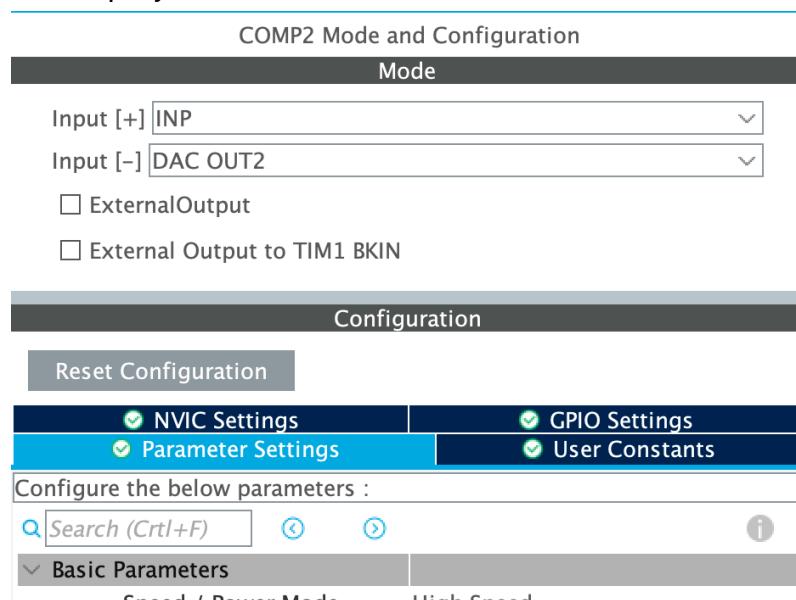
Cependant, nous avons remarqué que dans certains cas, lorsque le faisceau du capteur tape la limite entre le noir et le blanc, la distance parcourue s'incrémentait alors qu'il n'y a pas de mouvements. L'hypothèse que j'ai émise est alors que l'entrée CMOS alterne entre 1 et 0 comme on peut le voir sur le schéma suivant :



Problématique du bruit et solution par hystérésis

Pour contrer ce phénomène, j'ai mis en place un cycle d'hystérésis sur l'entrée qui permet des transitions certaines. Afin de pouvoir régler le seuil du capteur, on utilise un bloc comparateur du microcontrôleur avec le signal sur l'entrée + et le DAC interne sur l'autre.

Voici la configuration du projet :



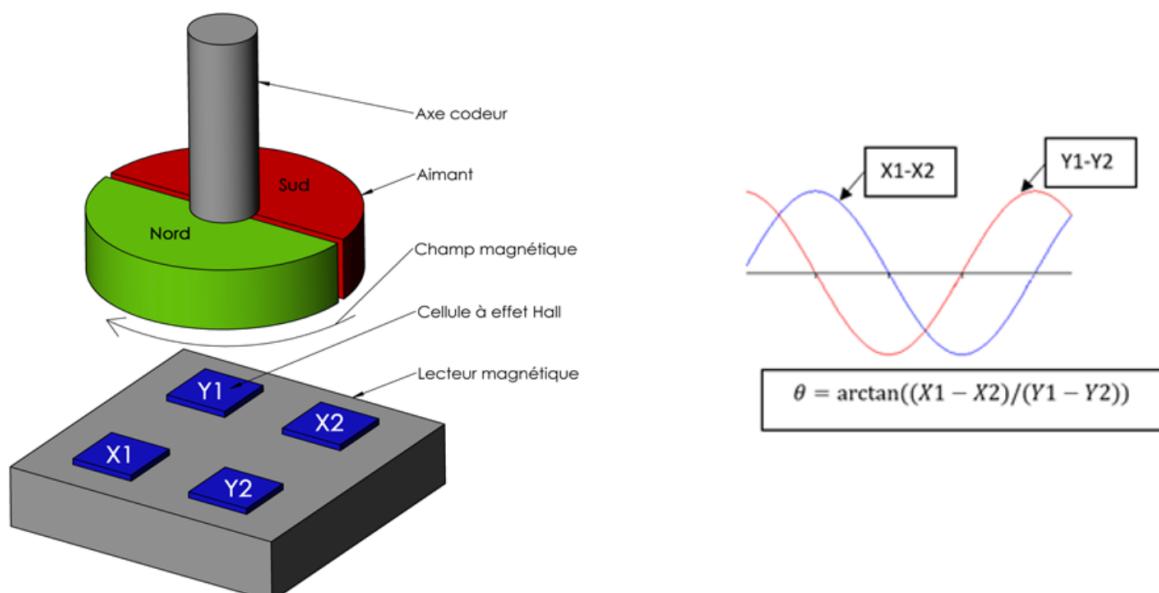
Speed / Power Mode	High Speed
Trigger Mode	Rising/Falling Edge Interrupt
Hysteresis Level	High
Output Configuration	
Blanking Source	None
Output Pol	COMP output on GPIO isn't inverted

Paramètres testés sur CubeMX

Cependant, la solution mise en œuvre ne m'a pas donné de résultats satisfaisants. Je n'ai pas réussi à supprimer complètement cet effet de bord et la précision était trop faible pour mettre en place un asservissement agressif.

Codeur magnétique

On est donc passé sur un capteur rotatif à effet hall.



Fonctionnement d'un codeur rotatif à effet hall

On récupère une tension analogique en sortie du capteur *codeur afin de déterminer l'incrément de position.

```

int imp=0;
int old_hall_value=0;
uint32_t *codeur;

/* calcul de la distance parcourue en fonction de la mesure ADC*/
void calcul_distance(){
    int hall_value=(int)*codeur;
    int delta=hall_value-old_hall_value;
    old_hall_value=hall_value;

    if(abs(delta)<2048){
        imp-=delta;
    }else if(delta>0){
        imp+=-4096+delta;
    }else{
        imp+=4096+delta;
    }
}

/* Routine d'interruption appellée toutes les 400us*/
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim == &htim6 )
    {
        calcul_distance(); //Mesure tension => distance parcourue
        if(compteur>=20){ //Toutes les 16 ms
            compteur=0;
            asser(); //Boucle d'asservissement du moteur
        }else{
            compteur++;
        }
    }
}

```

En faisant tourner le moteur à différentes vitesses, j'ai pu mesurer la vitesse maximale de rotation du capteur avant que celui-ci ne décroche.

- On obtient ainsi une vitesse de rotation maximale d'environ 7000 rpm

Comme on peut le voir, la STM32 est ici très utile. En effet, il est nécessaire de lire la tension au minimum deux fois par tour afin de détecter le sens de rotation du moteur. La position est déterminée en additionnant ou soustrayant (*en fonction du sens de marche*) la tension mesurée (*entre 0 et 4096*) à la distance déjà parcourue.

Asservissement de la vitesse

Le codeur précédent nous permet d'obtenir la position du rotor du moteur. Nous avons donc pu mettre en place un asservissement de la vitesse de la voiture via un correcteur

PID. Cet asservissement nous permet d'obtenir une vitesse constante malgré la décharge de la batterie et également de tirer le maximum de la voiture lors des phases de freinage et d'accélération.

Pour régler cet asservissement, nous avons mesuré la tension moteur sur l'oscilloscope afin de mesurer la période des oscillations, puis nous avons choisi les coefficients avec la méthode de *Ziegler Nichols*.

Asservissement de la direction

Nous avons mis en place un asservissement sur la consigne en direction de la voiture.

Les capteurs utilisés

Le lidar

Nous avons commencé par utiliser un RPLidar A2, mais sa fréquence de rafraîchissement ainsi que sa résolution étaient trop faibles pour réaliser un évitemennt de l'adversaire correct.

Le lidar que nous avons finalement utilisé est un *UBG04 LX* de Hokuyo. Il dispose d'une fréquence de 36Hz et de 1024 points par tour. Avec une portée de 5800mm, il nous permet de se situer efficacement. Les points situés à plus de 5.8m sont alors retournés avec une valeur très faible <20mm.

Les capteurs infrarouges Sharp

Le lidar ne disposant que d'un angle de mesure de 240 degrés, j'ai installé des capteurs infrarouges Sharp GP2YA41SK0F sur les côtés et GP2Y0A21YK derrière. Ces capteurs nous permettent de détecter la présence d'un adversaire ou d'un bord de piste lors des marches arrière. On récupère leurs valeurs sur les entrées analogiques du STM32.

Ils suivent une loi non linéaire pour retrouver la distance à partir de la tension.

- Pour le capteur arrière, portée 80cm : $Distance = 29.988 * POW(Volt, -1.173)$
- Pour les capteurs latéraux, portée 30cm :
 $Distance = 12.08 * POW(Volt, -1.058)$

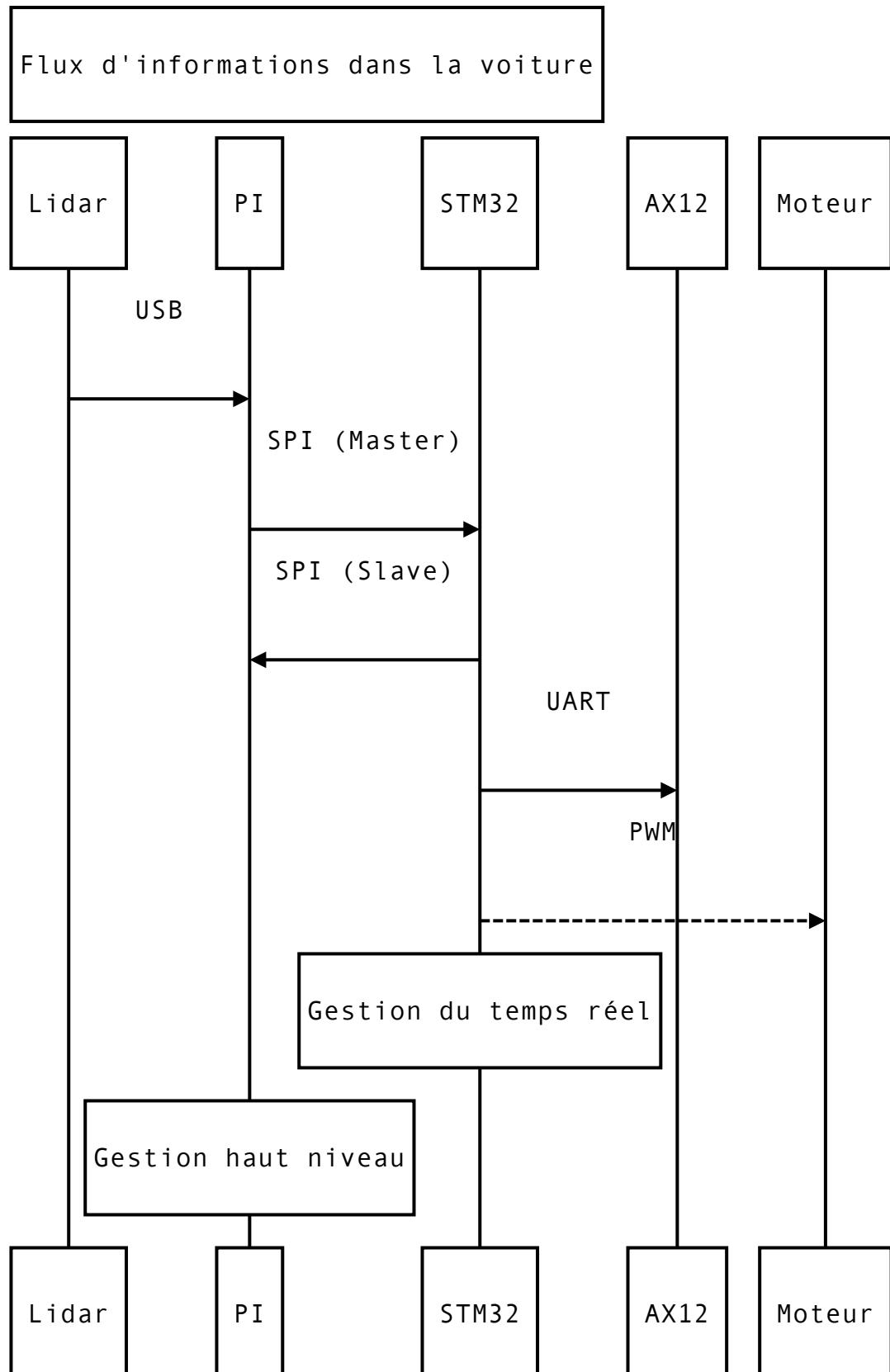
Voici le code qui fait la conversion entre la valeur des ADC récupérée directement via DMA et la distance de l'obstacle. S'il n'y a pas d'obstacle dans le champ, la valeur est initialisée à 255.

```
float coef_IR[] = {12.08,30,12.08};  
float power_IR[] = {-1.058, -1.176, -1.058};  
char limit_IR[] = {30, 80, 30};  
  
void capteurs_read(){  
    uint32_t distance;  
    for (uint8_t i=0;i<3;i++){  
        distance=coef_IR[i] * pow((float)(adc_buf[i+1]*3.3)/4096,  
power_IR[i]);  
        distance>limit_IR[i]?distance=255:  
(distance<1?distance=255:1);  
        spi_emit[i]=(uint8_t)distance;  
    }  
}
```

On peut observer que ce code assez lourd à faire tourner, la présence de multiplication et de calcul de puissance étant couteux en cycles CPU. On ne peut donc pas faire tourner cette fonction dans la routine d'interruption, c'est la seule fonction qui tourne dans le main.

La communication au sein de la voiture

Les différents éléments de la voiture doivent communiquer un certain nombre d'informations entre eux. Voici les différents protocoles utilisés :



Nous allons dans cette partie discuter de leur mise en place.

Communication PI <=> STM32

La carte Nucléo L432KC utilisée ne dispose que de deux périphériques USART. Comme

vu précédemment, l'un est utilisé pour communiquer avec l'AX-12 et j'ai préféré garder l'autre pour du debug potentiel. J'ai donc mis en œuvre une communication SPI entre le PI et la STM32. Le PI ne pouvant être que maître sur cette communication, le STM32 est esclave.

Les informations à faire transiter entre le PI et le STM sont :

- PI => STM32 : La vitesse, la direction
- STM32 => PI : Les trois capteurs infrarouges arrières

Afin que le statut d'esclave du SPI n'interfère pas avec les autres fonctions du microcontrôleur, j'ai décidé d'utiliser le DMA associé au SPI.

Le SPI est un bus duplex, j'ai donc mis en place des trames de communication.

Demande du PI :

Octet 0	Octet 1	Octet 2
0xA2	char direction	char vitesse
ID	Angle du servo en degrés	Vitesse (127 = 1.5ms)

Réponse du STM32 :

Octet 0	Octet 1	Octet 2
Capteur AR Gauche	Capteur AR Centre	Capteur AR Droit

La taille de la trame étant connue, l'utilisation du DMA en mode **circular** permet de s'affranchir complètement de la gestion du SPI.

```
//réception circulaire de 3 octets et émission des distance IR
HAL_SPI_TransmitReceive_DMA(&hsp1,spi_emit,spi_recep,3);
```

Cette ligne de code est utilisée uniquement lors de l'initialisation, le mode **circular** du DMA faisant le reste. En lisant les données du tableau `spi_recep[3]`, on a accès aux commandes en vitesse et en position. En remplissant le tableau `spi_emit[3]` avec les distances en cm mesurées par les capteurs IR, celles-ci seront transmises lors du prochain échange SPI.

Pour réaliser les différentes connections entre le STM32 et les différents éléments (*PI, capteurs, écran via i2c,...*) j'ai développé une carte électronique sur Eagle. Cette carte est très simple, elle utilise seulement deux couches et uniquement des composants traversants. Les prises utilisées pour connecter tous les éléments sont des JST-XH. La carte dispose de nombreux ports, on peut donc ajouter facilement des éléments sur la

voiture.

Gestion du Lidar

Le lidar Hokuyo est connecté en USB avec la Raspberry PI. Le code qui tourne sur le PI est écrit en python, j'ai donc utilisé une librairie Hokuyo disponible pour ce langage et écrit par-dessus une classe qui nous permet d'obtenir le même fonctionnement que le lidar sur webots. Ce point est important pour permettre un passage du code de webots vers la voiture le plus simple possible.

```
def array_scan(self):#dict va de 119 a -119
    angle=list(self.scan.keys())
    distance=list(self.scan.values())
    angle.reverse()
    distance.reverse()

    self.scan_array[self.taille_full_2-self.taille_2:self.taille_full_2 ,
    self.scan_array[self.taille_full_2:self.taille_full_2+self.taille_2 ,
    self.scan_array[self.taille_full_2-self.taille_2:self.taille_full_2 ,
    self.scan_array[self.taille_full_2:self.taille_full_2+self.taille_2 ,

    self.scan_array[self.taille_full_2-self.taille_2:self.taille_full_2+se
    self.scan_array[self.scan_array[:,0]<20,0]= 5800
```

La librairie renvoie un dictionnaire avec les angles allant de 119 à -119 degrés comme clés et les distances comme valeurs. A partir de ce dictionnaire, je crée un `array numpy` qui contient les distances dans la première colonne et l'angle (*de 0 à 360 degrés*) dans la seconde.

La communication avec l'extérieur

Pour le début du projet, nous programmions la PI via VNC. Cependant, nous nous sommes vite confrontés à plusieurs problèmes :

- L'interface graphique de la PI est lente
- Il est stipulé dans le règlement que la voiture doit s'arrêter si elle perd la communication avec le PC
- Lorsque l'on écrivait du code sur PC, le transfert vers la PI était compliqué

J'ai donc écrit un code Python qui permet de

- Déetecter l'adresse IP du PI et du PC
- Supprimer le code présent sur la PI
- Envoyer la dernière version du code sur la PI en SSH

- Ouvrir deux sockets, une de communication et une de *keep-alive*
- Lancer le code du PI
- Afficher les `print()` du PI sur le PC
- Donner l'ordre de départ de la voiture

Voici par exemple le code d'initialisation des sockets.

```
def init_socket(self, port):
    s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.settimeout(10)
    s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
    s.bind(('0.0.0.0', port))
    s.listen()
    client, address = s.accept()
    client.settimeout(1)
    return s,client
```

Les sockets permettent la communication client<=>serveur (*PC=Serveur <=> PI=Client*). La socket 1 permet de récupérer les print du PC et d'envoyer différents ordres tel que celui de départ.

La socket 2 envoie le signal '`op`' toutes les 100ms. Si la PI ne reçoit pas ce message pendant 1s, la voiture s'arrête et éteint tous les périphériques. Cela nous permet également d'éteindre la voiture à distance lors des essais par exemple.

```
def test_com(self):
    while self.on:
        try:
            message = self.socket2.recv(4096).decode()
        except socket.timeout:
            message = 0

        if message !='op':
            print("A bientôt dans le metro")
            self.voiture.stop()
            self.lidar.stop()
            self.stop()
```

Multithreading

Pour mener à bien toutes ces tâches, nous avons mis en œuvre le multithreading sur Python.

```
import threading
import time

class exemple(object):
    def __init__(self, voiture, lidar, port=11111):
        self.thread = threading.Thread(target=self.fonction_thread)
        self.thread.start()

    def fonction_thread(self):
        while 1:
            #on peut faire ce qu'on veut
            time.sleep(0.00001)
```

La boucle `while` permet au thread de tourner en continu et le `time.sleep` permet au gestionnaire de thread de passer à autre chose. (*Un autre thread, le main,...*)

Le multithreading a été utilisé pour :

- L'envoi des consignes au STM32
- La gestion du lidar
- Le test de com avec le PC

Conclusion

Je me suis occupé de la partie bas niveau du projet, et celle-ci à été fonctionnelle pour la course. Le très large spectre de compétences travaillées tout au long de l'année a été très intéressant et m'as permis de découvrir des notions nouvelles telles que le multi-threading où le système de socket. La mise en place pratique de correcteurs pour asservir la vitesse et la position a également été formatrice.

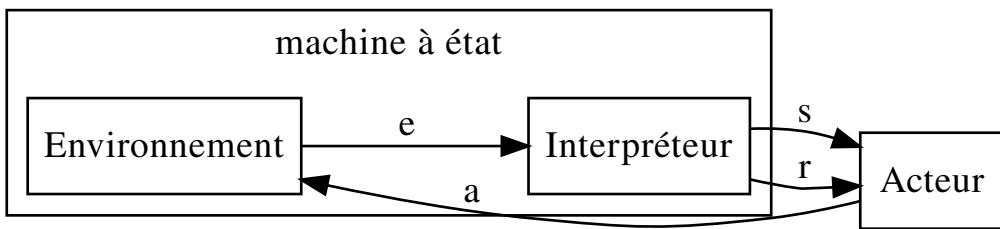
Partie de Léopold

J'ai travaillé sur la mise en place d'une solution de prise de décision par un algorithme de machine learning. La solution choisie est de l'apprentissage par renforcement.

Qu'est-ce que l'apprentissage par renforcement ?

La machine à état

Le système (l'**environnement**) est représenté par un **vecteur d'état**, e_t , qui est interprété par l'interpréteur pour devenir le vecteur d'observation s_t . Nous (l'**agent**) pouvons agir sur le système grâce à une **action**, a_t . Lorsque nous agissons sur le système, il nous renvoie le nouveau vecteur d'observation correspondant à son nouvel état, et une **récompense**, r_t , nous permettant d'évaluer la qualité de l'action. Le temps est discret $t \in \mathbb{Z}$. C'est une modélisation sous forme d'un processus markovien du système.



On nomme **espace d'observation**, \mathbb{S} l'ensemble des valeurs possibles pour s , et **espace d'action**, \mathbb{A} , l'ensemble des valeurs possibles pour a . On ne considère pas e_t puisque nous n'y avons pas accès.

Notre objectif est de créer une fonction $f : \mathbb{S} \rightarrow \mathbb{A}$, qui nous permet de déterminer l'action $a_t = f(s_t)$ permettant de maximiser la récompense totale $\sum_{t \in \mathbb{Z}} r_t$. Si il n'y a pas d'états terminaux, on n'est pas sûr que cette somme converge, donc on utilise une version pondérée $\sum_{t \in \mathbb{Z}} \gamma^t r_t$ avec $\gamma \in [0[$.

Dans un modèle où le nombre d'états et d'actions est raisonnable, on utilise un tableau rempli par les valeurs $Q(s, a)$. Si l'on se trouve dans l'état s_t , on cherche l'action a_t qui maximise $Q(s_t, a)$ et on l'applique au système. On ajuste la valeur de $Q(s_t, a_t)$ en fonction de r_t . C'est ce tableau Q qui donne son nom au Q-learning. Cela nécessite un espace mémoire en $O(\bar{\mathbb{S}} \cdot \bar{\mathbb{A}})$.

Le remplissage de Q

Au début de l'apprentissage, Q est initialisé avec uniquement des 0. Il y a plusieurs algorithmes utilisés pour remplir le tableau, le plus simple étant l'algorithme ϵ -glouton. L'acteur suit l'action $a_t = \arg \max_{a \in \mathbb{A}} Q(s_t, a)$ avec une probabilité $1 - \epsilon$, c'est l'**exploitation**, et il applique une action aléatoire avec une probabilité de ϵ , c'est l'**exploration**.

Pour remplir le tableau Q , il suffit donc de laisser l'algorithme parcourir l'environnement pendant un temps suffisamment long pour qu'il y ait convergence.

Modélisation du problème

Le vecteur d'état correspondant à notre problème est

$$\begin{pmatrix} x \\ y \\ \theta \\ \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix}$$

En effet, en connaissant ce vecteur et le circuit, on connaît toutes les grandeurs. Malheureusement, on ne connaît ni l'un ni l'autre, il faut donc trouver une modélisation différente pour qu'elle soit exploitable.

Notre robot se repère grâce à un lidar, un vecteur d'observation simple est donc un vecteur constitué de toutes les distances mesurées par le lidar. On considère qu'il y a N_l mesures par tour pour la suite, et que ces mesures sont entre 0 et 5m : la distance maximale du lidar (pour la suite, nous nommerons ce vecteur L_t). Pour le vecteur d'action, nous devons gérer la vitesse et l'angle de braquage, ce qui peut être représenté par une paire d'entiers entre -1 et 1 (pour la suite, nous nommerons ce vecteur C_t , car ce sont les commandes à envoyer au moteur et servomoteur).

Mais cela pose un problème, car on veut remplir un tableau fini. Or ici, $\mathbb{S} = [0; 5]^{N_l}$, ce qui n'est pas dénombrable. Même si en réalité, une fois en machine, la quantification des réels codés en `float32` résulte en un ensemble dénombrable, il reste trop vaste pour que le tableau Q soit stockable en mémoire. Le problème se pose aussi pour $\mathbb{A} = [-1; 1]^2$, dans une moindre mesure.

Quantification moins précise

Une solution serait de subdiviser les intervalles en un nombre raisonnable de sous intervalles. En subdivisant les intervalles en 100, $\bar{\mathbb{S}} = 100^{N_l}$ et $\bar{\mathbb{A}} = 10000$, la taille mémoire sera de $4 \cdot 10^{N_l+2}$ octets. Cela est plus raisonnable, mais reste trop élevé si on veut une résolution angulaire importante.

Réseau de neurones

Une autre solution serait de ne pas calculer Q directement, mais une approximation de Q . Pour cela, on peut utiliser un réseau de neurones artificiel $N : \mathbb{S} \rightarrow \mathbb{A}$. Le nombre de paramètres d'un tel réseau reste raisonnable par rapport au nombre de paramètres dans Q . En effet, on sait que toute fonction continue sur un sous-ensemble compact de \mathbb{R}^N peut être approximée avec une erreur arbitraire par un réseau de neurones assez profond ou large.

Autre vecteur d'observation

L'utilisation d'un réseau de neurones permet d'envisager des vecteurs d'observation plus complexes. En effet, pour l'instant, notre réseau de neurones n'a aucune mémoire ni notion de vitesse. Ainsi pour remédier à cela, on peut proposer d'utiliser un vecteur plus complexe :

$$\begin{pmatrix} L_t & L_{t-1} \\ C_t & C_{t-1} \end{pmatrix}$$

Framework TensorFlow

Pour réaliser notre système d'apprentissage, nous utilisons un framework python connu, TensorFlow. Ce module python s'appuie lui-même sur Keras, une bibliothèque de fonction permettant la manipulation de réseaux de neurones.

Politique

La modélisation utilisée par TensorFlow est différente de celle utilisée précédemment, car on n'utilise pas juste l'action qui maximise le retour mais une distribution de probabilité. Cette distribution, noté π , est générée par le réseau de neurones. Le comportement de l'acteur est donc entièrement caractérisé par cette politique.

Le formalisme GYM

Pour l'interface avec l'environnement, TensorFlow utilise une norme définie par OpenIA : GYM.

```
,-----.
| GYM_env
| -----|
| + action_space
| + observation_space
| + step(action) -> observation, reward, done, info
| + reset() -> observation, reward, done, info
`-----'
```

Spécification de l'environnement GYM

Il nous faudra donc développer notre propre implémentation de cette classe afin de pouvoir l'intégrer dans TensorFlow. Heureusement, TensorFlow fournit une classe abstraite permettant d'encapsuler une partie du traitement, comme la conversion de `numpy.array` classique vers des `TensorFlow.Tensor` que l'on ne sait pas aussi bien manipuler.

La boucle d'apprentissage

Apprentissage différé

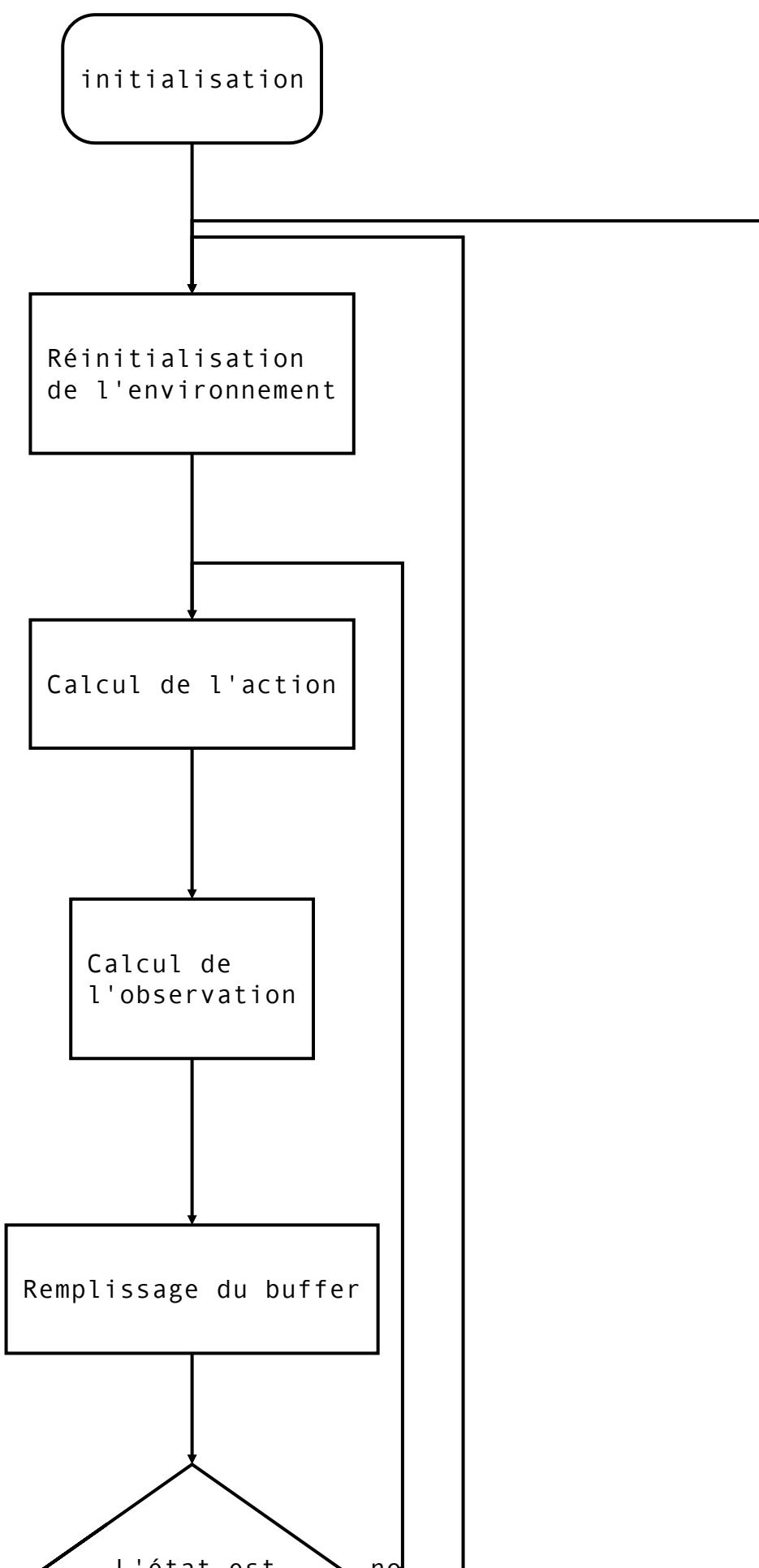
TensorFlow utilise de l'apprentissage différé, c'est-à-dire que ce n'est pas après chaque action que la politique est ajustée. Ici, on remplit un buffer avec des objets `transition`, qui contiennent s_t , a_t , r_t et s_{t+1} . On réalise plusieurs épisodes pour remplir le buffer, et on ajuste le réseau de neurones.

Sur-apprentissage

Un des problèmes qui peut se présenter est le sur-apprentissage. Par exemple, la politique pourrait être performante pour un circuit particulier. Pour contrer cela, on peut utiliser un ensemble de circuits. L'autre solution est d'entraîner la voiture sur un circuit, et d'évaluer sa qualité sur un autre.

Conclusion

On choisit d'utiliser deux circuits différents pour entraîner et évaluer. L'algorithme utilisé est le suivant:



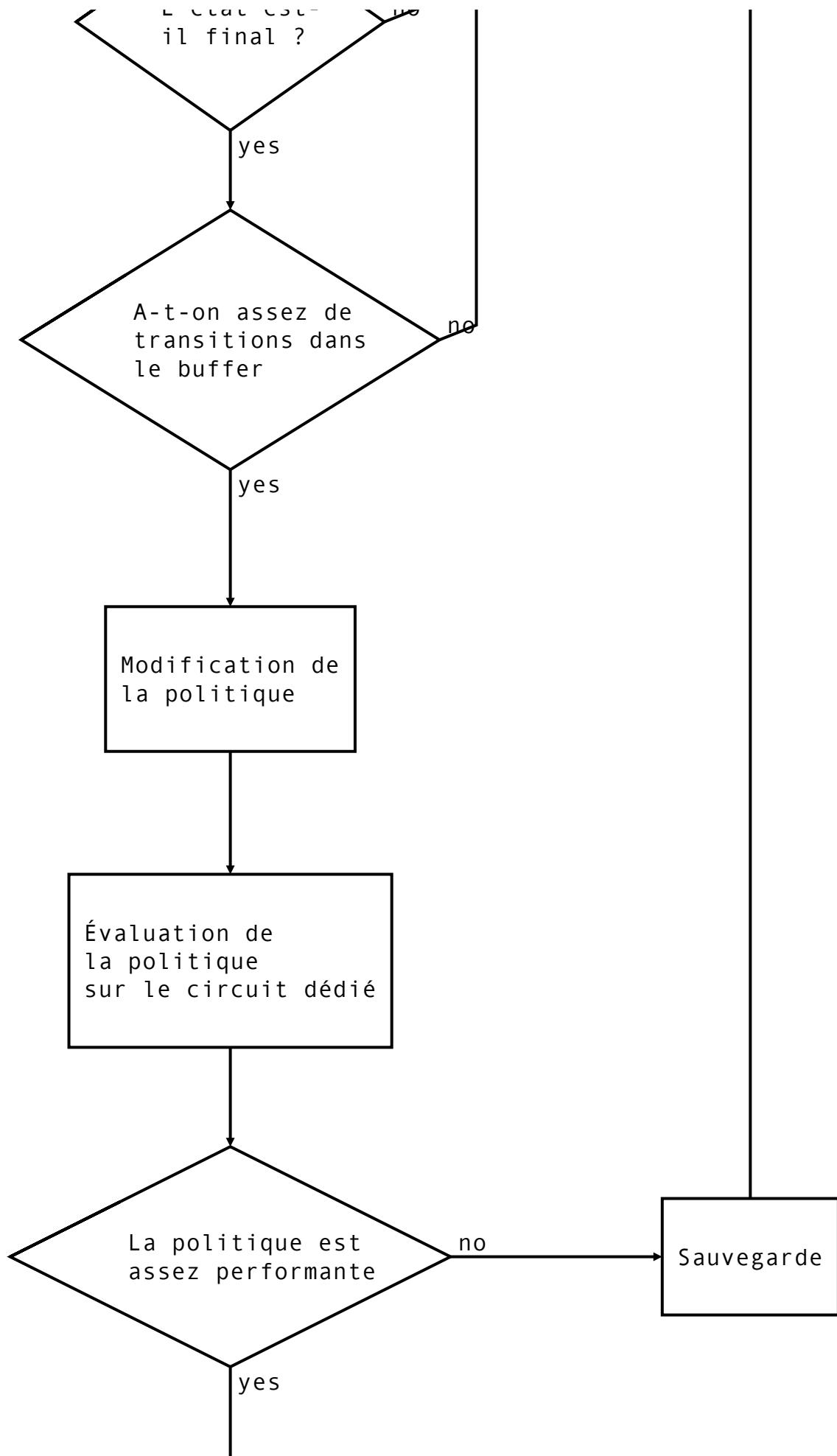


Figure : Principe de la boucle d'apprentissage

Implémentation

Simulateur python

Les parties importantes pour la définition de l'environnement, dans le fichier `interface_tf.py` sont:

- Créer une classe héritant de `tf_agents.environments.py_environment.PyEnvironment`.
- Surcharger les méthodes :
 - `_rest(self)` , pour réinitialiser l'environnement, et renvoyer le `time_step.restart` initial,
 - `_step(self, action)` , pour calculer l'état suivant en fonction de l'action. Il faut renvoyer un `time_step.transition(observation, récompense)` si c'est un état classique, et `time_step.termination(observation, récompense)` si c'est un état terminal.
- Surcharger les propriétés/attributs:
 - `action_spec` la description de l'espace d'action,
 - `observation_spec` la description de l'espace d'observation.

Les trois dernières lignes permettent d'utiliser une politique aléatoire pour tester la validité des descriptions des espaces d'action et d'observation.

Boucle d'apprentissage TensorFlow

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 import tensorflow as tf
5
6
7 from tf_agents.agents.reinforce import reinforce_agent
8 from tf_agents.drivers import dynamic_step_driver
9 from tf_agents.environments import tf_py_environment
10 from tf_agents.eval import metric_utils
11 from tf_agents.metrics import tf_metrics
12 from tf_agents.networks import actor_distribution_network
13 from tf_agents.replay_buffers import tf_uniform_replay_buffer
14 from tf_agents.trajectories import trajectory
15 from tf_agents.utils import common
16 from tf_agents.policies import policy_saver
17
18 tf.compat.v1.enable_v2_behavior()
19
20 from interface_tf import MiniSimEnv
21
22 num_iterations = 100000 # @param {type:"integer"}
23 collect_episodes_per_iteration = 3 # @param {type:"integer"}
24 replay_buffer_capacity = 2000 # @param {type:"integer"}
25
26 fc_layer_params = (30, 15) # Nombre de neurones par couche du réseau de
27
28 learning_rate = 1e-3 # @param {type:"number"}
29 log_interval = 25 # @param {type:"integer"}
30 num_eval_episodes = 5 # @param {type:"integer"}
31 eval_interval = 50 # @param {type:"integer"}
32
33 policy_dir = "cerveau_sm_delta"
34 """ Nom de l'essai """
35
36 train_py_env = MiniSimEnv()
37 eval_py_env = MiniSimEnv()
38 """ Instanciation des environnements """
39
40 train_env = tf_py_environment.TFPyEnvironment(train_py_env)
41 eval_env = tf_py_environment.TFPyEnvironment(eval_py_env)
42 """ Encapsulation des environnements avec des tenseurs"""
43
44 actor_net = actor_distribution_network.ActorDistributionNetwork(
45     train_env.observation_spec(),
46     train_env.action_spec(),
47     fc_layer_params=fc_layer_params)
48 """ Création d'un réseau de neurones correspondant à nos spécifications"""
49
50 optimizer = tf.compat.v1.train.AdamOptimizer(learning_rate=learning_rate)
51 """ Instanciation de l'optimiseur """
```

```
58         train_env.action_spec(),
59         actor_network=actor_net,
60         optimizer=optimizer,
61         normalize_returns=True,
62         train_step_counter=train_step_counter)
63 tf_agent.initialize()
64 """ Instanciation de l'agent """
65
66 eval_policy = tf_agent.policy
67 collect_policy = tf_agent.collect_policy
68
69 def compute_avg_return(environment, policy, num_episodes=10):
70     """ Calcul la récompense totale moyenne d'un politique sur un environnement """
71     total_return = 0.0
72     for _ in range(num_episodes):
73
74         time_step = environment.reset()
75         episode_return = 0.0
76
77         while not time_step.is_last():
78             action_step = policy.action(time_step)
79             time_step = environment.step(action_step.action)
80             episode_return += time_step.reward
81         total_return += episode_return
82
83     avg_return = total_return / num_episodes
84     return avg_return.numpy()[0]
85
86 replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
87     data_spec=tf_agent.collect_data_spec,
88     batch_size=train_env.batch_size,
89     max_length=replay_buffer_capacity)
90
91 def collect_episode(environment, policy, num_episodes):
92     """ rajoute des trajectoires dans le buffer"""
93     episode_counter = 0
94     environment.reset()
95
96     while episode_counter < num_episodes:
97         time_step = environment.current_time_step()
98         action_step = policy.action(time_step)
99         next_time_step = environment.step(action_step.action)
100        traj = trajectory.from_transition(time_step, action_step, next_time_step)
101
102        # Add trajectory to the replay buffer
103        replay_buffer.add_batch(traj)
```

```
110     # Reset the train step
117     tf_agent.train_step_counter.assign(0)
118
119     # Evaluate the agent's policy once before training.
120     avg_return = compute_avg_return(eval_env, tf_agent.policy, num_eval_episodes)
121     returns = [avg_return]
122
123     for _ in range(num_iterations):
124
125         # Accumule des trajectoires
126         collect_episode(
127             train_env, tf_agent.collect_policy, collect_episodes_per_iteration)
128
129         # On utilise ses trajectoires pour l'entraînement
130         experience = replay_buffer.gather_all()
131         train_loss = tf_agent.train(experience)
132         replay_buffer.clear()
133
134         step = tf_agent.train_step_counter.numpy()
135
136         if step % log_interval == 0:
137             print('step = {0}: loss = {1}'.format(step, train_loss.loss))
138
139         if step % eval_interval == 0:
140             avg_return = compute_avg_return(eval_env, tf_agent.policy, num_eval_episodes)
141             print('step = {0}: Average Return = {1}'.format(step, avg_return))
142             returns.append(avg_return)
143             tf_policy_saver.save(policy_dir)
144             if avg_return > 60 + 10*15: # Objectif de récompense
145                 break
146
147     except KeyboardInterrupt:
148         print("STOP")
149         tf_policy_saver.save(policy_dir)
150         tf_policy_saver.save(policy_dir+"_final")
151
152     steps = range(0, num_iterations + 1, eval_interval)
153     plt.plot(steps[:len(returns)], returns)
154     plt.ylabel('Average Return')
155     plt.xlabel('Step')
156     plt.savefig("rendement.png")
```

Code : Boucle d'apprentissage, les paramètres ne sont pas définitifs

Récompense

Une des problématiques qu'il faut résoudre est la distribution des récompenses. En effet, il faut arriver à créer une mesure des "Bonnes actions". Une solution pourrait être de donner comme récompense la distance parcourue, par exemple en intégrant la vitesse sur tout l'épisode. Notre solution est plutôt de donner deux types de récompenses:

- une récompense faible pour encourager la voiture à avancer en ligne droite
- une récompense importante lorsque la voiture passe une porte

Les portes sont disposées sur le circuit pour matérialiser la progression de la voiture sur le circuit.

Mais je donne aussi des pénalités:

- lorsque la voiture recule
- lorsqu'elle touche un mur

De plus, l'épisode prend fin lorsque la voiture touche un mur, ou lorsqu'il s'est écoulé 60 sec, afin de ne pas avoir des épisodes de longueur infinie si la voiture devient très performante.

Résultat

Voici le résultat après 3h de simulation:

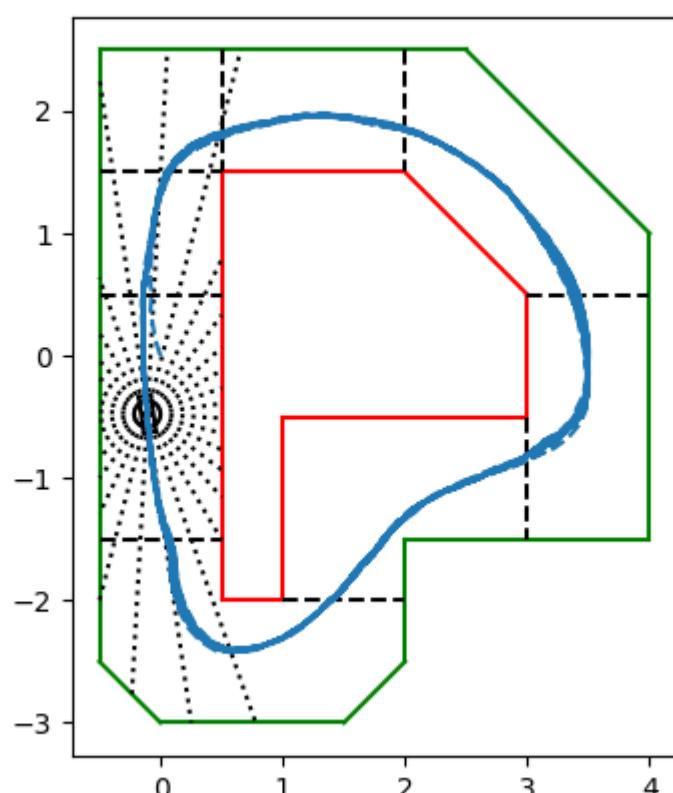


Figure : Trajectoire empruntée par la voiture en simulateur, sur un circuit simple pour l'entraînement

On a bien une politique permettant de diriger la voiture sur le circuit. On semble même atteindre une sorte de cycle limite donc on peut espérer que la politique soit un peu robuste. Ici, je présente un système avec le lidar à 360°. Le résultat de la simulation avec un lidar à 240° est présenté dans la conclusion.

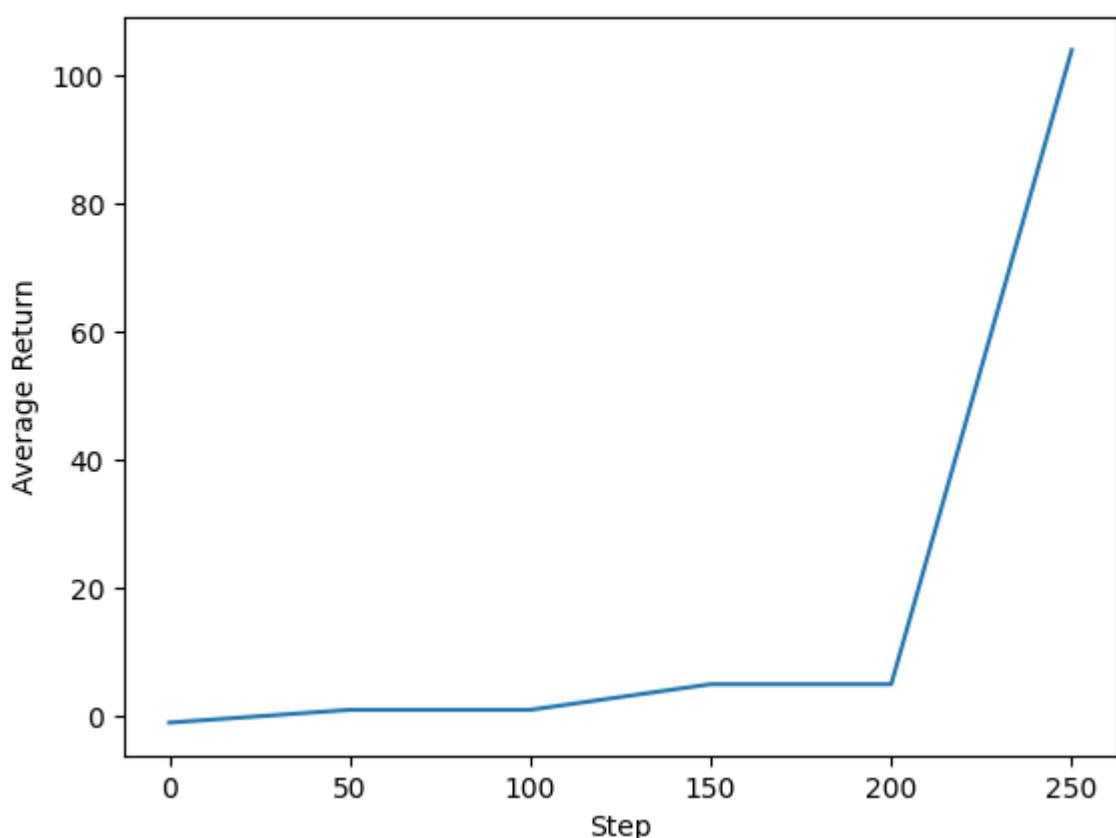


Figure : Évolution du retour moyen (somme des récompenses sur un épisode, moyenné sur plusieurs essais) pendant l'entraînement.

En regardant l'évolution du retour moyen au fil des étapes d'optimisation, on se rend compte qu'il y a eu une révélation au moment où la voiture a réussi à faire un tour en entier.

Simulation sur Webots

Le contrôleur utilisé par Webots pour l'apprentissage est dans le fichier `contrroleur_alpha.py`. On utilise `DeepBots`, un module python, pour servir d'interface GYM à l'environnement Webots, ainsi la boucle d'apprentissage conçue précédemment reste compatible.

Voiture réelle

Le déploiement de notre IA se fait grâce au module `policie_saver` de TensorFlow qui permet d'exporter la politique sans avoir besoin de connaître le modèle sous-jacent.

Malheureusement, je n'ai pas réussi à déposer TensorFlow sur notre Raspberry Pi4, malgré le fait qu'il existe des versions précompilées de TensorFlow pour Raspberry Pi4.

Malgré cela, nous avions quand même créé un module pour le déploiement dans le fichier `interface_materielle.py` bien qu'il n'ait pas été testé en conditions réelles.

Ici, on n'utilise que L_t comme vecteur d'observation, d'où le nom `CerveauSansMemoire`. On définit donc une interface avec `CerveauAbstrait` dans un souci de compatibilité. On pourrait créer de nouvelles classes si on change le vecteur d'observation.

Conclusion

Bien que nous n'ayons pas pu déployer la politique sur la machine réelle, nous pensons que cela permettrait d'avoir un bon contrôle de la voiture. Outre le déploiement, il reste d'autre pistes à étudier:

- Tester d'autres architectures de réseau de neurones, qui peuvent prendre en compte le caractère particulier de L_t , par exemple avec une convolution pour détecter les angles
- L'entraînement successif sur des environnements de plus en plus complexe pour avoir un apprentissage plus efficace
- L'entraînement sur des simulateurs plus poussés, avec des paramètres aléatoires pour augmenter la robustesse du modèle
- L'utilisation d'un vecteur d'observation différent, par exemple en donnant les caractéristiques des clusters déterminés par Rémi Lembeye.

Mais même sans ces améliorations, je suis confiant en la capacité de l'apprentissage machine à trouver une solution performante au problème de calcul de trajectoire. En effet, en lançant notre IA dans une reproduction du circuit de la course, on obtient la trajectoire suivante, qui est assez satisfaisante (en bleu, la trajectoire, les tirets noirs sont les portes et les pointillés les rayons du lidar) :

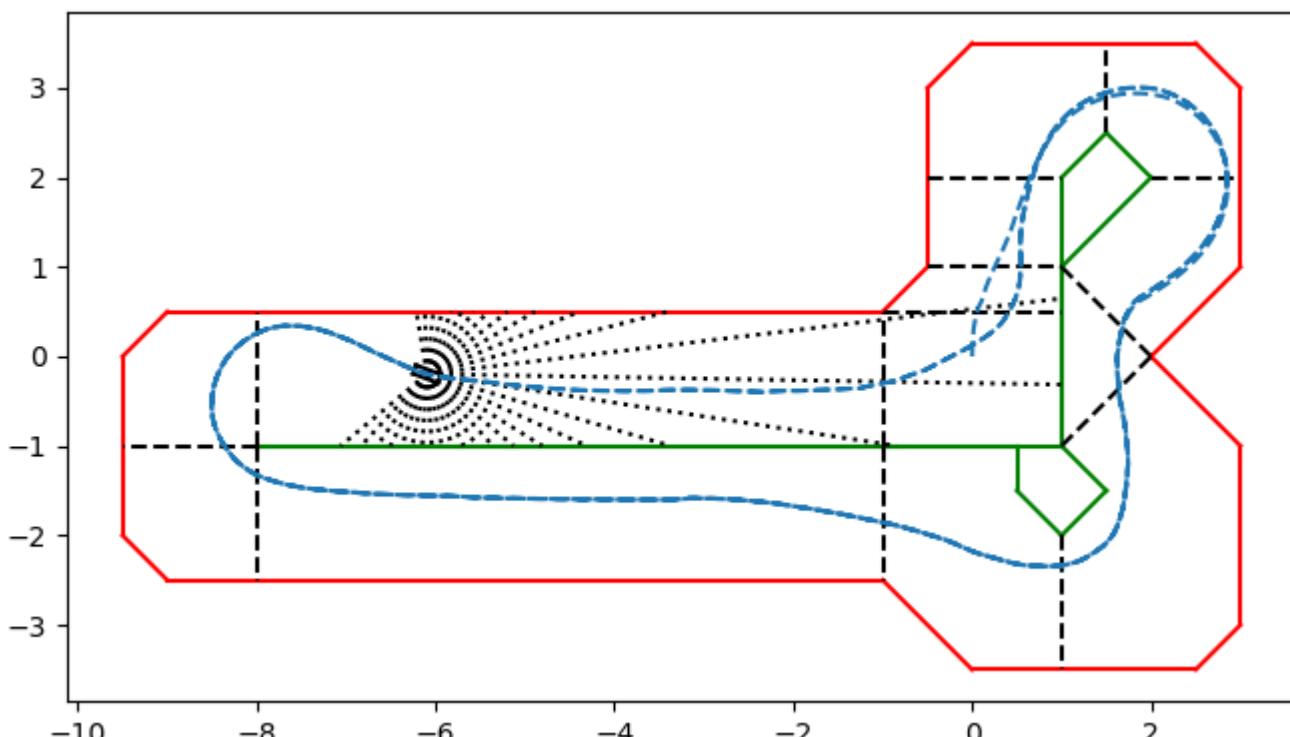


Figure : Trajectoire empruntée par la voiture en simulateur, avec un circuit proche du circuit de la course.

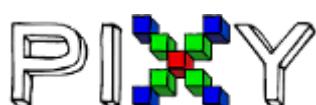
Partie d'Amaury

Au début des séances de TER, afin de pouvoir détecter le sens de la piste et le sens de redémarrage si la voiture rencontre un mur, 2 hypothèses furent étudiées :

- utiliser une caméra effectuant déjà un traitement de l'image avec ses propres algorithmes déjà implémentés → caméra retenue : Pixy2
- utiliser une caméra simple et appliquer nous même le traitement des images reçues → caméra retenue : Raspberry Pi Camera V2

Dans un premier temps, la caméra Pixy2 paraissait la plus adaptée à notre problème car elle possédait déjà son propre traitement de l'image qu'il fallait juste adapter à notre situation. Cependant, comme je vais l'expliquer, la solution finale est d'utiliser la caméra du Raspberry Pi et de coder nous même le traitement de l'image.

Travail sur la caméra Pixy2



J'ai commencé par travailler sur la caméra Pixy2. Mais avant de commencer à travailler avec les résultats du traitement de l'image effectué par la caméra, il fallait commencer par étalonner cette dernière. C'est ce que l'on peut faire à l'aide du logiciel PixyMon.

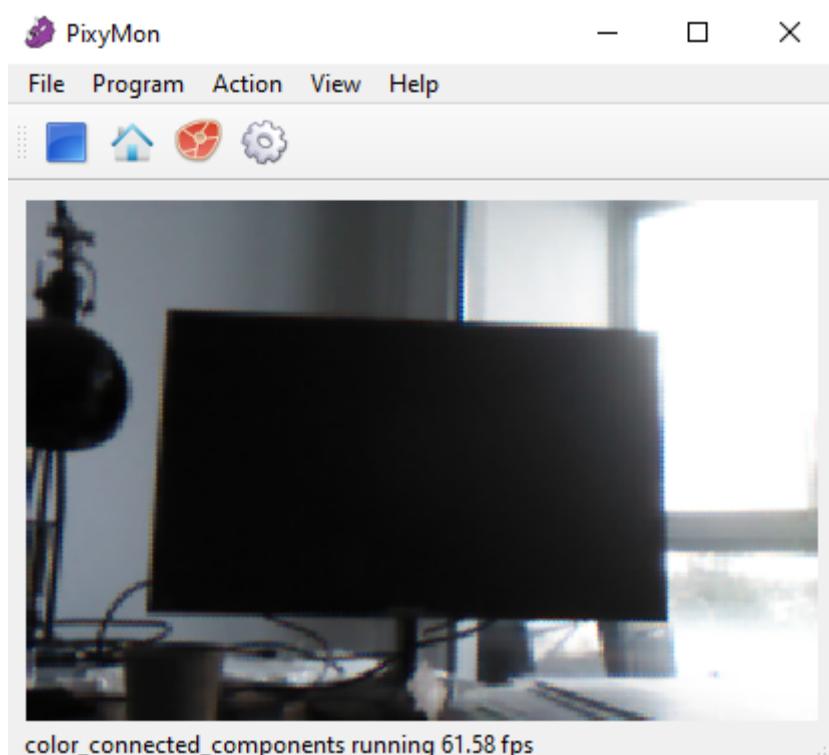


Figure : Écran de fonctionnement du logiciel PixyMon V2

En effet, notre but est d'utiliser la caméra dans un seul de ces modes de fonctionnement, celui de la reconnaissance de couleur. La caméra doit reconnaître les couleurs des bords de la piste et nous renvoyer les coordonnées de ces bords. Un léger traitement des données renvoyées par la caméra nous permettrait alors de déterminer le sens de la piste : il suffit de vérifier de quel côté de l'écran on voit le vert et le rouge.

Cependant, afin de reconnaître les couleurs, la caméra doit être étalonnée, c'est à dire que l'on doit lui définir ce qu'est la couleur rouge et ce qu'est la couleur verte. Le résultat de cet étalonnage est montré sur la Figure suivante :

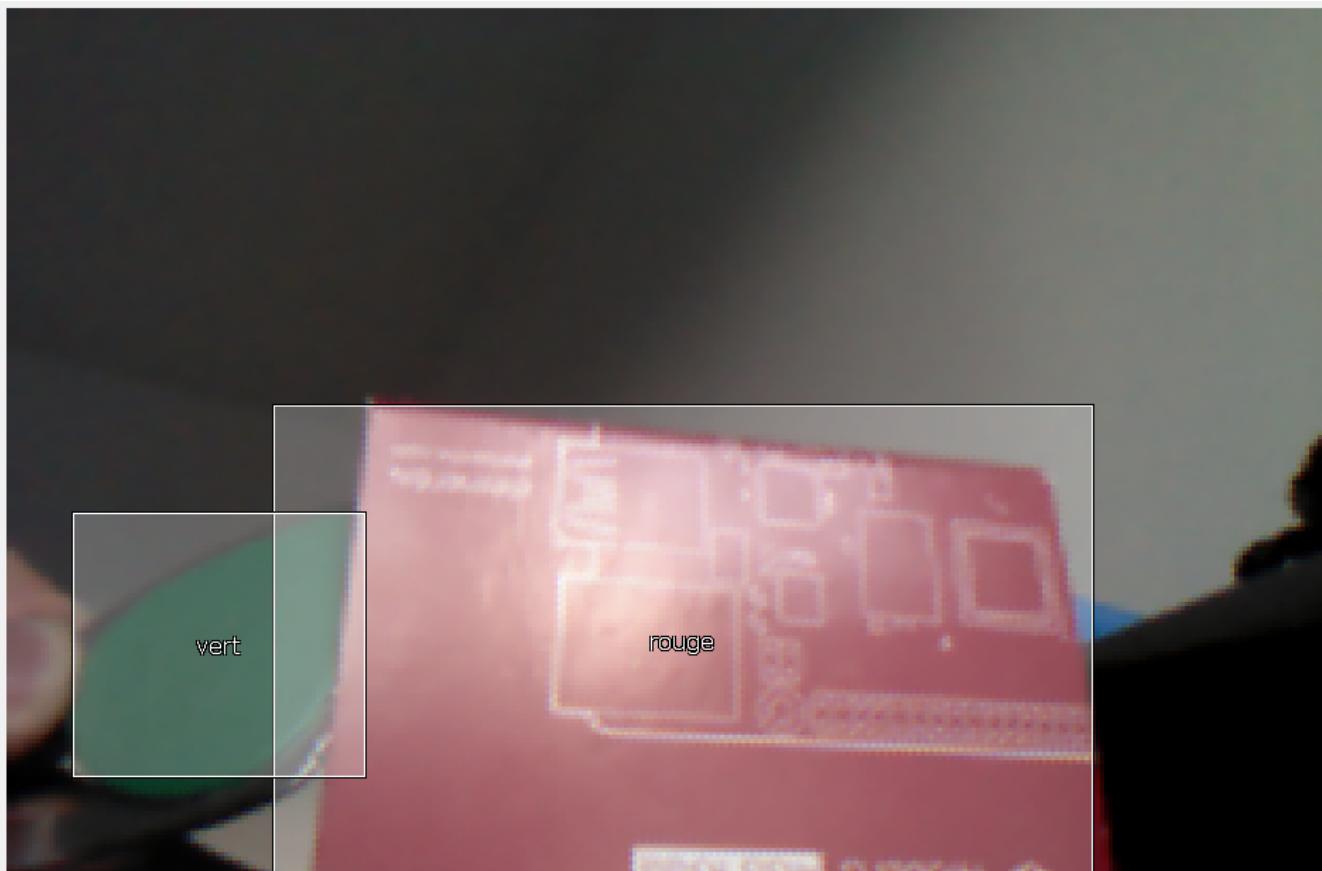


Figure : Reconnaissance de couleur de la Pixy2 après étalonnage

Une fois les différentes couleurs enregistrées dans la caméra, le but était de récupérer les données des carrés de couleur renvoyés par la caméra. A l'aide d'un algorithme simple comparant la latitude des carrés rouges et des carrés verts, nous aurions alors pu déterminer le sens de la piste.

Cependant, cette caméra pose plusieurs problèmes :

- Elle détecte très mal les variations de luminosité : elle peut être efficace dans le cas de couleurs brillantes mais comme elle détecte très mal les nuances de gris. Ainsi, elle sera peu efficace pour la détection de couleurs très foncées ou très claires.
- Cette caméra possède déjà son traitement de l'image. Cela peut paraître comme étant un avantage au premier abord, mais nous ne maîtrisons pas du tout le format

des données de sortie et ne pouvons pas demander à la caméra de faire exactement ce que nous voulons : on doit se contenter de ce qu'elle nous propose.

Pour pallier au premier problème, j'ai donc réglé toutes les sensibilités afin de détecter au mieux le rouge et le vert comme on peut le voir sur la Figure suivante :

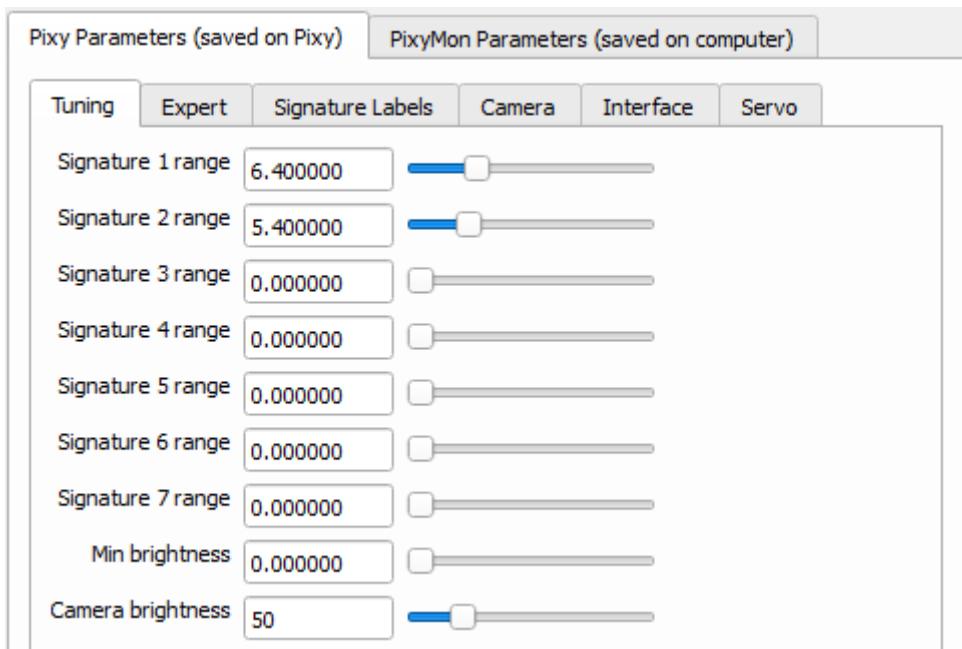


Figure : Réglage de détection des couleurs avec le logiciel Pixymon V2

Avec cette méthode, je n'obtenais jamais une détection 100% fiable des deux couleurs à cause de la luminosité, et même en faisant la détection de plusieurs nuances de rouge et de vert, à partir du moment où ces couleurs étaient trop foncées ou trop claires, la caméra ne les détectait pas.

Pensant que ce problème trouverait une solution une fois que l'on commencerait les essais sur la piste, et que les couleurs des murs seraient vives et brillantes, j'ai voulu utiliser la caméra en détection d'objet. Je voulais que la caméra détecte les formes trapézoïdales des murs afin qu'elle puisse nous aider à anticiper les virages. C'est ce que l'on peut voir sur les Figures suivantes :

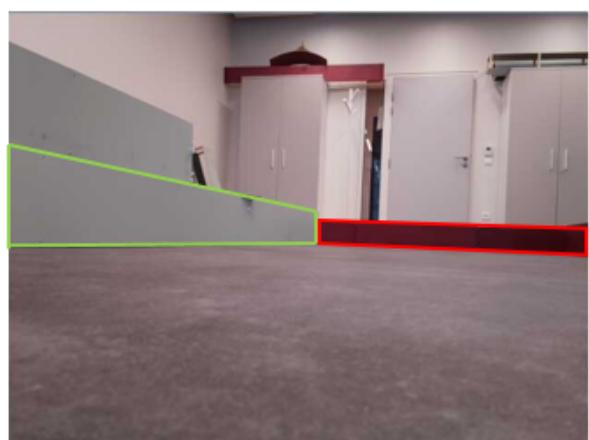
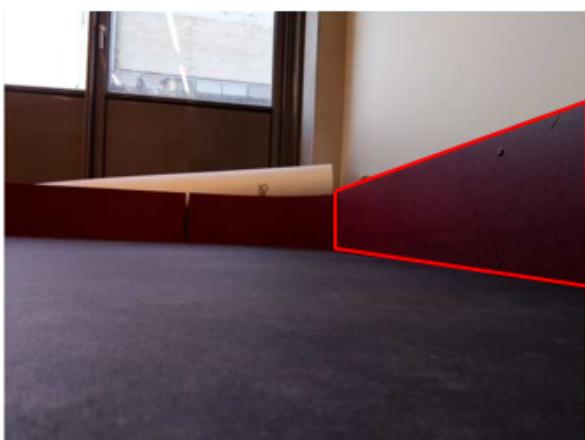


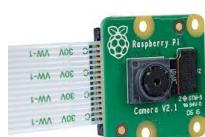
Figure : Mise en place de la détection de forme des murs

Cette méthode c'est avérée très longue à mettre en place pour une fiabilité trop faible et elle n'a donc pas été retenue.

A ce moment du projet, j'ignorais que les couleurs choisies pour les murs de la piste ne seraient pas détectées par la Pixy2 (rouge trop foncé et vert trop clair).

Pour pallier au second problème, celui qui fait que je ne maîtrise pas les données de sortie du traitement de l'image propre à la caméra, la solution fut d'essayer une autre caméra, ce que nous allons maintenant traiter.

Travail sur la caméra v2 de la Raspberry pi 4



Plusieurs avantages venaient avec cette seconde caméra :

- nous contrôlons totalement le traitement que nous effectuons sur les différentes images
- cette caméra étant faite pour fonctionner avec un Raspberry Pi, elle paraît parfaitement adaptée à notre véhicule

Pour commencer l'utilisation de cette caméra, il faut télécharger les bibliothèques nécessaires au traitement de l'image pour détecter des couleurs ou des objets facilement : la librairie OpenCV. Il faut ensuite fixer les paramètres d'utilisation de la caméra. Pour la détection de couleurs, les principaux paramètres à fixer sont la résolution et la luminosité.

Dans un premier temps, la méthode envisagée était de compter le nombre de pixels verts et de pixels rouges puis de définir les barycentres associés à chacune de ces deux couleurs. Enfin, en comparant la position de ces barycentres, on aurait pu définir dans quel sens allait la piste. Bien sûr, afin de s'affranchir du rouge et du vert qui serait détectés au-dessus de la piste, il a fallu déterminer une "bande utile" qui ne prendrait en compte que les pixels sous une certaine hauteur.

Détermination de la bande utile :

Lorsque l'on observe un mur de près ou de loin, la hauteur de la bande utile varie, c'est pourquoi nous avons décidé de prendre le cas où la bande utile est la plus utile : lorsqu'on est loin d'un mur et que l'extérieur de la piste est très présent sur l'image.





Figure : Images complètes (sans bande utile)



Figure : Bande utile calée sur l'image de gauche

On teste l'algorithme pour l'image suivante :



Figure : Image test de l'algorithme

L'algorithme fonctionne bien et renvoie les coordonnées des barycentres rouge ($x=472$; $y=325$) et vert ($x=216$; $y=385$) pour une image de résolution (480,640). En comparant la position des deux barycentres, on pouvait détecter qui du rouge ou du vert est le plus à droite (ou à gauche) et ainsi déterminer le sens de course.

Cette solution paraissait parfaitement adaptée à notre problème jusqu'à ce que l'on connaisse les couleurs finales de la piste. En effet, le rouge reste bien détecté par la caméra V2 de la Raspberry Pi mais ce n'est pas le cas du vert clair qui est dans la majorité des cas perçus comme une nuance de gris. C'est pourquoi nous avons décidé de changer de méthode de détermination du sens de course.

Méthode K-means:

Cette méthode consiste à considérer l'image comme étant un ensemble de pixels et vise à diviser les pixels en k groupes, appelés clusters, homogènes et compacts. Le nombre de groupes est fixé par l'utilisateur. Le but est de regrouper les pixels de couleurs proches en une même couleur. Cette couleur unique pour chaque groupe est considérée comme la couleur moyenne du groupe. L'objectif est de réussir à différencier le rouge, le vert et le sol. Voici un exemple d'application de la méthode K-means avec $k=4$ (Attention, on travaille toujours sur la bande utile).

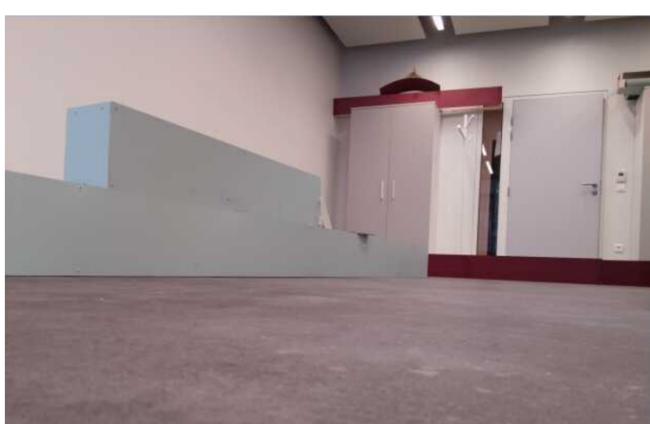
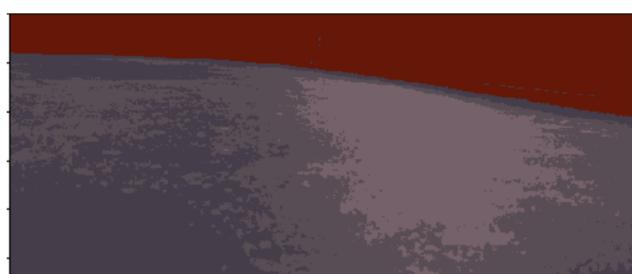
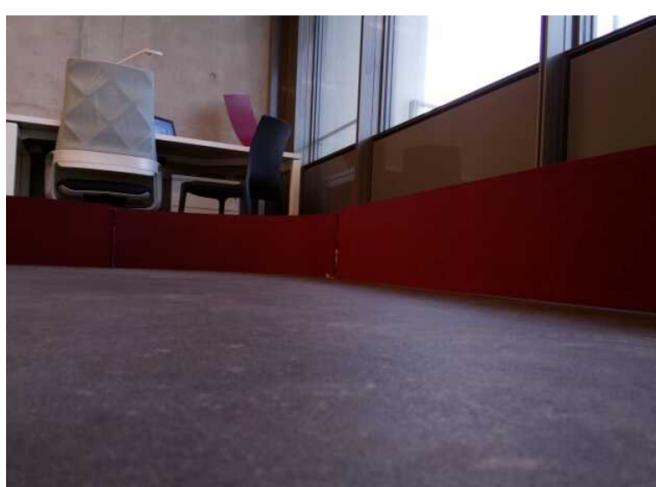




Figure : Application de la méthode K-means avec k=5 dans 2 configurations

On remarque bien que cette méthode permet de distinguer le rouge, le vert et le gris du sol. Ainsi, après avoir déterminé les différents clusters, on passe l'image en hsv (Hue Saturation Value) et on détermine quels clusters correspondent au vert et au rouge assez facilement en appliquant des filtres sur les valeurs de h,s et v.

Une fois que l'on connaît le cluster vert et le cluster rouge, on peut à nouveau appliquer la méthode des barycentres qui est expliquée précédemment et ainsi savoir le sens de course. L'algorithme fonctionne bien et renvoie les coordonnées des barycentres rouge ($x=533;y=325$) et vert ($x=207;y=317$) pour une image de résolution (480,640). Cela signifie que les deux clusters sont proches du milieu de l'image selon la hauteur et que le vert est nettement plus à gauche que le rouge donc que le sens de piste est faux.

Pourquoi traiter l'image en hsv (Hue, Saturation, Value) :

Les images capturées et lues le sont tout d'abord en RGB (Red, Green, Blue). C'est un code précis qui détermine chaque couleur en fonction de sa proportion de Rouge, de Vert et de Bleu.

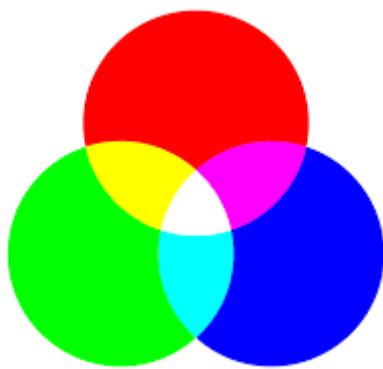


Figure : Présentation du code couleur RGB

Cependant, pour notre application, nous avons seulement besoin de détecter la couleur rouge et la couleur verte, et c'est là qu'intervient le code couleur HSV.

En effet, en code HSV, la couleur n'est représentée que par H qui est compris entre 0 et 360 degrés. La saturation et la Value ne font que régler l'intensité et la brillance de cette couleur.



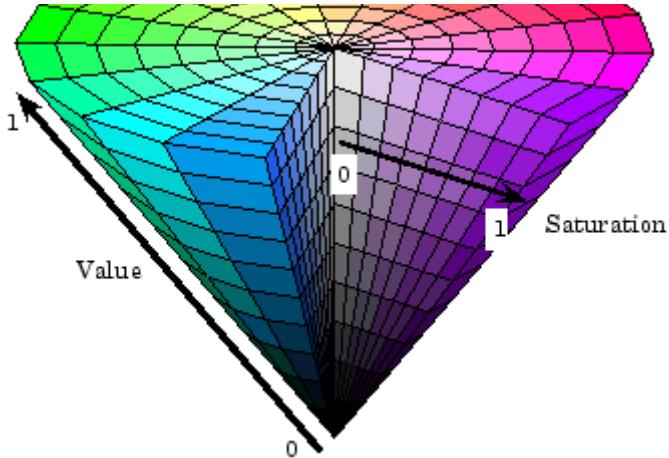


Figure : Présentation du code couleur HSV

Ainsi, il est plus facile de traiter un unique paramètre H pour déterminer la couleur des pixels plutôt que de prendre en compte 3 paramètres (R,G,B). C'est pourquoi, nous transformons la bande utile en HSV quand nous voulons déterminer la couleur des clusters. Toutefois, la détermination des clusters en eux-mêmes peut très bien s'effectuer directement en RGB. Ce n'est que lorsqu'on veut savoir quel cluster est rouge et lequel est vert que l'on passe l'image en hsv.

Comment faire pour traiter l'image en continu :

Pour traiter en continu les images perçues par la caméra, deux options s'offraient à nous : soit la caméra filmait en continu et nous traitions en direct chaque frame, soit la caméra prenait une photo toutes les secondes et nous traitions cette photo. On a le tableau de comparaison suivant :

Méthode	Avantages	Inconvénients
vidéo	traitement continu	Nécessite un traitement lourd
photos	traitement bien moins lourd	traitement toutes les secondes

Selon les besoins de notre utilisation, le choix a été la méthode qui prend des photos toutes les secondes. En effet, un traitement toutes les secondes est suffisant pour avoir une bonne réaction si la voiture se trompe de sens. De plus, le traitement ne se fait pas sur une vidéo mais sur une image donc il demande bien moins de puissance de calcul pour traiter les données. On a alors le code suivant :

```
1 def camera_stream():
2     while True:
3         camera.capture('image.jpg')
4         image = cv2.imread("image.jpg")
5         sens=sens_detect(image)
6         sleep(1)
```

Méthode pour savoir de quel côté repartir lorsqu'on rencontre un mur :

Cette fonction est assez simple à mettre en place. Lorsque la voiture s'arrête et rentre dans un mur, on prend une photo du mur. On passe l'image en hsv (Hue, Saturation, Value) et on ne regarde que la valeur de h. Si la moyenne des valeurs de h se trouve dans l'ensemble correspondant au rouge alors il faudra tourner à droite pour repartir dans le bon sens. Sinon, il faudra repartir à gauche.

```
1 def arret_mur_repartir(frame):
2     image_hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV) #on passe l'image
3
4     h,s,v=cv2.split(image_hsv) #on isole h
5     h[h>160]= 180-h[h>160] #on regroupe tout le rouge entre 0 et 20
6     moyenne=h.mean()
7     if moyenne>0 and moyenne<30: #mur rouge
8         turn_right=True
9     else : #mur vert
10        turn_right=False
11    return turn_right
12    #print(turn_right) #vérification visuelle
```

Conclusion

Je pensais au départ que le traitement de l'image serait assez rapide à effectuer avec une caméra possédant déjà ses propres algorithmes de traitement de l'image. Cependant, même si la reconnaissance de forme est relativement aisée à réaliser, j'ai très vite pris conscience de la difficulté à reconnaître une couleur car la vision de la caméra dépend beaucoup trop de la luminosité extérieure ou encore de l'ombre : dès que l'on travaille avec une luminosité changeante avec des couleurs soit très sombres soit très claires, elles ne sont pas détectées correctement.

Ainsi, ce projet m'a permis d'implémenter moi-même des algorithmes de traitement de l'image de plus en plus complexes au fur et à mesure que je rencontrais des problèmes de détermination de couleurs. J'ai aussi vu comment ces algorithmes pouvaient trouver une application concrète à travers les technologies actuelles comme les voitures autonomes.

Partie d'Hardy

Création d'un simulateur

Premièrement, mon travail s'est porté sur la création d'un simulateur afin de pouvoir travailler sur l'élaboration d'une stratégie de course. Ce simulateur a aussi pour but de servir d'environnement d'apprentissage pour un algorithme par renforcement.

Etant donné que le but est d'entraîner un algorithme sur ce simulateur, il se doit d'être le plus réaliste possible afin de faciliter le passage du simulateur à la réalité, voir [1].

Choix de l'environnement de simulation

Nous nous sommes très vite penchés sur l'environnement Webots. Webots est un environnement de simulation doté d'un moteur physique, il permet donc de reproduire des comportements du monde réel. Nous nous sommes intéressés à Webots car il présente plusieurs avantages. Premièrement, il dispose de plusieurs modèles de robots déjà créés qui sont importables facilement. Des capteurs déjà implémentés comme plusieurs lidars et une caméra, et la possibilité de coder en C++, java, python, matlab ce qui permet de garder une certaine liberté dans le développement des algorithmes de contrôle de la voiture. De plus, la documentation constructeur de Webots est assez fournie.

Choix du modèle de voiture

Webots contient plusieurs modèles de voiture à taille réelle. Cependant, notre voiture étant à échelle 1/10, nous doutons de la capacité de l'algorithme à passer du simulateur à la réalité si l'échelle est trop différente. Nous avons alors choisi le modèle de voiture ALTINO d'environ 30cm de longueur. La physique de ce modèle est basée sur le modèle d'Ackermann qui nous permet de calculer l'angle de braquage de chaque roue.

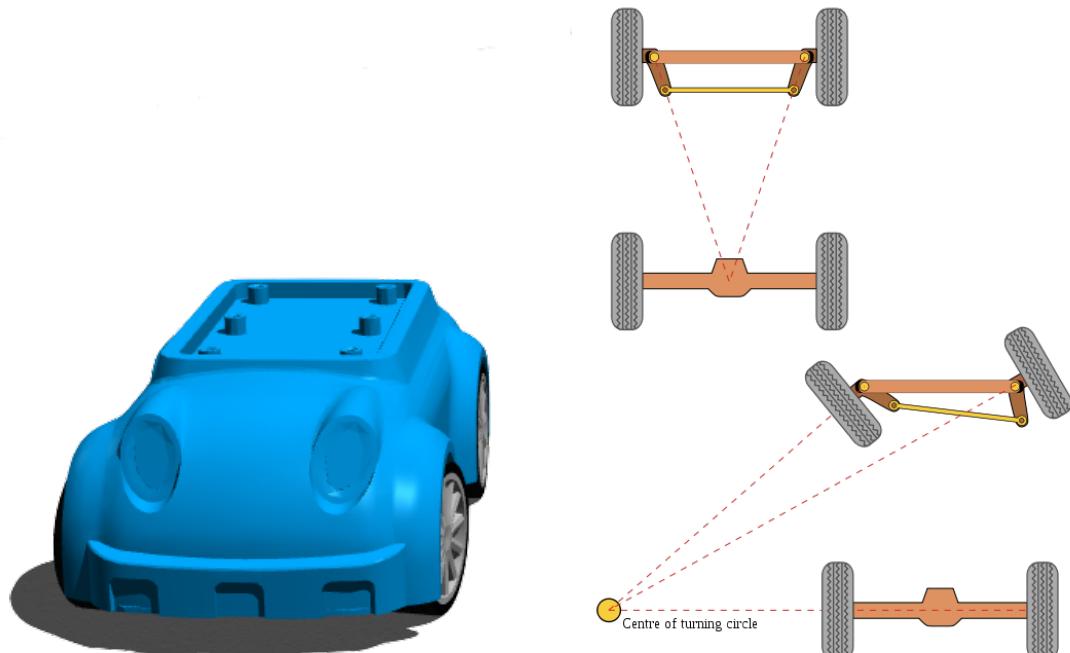


Figure : modèle ALTINO à gauche et modèle Ackermann à droite

De plus, afin de rendre le modèle le plus réaliste possible, nous l'avons retravaillé afin que la voiture réelle et le modèle aient la même longueur, largeur, taille de roue, écartement des roues, masse, et coefficient d'adhérence des pneus.

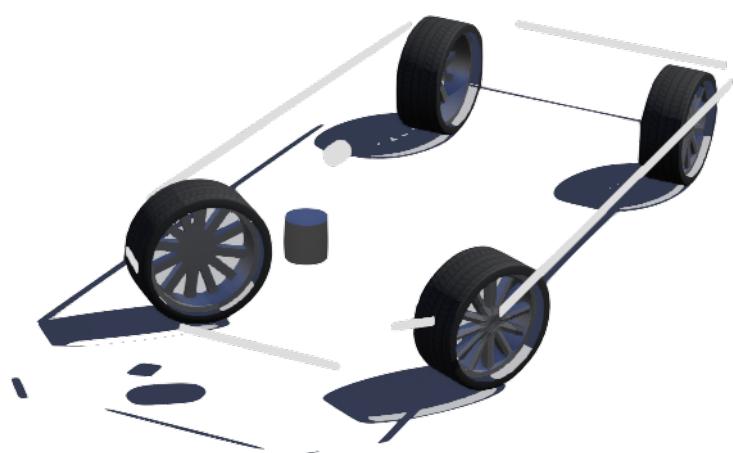
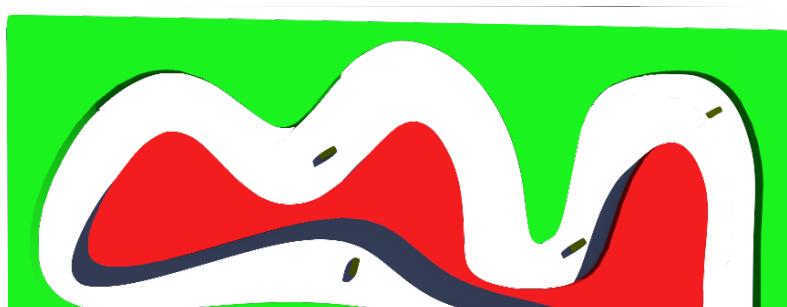
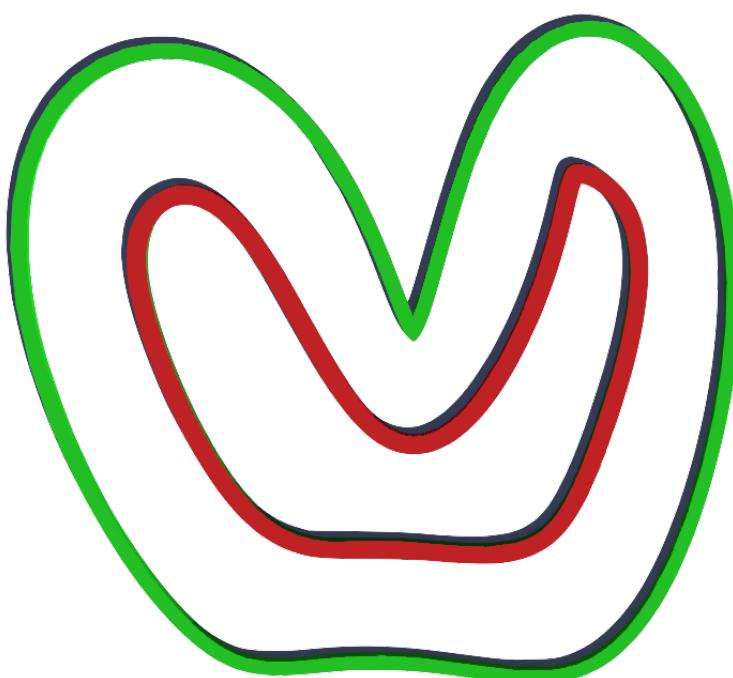
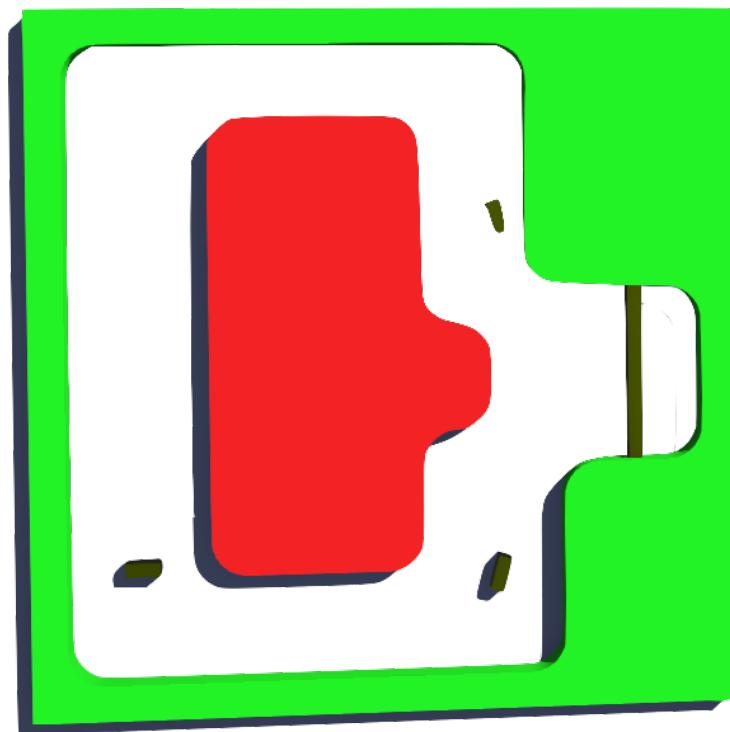


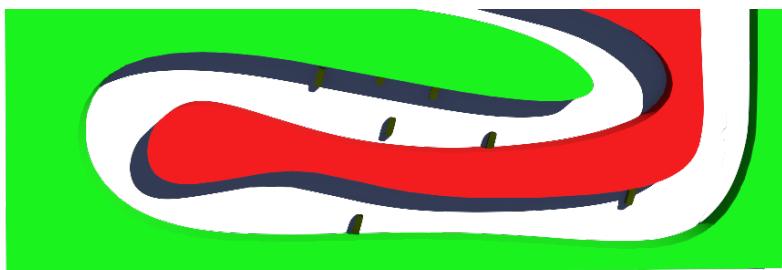
Figure : Modèle utilisé avec ses différents capteurs

On peut voir ci-dessus le modèle que nous avons construit (la carrosserie est ici invisible). Le cylindre bleu est le lidar que nous avons paramétré de manière à avoir les mêmes performances que notre lidar réel. Le petit cylindre blanc est la caméra : elle nous permet de connaître le sens de la course. Enfin les fines barres blanches sont des touch sensors : ils renvoient la valeur 1 dès qu'un objet entre dans leur hitbox. Ici, ils servent à redémarrer la simulation si la voiture entre en collision avec un mur ou un obstacle.

Réalisation de circuits sur simulateur

Les circuits ont été réalisés sur Fusion360 et les obstacles ajoutés depuis Webots. Ces circuits ont été faits pour présenter une difficulté croissante afin d'entraîner un algorithme sur des circuits du plus simple au plus complexe car cela permet de rendre plus efficace le passage du simulateur à la réalité, voir [2].





Développement de la stratégie de course

En collaboration avec les autres membres du groupe, nous avons décidé de contrôler la voiture via 2 paramètres : la vitesse de la voiture et l'angle de braquage des roues.

Détermination de la consigne de vitesse de la voiture

Afin de déterminer la vitesse de la voiture, il est naturel de prendre en compte la distance se trouvant à l'avant de la voiture.

Pour commencer nous avons pris la distance renvoyée par le lidar pour le rayon directement à l'avant de la voiture. Cette méthode fonctionnait bien sur le simulateur. Or, le problème de cette méthode est qu'elle donne lieu à des effets de tout ou rien. Par exemple, si un obstacle se trouve bien dans la direction de la voiture, elle va ralentir, mais s'il est faiblement désaxé par rapport à la voiture, celle-ci ne va pas du tout le prendre en compte.

Nous avons alors décidé de prendre en compte un ensemble de rayon à l'avant de la voiture. On définit alors une ouverture en degrés dans laquelle on vient récupérer les distances du lidar et dont on fait une moyenne. Cette méthode permet de mieux s'adapter à l'environnement. Cependant, elle a un certain nombre d'inconvénients.

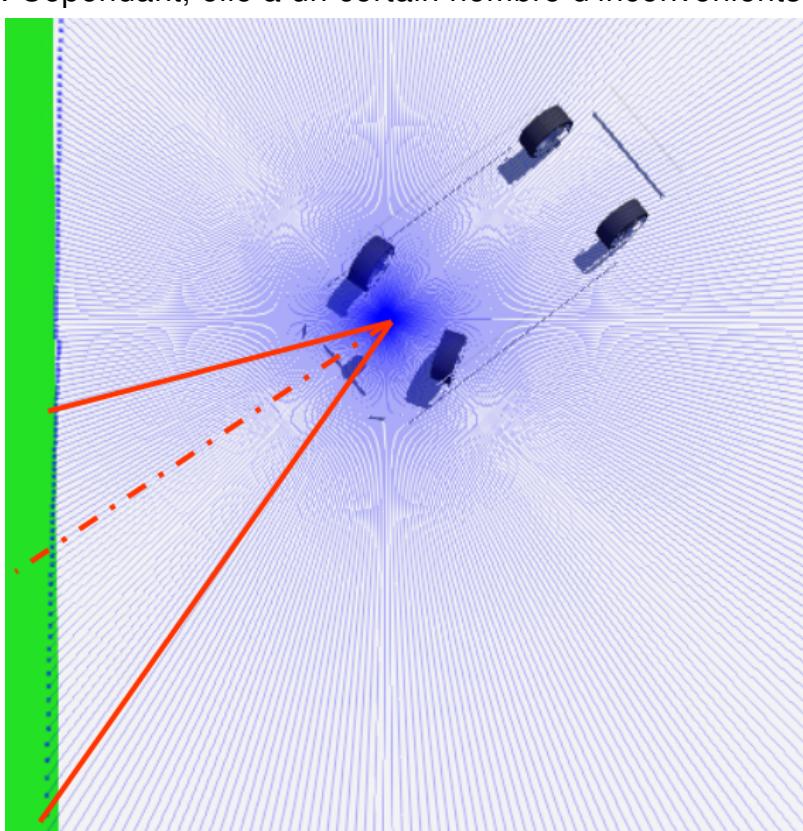
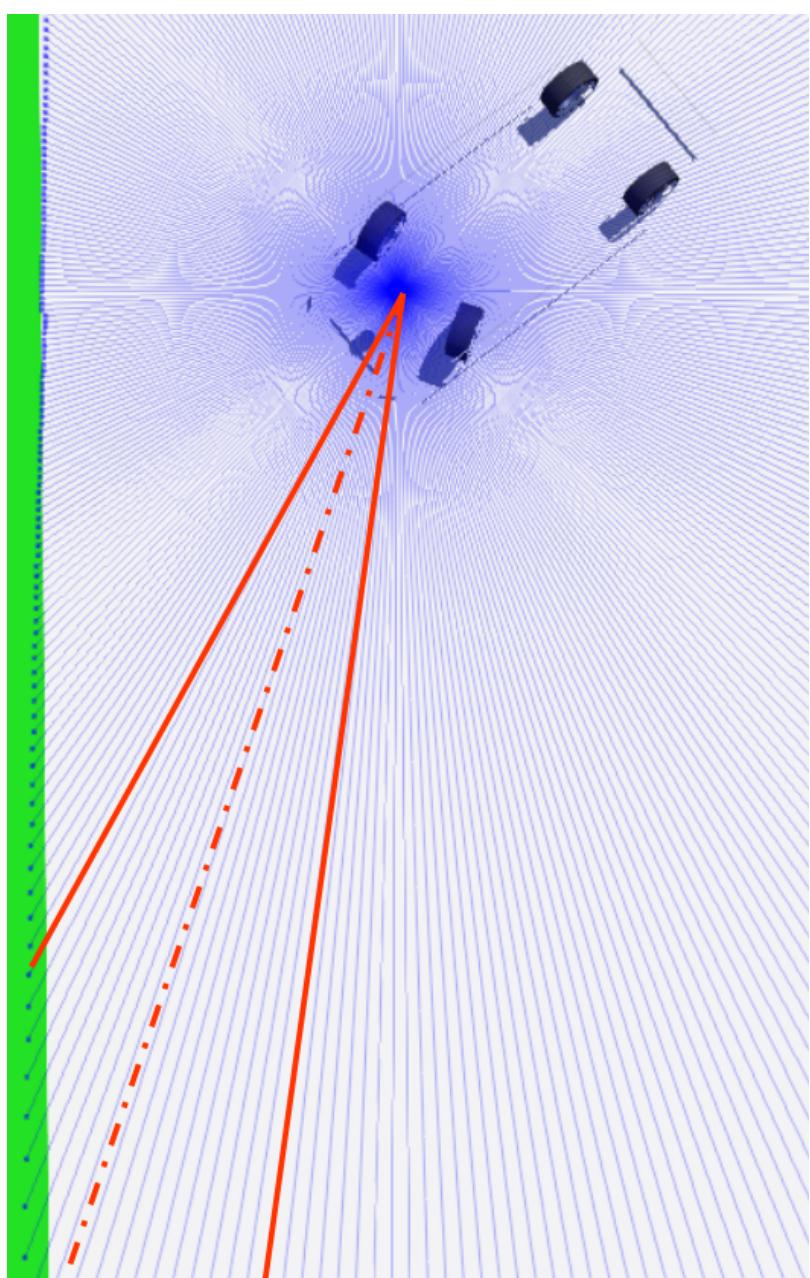




Figure : Illustration des problèmes causés par cette méthode

Comme on peut le voir ci-dessus, dans ce type de configuration, la voiture a très peu de place devant elle et a donc une vitesse nulle. Or, elle pourrait se sortir de cette situation en braquant au maximum et en avançant. Nous avons alors décidé de regarder les rayons du lidar dans la direction des roues et non pas en face de la voiture car cela permet de s'affranchir de ces situations. Comme le montre la figure suivante en alignant le cône dans lequel nous observons les distances avec l'angle des roues, la distance prise en compte pour calculer la vitesse est bien plus importante.



Cependant, d'autres problèmes sont encore présents avec cette méthode. Nous avons

notamment remarqué que lorsque la voiture était collée à un obstacle, elle ne s'arrêtait pas. En effet, comme on peut le voir sur la figure ci-dessous, lorsqu'un obstacle est proche et un peu désaxé par rapport à la voiture, il est susceptible de ne pas être pris en compte alors même que dans cette condition, la voiture ne peut pas passer cet obstacle.

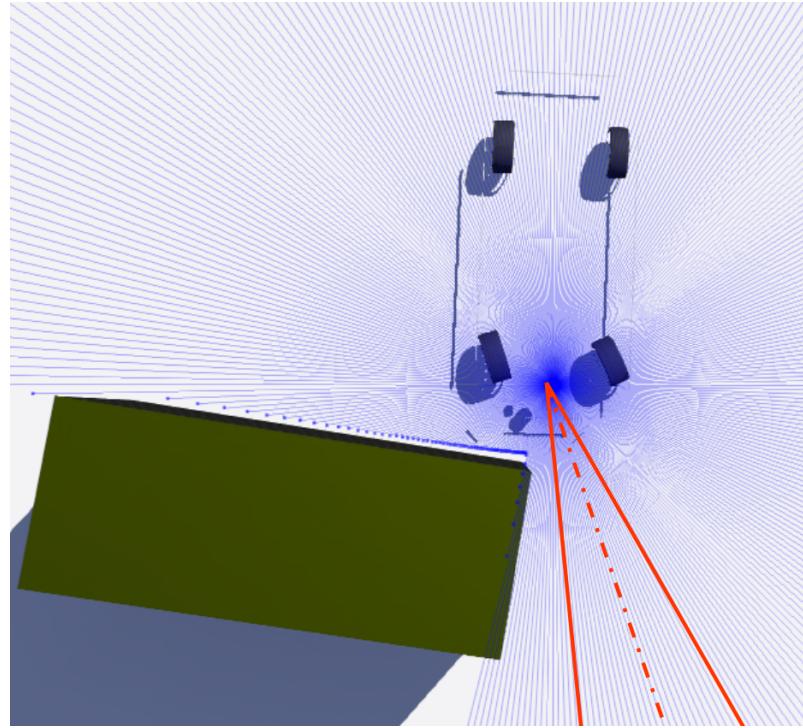
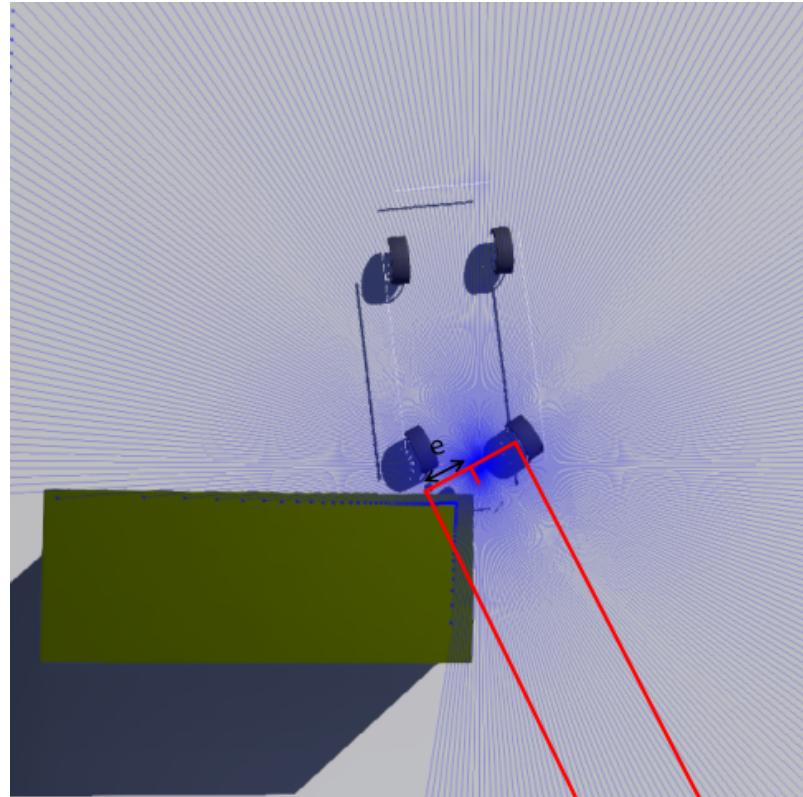
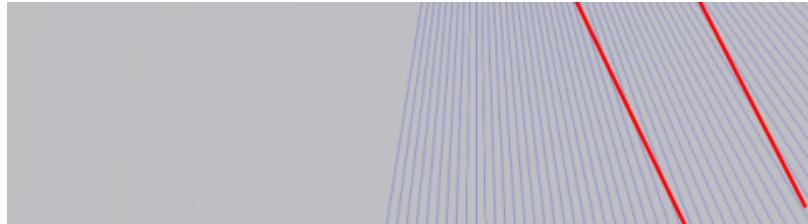


Figure : Problème posé par cette méthode de calcul

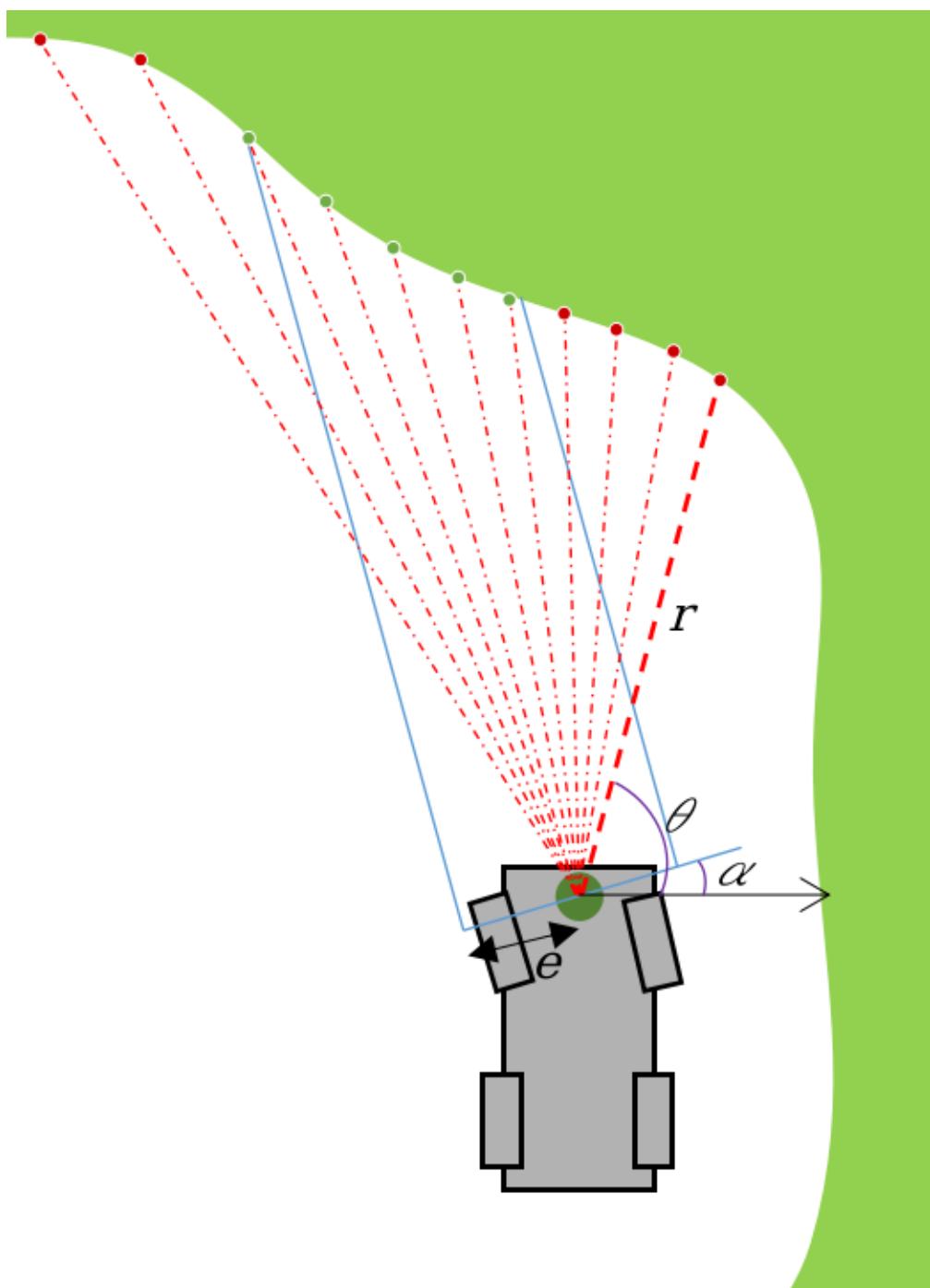
Afin de pallier cette difficulté, nous décidons d'utiliser un tube dans lequel on regarde les distances au lieu d'un cône. De cette manière, comme on peut le voir sur la figure suivante, les obstacles proches sont pris en compte.





La question est donc de définir les rayons qui sont dans le tube. Sur le schéma suivant, on peut voir les grandeurs importantes représentées. On a α qui est l'angle de braquage, θ représente l'angle d'un rayon du lidar, r est la distance renvoyée par le lidar, e est la demi-largueur du tube considéré.

Ainsi, un rayon qui entre dans le tube défini doit respecter l'équation suivante :
$$r < \frac{e}{\cos(\theta-\alpha)}$$



Cette dernière méthode est celle qui a été retenue pour la course. Maintenant que nous

avons défini les distances qui nous intéressent pour déterminer la vitesse de la voiture, il faut calculer une vitesse à partir de ces distances.

Afin de s'arrêter lorsqu'un obstacle est proche de la voiture, nous prenons en compte seulement le plus petit rayon respectant l'équation définie au-dessus. Ensuite, on applique la fonction suivante à la distance retenue.

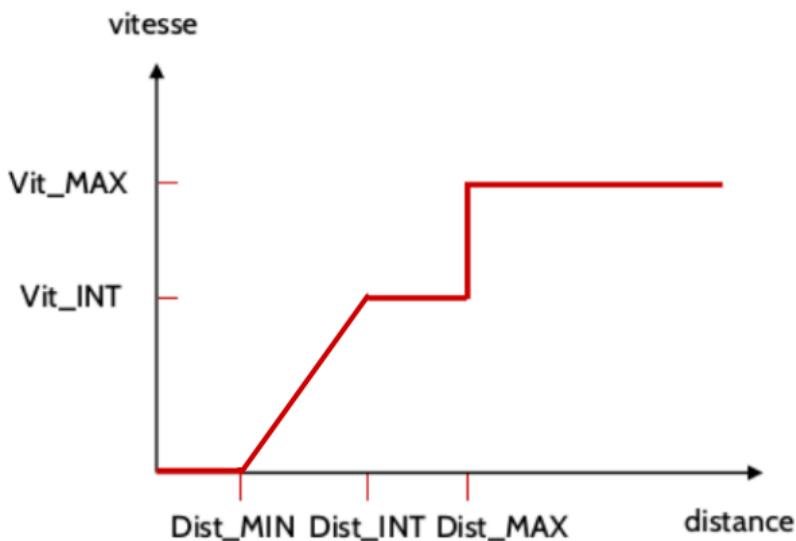


Figure : Loi de vitesse en fonction de la distance

Cette fonction permet, en ajustant $Dist_{MIN}$, de s'arrêter si un obstacle est proche et d'aller à une vitesse très soutenue, en ajustant Vit_{MAX} et $Dist_{MAX}$, si on a une grande distance devant la voiture.

Détermination de la consigne de l'angle de braquage de la voiture

La consigne de l'angle de braquage est basée sur un algorithme relativement simple que l'on a baptisé "méthode des rayons". Cette méthode consiste à sommer les distances renvoyées par le lidar dans le quart avant gauche et droite et à comparer ces deux sommes. La soustraction des deux sommes nous donne une erreur que le contrôleur PID de la direction va essayer d'annuler.

Cette méthode fonctionne très bien sur le virage d'angle de 0° à 90° , seulement nous nous sommes rendu compte que pour les virages en épingle, cette méthode n'est pas efficace. Il y a alors la nécessité de détecter les épingles. De plus, cette méthode n'est pas efficace pour éviter les obstacles, on doit donc aussi mettre en place une méthode de détection d'obstacles.

Détection d'épingles et d'obstacles

Pour détecter les épingles et les obstacles, on va réaliser un clustering des différents points du lidar.

Pour cela on va parcourir les points du lidar et si un point et son suivant sont à une

distance supérieure à un certain seuil, ils sont placés dans deux clusters différents.

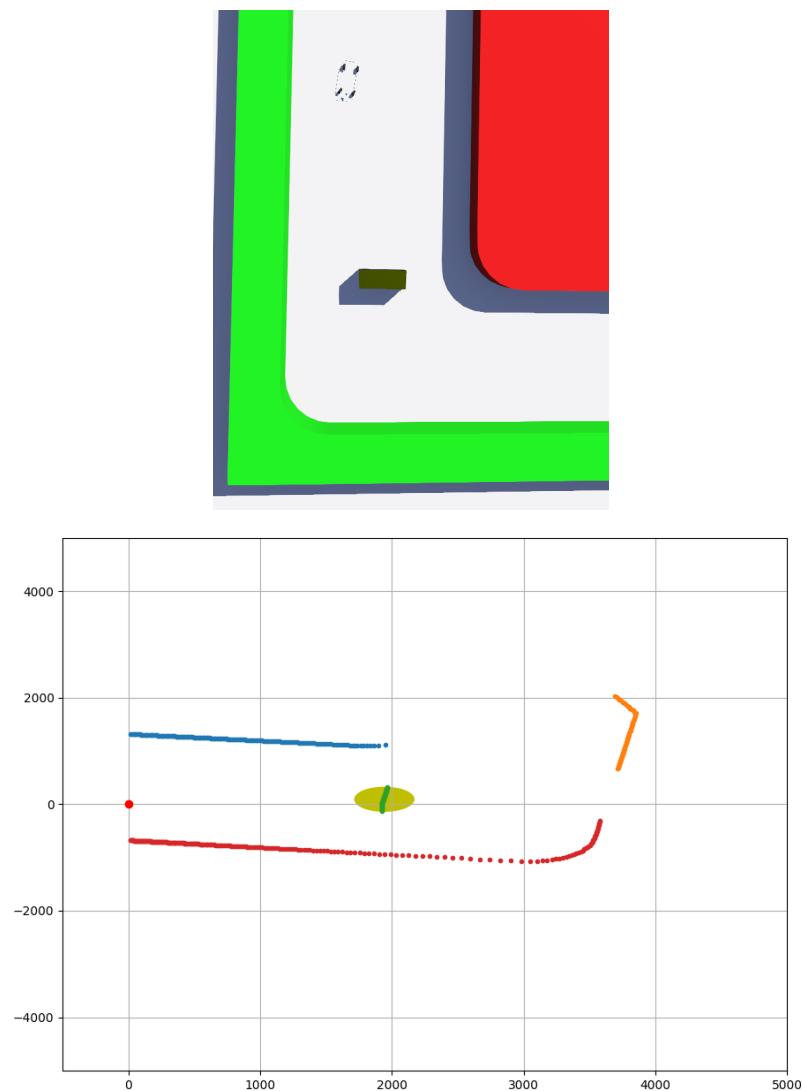


Figure : Exemple du clustering des points du lidar

Ainsi via ce clustering, il est possible de détecter les obstacles et les épingle.

Pour la détection d'obstacles, on procède de la manière suivante :

Pour chaque cluster, on définit un point médian au premier et au dernier point du cluster, et un cercle de diamètre la distance entre le premier point et le dernier point du cluster. Avec le point médian, on a une approximation de la localisation du cluster, et avec le cercle, on peut calculer une densité de points, c'est à dire que l'on va diviser la surface du cercle par le nombre de points du cluster. Ainsi, au-delà d'une certaine densité, on considère le cluster comme un obstacle et on va pouvoir mettre en place une stratégie de course alternative qui permettra de l'éviter. Cette méthode permet de différencier les murs qui sont, en général, des grands clusters avec beaucoup de points peu denses, et les obstacles qui possèdent moins de points mais dans une surface moins importante. On peut d'ailleurs voir sur la figure ci-dessus que le cluster vert est inclus dans un cercle jaune, ce qui signifie qu'il a bien été détecté comme un obstacle, contrairement aux murs qui l'entourent.



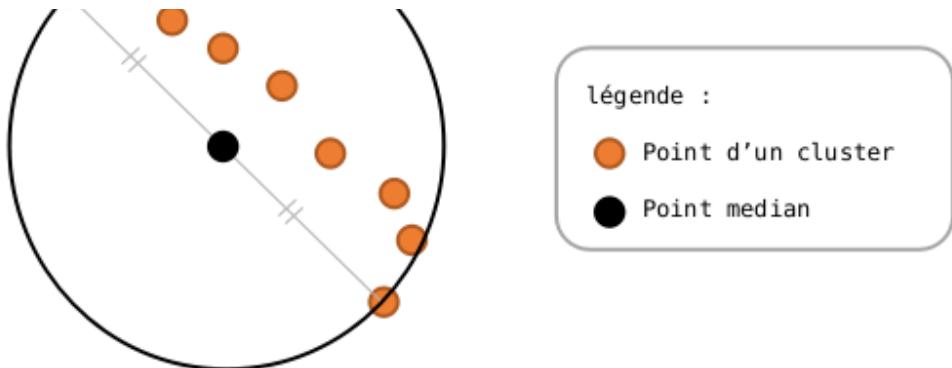


Figure : Principe de la détection d'obstacles

Pour ce qui est de la détection d'épingles, on définit 3 zones : la zone Est, en jaune sur la figure ci-dessous, la zone Ouest, en violet, et la zone Nord, en bleu.

Ensuite pour chaque cluster on va calculer la proportion de points du cluster se trouvant dans ces 3 zones.

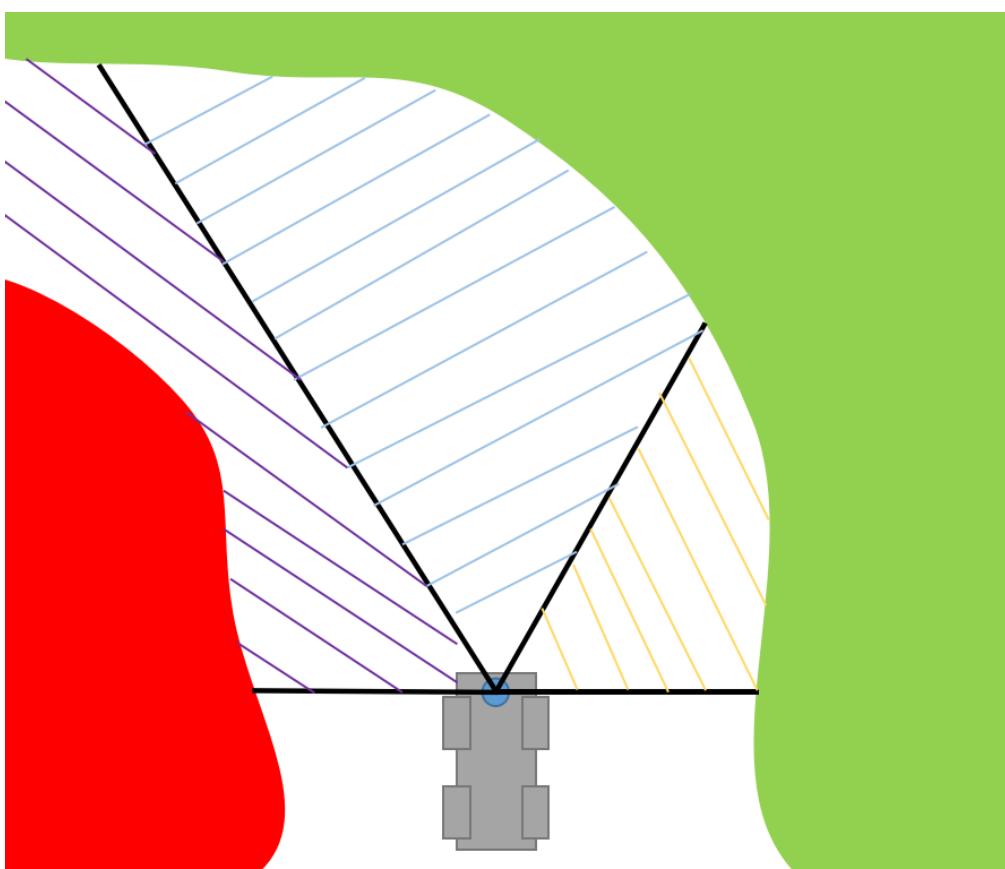


Figure : Définition des zones Est,Nord et Ouest

Ainsi, pour détecter une épingle ou un virage serré à gauche, on regarde si le cluster qui possède le plus de points dans la zone Nord possède aussi au moins 70% des points de la zone Ouest et 20% des points de la zone Est, comme cela pourrait être le cas dans la figure ci-dessus. Pour détecter les épingles à droite, on procède de la même manière en intervertissant Est et Ouest.

Par exemple, dans la figure suivante, on voit clairement que le cluster bleu représente

l'intégralité de la zone Nord et Ouest. De plus, il intègre une partie de la zone Est ce qui signifie qu'une épingle est imminente.

Ainsi, lorsqu'on détecte une épingle, on peut appliquer une stratégie de course différente.

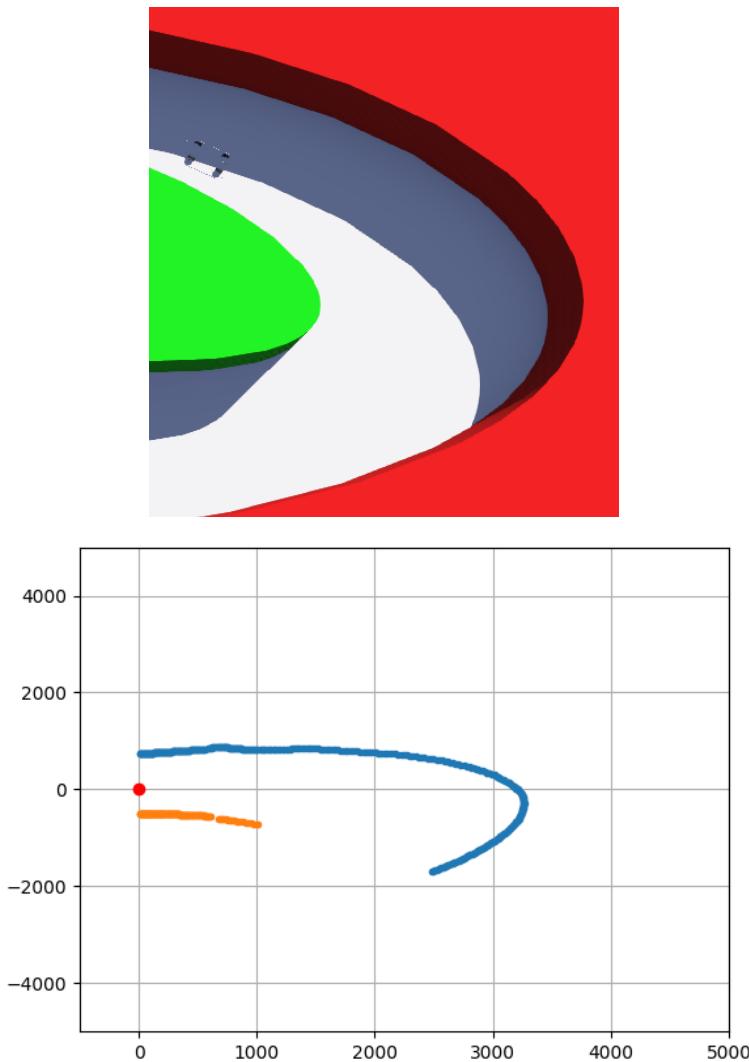


Figure : Exemple de détection d'une épingle

Cependant nous nous sommes assez vite rendu compte que la détection d'obstacles était défaillante si un obstacle se trouve dans l'épingle. En effet, les obstacles cassent en deux les clusters constitués par les murs. Ainsi, aucun cluster ne remplit les conditions citées au-dessus.

Il a donc été nécessaire de rendre plus robuste la détection d'épingles. Pour cela, lorsque l'on détecte un obstacle, on va calculer la distance entre le dernier point du cluster précédent et le premier point du cluster suivant. Ainsi, si cette distance est inférieure à un certain seuil, on fusionne les 3 clusters que sont l'obstacle, le cluster précédent et le cluster suivant. Par exemple, dans la figure suivante, les clusters bleu, vert et orange vont être fusionnés afin de pouvoir détecter l'épingle.

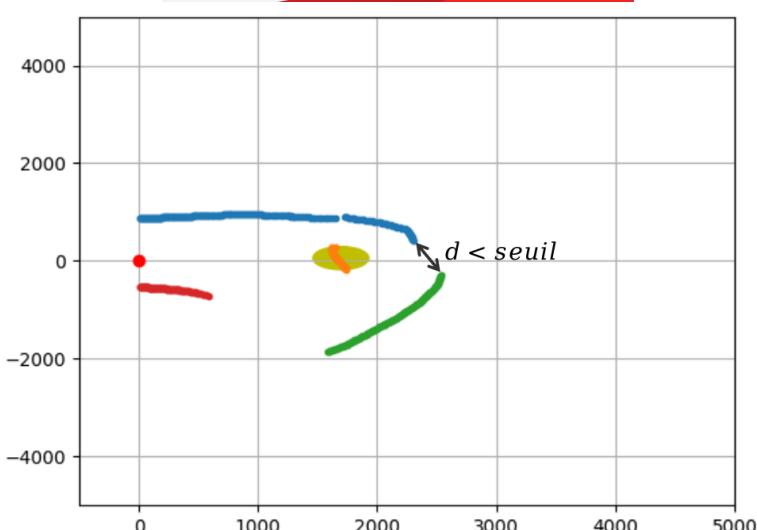
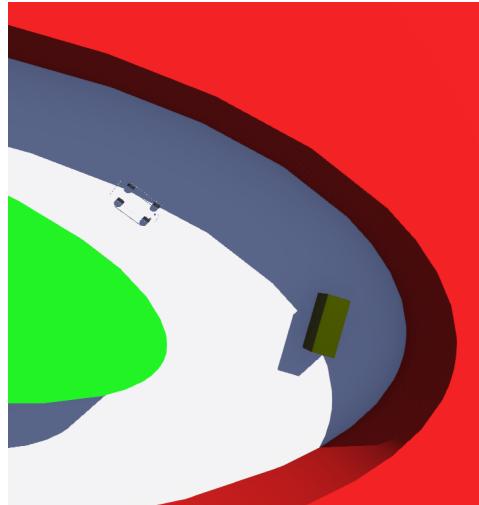


Figure : Exemple de fusion de 3 clusters dans le but de détecter une épingle

On peut tout de même noter que même avec cet ajout, notre technique reste défaillante dans certains cas. Par exemple, sur la figure suivante, on peut voir que si une voiture est dans un mur il se peut qu'il casse un cluster en deux et empêche la détection de l'épingle.

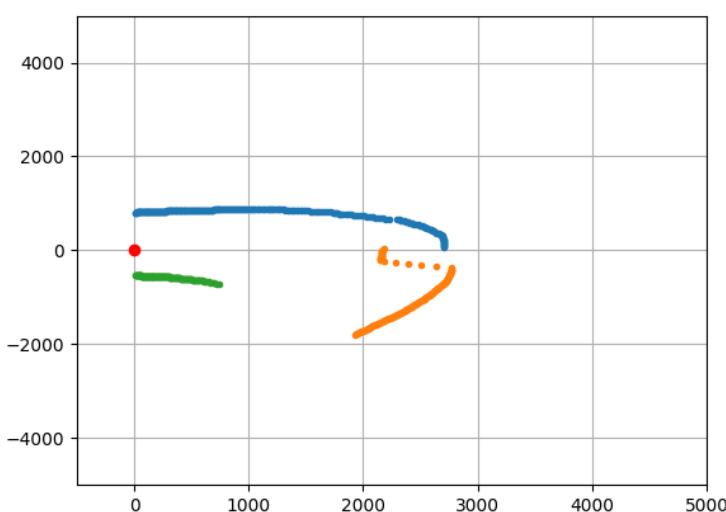


Figure : Exemple de défaillance de la détection d'épingles

Sur le même principe que précédemment, pour pallier ce problème, on va parcourir les différents clusters (qui ne sont pas des obstacles) et observer la distance entre un cluster et son suivant. Si cette distance est inférieure à un certain seuil, les deux clusters seront fusionnés.

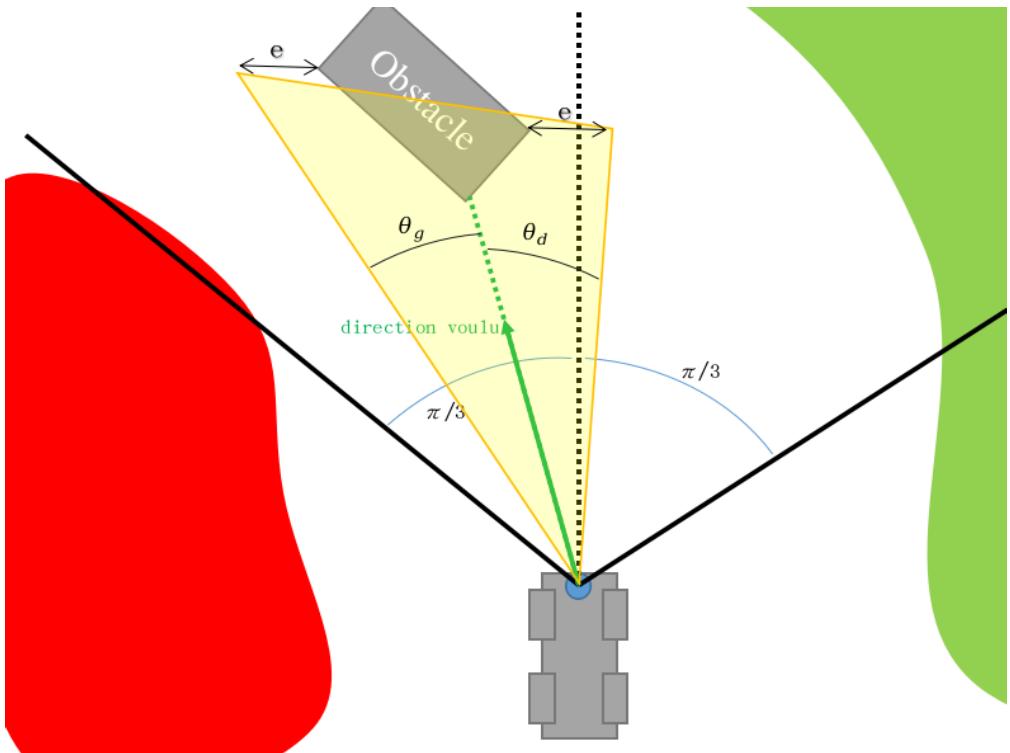
Stratégie de course en cas d'obstacle

Une fois que l'on a détecté les obstacles, il faut définir lorsqu'un obstacle est problématique ou non. Premièrement, on ne considère un obstacle que s'il est dans un arc de cercle de $\frac{2\pi}{3}$ à l'avant de la voiture et à une distance de moins de 3 mètres.

De plus, dans ce cas, on regarde si la direction voulue pointe vers cet obstacle (avec une certaine tolérance e). Si c'est le cas, on calcule les deux angles θ_d et θ_g .

Dans un premier temps, on remplace la direction voulue par le plus petit de ces deux angles. Seulement, cette stratégie mène parfois la voiture vers des impasses car il arrive que l'obstacle se trouve proche d'un mur et qu'il n'y ait pas assez de place pour que notre voiture passe.





Dans un second temps, nous avons alors décidé de calculer les distances à gauche et à droite de l'obstacle. La nouvelle stratégie d'évitement consiste à se diriger vers le côté de l'obstacle présentant la plus grande distance à un mur tant que l'obstacle est à moins de 3 mètres et à plus de 1.5 mètre de notre voiture. Lorsque l'obstacle est à moins de 1.5 m de notre voiture, on reste du même côté de l'obstacle et on vise le point médian entre l'obstacle et le mur.

Stratégie de course en cas d'épingle

En cas de détection d'épingle, la stratégie est assez simple : on braque au maximum dans le sens voulu, sauf en cas d'obstacle.

Nous pourrons noter que si on braque simplement, il est fréquent que la voiture frappe les murs intérieurs de l'épingle. Il est donc nécessaire de contrôler la distance intérieure à notre voiture. Pour cela, on observe les distances à l'intérieur de la voiture et si une des distances est inférieure à un certain seuil, la consigne de l'angle de braquage change pour que la voiture aille droit. Cela permet d'éviter de nombreux chocs avec les intérieurs des virages.

Conclusion

Pour conclure à propos de cette partie, cette stratégie de course s'est avérée assez efficace et fiable. En effet, sur le circuit de la course sans obstacle, la voiture peut réaliser plusieurs tours de piste sans se heurter à un mur. Cependant, la détection d'obstacles montre quelques lacunes. Elle est assez fiable pour des obstacles fixes positionnés en milieu de piste. Cependant, pour des obstacles collés au mur, le clustering considère cet obstacle comme la continuité du mur, la stratégie d'évitement n'est donc pas déclenchée. De plus, la stratégie d'évitement ne semblait pas être très efficace face à des obstacles

mouvants comme des voitures qui oscillaient sur la piste. La détection d'épingle présente aussi encore quelques défaillances lorsque beaucoup d'obstacles étaient présent dans l'épingle. Pour améliorer cette stratégie, il serait important d'arriver à détecter les voitures collées à des murs comme des obstacles afin de les éviter de manière efficace.

Conclusion générale

Ce TER a été très intéressant. L'objectif de la course nous a permis de nous focaliser sur le projet pour obtenir des résultats concrets. Nous nous sommes chacun focalisés sur un domaine particulier, ce qui nous permettait de travailler de notre côté et de pouvoir mettre en commun lors des réunions. Les solutions mises en œuvre ont pour la plupart été fonctionnelles, et quand ça n'a pas été le cas, nous avons pu trouver des solutions aux problématiques posées.

Le passage de la simulation au système réel a été plus ou moins aisée : la détection d'épingle a par exemple fonctionné directement alors que ça n'a pas été le cas de l'évitement d'obstacle.

Nous n'avons pas pu aller au bout du projet et implémenter l'IA sur la voiture, mais les notions d'asservissement, de traitement du signal et d'informatique industrielle abordées tout au long de l'année ont été très enrichissantes. Nous avons mis en place une base solide avec une voiture capable de tourner à bonne vitesse sur n'importe quelle forme de circuit.

Les suites du projet reposent sur l'amélioration de l'algorithme d'évitement et l'implémentation de l'IA.

Bonus



Figure : Rien à voir avec le TER mais ces gens s'appellent Ackermann et quand on cherche le modèle d'Ackermann sur Google on tombe sur eux, leur plus grand fait d'arme vous me demanderez ?

C'est une grande famille de complotistes anti masque, anti confinement, ils sont bien marrant je conseille !

Références

- [1] Wenshuai Zhao and Jorge Peña Queralta and Tomi Westerlund (2020).
Sim-to-Real Transfer in Deep Reinforcement Learning for Robotics: a Survey
- [2] Thomas Chaffre and Julien Moras and Adrien Chan-Hon-Tong and Julien Marzat (2020).
Sim-to-Real Transfer with Incremental Environment Complexity for Reinforcement Learning of Depth-Based Robot Navigation