Projet ANDROIDE

Étude de Trajectoires d'un Robot Roulant Autonome & Visualisation

Université Paris 6 : M1 Semestre 2

Luka LAVAL & Etienne PENAULT & Ruizheng XU

Nous tenons d'abord à remercier toutes les personnes qui ont contribué au succès de ce projet.

Nous voulons remercier évidemment notre encadrant Jean-Michel ILIE ainsi que son collaborateur Farouk VALLETTE.

Nous voulons remercier également Nell FLAHARTY et le deuxième groupe de projet : Ethan ABITBOL, Elias BENDJABALLAH, Jules CASSAN, et Jérémy DUFOURMANTELLE.

Merci !

Table des matières

1. Introduction **4**

2. Mise en context **4**
2.A. Etat de l'art . . . 4
2.B. Context du projet . . . 5
2.B.a. Architecture du projet E-HoA par ROS . . . 5
2.B.b. Robot réel (Matériel) . . . 6
2.B.c. Simulation (*Gazebo*) . . . 7

3. Contribution **8**
3.A. Etude des sections de route . . . 8
3.A.a. Courbes de Bézier . . . 8
3.A.b. Classification . . . 9
3.B. Développement d'un outil de visualisation et de test . . . 13
3.B.a. Interface et fonctionnalités . . . 13
3.B.b. Interaction entre le Visualiseur et ROS . . . 18
3.B.c. Simulation avec *Gazebo* . . . 20
3.C. Evaluation de la méthode de classification . . . 22
3.C.a. Protocole de validation . . . 23
3.C.b. Expérimentation . . . 23
3.C.c. Etude des résultats . . . 24

4. Conclusion **26**
4.A. Mise en perspective du travail . . . 26
4.B. Pour aller plus loin . . . 26

5. Bibliographie **28**

1. Introduction

Le sujet des véhicules autonomes prend de plus en plus de place depuis plusieurs années. Un chercheur du LIP6, Jean-Michel Ilié, développe un robot roulant intelligent dont on peut faire l'analogie avec un véhicule terrestre autonome. Celui-ci a été réalisé dans le cadre du projet E-HoA (Embedded Higher Order Agent) qui se base sur le principe du Belief Desire Intention. Avec cette architecture multi-processus, pour prendre des décisions, le robot se base sur ses connaissances, prend en compte son environnement pour réagir aux changements et s'adapter à l'inattendu en temps réel.

Par notre projet, nous voulons qu'un véhicule rencontrant un nouveau virage puisse directement déterminer sa difficulté pour adapter son comportement dans le but de poursuivre son chemin. Aux vues du manque de données, nous adoptons une approche géométrique pour déterminer cette difficulté à partir de trois points caractéristiques du virage pour lui attribuer une classe. Ainsi, le véhicule autonome pourra se référer à ses expériences passées ou celle d'autres véhicules face à un virage de même classe pour adapter son comportement. Une fois la recherche de classification assez poussée, nous avons développé un outil de visualisation et de test pour évaluer la justesse de la classification.

Après une mise en contexte globale, nous vous présenterons plus en détails notre travail dans la section contribution. Cette section se découpe en trois parties: d'abord l'étude des sections de routes puis le développement de l'outil mentionné plus haut et enfin l'évaluation de notre méthode de classification. Ceci avant de conclure avec une mise en perspective du projet et des idées d'ouvertures.

2. Mise en context

2.A. Etat de l'art

Aujourd'hui, maintes approches ont été abordées dans le domaine des robots roulant et plus particulièrement à propos de la conduite. Étant donné un sujet aussi vaste et complexe, il a déjà été abordé selon des angles différents.

Comme nous le montre l'article [8], la recherche par les courbes de Bézier s'avère convaincante pour représenter des virages, et plus particulièrement pour une génération de trajectoire à la volée lorsque le robot est en train de rouler (par division successive des courbes). Cependant d'autres approches par clustering ont également donné des bons résultats. Comme en témoigne l'article [6], la classification de type de route est quelque chose s'avérant décent et qui à déjà fait ses preuves.

Ce projet a pour but d'aborder une approche nouvelle vis à vis des méthodes déjà explorées afin d'en tirer les bénéfices: il s'agira de coupler la notion de courbe de Bézier à la classification.

2.B. Context du projet

2.B.a. Architecture du projet E-HoA par ROS

L'architecture du projet E-HoA est basée sur ROS (Robot Operating System). Ce dernier est un système d'exploitation open source ayant pour but de faciliter le développement d'applications autour des robots.

ROS est parfaitement adapté au robot sur lequel nous travaillons (Turtlebot 3 Burger). Ce système a pour but de faciliter la tâche du développeur par son architecture décomposée sous forme de nœuds ainsi que par des communications à travers des topics auxquels nous devons souscrire pour la communication inter-nœuds.

Architecture visée pour E-HoA

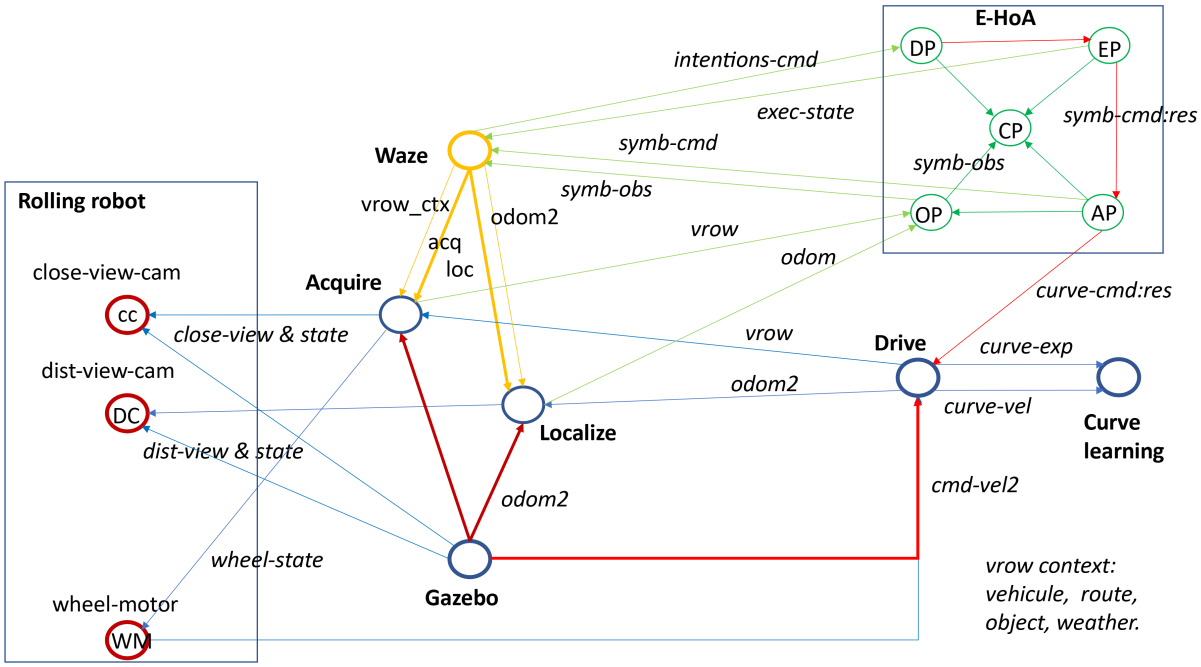

Figure 1: Architecture ROS d'E-HoA souhaitée

Sur la Figure 1, notre groupe se charge du développement la partie **Jaune**. Notre but au sein des communications ROS est d'interagir avec le nœud *Drive* (de l'autre groupe encadré par M. ILIE), par le biai du nœud *Localize*.

Architecture actuelle de notre programme

L'architecture précédente faisait référence à l'architecture ROS finale d'E-HoA , cependant, étant en phase de développement, nous nous devons de créer quelque chose de fonctionel et de modulable avant de merge notre travail au projet final.

Actuellement, nous ne passons pas par le nœud *Localize* car il n'est pas encore implémenté. Cependant, nous récupérons les données sérialisées (par le nœud *Odometry* à l'aide de *get_pos_ and_twist* dont nous parlerons plus tard), ce qui ne posera aucun soucis par le futur étant donné les normes établies pour le nœud *Localize*. Le passage d'un nœud à l'autre devrais être sans conflits.

Voici l'architecture actuelle de notre application:

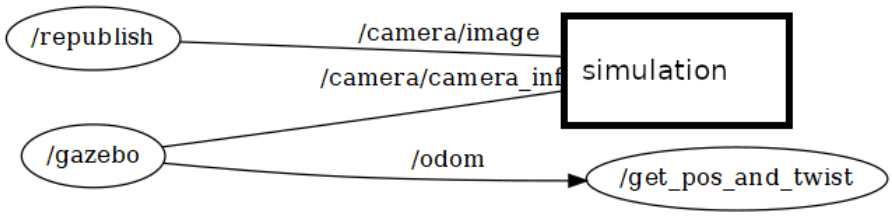

Figure 2: Notre architecture ROS

Remarque: Il est important de noter que nous avons beaucoup échangé afin de bien comprendre son architecture pour s'assurer de la compatibilité de notre travail.

2.B.b. Robot réel (Matériel)

Le robot étudié est le TurtleBot 3 Burger, qui est un robot mobile, programmable et fortement personnalisable basé sur ROS. Initialement, La technologie de base qui est utilisée par ces fonctionnalités est **SLAM** (Simultaneous Localization And Mapping), **Navigation** et **Manipulation**, et ainsi il est capable d'effectuer plusieurs tâches comme la conduite autonome (Navigation), construire une carte de l'environnement (SLAM), manipuler des objets en intégrant un manipulateur (Manipulation), etc...

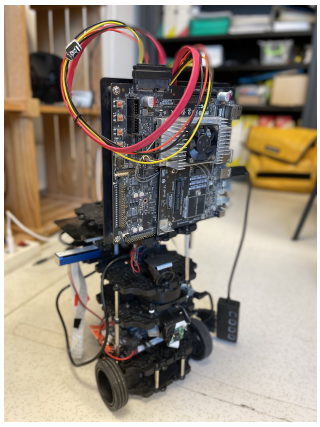

Figure 3: TurtleBot3 Burger

Pour le moment, notre robot possède les composants suivant :

- Un LiDAR
- Une carte Rasberry Pi 4
- Une carte OpenCR (contrôle les roues)
- Une batterie
- Un caméra branchée en USB, qui est utilisée pour la détection de codes ArUco (pour donner des ordres au robots)
- Une carte Nvidia Jetson Tx2 embarquée

- Un raspicam, qui capture une image (trapèze) de la route
- Un stick Movidius, qui offre une capacité de développement de réseaux neuronaux dans la reconnaissance d'image (l'image trapèze capturé par la raspicam)

2.B.c. Simulation (*Gazebo*)

Des packages ROS sont disponibles pour simuler le turtlebot directement sur le logiciel de simulation *Gazebo*. Celui-ci permet de simuler avec une grande précision un environnement (forces de frottements, gravité...) avec la modélisation fidèle du véhicule Turtlebot3 est de ses caractéristiques.

ROS fournit de plus le package *Autorace*. Celui-ci permet de mettre en place l'environnement dans *Gazebo* et de simuler une conduite autonome du véhicule avec leur algorithme *Autorace* (image ci-contre). Les nœuds ROS mis en jeux lors de la simulation sur *Gazebo* sont les mêmes que pour le robot réel et la procédure de lancement est très similaire. Ceci facilite grandement les tests pour ne pas avoir à les faire systématiquement sur robot réel.

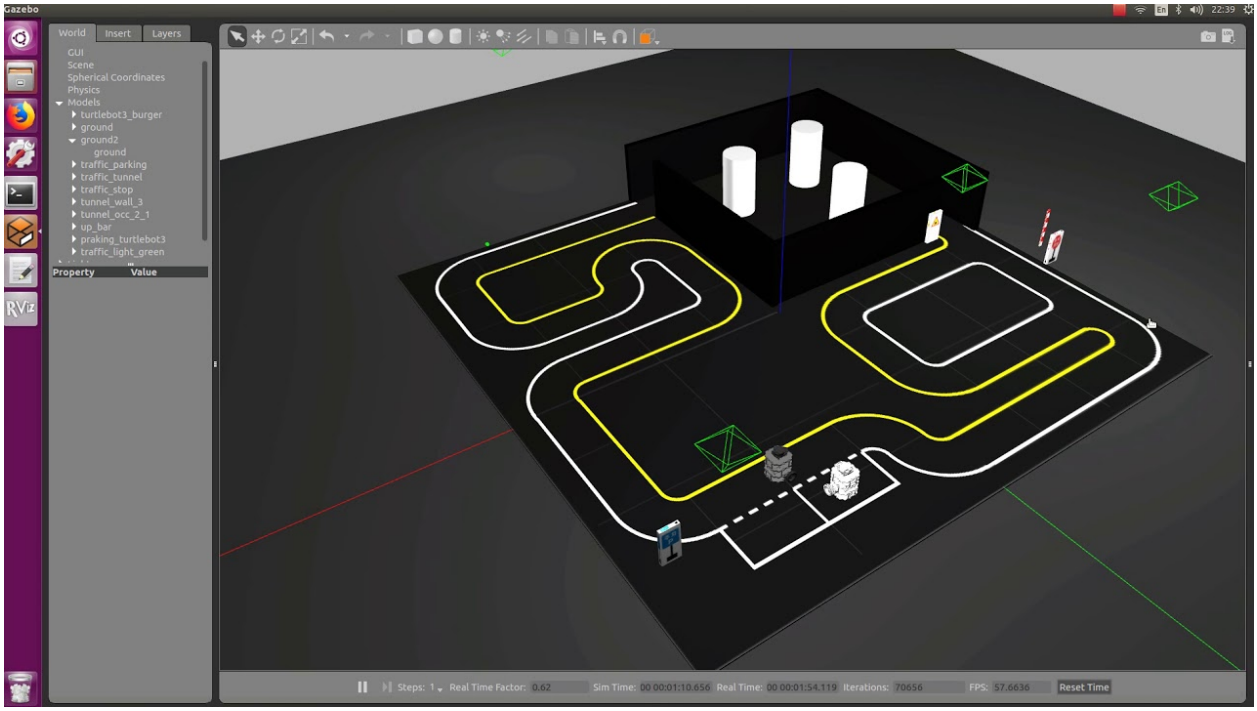

Figure 4: Interface de *Gazebo* (*Autorace*)

3. Contribution

3.A. Etude des sections de route

3.A.a. Courbes de Bézier

Les courbes de Bézier ont à l'origine été développées pour la conception de carrosserie de voitures. Aujourd'hui elles sont très répandues dans de nombreux domaines souvent liés à la synthèse d'images. Une courbe de Bézier est caractérisée par un ensemble de points appelés points de contrôle. Le nombre de points associés à la courbe lui donne un ordre : deux points pour une courbe linéaire, trois points pour une quadratique, quatre pour une cubique... Dans ce projet, nous n'utilisons que des courbes de Bézier quadratique donc caractérisées par trois points.

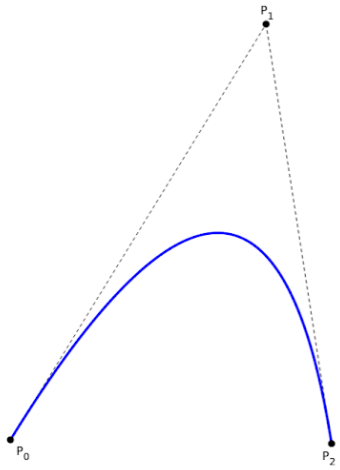

Figure 5: Courbe de Bézier quadratique

Soient P_0, P_1, P_2 les trois points de contrôle d'une courbe de Bézier B. Étant donné $t \in [0, 1]$, la courbe se définit par:

$$B(t) = P_1 + (1-t)^2 * (P_0 - P_1) + t^2 * (P_2 - P_1)$$

La dérivée d'une courbe de Bézier d'ordre n donne une courbe de Bézier d'ordre $n-1$. Dans notre cas, on a:

$$B'(t) = 2 * (1-t) * (P_1 - P_0) + 2 * t * (P_2 - P_1)$$

et donc:

$$B''(t) = 2 * (P_2 - 2 * P_1 + P_0)$$

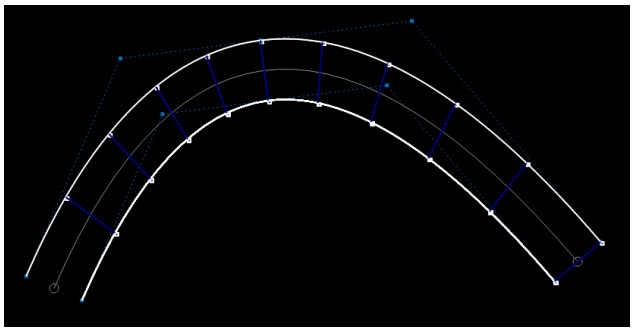

Figure 6: Route générée par le Road Editor

Pour notre étude, nous assimilons un virage à une courbe de Bézier quadratique. Ainsi, une route (caractérisée par une suite de virages) est représentée par un ensemble de courbes de Bézier quadratique. Cette approximation nous paraît réaliste pour la nature même de ces courbes (pas d'angle "abrupt" par exemple). L'utilisation des courbes de Bézier nous permet de nous appuyer sur le travail de Nell Flaharty. Dans le cadre de son stage, Nell a été amené à tra-

vailler sur le Road Editor, un outil permettant la construction de routes à partir de courbes de Bézier en tenant compte des critères de différence d'angles. Son travail se base notamment sur l'article [7]. Le Road Editor permet à son utilisateur de tracer une route dynamiquement en déplaçant les points de contrôles de la courbe de Bézier la représentant. Il renvoie ensuite un fichier au format .json représentant cette route (largeur courbure etc...) caractérisée par deux listes de points calculés à partir des trois points de contrôle. Nous détaillons dans la partie **Développement** d'un outil de visualisation et de test leur utilisation.

Jusqu'ici, nous avons donc encadré la notion de virage, et par extension de route, que nous utilisons dans tout le projet.

3.A.b. Classification

Dans le cadre global du projet E-HoA, nous cherchons à adapter le comportement du véhicule en fonction de son environnement. Ici, face à un nouveau virage, le véhicule doit pouvoir déterminer le comportement à adopter pour optimiser sa trajectoire. C'est pourquoi nous cherchons à créer des classes de virages qui regroupent des virages ayant des propriétés communes. De cette manière, le véhicule autonome pourra se référer à ses expériences (ou celles d'autres véhicules similaires ou non) pour aborder un nouveau virage de même classe de façon optimale.

Dans cette partie, nous allons détailler notre approche pour déterminer une classification de virages en fonction de leur difficulté. Naturellement la notion de difficulté est ici très instinctive voire abstraite. Nous l'avons défini comme suivant: sur une route réaliste, plus un virage est difficile, moins la vitesse du véhicule peut être importante pour parcourir entièrement ce virage sans en sortir. Pour cette classification, nous avons étudié plusieurs pistes dont certaines sont présentées ci-dessous.

Angle

De prime abord, avant de plonger dans les caractéristiques des courbes quadratiques de Bézier, nous avons fait le rapprochement entre l'angle que forment les trois points de contrôles d'une courbe de Bézier par rapport à sa dite classe pour considérer l'angle formé par les trois points de contrôles comme classe.

Le calcul de l'angle se fait de la manière suivante:

$$\delta(A, B, C) = \arccos \frac{(A - B) \cdot (C - B)}{||A - B|| * ||C - B||}$$

Cette formule nous permet éventuellement d'étendre l'étude à des points en trois dimensions (x, y, z) même si notre étude reste strictement en deux dimensions (x, y). L'avantage de cette approche est que le calcul est très basique et il est facile de lier plusieurs types de virages entre eux selon la précision accordée au calcul de l'angle.

Afin de vérifier si l'angle est un bon critère (voir le seul ?) pour classifier nos courbes selon la difficulté avec laquelle le robot va les aborder, nous avons décidé dans un premier temps de générer une série de courbes étant toutes du même ordre de grandeur en termes d'angle (que forment les trois points de contrôle).

Une fois ce jeu de courbe créé, nous avons comparé les courbes afin de nous donner une intuition sur la validité de la potentielle classe:

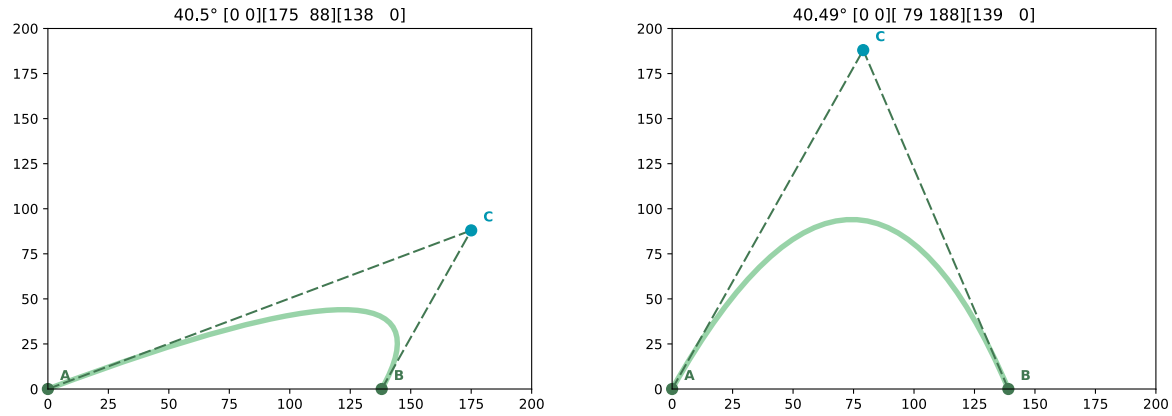

Figure 7: 2 courbes de Bézier ayant approximativement le même angle (40.5°)

Comme ces courbes en témoignent, le résultat est flagrant. L'angle seul n'est vraiment pas un critère fiable pour définir la difficulté d'une courbe. La figure ci-dessus nous prouve bel et bien que deux courbes ayant plus ou moins le même angle formé par ses points de contrôle ne garantit en rien une même difficulté pour un virage. La courbure de la courbe de gauche (40.5°) est bien plus difficile à aborder que celle de droite à (40.49°).

Remarque: Si nous considérons uniquement $C_x \in [A_x, B_x]$, alors nous aurions une approximation acceptable/réaliste de la difficulté par l'angle. Il est très possible d'envisager ce cas, il suffirait de proposer un sous-échantillonnage de la courbe de Bézier ne respectant pas le critère en deux sous-courbes de Bézier (toujours quadratiques) respectant ce critère.

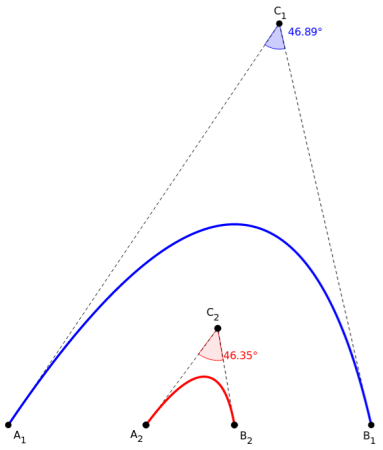

Figure 8: Problème d'échelle

De plus, à l'aide de Géogebra, nous nous sommes rendu compte que malgré l'apparence plus ou moins réaliste de cette classification, elle ne prend pas en compte l'échelle du virage qui elle a son importance. Par exemple, les deux courbes ci-contre ont le même angle formé par leurs trois points de contrôle. En revanche, le virage le plus petit est naturellement plus difficile à aborder que le plus grand pour un véhicule.

Il faut trouver une autre approche que celle de l'angle uniquement. Cependant nous gardons ce critère dans un coin de nos têtes afin de l'utiliser couplé à d'autres.

Etude géométrique des points de contrôle

La seconde approche envisagée consiste à étudier les liens purement géométriques des points de contrôles entre eux. L'idée consiste à classer un virage en fonction de ses vecteurs $\{\vec{i}, \vec{j}, \vec{k}\}$. La notion de vecteur permet de plus de prendre en compte quand $B_x \in [A_x, C_x]$.

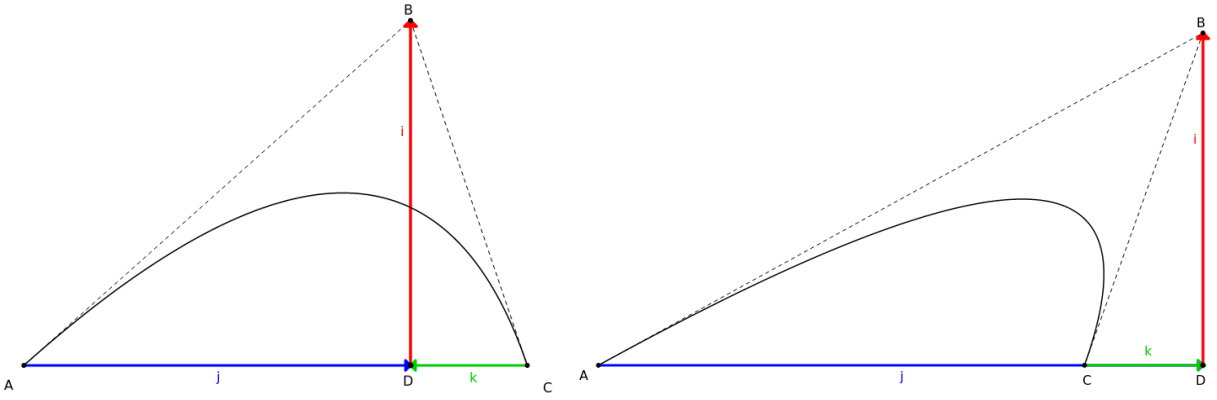

Figure 9: Analyse de la géometrie des deux cas de courbes

Observation 1: Pour les courbes respectant $B_x \in [A_x, C_x]$, on remarque que plus le vecteur $\vec{i}$ est petit, plus le virage est facile. Par exemple, pour un vecteur $\vec{i}$ de norme nulle, la courbe est une ligne droite. À l'inverse, pour les courbes ne respectant pas le critère, plus le vecteur $\vec{i}$ est petit, plus la courbe est difficile à suivre (virage très serré pour un vecteur de norme proche de 0).

Observation 2: Pour toutes les courbes (respectant ou non le critère), pour un vecteur $\vec{i}$ donné, plus le point D est éloigné du centre du segment $[A, C]$, plus la courbe est difficile.

Basé sur ces deux observations, une idée de formule vient naturellement pour classer une courbe respectant le critère:

$$||\vec{i}|| * ||\vec{MD}|| \text{ où } M \text{ est le milieu du segment } [A, C]$$

De la même manière, la formule naïve pour une courbe ne respectant pas le critère aurait été:

$$\frac{||\vec{MD}||}{||\vec{i}||} \text{ si } ||\vec{i}|| \neq 0 \text{ et } +\infty \text{ sinon}$$

Ces deux formules respectent les évolutions globales des observations 1 et 2. Cependant, leur découverte reste largement abstraite et elles ne sont pas robustes: la première formule score à 0 les courbes où $D = M$ alors qu'elles n'appartiennent pas à la classe de celles où $||\vec{i}|| = 0$ (ligne droite). Laisser ce critère reviendrait à assimiler toutes les courbes symétriques

par rapport au segment $[M, B]$ à des lignes droites ce qui est absurde. De plus, il est difficile de trouver des facteurs pour renforcer la cohérence entre les classes (c'est à dire favoriser $||\vec{MD}||$ par rapport à $||\vec{i}||$ ou l'inverse avec des facteurs multiplicateurs ; passer à une formule exponentielle plutôt que d'utiliser un produit ; ... bref, trouver un lien d'importance entre les deux observations). La recherche paraissait finalement trop brouillon pour être poursuivie dans le cadre de ce projet.

Courbure

Finalement, l'étude de la courbure de la courbe nous paraît être l'une des méthodes de classification la plus adéquate. La réflexion primaire pour trouver cette classification consistait à approcher le creux du virage (son sommet) par un cercle et retenir son rayon comme score du virage. Ceci nous permet de prendre en compte dans une même valeur, l'échelle du virage et son angle. Finalement il s'avère que c'est une approche très similaire au concept de courbure d'une courbe.

De manière intuitive, la courbure est la quantité par laquelle une courbe s'écarte d'une ligne droite en un point donné. Plus formellement, sur ce schéma, la courbure de C au point P a pour valeur $k = 1/R$.

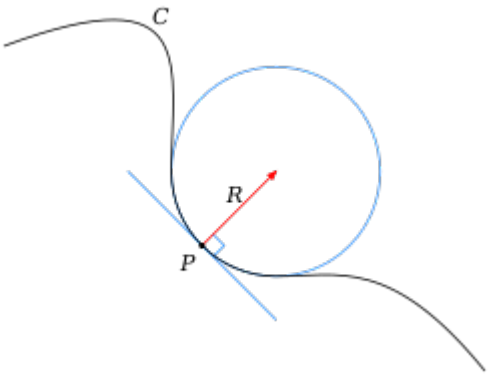

Figure 10: Illustration de la courbure

Ainsi, il faut pouvoir déterminer le creux du virage. Celui-ci est par exemple atteint en $t = 0.5$ pour une courbe symétrique. En pratique les courbes sont rarement symétriques, une manière de déterminer le creux de la courbe est détaillée dans l'article [7].

Soit $t \in [0, 1]$ tel que $B(t)$ est le creux de la courbe de Bézier quadratique B. On calcul la courbure en t de la manière suivante:

$$k = \frac{|\det(B'(t), B''(t))|}{||B'(t)||^3}$$

En pratique, nous avons utilisé un calcul plus direct utilisant les trois points de contrôles A, B et C :

$$k = \frac{|2 * det(B - A, C - B)|}{||C - 2 * B + A||^2}$$

Les scores associés aux courbes nous paraissent cohérents. Nous sélectionnons cette méthode de score et par extension de classification pour la suite.

Angle avec distance au sommet

Nous nous sommes rendu compte que la première approche d'étude de l'angle pour la classification restait une bonne piste lorsque la courbe respecte le critère $B_x \in [A_x, C_x]$ et en tenant compte de la distance au sommet de la courbe. Nous associons ainsi une échelle à l'angle.

Le calcul de l'angle est expliqué dans la partie **Angle** précédente. La distance au sommet de la courbe est approximée par la norme euclidienne entre le point de contrôle A (d'où arrive le véhicule) et le point du sommet de la courbe. Ceci équivaut à une distance entre les deux points à vol d'oiseau au lieu de suivre la courbe. L'approximation paraît largement acceptable lorsqu'on se restreint à des courbes quadratiques comme c'est le cas ici.

Soit d la distance entre le point de contrôle A et le sommet de la courbe S, et à l'angle $A\hat{B}C$ en degrés, le score de la courbe est calculé de la manière suivante:

$score = d * a$

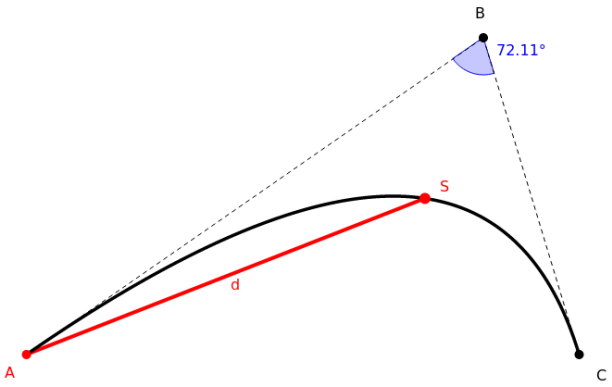

Figure 11: Distance au sommet

Intuitivement, on voit que le score augmente proportionnellement à l'angle et la distance initiale au sommet. Le fait de le prendre en compte est finalement relativement instinctif puisqu'il représente entre autres la distance au point le plus critique du virage c-a-d la distance de ralentissement théorique avant ce point critique. Cette formule de classification est aussi sélectionnée pour la suite.

3.B. Développement d'un outil de visualisation et de test

3.B.a. Interface et fonctionnalités

Visualisor

Le développement de cet outil de visualisation et de test est basé sur le langage Python, avec la librairie PyQt5 pour l'interface.

Création de la carte et des virages

Avant de rentrer dans l'implémentation de l'application RoadMap Visualisor, nous allons expliquer la génération des données nécessaires par les outils existants tels que WebMap Editor et Road Editor, et ces données seront par la suite importées par notre application pour

visualiser les différents virages.

Tout d'abord, la création d'une carte avec WebMap Editor, présentée sous forme de graphe comme ci-dessous.

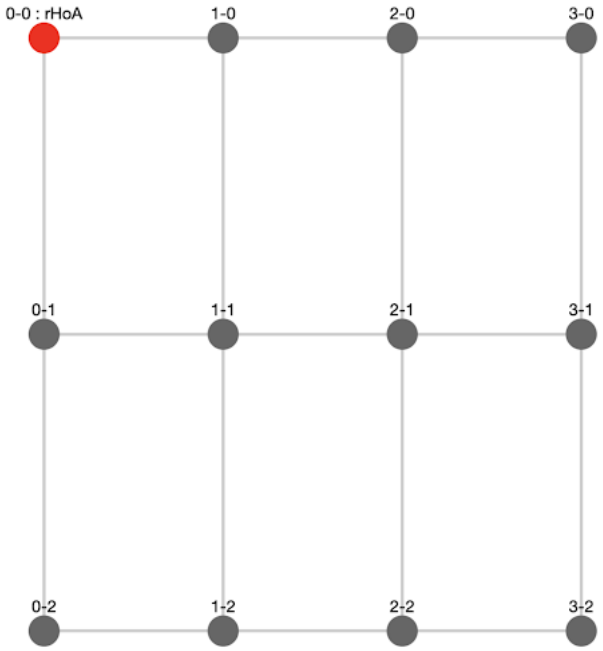

Figure 12: Carte du WebMap Editor

Les sommets représentent des intersections, qui peuvent avoir des éléments tels que le robot, la pharmacie, la station d'essence, etc... Et les arêtes représentent donc des sections avec un seul virage (un chemin dans le graphe est une suite de sections qui forment alors plusieurs virages, et un circuit est un chemin qui revient sur lui-même).
Ainsi, les données intéressantes que cette carte nous fournit sont l'emplacement des différents éléments sur la carte, et les intersections qui sont à l'extrémité de chaque section. Une fois la carte établie, on doit attribuer un virage à chaque section, on passe alors à l'outil Road Editor.

Road Editor nous permet de créer des routes comportant un ou plusieurs virages, et chaque virage est construit avec les courbes de Bézier. Nous nous intéressons seulement à des routes à un seul virage, et une fois cette route créée, l'éditeur nous renvoie un fichier .json avec les informations pertinentes suivantes :

- La liste des points sur les deux lignes, permettra à reconstruire la route
- Les points de contrôles

À ce stade, avec le fichier de la carte et les différents fichiers de la route, nous utilisons un script Python pour les fusionner, et on obtient un fichier *join.json*, qui fait le lien entre la carte et les virages associés à chaque section.

Visualisation des données

La première fonctionnalité principale de notre application est de visualiser la carte ainsi que les différentes sections. Au lancement, on importe le bon fichier, et on verra l'interface suivante :

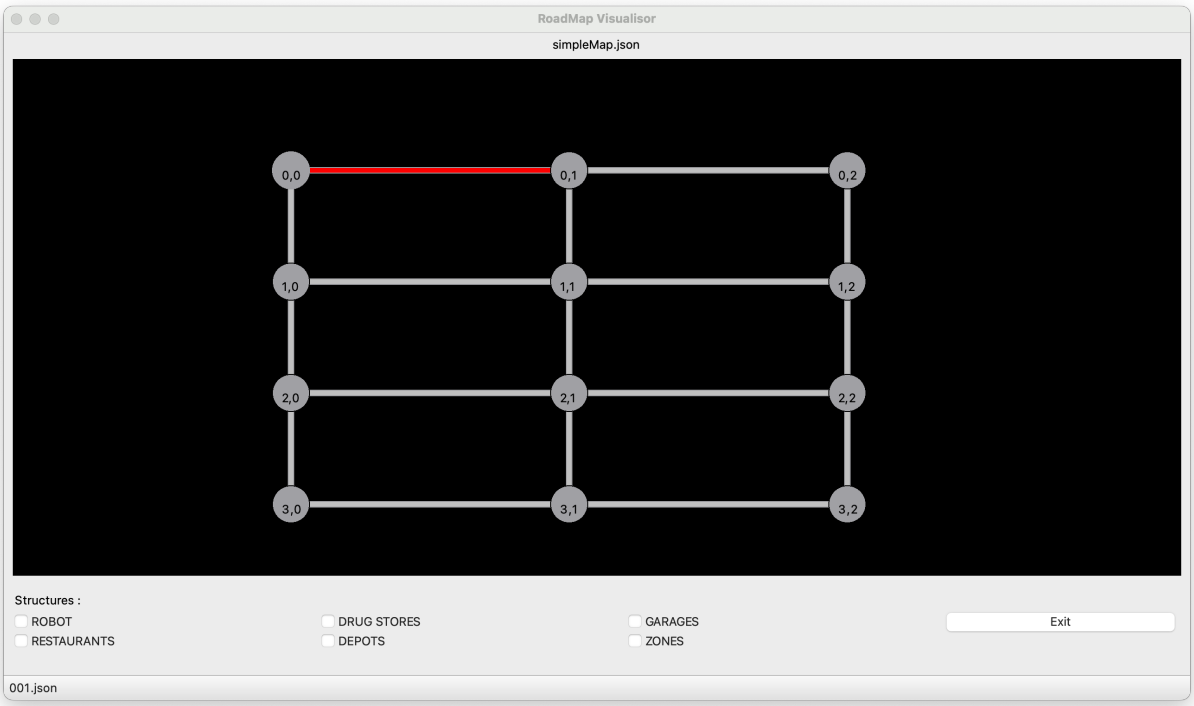

Figure 13: Carte sur le Visualisor

Sur cette interface, la carte est représentée de la même manière (sous forme d'un graphe) que précédemment pour faciliter l'utilisation entre les différents outils, et l'utilisateur a la possibilité de cocher les cases des différents éléments pour connaître leur emplacement dans les intersections. Et évidemment, en cliquant sur une section, le virage correspond sera affiché dans une nouvelle fenêtre :

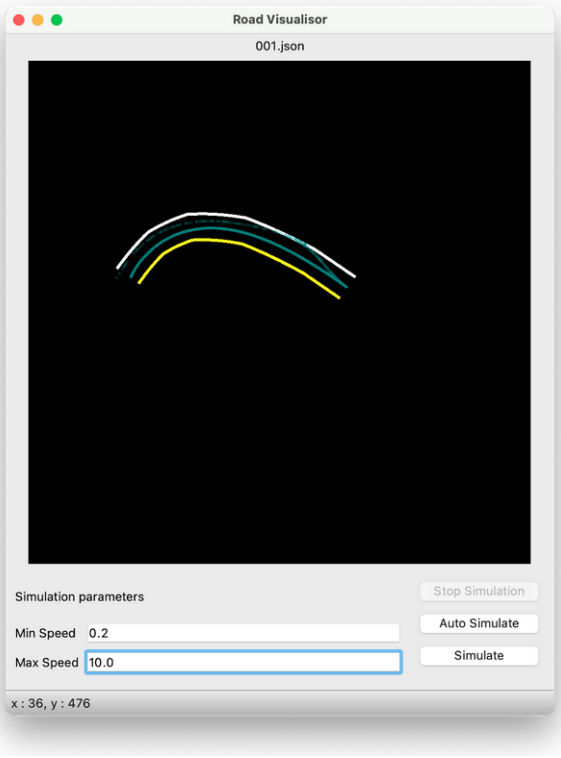

Figure 14: Section de route sur le Visualisor

À travers cette page de l'application, l'utilisateur pourra visualiser le virage avec la ligne jaune la bordure gauche et la ligne blanche la bordure droite de la route. De plus, les trajectoires bleues sont les historiques du robot, c'est-à-dire le chemin que le robot a effectué pendant les précédentes simulations.

Simulation d'une course

La deuxième fonctionnalité principale de l'application est la simulation d'course d'un robot TurtleBot 3 dans le virage correspondant, et c'est une étape essentielle pour récupérer une vitesse optimale pour ce virage.

Toujours sur la même interface qui affiche les virages, pour lancer une simulation, il suffit pour les utilisateurs d'appuyer sur le bouton "Simulate" ou "Auto Simulate", tandis que l'application va lancer le simulateur *Gazebo* et *Autorace* en processus, sans oublier le nœud qui récupère la position du robot dans le simulateur. Ensuite, le robot apparaît au début du virage, et va essayer d'arriver à la fin sous une vitesse donnée. Et tout au long de la simulation, l'application affiche également en temps réel sa trajectoire en laissant un point à l'arrière du robot à chaque déplacement, accompagnant de sa direction avec une flèche qui indique son orientation.

La condition d'arrêt d'une simulation est simple, lorsque le robot sort de la route pendant sa course, alors l'application mettra la simulation à sa fin. Pour cela, on construit un

polygone à partir de notre route qui est composé d'un ensemble de points, et on vérifie si le dernier point du trajectoire du robot est à l'intérieur du polygone.

Une fois la simulation terminée, il faut dans un premier temps vérifier si la simulation est réussie. La vérification consiste à calculer la distance entre :

- L'endroit où le robot s'est arrêté. Qui est le dernier point de la trajectoire.
- La fin de la route. Qui est calculé à partir de pj et pb, qui correspondent repectivement au dernier point de la liste des points pour tracer la bordure jaunes et la bordure blanches, et on cherche le point milieu des points pj et pb, avec la formule : $(\frac{pj_x + pb_x}{2}, \frac{pj_y + pb_y}{2})$.

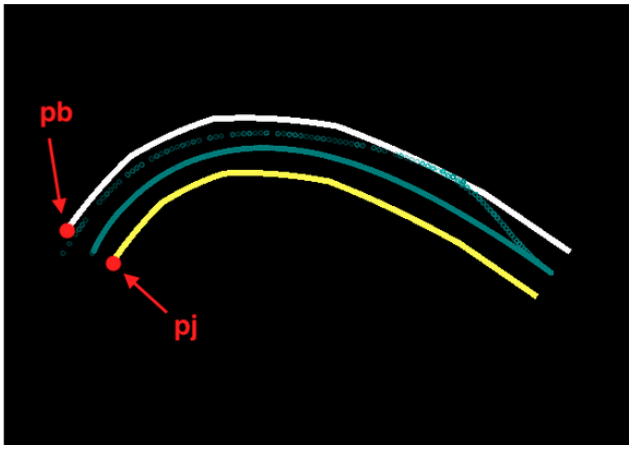

Figure 15: Point de réussite

Et on calcule la distance euclidienne des deux points avec : $\sqrt{(x2-x1)^2+(y2-y1)^2}$. Si cette distance est inférieure au seuil que l'on a déterminé, alors la simulation est considérée comme réussie, sinon échec.

Enfin, un fichier contenant la vitesse utilisée, la trajectoire du robot, et le résultat (succès ou échec) de cette simulation sera sauvegardé. Ce fichier est donc une expérience du robot sur cette route.

Remarque: Auto Simulate permet de faire plusieurs simulations à la chaîne avec un algorithme de recherche basé sur la recherche dichotomique pour trouver la vitesse optimale. Le détail du protocole est expliqué dans la partie du protocole d'évaluation.

Road Slicer

Le Road Slicer permet de faire le lien entre l'outil de visualisation et le robot réel. C'est un protocole qui permet de mettre à l'échelle du Turtlebot réel une route (au format json) passée en paramètres. Une fois à l'échelle, la route est découpée en fichiers .png au format

A4 prêts à être imprimés. Il suffit ensuite de fixer les feuilles imprimées dans le bon ordre sur le sol. Ceci couplé au visualiseur permet de suivre en direct les trajectoires réelles du TurtleBot, mais aussi de les garder en mémoire pour les analyser plus tard avec la fonctionnalité d'historique mise en place.

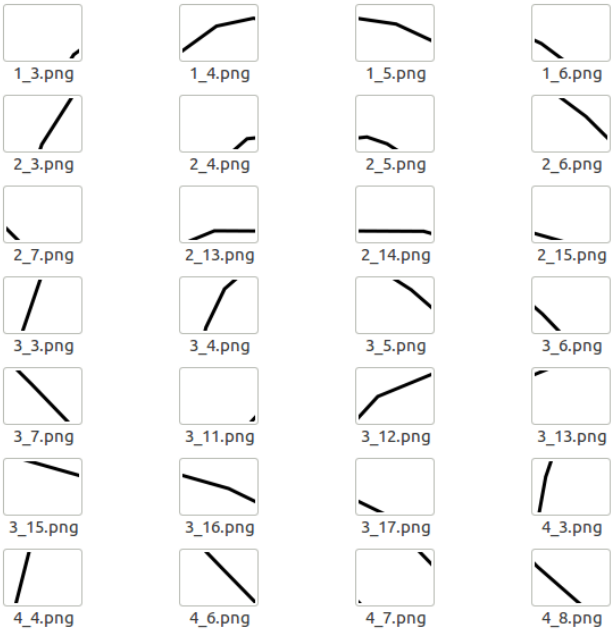

Figure 16: Exemple de route slicée à imprimer

3.B.b. Interaction entre le Visualiseur et ROS

Le cœur de cette section porte sur le lien entre l'interface d'un côté, et l'exécution des processus ROS de l'autre. Nous devons interagir avec ROS afin de récupérer des données précieuses lors de l'utilisation de l'interface ainsi que lui donner des instructions.

Comme évoqué précédemment, pour interagir avec ROS et son écosystème, nous devons adopter des normes pré-établies: Communiquer à l'aide de nœuds par des topiques auxquels nous devons souscrire.

Quelles informations sont utiles pour notre interface ?

Nous souhaitons représenter le robot se déplaçant en parallèle de la réalité (ou de la simulation dans notre cas): nous cherchons donc à recueillir les données spatiales et télémétriques du robot en temps réel.

Pour ce faire nous avons créer le nœud *get_pos_and_twist* qui souscrit au nœud ROS *Odometry* nous donnant des informations vitales sur ce que l'on cherchait à récupérer initialement:

- Sa position
- Son orientation

- Sa vitesses linéaires dans les directions x, y et z
- Sa vitesse angulaire dans les directions x, y et z

Une fois cela fait, nous réceptionnons de manière périodique les informations émises par ROS afin de les actualiser à la même fréquence dans notre programme gérant l'interface.

Remarque: Nous devons lancer ce processus en parallèle de l'interface étant donnée que la récupération d'information est bloquante. La procédure de récupération des données se fait dans *Pos.py*.

Que devons nous communiquer à ROS ?

Précédement, nous avons vu la phase de récupération des données. Cependant, nous devont également interagir avec ROS pour lui donner les instructions nécessaires afin de débuter la simulation.

Ne travaillant pas à même le robot physique, nous devons initialiser *Gazebo* par le biai de ROS pour créer le pont entre les deux. Une fois cela fais, il ne nous reste plus qu'à lancer la simulation du robot dans *Gazebo* par le package *Autorace*.

Cependant, *Autorace* demande lui aussi une configuration auprès de ROS au préalable: Il faut initialiser la simulation de la caméra du simulateur *Autorace* afin qu'il puisse voir les bordures de la route, ce qui nous fais une deuxième chose à éxécuter.
Enfin, nous pouvons lancer toujours par le biais de ROS la simulation d'*Autorace* pour faire bouger le robot !

Pour résumer, nous devons lancer en parallèle trois processus via ROS:

- *Gazebo*
- La caméra d'*Autorace*
- La simulation par *Autorace*

Voici un schéma sous forme de ligne temporelle afin de mieux comprendre le parallélisme des tâches précedemment expliquées:

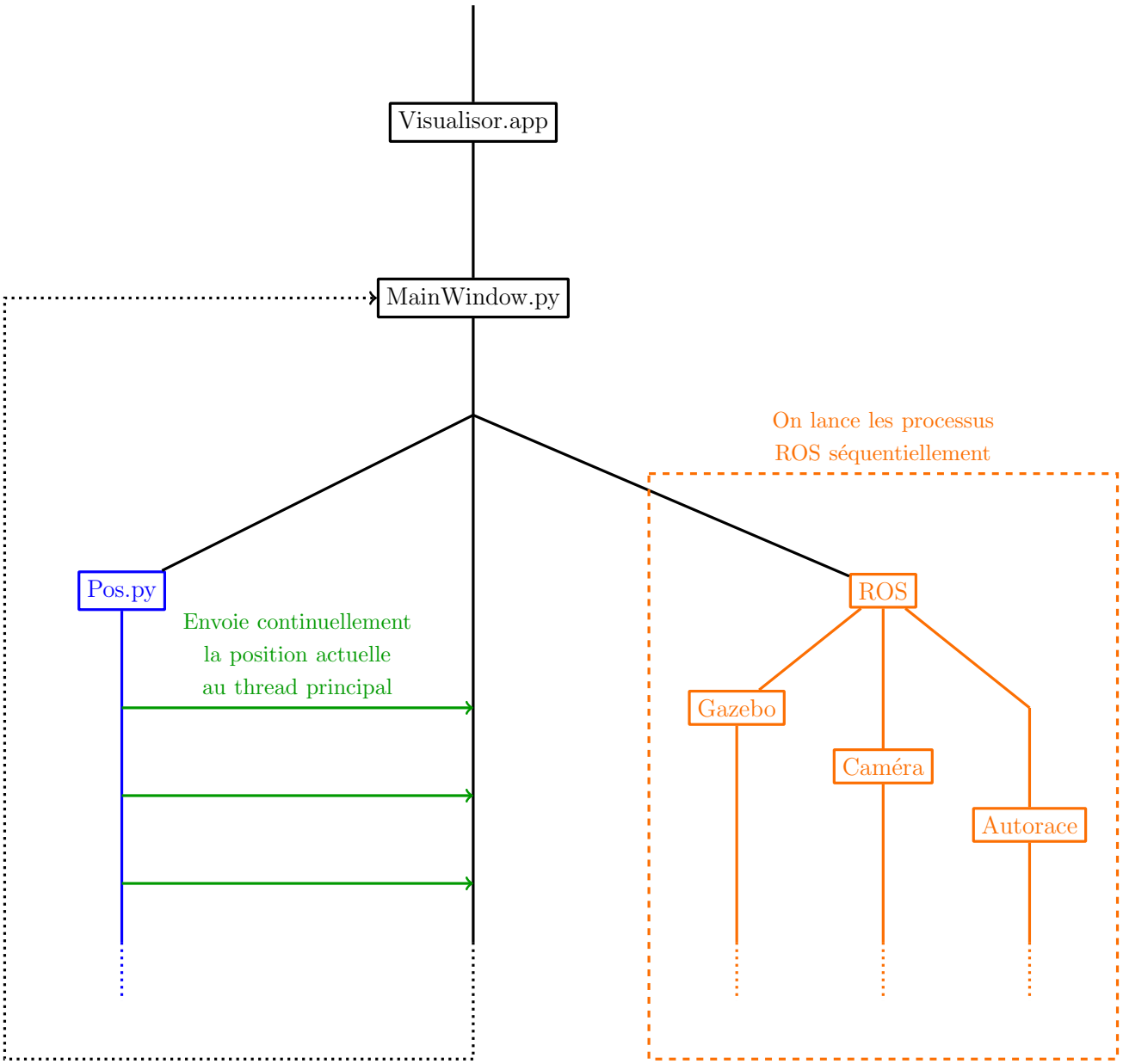

Figure 17: Éxecution et fonctionnement de notre application de visualisation

Remarque: Notons que l'invocation des processus ROS est modulable. En effet, nous appelons ici les processus basiques fournit par *ROS*, *Gazebo* et *Autorace*. Cependant nous pouvons tout à fais remplacer ces processus par ceux que nous créons. Cela a était volontairement pensé pour, afin de ne pas se limiter à la simulation par *Gazebo*, mais bien par la suite utiliser d'autres processus ROS. Par cette méthodologie, il sera très simple de fusionner notre travail avec celui de l'équipe Drive (comme le simulateur par exemple).

3.B.c. Simulation avec *Gazebo*

Pour ne pas à avoir à mettre en place le TurtleBot réel pour chaque test, nous avons utilisé une simulation du TurtleBot sur le logiciel de simulation *Gazebo*. *Gazebo* permet de créer un environnement virtuel réaliste pour étudier le Turtlebot (beaucoup d'outils sont mis

à disposition, la gravité et les autres forces qui s'appliquent au véhicule sont évidemment simulées par le logiciel). L'idée ici est de simuler le TurtleBot sur *Gazebo* dans certaines conditions et de récupérer les résultats du comportement du TurtleBot.

Environnement de simulation

On veut évidemment pouvoir adapter l'environnement de simulation à nos besoins. Pour cela on va influer sur différents fichiers directement dans les fichiers des packages ROS.

D'abord la route: à partir d'une route .json générée par le Road Editor, nous créons un .png ce celle-ci en respectant le fond noir, la ligne blanche et la ligne jaune. Ce fichier est remplacé directement dans le package *turtlebot3_simulation* de ROS. Les dimensions sont évidemment à adapter pour respecter l'échelle du robot dans la simulation. Ce changement se fait dynamiquement sur l'interface à travers l'appel de $Visualisor/map/setup\_map.py$ prenant un virage .json en paramètre. Ce script modifie directement le fichier concerné du package *turtlebot3_simulation*.

De plus, il est important d'avoir une luminosité adaptée à la caméra simulée du Turtlebot. Pour cela, on ajoute quelques spots de lumières au-dessus du plateau de simulation. Même si le plateau de simulation de *Gazebo* n'est pas amené à être visualisé dans l'application finale, les lumières sont très importantes pour une bonne reconnaissance des couleurs des traits de la route par la caméra du robot. Toutes ces modifications sont visibles dans le fichier *turtlebot3_autorace.world* du dossier $.stuff$.

Gazebo se charge des différentes forces comme la gravité ou les forces de frottements entre le véhicule et le sol. Ici, aucun paramètres changés de notre part pour rester le plus proche possible du robot réel.

Gestion de la vitesse

Nous voulons aussi pouvoir agir sur la vitesse du véhicule simulé. Pour cela, rendez-vous directement dans le nœud *control_lane* du package *turtlebot3_autorace_control*. On transforme la formule originale d'*Autorace* :

```
min(self.MAX_VEL*((1-abs(error)/500)**2.2),0.2)
```

en une nouvelle formule plus flexible nous permettant d'agir sur la vitesse du véhicule

```
max(min(self.MAX_VEL*((1-abs(error)/500)**2.2),MAX_LIN),MIN_LIN)
```

Avec ces quelques transformations, nous nous permettons de modifier dynamiquement les valeurs de MAX_LIN et MIN_LIN encadrant ainsi la vitesse minimale et maximale imposée directement depuis notre interface. Ceci nous permet notamment de garder la gestion automatique des accélérations les décélérations linéaires qui restent gérées par l'algorithme d'*Autorace*.

Positionnement initial du Turtlebot

Le placement du robot est primordial. Naturellement, on souhaite que le robot soit au début du virage orienté dans la bonne direction. Pour cela on place le robot aux coordonnées du premier point de contrôle du virage orienté vers le milieu des deux prochains points de route jaune et droit comme illustré ci-contre. Ce changement se fait dynamiquement sur l'interface à travers l'appel de *Visualisor/pose/setup_pose.py* prenant un virage .json en paramètre. Ce script modifie directement le fichier *.launch* concerné.

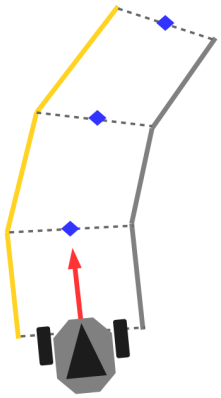

Figure 18: Position initiale

Modifications annexes

Il ne reste plus qu'à masquer l'interface de *Gazebo* pour n'en récupérer que les données coordonnées du robot fournies de manière régulière. L'accélération de la simulation n'est malheureusement pas évidente à mettre en place sans ordinateur très puissant, nous nous limitons donc à une simulation dont la vitesse est proche de la vitesse réelle supposée. Notre travail avec *Gazebo* s'arrête aux modifications mentionnées ci-dessus.

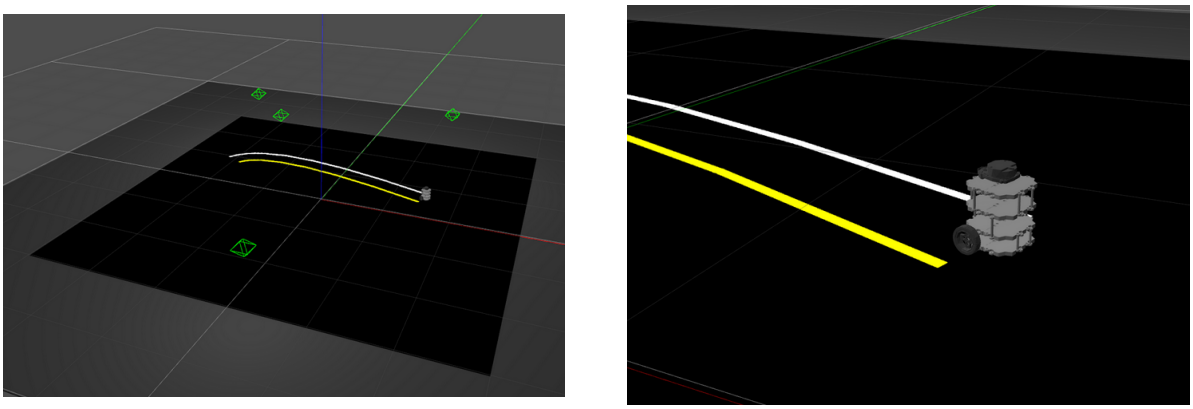

Figure 19: Simulation Turtlebot sur route (initialisation)

3.C. Evaluation de la méthode de classification

Dans cette partie nous cherchons à évaluer la classification retenue. Dans la suite nous détaillons le protocole d'évaluation mis en place, l'expérimentation à l'aide de l'outil développé et nous étudions les résultats obtenus.

3.C.a. Protocole de validation

À travers ce protocole, on cherche à évaluer les classifications de virages proposés. On cherche v_{opt} la vitesse "optimale" du véhicule dans un virage qui en réalité donnera une idée de la vitesse maximale pour prendre ce virage (donc pas réellement la vitesse optimale). Dans la suite, on considère qu'une vitesse pour un virage et un véhicule donnés est validée si et seulement si, en partant du premier point de contrôle, le véhicule atteint le point de réussite sans sortir du virage avec cette vitesse. De manière analogue, une vitesse est invalidée si le véhicule sort du virage avant d'atteindre le point de réussite. Ainsi nous définissons v_{opt} pour un virage donné, comme étant la vitesse minimale maximale validée du véhicule dans ce virage.

Pour déterminer v_{opt} pour un virage donné, on va lancer plusieurs fois la simulation (à l'aide de l'outil développé précédemment) sur ce virage avec des vitesses différentes. Nous n'influons ici que sur la vitesse minimale imposée (avec une borne très haute pour la vitesse maximale). La recherche de v_{opt} se fait par une approche dichotomique (à l'idée d'une recherche dichotomique dans un tableau en $\mathcal{O}(log_2(n))$ entre 0.2 et 10 (0.2 étant la vitesse de base proposée par *Autorace*). Chaque simulation est sauvegardée dans un historique de simulation. Une run de l'historique est structuré de la manière suivante : un pointeur vers le fichier .json du virage, les trois points de contrôles du virage (dont on peut déduire la classe), les vitesses v_{min} et v_{max} imposées dans la simulation et enfin la liste des positions du véhicule. Ces données sont ensuite analysées.

Enfin discutons des terminologies des classes. Une classe est définie comme cohérente si et seulement si les v_{opt} de tous les virages associés sont suffisamment proches. Si toutes les classes sont cohérentes, alors la classification est validée. Pour aller plus loin, on pourra tenter d'expliquer les résultats observés (voir Étude des résultats).

3.C.b. Expérimentation

La première partie de l'expérimentation consiste à d'abord classifier nos virages, pour cela, nous avons décidé d'utiliser deux approches, l'une avec angle et distance, et l'autre la courbure, toutes les deux présentées dans la partie Classification.

Pour l'expérimentation, nous automatisons les tests à faire avec un script. Il nous suffit d'ouvrir un virage dans l'application et donner les deux bornes des vitesses pour la recherche dichotomique. La validation et l'invalidation d'une vitesse sont automatisées. En pratique, la simulation a besoin en moyenne de 15 secondes pour faire une run simulée. En moyenne, il nous faut environ 2 minutes pour déterminer la v_{opt} d'un virage (plus ou moins 8 simulations).

L'étude est réalisée sur 6 classes. Le temps total de simulation a été de 3 heures (prenant en compte le changement manuel de virages). Les 6 classes sont construites autour des intervalles de scores en fonction de la méthode utilisée.

3.C.c. Etude des résultats

Dans le cas idéal, pour valider une classification, la figure devrait avoir une forme d'escalier, c'est-à-dire que chaque classe doit avoir son intervalle de vitesse sans chevaucher sur les autres. Et concernant les virages réussis avec une vitesse dans l'extrême (soit trop petite, soit trop grande), ils devraient se retrouver dans une même classe.

Passons à l'étude des résultats obtenus avec l'expérimentation. Nous allons observer la répartition des virages dans chaque classe, en fonction de leur vitesse optimale. Pour cela, nous allons projeter ces données sur un plan à deux dimensions, en abscisse les vitesses optimales, et en ordonnée les différentes classes. Les données abérantes comme les virages avec une vitesse optimale de 0, ou des virages avec une vitesse incohérente (tel qu'une route droite mais avec une vitesse très faible) seront supprimées de notre dataset pour avoir une évaluation plus précise. Et on va commencer par analyser les résultats de la classification avec l'angle et distance :

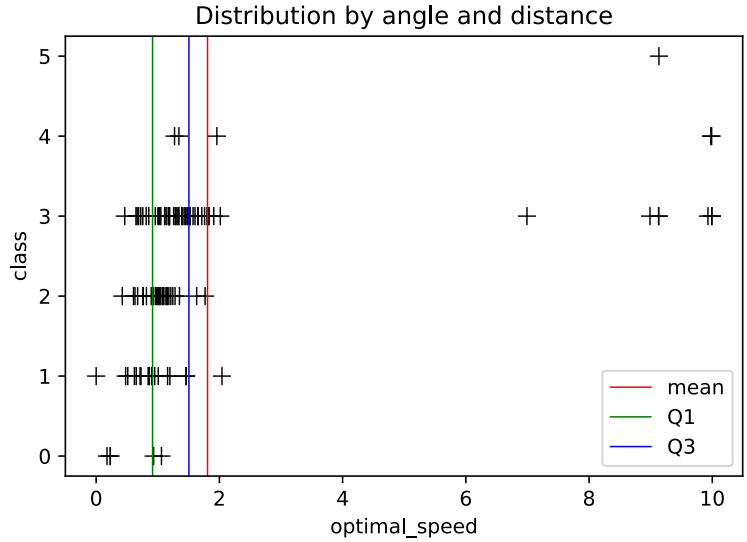

Figure 20: Répartition des virages dans chaque classe selon l'angle et distance

Il est clair que la répartition n'est pas si concluant avec cette méthode. La plupart des virages ont été attribué à la classe 3 (68 sur 123, donc plus de la moitié), et que les virages des classes 1, 2 ont des vitesses optimales très proches, donc ces deux classes sont difficilement distinguables selon la Figure 20 (et c'est aussi le cas lorsqu'on obeserve les images des virages). En plus de cela, les virages avec une grande vitesse optimale ont été attribués dans des classes différentes (3, 4, 5), alors que ces virages devraient être dans la même classe puisque le robot est capable de les prendre facilement. Prenons comme exemple les virages n°134 et n°176, qui sont similaires à l'image :

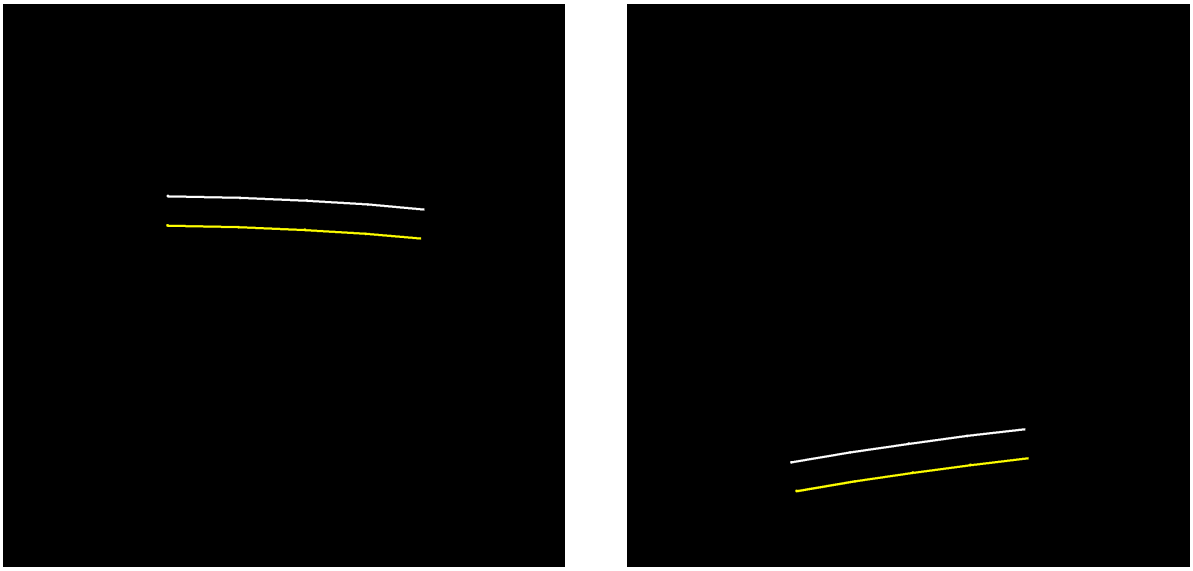

Figure 21: Virage n°91 (gauche) et n°134 (droite)

Ces virages ont tous deux une vitesse optimale proche de 10, mais ont été attribué dans deux classes différentes. Cette distribution ne peut être expliquer que par le calcul des scores qui n'est pas assez optimisé, on peut supposer que c'est la distance qui a induit cette différence puisque les deux virages ont tous les deux un angle proche voir identique. Donc pour cette méthode, il faut essayer de changer le processus de calcul de score, par exemple en ajoutant d'autres facteurs à prendre en compte, ou de diminuer/augmenter l'impact de la distance.

Ensuite, passons à l'analyse des résultats de la classification selon la courbure :

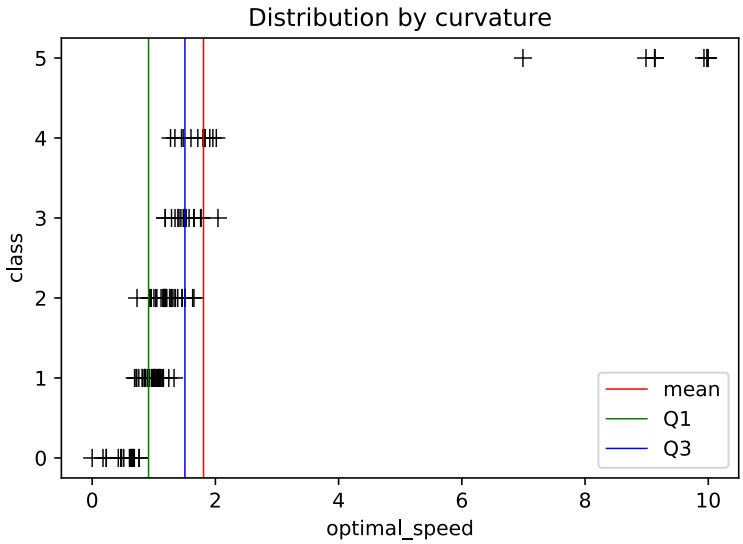

Figure 22: Répartition des virages dans chaque classe selon la courbure

Sur la Figure 22, nous pouvons observer que les virages sont plus ou moins bien répartis par rapport à leur vitesse, et les données commence à avoir une forme d'escalier, ce qui

est la répartition que l'on cherche à atteindre. Par ailleurs, nous remarquons quand même que les virages des classes 3 et 4 se chevauchent pas mal entre eux, sinon la répartition est globalement satisfaisante. Nous pouvons aussi noter que pour les virages réussis à grande vitesse (avec une vitesse optimale proche de 10) sont tous attribués dans la dernière classe qui est la classe 5, et c'est aussi ce qui caractérise une bonne classification.

D'un autre côté, pendant l'expérimentation nous avons abouti deux facteurs importants qui influence la qualité de notre résultat :

- La recherche des meilleurs hyperparamètres qui est très important pour pouvoir trouver un résultat satisfaisant (nos résusltats sont obtenus après avoir tester seulement quelques paramètres). Les deux méthodes possèdent les mêmes hyperparamètres qui sont :
  - Le nombre de classe. Dans notre cas, nous avons choisi 6 classes.
  - L'intervalle de scores pour chaque classe. Ce qui détermine l'appartenance de chaque virage dans sa classe en fonction du score évalué.
- La taille du dataset. Il faut noter que le temps de simulation pour les virages est assez important, donc nous n'avons pas pu construire un dataset de grande taille. Le manque de données est aussi un facteur que l'on ne peut pas négliger dans cette partie.

Pour finir, la classification avec courbure nous semble plus adaptée pour classifier nos virages, et ce résultat n'est pas encore à l'optimal, puisque les hyperparamètres peuvent encore être optimisés en automatisant tout le processus par une recherche locale par exemple, qui consisterait à génèrer et évaluer les différents paramètres, puis de choisir le meilleur pour réitérer l'ensemble des processus jusqu'à l'obtention d'un résultat satisfaisant.

4. Conclusion

4.A. Mise en perspective du travail

Ce travail entre dans le cadre d'un projet beaucoup plus global qui a commencé bien avant nous et continuera bien après. Ceci explique plusieurs points cruciaux de notre travail qui ne ressortent pas forcément de ce rapport. D'abord il a été primordial d'acquérir une compréhension large du travail complexe fait jusque là dans le cadre du projet E-HoA, puis comment y contribuer de manière utile à notre échelle et utiliser les outils à disposition (Road Editor, WebMap Editor...). Finalement, nous apportons des conclusions et un outil à la fois complémentaire et dépendant de ce qui a été fait jusqu'ici.

4.B. Pour aller plus loin

Le protocole d'évaluation adopté peut être amélioré. L'idée serait d'étudier la vitesse maximale du véhicule simulé à l'entrée du virage et laisser l'algorithme de conduite libre de

chercher à accélérer / ralentir / s'orienter pour passer ce virage. L'approche reste donc la même avec une recherche dichotomique de la vitesse maximale à l'entrée du virage mais les conclusions apportées pourraient être plus réalistes. Cependant ce protocole nécessite une maîtrise de la vélocité instantanée du véhicule en début de simulation sur *Gazebo* que nous n'avons pas.

Nous avons imaginé une autre méthode de classification. Celle-ci, basée sur l'apprentissage, consiste à utiliser un modèle de régression (réseau de neurones, SVM, etc ...), en lui fournissant les trois points de contrôles d'un virage avec sa vitesse optimale en sortie pour l'entraînement. Puis, il nous suffit de passer les points de contrôle des nouveaux virages dans le modèle pour prédire sa vitesse optimale à ne pas dépasser, et ainsi en déduire la classe correspondante. Cette approche n'était pas envisageable pour ce projet du fait du manque de données et de la difficulté pour en construire. La recherche de la vitesse optimale avec la simulation réaliste par *Gazebo* est beaucoup trop lente pour avoir un jeu de données conséquent.

5. Bibliographie

References

[1] Maruan Al-Shedivat, Trapit Bansal, Yuri Burda, Ilya Sutskever, Igor Mordatch, and Pieter Abbeel. Continuous Adaptation via Meta-Learning in Nonstationary and Competitive Environments. *arXiv:1710.03641 [cs]*, February 2018.

[2] Peng Cao, Zhandong Xu, Qiaochu Fan, and Xiaobo Liu. Analysing driving efficiency of mandatory lane change decision for autonomous vehicles. *IET intelligent transport systems*, 13(3):506–514, 2019.

[3] Ji-wung Choi, Renwick Curry, and Gabriel Elkaim. Path Planning Based on Bézier Curve for Autonomous Ground Vehicles. In *Advances in Electrical and Electronics Engineering - IAENG Special Edition of the World Congress on Engineering and Computer Science 2008*, pages 158–166, October 2008.

[4] David González, Joshue Pérez, Ray Lattarulo, Vicente Milanés, and Fawzi Nashashibi. Continuous curvature planning with obstacle avoidance capabilities in urban scenarios. In *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pages 1430–1435, October 2014. ISSN: 2153-0017.

[5] Christos Katrakazas, Mohammed Quddus, Wen-Hua Chen, and Lipika Deka. Real-time motion planning methods for autonomous on-road driving: State-of-the-art and future research directions. *Transportation Research Part C: Emerging Technologies*, 60:416–442, November 2015.

[6] Sang chan Moon, Soon-Geul Lee, and Dong-Han Kim. Classification of Roadway Type Based on Roadway Characteristics Using RCCC. *International Journal of Applied Engineering Research*, 10:27587–27592, 2015.

[7] Gabriel Suchowolski Morelli. Quadratic bezier offsetting with selective subdivision. page 9, July 2012.

[8] Kenneth Renny Simba, Naoki Uchiyama, and Shigenori Sano. Real-time smooth trajectory generation for nonholonomic mobile robots using Bézier curves. *Robotics and Computer-Integrated Manufacturing*, 41:31–42, October 2016.

[9] Nickolas S. Sapidis and William H. Frey. Controlling the curvature of a quadratic Bézier curve. *Computer Aided Geometric Design*, 9(2):85–91, 1992.

[10] Ihn-Sik Weon, Soon-Geul Lee*, and Soo-Ho Woo. Lane Departure Detecting with Classification of Roadway Based on Bezier Curve Fitting Using DGPS/GIS. *Tehnički vjesnik*, 28(1):248–255, February 2021. Publisher: Strojarski fakultet u Slavonskom Brodu; Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek; Graevinski i arhitektonski fakultet Osijek.