Projet Ingénierie et Entrepreneuriat

Projet 86

Génération rapide de trajectoire pour voiture autonome

Étudiants :

Médéric Fourmy
William Jussiau
Guillermo Prieto
Martin Renou
Jan Zeman

Tutrice en gestion de projet :
Frédérique Martin, AIRBUS

Responsable technique :
Diane Bury, CNRS-LAAS

Client :
Nicolas Mansard, CNRS-LAAS

Résumé

Ce document présente le travail effectué pendant l'année scolaire 2017-2018 dans le cadre du PIE « Génération rapide de trajectoire pour voiture autonome, sous l'encadrement de deux tuteurs du CNRS-LAAS et d'une tutrice en gestion de projet, d'Airbus.

Le projet a consisté à s'intéresser à une approche développée au LAAS pour la génération rapide de trajectoire, basée sur l'interaction entre un réseau de neurones et un algorithme de contrôle prédictif.

Afin de démontrer la pertinence d'un tel algorithme et son intérêt par rapport à la méthode traditionnelle de contrôle prédictif, nous avons développé un simulateur de voiture autonome exploitant cette méthode. Celui-ci se base sur l'utilisation des logiciels ROS et Gazebo, classiques dans le domaine de la robotique.

Mots-clés : voiture autonome, simulateur, contrôle prédictif, réseau de neurones, IREPA, ROS, Gazebo, CNRS-LAAS

Déclaration d'authenticité

Nous déclarons solennellement que le travail présenté a été produit entièrement par notre équipe d'étudiants pendant l'année scolaire 2017-2018.

Dans le cas de l'utilisation de ressources extérieures, celles-ci seront dûment citées et leurs auteurs remerciés.

Table des matières

Introduction

Le contexte

Avec le développement récent de voitures dites autonomes dont l'accessibilité au grand public est prédite pour les années 2020 [1][2], les thématiques liées au pilotage de ces engins sont devenues très actuelles. Le CNRS-LAAS [3], acteur incontournable de la recherche en robotique en France, a dans ce contexte mandaté un projet sur la génération rapide de trajectoire pour un véhicule autonome en partenariat avec l'ISAE-SUPAERO. Ce projet prend la forme d'un Projet Ingénierie et Entrepreneuriat (PIE) proposé à une équipe de 5 étudiants de troisième année, encadrés par deux chercheurs du LAAS et une tutrice en gestion de projet, d'Airbus.

Le projet

Le projet fait suite à un projet réalisé l'année dernière, dans lequel les étudiants ont appliqué des techniques de commande prédictive à un drone quadrirotor. L'un des problèmes majeurs de cette approche est que le contrôleur de bord doit recalculer entièrement la trajectoire à une fréquence donnée, opération gourmande en temps et en énergie. Ici, on s'intéressera à l'implémentation d'un algorithme développé au LAAS qui améliore l'approche précédente : un pré-calcul hors-ligne détermine de bonnes trajectoires et les stocke dans un réseau de neurones. Celui-ci, embarqué sur le système réel, donne une bonne première approximation de la trajectoire en temps réel (et avec un faible coût calculatoire), qui est ensuite raffinée par un contrôleur prédictif. Notre travail se base sur une publication scientifique issue du LAAS où l'algorithme IREPA (pour *Iterative Roadmap Extension and Policy Approximation*) et son utilisation sont expliquées plus en détails [4].

L'objectif du projet est de développer pour le LAAS un démonstrateur de voiture autonome exploitant cette approche, afin de mettre en avant sa pertinence par rapport au contrôle prédictif traditionnel. Nous avons choisi de nous baser sur l'utilisation du logiciel de communication ROS [5] et du simulateur Gazebo [6], deux références en robotique.

Ce document

Ce rapport se décompose en trois parties. Dans une première, on abordera la façon dont le projet a été géré : méthodes de travail, répartition des tâches et des responsabilités, conventions, définition de jalons... Dans une seconde partie, on détaillera la manière dont on a réalisé le projet : architecture, logiciels utilisés, code, méthodes employées... Enfin, dans une dernière partie, on mettra en avant notre contribution et on donnera quelques pistes pour une éventuelle suite.

Chapitre 1

Gestion de projet

1.1 Description du projet

1.1.1 Objectifs, périmètre

Objectif

L'objectif de ce projet est unique et a été bien défini au départ par le mandataire du projet. Il s'agit de :

Prouver, en développant un démonstrateur, la pertinence de la méthode de génération rapide de trajectoire basée sur l'approche [4] dans le cas de la voiture autonome.

Périmètre

Le périmètre du projet a été contraint de sorte que l'objectif reste réalisable dans le temps imparti. En particulier, certaines des limites du simulateur sont les suivantes :

- la voiture connaît parfaitement son environnement : on considère que la chaîne de perception (traitement d'images, de données radar/lidar, etc.) qui existerait sur un système réel nous fournit toutes les informations nécessaires et exemptes de bruit de mesure,
- le modèle et les paramètres de la voiture sont supposés entièrement connus et déterministes,
- les calculs seront effectués et leurs performances seront évaluées sur une architecture de calcul classique (non embarquée, ex. processeur Intel i7, exécution du code à 10 Hz). Cela ne devra pas empêcher le portage sur un calculateur embarqué,
- la voiture ne sera pas manipulée en temps réel par un utilisateur disposant d'un clavier ou d'un joystick. Les paramètres de l'environnement (obstacles) et de la trajectoire (départ, arrivée) seront choisis par l'utilisateur avant la simulation.

Exigences fonctionnelles

En l'absence de spécifications fonctionnelles précises sur la forme, nous avons pu décider (avec accord du client) quelques contraintes haut niveau, qui sont les suivantes :

1. le démonstrateur devra être facile à lancer et à manipuler pour un utilisateur sans connaissance préalable,

2. le code devra être le plus modulaire et réutilisable possible, afin de favoriser une éventuelle extension du projet.

Quelques exigences, énoncées implicitement dans la description du projet, sont aussi détaillées ci-dessous :

- Documentation : le démonstrateur doit posséder une documentation technique, qui comprend :
  - une procédure explicative d'installation,
  - une liste exhaustive des ressources externes nécessaires,
  - un manuel d'instructions pour l'utilisation du logiciel.
- Démonstrateur : le démonstrateur doit respecter les exigences suivantes :
  - simuler une voiture dans le cas nominal,
  - simuler et représenter un environnement contenant des obstacles,
  - opérer sous un environnement Linux,
  - s'exécuter en temps réel,
  - ses ressources externes (ex. librairies, logiciels) doivent être bien identifiées et répertoriées.
- Algorithme : la génération rapide de trajectoire devra respecter certaines règles :
  - l'algorithme se base sur l'article cosigné par notre tuteur Nicolas Mansard : "Using a memory of motion to efficiently warm-start a nonlinear predictive controller" [4]. Par abus de langage, dans la suite du rapport, on pourra désigner la méthode décrite dans cette article par le sigle IREPA (qui correspond en fait à la partie apprentissage de l'estimateur),
  - l'algorithme doit permettre de générer une trajectoire entre deux points du plan,
  - l'algorithme doit empêcher toute collision avec un obstacle.
- Interface utilisateur : le logiciel disposera d'une interface utilisateur basique respectant les contraintes :
  - l'utilisateur doit pouvoir choisir les points de départ et d'arrivée de la trajectoire,

Contraintes :

Le contexte dans lequel s'inscrit le projet a pour conséquence l'apparition de quelques contraintes que nous devrons prendre en compte dans le développement, à savoir :

- Développement logiciel :
  - Utilisation de logiciels et librairies : le projet devra s'inscrire dans une continuité relative avec le projet de l'année dernière sur le drone quadrirotor. Ainsi, il nous devrons utiliser l'outil de gestion de sources Git/GitHub pour la collaboration, ainsi que la librairie de calcul ACADO [7] pour le contrôle prédictif,
  - Philosophie de développement : dans un souci d'universalité, nous nous imposerons des normes de codage (GNU Coding Standards [8] pour C et bash, PEP8 [9] pour Python).
- Contrainte matérielle : il n'est pas possible d'installer les logiciels sur les ordinateurs des salles informatiques à l'ISAE. Il est donc nécessaire que chacun ait un ordinateur personnel assez puissant et où Linux est installé.

- Travaux effectués en amont : en plus du PIE de l'année 2016-2017, nous disposons du code de recherche produit au LAAS. La méthode IREPA a déjà été implémentée sur des cas différents et en utilisant une architecture différente. Nous nous inspirerons de certaines parties de ce code.

Nous profiterons aussi de l'expertise de Diane Bury, notre encadrante technique au LAAS, dans la mesure de ses disponibilités, qui pourra nous aiguiller sur la partie simulation car elle participé au PIE de l'année précédente (qui utilise les mêmes outils que nous - ROS, Gazebo).

1.1.2 Livrables du projet

À l'issue du projet, nous devrons fournir aux différentes parties prenantes les ressources suivantes :

— **Un rapport final**. Il regroupe tout le travail qui a été effectué, de la gestion de projet à l'exécution (développement logiciel) et les résultats auxquels nous sommes arrivés. Il sera accompagné d'un résumé sur une page (A4).
— **Le code source du démonstrateur**, qui permet de simuler une voiture autonome en utilisant l'algorithme de génération de trajectoire IREPA,
— **Une documentation technique**, comprenant un manuel d'installation et un manuel utilisateur. Un fichier README détaillé sera accepté.

1.2 Processus de développement

1.2.1 Définition détaillée du projet

Afin de pouvoir répartir les tâches dans le groupe, nous avons décomposé le projet en paquets de tâches. Au fur et à mesure de l'avancée du projet, cette répartition a été largement modifiée. À la date de rédaction du rapport, elle est la suivante :

— gestion du projet : organisation, définition des méthodes et outils, suivi du projet,
— développement du moteur de calcul, qui comprend le contrôle optimal et l'algorithme de contrôle IREPA,
— développement de l'aspect graphique et simulation, qui comprend la simulation et visualisation avec Gazebo,
— documentation et intégration : gestion en continu du suivi des spécifications

Product Breakdown Structure

On propose ici le diagramme PBS (Product Breakdown Structure), qui montre la structure du produit à livrer. Comme prévu, celui-ci comprend un moteur de calcul comprenant l'algorithme IREPA et le calcul du contrôle optimal. On note aussi une branche comprenant les obstacles (stockés dans un fichier dédié). La front-end comporte la simulation-visualisation avec Gazebo.

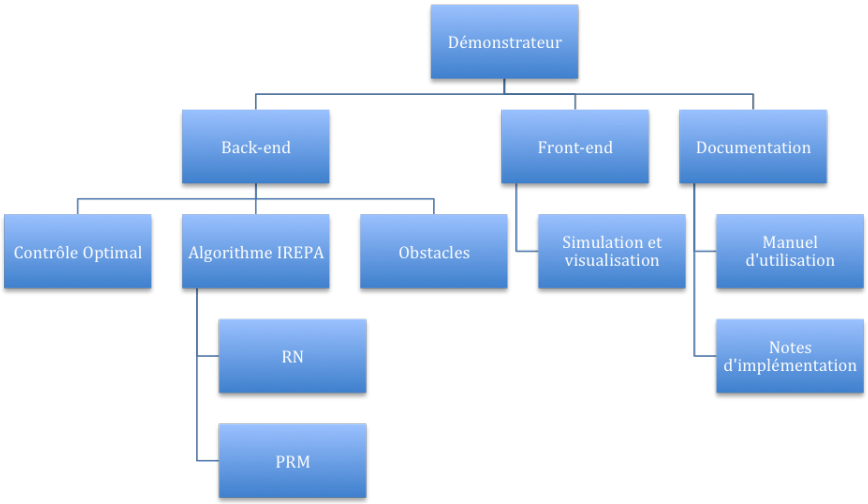

FIGURE 1.1 – Diagramme PBS

1.2.2 Organisation de l'équipe

Pour pouvoir effectuer le projet, il a été nécessaire d'organiser l'équipe en pôles, autour de tâches spécifiques. Ces attributions ont été faites en fonction des connaissances, souhaits et filières de chacun. Les rôles attribués ont été les suivants :

— Gestion de projet : William Jussiau
— Algorithme IREPA : Médéric Fourmy & Guillermo Prieto
— Contrôle optimal : Martin Renou

— Simulation-visualisation : Jan Zeman

Certaines tâches dans des pôles requérant peu d'expertise pourront être effectuées par un membre de l'équipe qui n'est à l'origine pas affecté à ce pôle.

La répartition des tâches est résumée dans le diagramme RACI joint (légende : A - accountable en orange ; R - responsible en jaune ; C - consulted en vert ; I - informed en bleu).

Diagramme RACI	Equipe PIE86					Client		Encadrante	Tutrice gestion de projet
	Médéric Fourmy	William Jussiau	Guillermo Prieto	Martin Renou	Jan Zeman	Nicolas Mansard		Diane Bury	Frédérique Martin
Développer le contrôle optimal (modèle dynamique et calcul)	R	R	I	A	R	I		C	
Développer un modèle d'environnement	I	R	I	R	A	I		C	
Mettre en place la visualisation	I	R	I	R	A	I		C	
Étudier et implémenter l'algorithme prédictif (PRM)	A	I	R	I	I	I	C	I	
Construire et entrainer le réseau de neurones	R	I	A	I	I	I	C	I	
Gestion de projet	R	A	R	R	R			I	C
Suivi qualité	R	R	R	A	R			I	

FIGURE 1.2 – Diagramme RACI du projet

1.2.3 Organisation du travail

Outils, méthodes

Cette partie pourra être redondante avec certaines informations présentes plus haut dans le rapport. On y décrit les outils et méthodes utilisés.

- Outils
  - Git/GitHub pour la gestion des sources,
  - l'utilitaire Travis [10] intégré à GitHub, qui ne permet la gestion automatique des normes de codage (en particulier : l'impossibilité pour un développeur de proposer un code non conforme),
  - Framateam.org (basé sur Mattermost [11]) pour les communications techniques. Il est possible d'y créer des canaux afin de discuter en petit comité, d'y partager des documents, du code, etc. Ont accès à l'outil : les 5 membres du groupe et Diane Bury,
  - Facebook Messenger pour les communications "projet" internes (ex. réunions),
  - e-mailing pour les communications externes (ex. avec Nicolas Mansard ou Frédérique Martin).
- Méthodes
  - Gestion des sources : procédé recommandé par les utilisateurs de Git et formalisé par Martin Renou lors de la réunion du 20/12/17. La gestion des sources se

fera en langue anglaise : toutes les étiquettes associées à des actions (ex. commentaire d'un *commit* ou d'une *pull request*) devront être rédigées en anglais,
— Normes de codage : normes liées au langage (C/C++, bash : GNU Coding Standards [8] ; Python : PEP8 [9]). Le choix du langage reste à discrétion de l'étudiant, en accord avec les autres membres du groupe si nécessaire,
— Rédaction de code : le code est rédigé et commenté en anglais,

Définition des jalons

Aucun jalon n'a été explicitement imposé par le client - ont seulement été évoqués des simulateurs prenant en compte différentes complexités (dans le modèle dynamique, l'environnement...). Nous avons ainsi défini au sein de l'équipe, avec l'aval du client, plusieurs complexités possibles :

- Relativement au modèle dynamique :
  — Modèle point sur grille cartésienne : $X = \begin{bmatrix} x \\ y \end{bmatrix}$, $U = \begin{bmatrix} dx \\ dy \end{bmatrix}$,
  — Modèle unicycle : $X = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$, $U = \begin{bmatrix} v \\ \omega \end{bmatrix}$
  — Modèle de voiture à définir exactement (possibilités : angulation des roues et vitesses différentielles en virage, inertie des roues, etc.)
- Relativement à l'environnement :
  — Circulation sur plan libre,
  — Circulation avec obstacles fixes (cylindres),
  — Circulation avec obstacles fixes de forme avancée (polyèdres) [1].

On a donc défini les jalons suivants :

- J1 : modèle point basique, simulé sur Gazebo avec prise en compte d'obstacles fixes par *Probabilistic RoadMap* (PRM) simple.
- J2 : modèle uni-cycle, simulé sous Gazebo avec algorithme IREPA complet.
- J3 : modèle de voiture (à définir), simulé sous Gazebo avec IREPA complet.

Nous avons aussi évoqué un dernier jalon J4 dans lequel nous pourrions développer des utilitaires qui montrent le processus de génération de trajectoire avec IREPA : graphes, trajectoires tracée en temps réel au sol, etc. Comme il ne s'agit pas de fonctions demandées, ce jalon est seulement accessoire. Il pourra aussi contenir un nouveau modèle de voiture.

Logique de développement

On utilisera pour le développement une approche incrémentale : chaque jalon correspond à une version fonctionnelle du logiciel, implémentant des modèles de plus en plus complets.

Dans la mesure du possible, les jalons sont indépendants. Cependant, chaque tâche d'un lot de travail requiert que la tâche de l'itération précédente (du même lot de travail) ait été effectuée.

1. La circulation avec des obstacles mouvants n'a pour l'instant pas semblé pertinente au vu de l'algorithme utilisé.

Lorsque les tâches correspondant à un jalon ont été effectuées dans chaque lot de travail, on peut procéder à l'intégration (mise en commun), le test et la validation de cette itération.

Tâches du projet

À partir des jalons et des lots de travail, nous avons pu déterminer les tâches plus atomiques à réaliser afin de valider les jalons. Les tableaux ci-dessous présentent toutes les tâches à remplir, de deux manières différente : tâches par jalon (Fig. A.1) ou par lot de travail (Fig. A.2). Ceux-ci sont disponibles dans l'annexe jointe au présent document.

Nous avons aussi formalisé cela sous Microsoft Project avec un diagramme de Gantt (prenant en compte certains liens temporels entre tâches) et l'emploi du temps (le calendrier de travail étant pris en compte). Les sur-utilisations de ressources ont été laissées telles que calculées par le logiciel - cela correspond à du travail supplémentaire à fournir par rapport au calendrier de base.

Évaluation du temps

D'après les tâches précédemment mises en avant, nous avons pu évaluer le temps nécessaire à la réalisation du projet. Il a été et sera nécessaire de le réévaluer aussi souvent que possible, en fonction de l'avancement des tâches et d'éventuelles complications.

Tâches	Temps alloué
– Gestion de projet	100 h
– Mise en route	120 h
– Développement	140 h
– Intégration + tests	75 + 10 h
Réalisation du projet	445 h
Marge	80 h
Charge de travail totale	525 h
Charge de travail par personne	125 h

FIGURE 1.3 – Évaluation du temps alloué au projet

1.2.4 Risques identifiés

Nous avons pu identifier un certain nombre de risques, regroupés dans le tableau 1.4 ci-joint. Seuls ceux présentant un fort indice de criticité (nous avons choisi les indices supérieurs à 9/16, correspondant à un score supérieur à 3/4 simultanément en probabilité et gravité) seront sujet à un plan d'action, afin que le projet soit perturbé le moins possible.

Les risques les plus critiques sont décrits dans le tableau 1.5.

Leur impact sera mitigé de la manière décrite dans le tableau 1.6. Notons que l'action préventive pour le risque d'incompatibilité logicielle (qui consiste à choisir le logiciel en fonction du matériel disponible) n'est pas satisfaisante : nos possibilités seraient bridées. Après beaucoup d'efforts, nous somme parvenus à installer et faire fonctionner tous nos logiciels sur les ordinateurs personnels de tous les membres du groupe.

Analyse des risques		Gravité			
		1- Faible	2 -Moyenne	3 - Grave	4 - Catastrophique
Probabilité	4 -Très probable				Temps de développement mal évalué
	3 - Probable			Difficulté d'intégration des briques logicielles ; Incompatibilité logicielle	
	2 - Improbable		Indisponibilité des encadrants	Problèmes de prise en main des outils ; Perte de modifications en cours de travail	
	1 - Très improbable	Arrêt de maintenance de l'un des outils			Perte des sources

FIGURE 1.4 – Risques mis en avant

Risque	Cause	Conséquence
Temps de développement mal évalué	Manque d'expertise et de connaissances du domaine	Blocage projet, retard sur les livrables
Difficulté d'intégration des briques logicielles	Spécifications vagues ou non-respect des spécifications	Retard, modifications à apporter, revue des spécifications
Incompatibilité logicielle	Mauvais référencement des versions ou matériel non adapté aux logiciels	Retard projet ou changement de logiciels utilisés

FIGURE 1.5 – Risques les plus importantes, leurs causes et conséquences

Risque	Criticité (1-16)	Actions préventives	Actions correctives
Temps de développement mal évalué	16	Prévoir des marges dans le temps alloué au projet Profiter de l'expertise des encadrants	Travail supplémentaire Utilisation des marges Redéfinition des livrables en accord avec le client
Difficulté d'intégration des briques logicielles	9	Définir des spécifications précises Suivre les spécifications (contrôle qualité)	Méthode agile Communication dans le groupe et retours fréquents
Incompatibilité logicielle	9	~~Choisir les logiciels en fonction du matériel disponible → non réalisable~~	Avoir recours à un support informatique (ex. service informatique de l'école)

FIGURE 1.6 – Gestion des risques les plus critiques

Chapitre 2

Développement du démonstrateur

Dans ce chapitre, on détaille le travail effectué. La présentation a été orientée autour de 4 grands axes, qui sont les suivants :

- La commande et la génération de trajectoire - contexte, principe et apport de l'algorithme IREPA,
- La modélisation de l'environnement et la simulation avec Gazebo,
- La résolution des problèmes de commande optimale avec l'outil ACADO,
- L'architecture finale du code, l'implémentation sous ROS et les résultats.

2.1 Introduction à la génération de trajectoire

Dans cette section, on va décrire le contexte dans lequel s'inscrit l'algorithme de génération de trajectoire que nous allons mettre en place.

Comme évoqué à plusieurs reprises dans ce rapport, il s'agit de permettre à une voiture de naviguer de manière autonome dans un environnement présentant des obstacles. On cherchera donc un algorithme de contrôle, global, qui permette d'éviter les collisions, en opposition avec un asservissement plus local. Il ne sera donc pas question ici de correcteur PID ou de commande par retour d'état, pour lesquels l'implémentation de base ne permet pas de formuler des contraintes.

2.1.1 Contexte de l'approche

Pour la commande des systèmes dynamiques complexes dans des environnements accidentés, les approches classiques d'automatique peuvent se révéler insuffisantes. Parmi les techniques qui existent, on peut établir deux groupes distincts d'approches :

1. Approches avec modèle (ou *model-based*),
2. Approches sans modèle (ou *model-free*).

2.1.2 Approches *model-based*

Ces approches nécessitent que l'on puisse décrire le système (ou le processus) sous la forme d'équations. Typiquement, il s'agit de pouvoir prédire l'état futur d'un système à partir de son état présent et d'éventuelles entrées contrôlées. Un problème de la sorte

pourra souvent être formalisé par une équation d'évolution, dont une forme très basique est la suivante :

$$\dot{X} = f(X, U)$$

où X est le vecteur d'état du système et U le vecteur de commande appliqué au système.

Parmi ces approches, on s'intéressera tout particulièrement au contrôle prédictif (ou *Model Predictive Control* (MPC)), une approche dans laquelle le but est de trouver la commande U qui amène le système dans l'état désiré $X^\star$, en anticipant le comportement du système selon son équation d'évolution sur un horizon fini glissant, T. Le calcul est réitéré à chaque pas de temps sur un horizon de même longueur (d'où l'adjectif "glissant"). Le procédé est illustré par la figure 2.1.

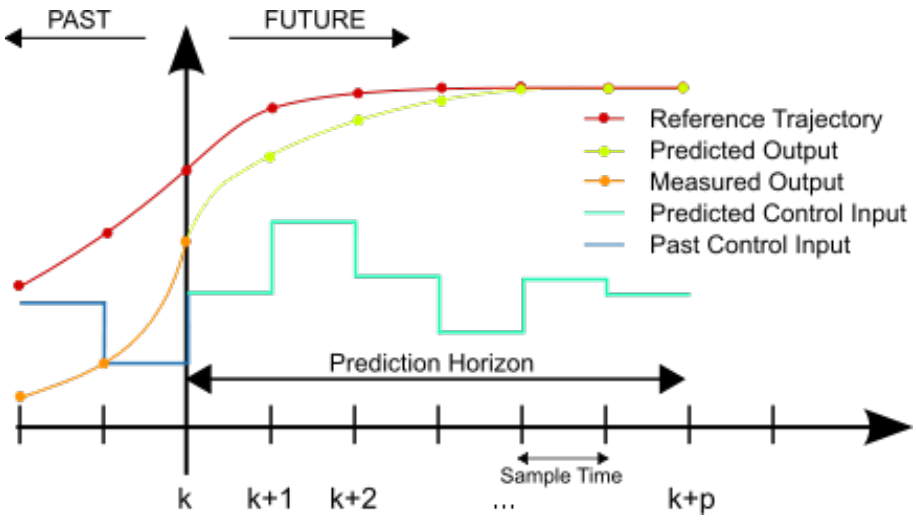

FIGURE 2.1 – Illustration du MPC, par Martin Behrendt (licence CC BY-SA 3.0)

L'un des problèmes qui ressort de cette méthode est une grande sensibilité à la modélisation : une faible erreur de modélisation (dynamique négligée, approximation, etc.) est susceptible d'avoir un impact négatif non négligeable lors du passage sur le système réel. De plus, le temps de calcul d'une telle approche est élevé : c'est une optimisation avec de l'ordre de $T \cdot (n_x + n_u)$ paramètres (pour la version de base, où n_x, n_u sont le nombre d'états et de commandes du système, respectivement), effectuée à chaque pas de temps (fréquence de l'ordre de 10 fois par seconde). Cela rend la technique applicable seulement à des systèmes de faible dimension seulement. De nombreux travaux tentent d'élargir l'approche pour la rendre plus largement applicable [12].

2.1.3 Approches *model-free*

À l'opposé, les méthodes sans modèle ne nécessitent aucune connaissance a priori sur le système. Parmi les plus connues, on pourra citer l'apprentissage par renforcement, déjà appliqué avec succès au contrôle de systèmes [13]. Ces approches peuvent apprendre (ex. par essai-erreur) la meilleure commande à appliquer à un système pour effectuer une tâche spécifique. Les avantages de ces techniques sont notamment un coût d'exécution très faible et une robustesse élevée, qui sont exactement les inconvénients des approches avec modèle. Toutefois, elles requièrent souvent un temps d'entraînement (hors-ligne)

prohibitif pour des systèmes de dimension élevée, notamment car la plus grande portion possible de l'espace d'état doit être explorée (et cela ne peut jamais être assuré). Par leurs avantages et inconvénients, elles peuvent être vues comme complémentaires des approches précédentes.

2.1.4 Complémentarité des approches et algorithme IREPA

Étant donné les caractéristiques des modèles évoquées ci-dessus, nous constatons qu'elles semblent complémentaires. Elles ont été combinées à plusieurs reprises dans des travaux de recherches, de deux manières :

1. Utilisation de techniques *model-based* pour accélérer l'entraînement de techniques *model-free* [14].
2. Apprentissage de modèles ou fonctions pour augmenter la performance de techniques *model-based*. Par exemple, la fonction approximée peut être utilisée pour initialiser le problème d'optimisation du MPC.

L'algorithme sur lequel nous nous basons est développé dans [4] : c'est l'algorithme IREPA [1]. Il exploite les deux familles susmentionnées, à savoir : i) une politique de contrôle est apprise hors-ligne grâce à une méthode avec modèle, et ii) cette politique est ensuite utilisée en ligne pour initialiser le MPC.

La méthode est détaillée dans la section 2.2.3.

2.2 Contrôle optimal et génération de trajectoires

2.2.1 Problème de contrôle optimal

Définition

Pour mettre en oeuvre l'algorithme IREPA, il est nécessaire de résoudre plusieurs problèmes de contrôle optimal. On définit un problème de contrôle optimal comme :

déterminer la commande d'un système qui minimise (ou maximise) un critère de performance, éventuellement sous des contraintes.

Appliqué à notre système, il s'agira de déterminer la meilleure commande à appliquer à la voiture pour effectuer un trajet, selon son modèle et en évitant des obstacles. Le critère de performance sera traité en suivant.

Définition formelle

De façon plus formelle, on définit le problème de la sorte :

1. Dans un souci de concision, on l'évoque par métonymie comme l'algorithme IREPA dans ce rapport (IREPA ne constitue qu'une partie de l'algorithme complet).

$$\begin{aligned}
&\arg\min_{X,U} \quad \mathcal{J}(X,U)\\
&\text{sous les contraintes :}\\
&\qquad \dot{X} = f(X,U),\ \forall t \in [0,T]\\
&\qquad X \in \mathcal{X} \cap \Omega_{free},\ \forall t \in [0,T]\\
&\qquad X(0) = X_0\\
&\qquad X(T) = X_f
\end{aligned}$$

Où les termes sont définis de la manière suivante :

- T l'horizon du MPC sur lequel on résout le problème,
- X, U les vecteurs d'état et de commande du système,
- $\dot{X} = f(X,U)$ l'équation d'évolution du système,
- $\mathcal{J}$ le critère à minimiser. La forme choisie sera discutée plus loin,
- $\mathcal{X} \cap \Omega_{free}$ l'espace des configurations de X qui sont libres (pas d'obstacle),
- X_0, X_f les états de départ et d'arrivée.

En bref, les termes de contraintes regroupent le modèle dynamique (qui doit être vérifié à tout temps) et la non-collision avec des obstacles.

Critère de performance

Dans l'implémentation de base, on utilisera un critère qui mène à la solution de temps minimal. Cela correspond à $\mathcal{J}(X,U) = \int_0^T dt = T$.

On pourra explorer des approches classiques où l'on considère des critères de performance quadratiques :

— à énergie minimale : $\mathcal{J}(X,U) = \int U^T R U dt$, où R est une matrice de poids,

— à état minimal : $\mathcal{J}(X,U) = \int X^T Q X dt$, où Q est une matrice de poids,

— prenant en compte ces deux contributions : $\mathcal{J}(X,U) = \int U^T R U + X^T Q X dt$.

Un outil : ACADO

Pour résoudre ce problème, on utilise un outil libre développé à la KU Leuven de Bruxelles : ACADO [7]. Pour citer la présentation que les auteurs font de leur outil, "ACADO Toolkit is a software environment and algorithm collection for automatic control and dynamic optimization. It provides a general framework for using a great variety

of algorithms for direct optimal control, including model predictive control, state and parameter estimation and robust optimization".

À l'aide de cet outil, on sera donc capables de résoudre facilement et efficacement nos problèmes d'optimisation.

2.2.2 Méthodes de génération de trajectoires

Introduction aux méthodes classiques

Certaines méthodes classiques de génération de trajectoires visent à déterminer la politique de contrôle optimale. C'est-à-dire, une fonction $\pi^\star : (x, u) \rightarrow x$ qui, à partir d'un état du système, détermine le meilleur contrôle pour amener le système à un nouvel état.

L'une de ces méthodes classiques se base sur l'utilisation d'une *probabilistic roadmap* (PRM). Dans un premier temps, l'idée est de construire une carte qui relie de façon non-optimale un nombre fini de points, choisis aléatoirement dans l'espace d'état du système. Un chemin entre deux états lointains peut alors être déterminée par un algorithme de type $\mathcal{A}^\star$. Ensuite, cette trajectoire est optimisée par contrôle optimal (solveur *optimal control problem* (OCP) dans ACADO).

La figure 2.2 détaille le fonctionnement de la dite méthode, basée sur une PRM. .

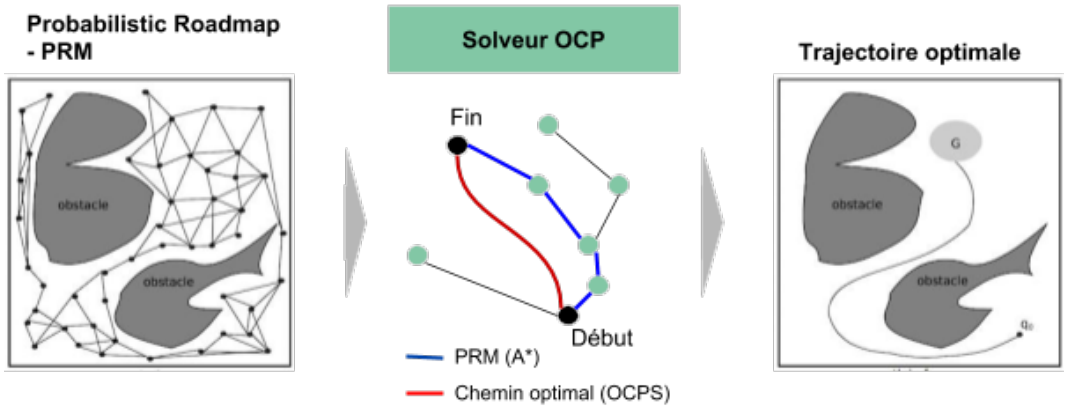

FIGURE 2.2 – Schéma de résolution du probléme de contrôle optimal (OCP) classique utilisant une PRM (*probabilistic roadmap*)

Méthodes d'approximation de politique de contrôle

Une extension de cette famille de méthodes consiste à utiliser un estimateur de la politique de contrôle. Il s'agit de construire une fonction de dimension plus faible qui approxime la politique de contrôle. Le plus souvent, le calcul de cet estimateur est long donc effectué hors-ligne, ce qui allège le coût de calcul en ligne. La partie 2.2.3 présente l'algorithme utilisé dans ce projet pour établir l'estimateur de politique de contrôle.

La figure 2.3 montre le schéma de fonctionnement de cette famille de méthodes.

Dans le cadre de ce projet, l'estimateur utilisé est un réseaux de neurones *feed-forward* du type perceptron multi-couches.

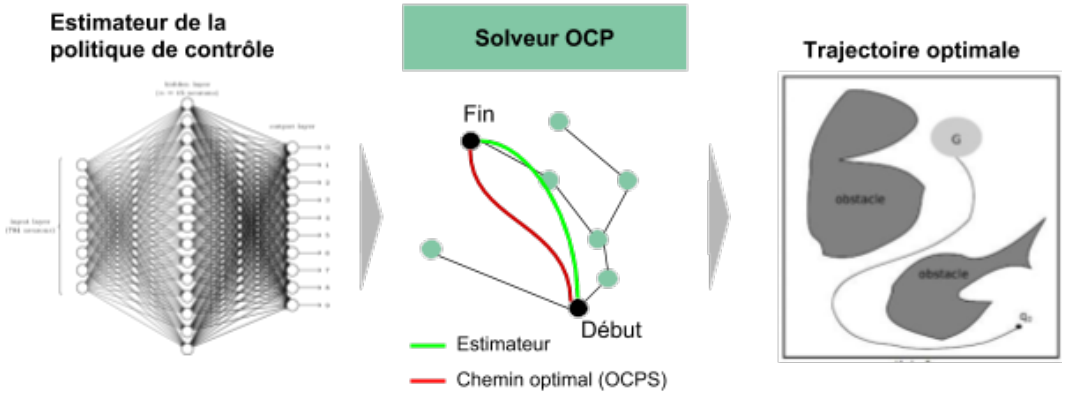

FIGURE 2.3 – Schéma de résolution du problème de contrôle optimal (OCP) utilisant un estimateur de la politique de contrôle optimale

Limitations des méthodes classiques, avantages des méthodes d'approximation

Bien que relativement efficaces, les méthodes sans approximation souffrent de plusieurs limitations. D'une part, les approches par PRM nécessitent que l'état de départ se trouve dans la PRM, sans quoi on doit chercher l'état de la PRM de le plus proche de l'état courant. D'autre part, elles souffrent d'un important temps de calcul en ligne et sont souvent piégés par des minimum locaux dans l'espace de solutions au problème de contrôle optimal. Les méthodes d'approximation de la politique de contrôle optimale réussissent à surmonter ces difficultés, comme l'explique la figure 2.4.

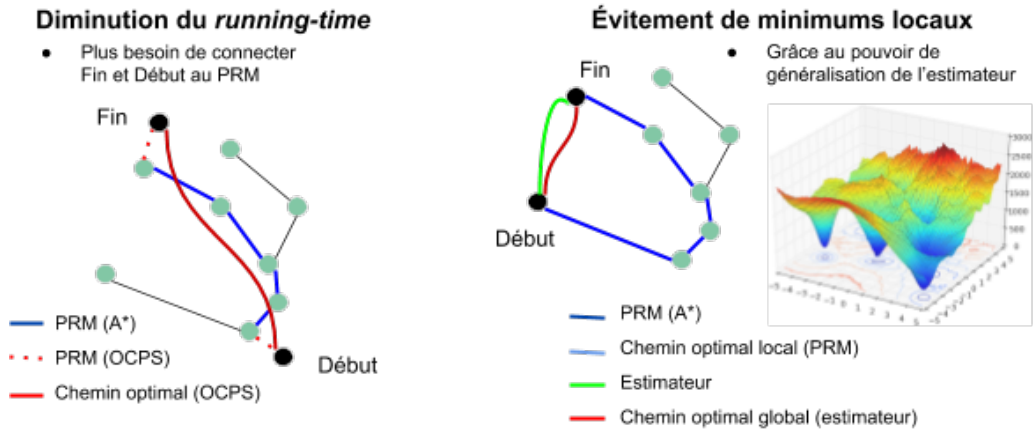

FIGURE 2.4 – Limitations des méthodes classiques de génération de trajectoires

2.2.3 Apprentissage de l'approximateur de politique de contrôle : la méthode IREPA

L'algorithme sur lequel nous nous basons est l'algorithme IREPA, développé dans [4]. Son fonctionnement est détaillé dans la figure 2.5.

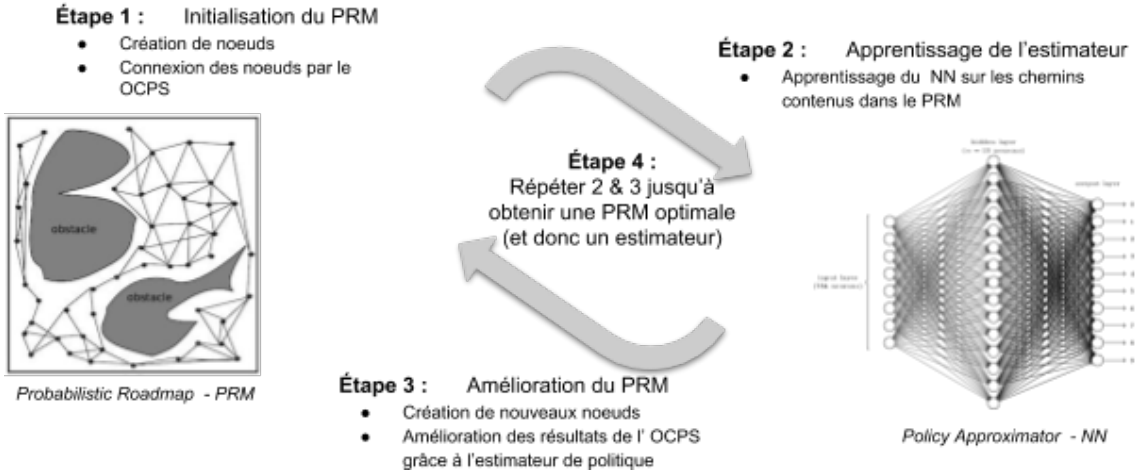

FIGURE 2.5 – Algorithme IREPA d'apprentissage de l'estimateur de la politique de contrôle optimale

2.2.4 Interaction avec la génération de trajectoire

L'algorithme IREPA permet donc d'entraîner un réseau de neurones dont la fonction est d'approximer une trajectoire entre deux points donnés. Cette approximation est alors donnée à l'outil de contrôle optimal en tant qu'initialisation d'un problème d'optimisation. Une trajectoire optimisée est finalement retournée en un nombre d'itérations faible.

Le bloc dont le rôle est de réaliser cette tâche (optimisation à partir de l'approximation) est dénommé *Path Finder*. Il prend en entrée deux états à rejoindre (typiquement, des positions dans le plan) ainsi qu'une trajectoire initiale, et retourne une trajectoire optimisée entre ces deux états. En ressources externes, il utilisera le réseau de neurones pour l'initialisation et ACADO pour le contrôle optimal (problème discuté plus loin, dans la section dédiée).

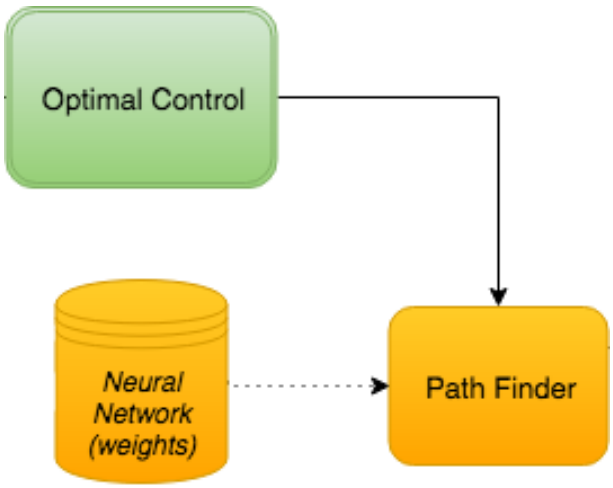

FIGURE 2.6 – *Path Finder* pour la génération de trajectoire

2.3 Modélisation de l'environnement et simulation

Les tâches de simulation et visualisation sont prises en charge par un unique logiciel : Gazebo. Nous détaillons plus bas le fonctionnement.

2.3.1 Simulation

Lorsqu'on évoque la simulation, on se réfère à l'étape qui permet de calculer l'évolution d'un système dans un environnement. C'est-à-dire, étant donnés un système commandé (la voiture), un environnement contenant des obstacles et des modèles physiques (gravité, frottements, collision), être capable de prédire l'état du monde dans le futur. Plus formellement, on veut connaître l'état du monde $\mathcal{W}_{k+1}$ à partir d'un état $\mathcal{W}_k$ comprenant le système $\mathcal{S}$, l'environnement Ω et des modèles physiques Φ – ces données étant considérés à l'instant k :

$$\mathcal{W}_{k+1} = \mathcal{G}(\mathcal{W}_k) = \mathcal{G}(\mathcal{S}_k, \Omega_k, \Phi_k)$$

Dans une première itération, nous avons géré cela manuellement, en développant un simulateur basique. Celui-ci utilisait l'hypothèse que la voiture n'interagissait pas avec l'environnement (pas de collision) et les modèles physiques n'ont pas été nécessaires. En bref, ce premier simulateur nous permettait d'avoir la position suivante de la voiture X_{k+1} à partir de sa position courante X_k et d'une commande U_k, tous les autres facteurs étant considérés sans impact sur le comportement du système. Il a ensuite suffi d'afficher la voiture dans son état courant à chaque pas de temps sur Gazebo. Mais Gazebo est plus puissant et peut lui-même simuler un modèle.

Pour la suite, nous avons donc utilisé le moteur de Gazebo `gzserver`, qui gère tous les modèles nécessaires en plus de celui de la voiture (que nous lui fournissons), en particulier les collisions, la gravité et les frottements. Il suffit de donner à Gazebo tous ces paramètres dans un format spécifique, et on peut ainsi simuler un monde et une voiture.

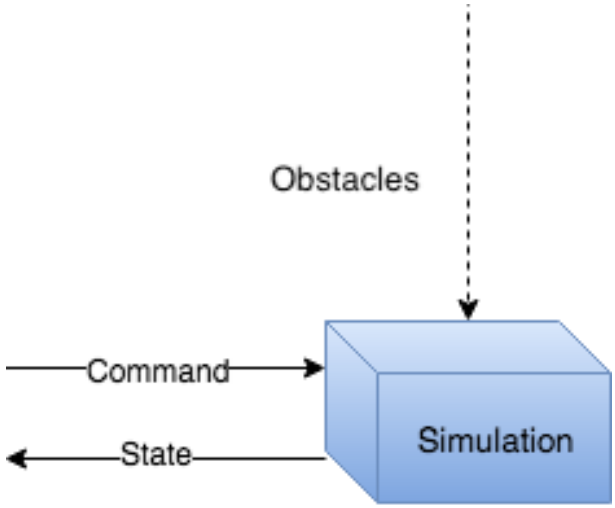

FIGURE 2.7 – Simulation avec Gazebo

2.3.2 Visualisation

Avec la simulation sous Gazebo s'accompagne un module graphique appelé `gzclient`, qui permet d'avoir un aperçu du monde simulé. Nous avons donc utilisé cet outil intégré en tant qu'interface graphique pour l'utilisateur.

2.3.3 Réalisation de la simulation

Modèle de voiture

La voiture est donnée sous la forme d'un fichier `urdf` qui est une spécialisation – propre à Gazebo – du format `xml` pour la description de robots.

Un fichier `urdf` basique contient, en plus du conventionnel nom de modèle, plusieurs sections importantes :

- `visual` : définit le visuel de l'objet, qui sera affiché à l'écran. Pour notre voiture, il a été admis qu'une représentation minimaliste serait suffisante – il s'agit donc d'un pavé droit. On pourra, dans une version plus avancée, y ajouter des fioritures comme des roues, lui donner une forme plus proche d'une voiture réelle ou ajouter des textures.
- `collision` : définit la zone de l'objet qui est sujette à collisions. Dans la plupart des cas (dont le nôtre), il s'agit de l'objet en entier. On pourrait définir un modèle de collision qui englobe légèrement le modèle visuel, afin de préserver ce dernier.
- `plugin` : définit la façon dont le modèle interagit avec l'extérieur de Gazebo, notamment comment le mouvement du modèle peut être contrôlé. Cette partie est discutée plus en détails ci-dessous.

Contrôle de la voiture : un plugin C++

Pour contrôler notre modèle de voiture, il est nécessaire de pouvoir faire passer des informations de l'extérieur vers l'intérieur de Gazebo (ex. envoyer une commande à la voiture), et vice-versa (ex. publier l'état actuel de la voiture). Cela permet un contrôle de la voiture en boucle fermée (i.e. la commande envoyée est fonction de l'état actuel de la voiture – à l'opposé d'une boucle ouverte, où la commande serait envoyée en fonction du temps).

Un plugin doit être codé manuellement, et Gazebo ne propose pas d'outil officiel de support du langage Python, ce qui nous a obligé a coder cette partie en C++. De nombreux tutoriels de Gazebo expliquent la création d'un plugin pour le contrôle d'un modèle, donc cela n'a pas posé de problème majeur. Il est même possible de lier un plugin Gazebo avec des instructions ROS, ce que nous avons fait, pour pouvoir commander la voiture à partir de l'algorithme que nous inclurons dans ROS.

Obstacles

Les obstacles sont à l'origine définis par l'utilisateur dans un fichier `obstacles.obs` qui a une syntaxe `xml`. Cela permet à l'utilisateur de préciser chacun des obstacles, cylindriques pour l'instant, sous la forme de trois paramètres : une position et un rayon — (x, y, R).

Mais cela n'est pas suffisant pour afficher et simuler les obstacles avec Gazebo. Plusieurs étapes intermédiaires doivent être effectuées avant d'obtenir les obstacles dans la simulation : extraire les paramètres de chaque obstacle et créer un modèle `urdf` propre à chaque obstacle, puis afficher ce modèle dans le monde simulé.

Chaque obstacle ayant une structure de fichier `urdf` proche (les champs sont les mêmes, seuls les paramètres de taille et placement sont modifiés), nous avons utilisé un utilitaire livré avec ROS : `xacro` (contraction de "xml" et "macro"). À partir d'un fichier "type" d'obstacle cylindrique (i.e. avec des paramètres formels), il sera possible de définir des fichiers proches dont les paramètres seront choisis.

Pour la prise en compte des obstacles dans le monde simulé, Gazebo offre un service disponible depuis ROS : `spawn_model`.

On affichera donc les cylindres avec l'algorithme suivant :

Pour chaque obstacle dans le fichier `obstacles.obs` :

1. Lire les paramètres de l'obstacle,
2. Exécuter l'utilitaire `xacro` pour créer le fichier `urdf` associé,
3. Générer l'obstacle dans le monde avec `gazebo_spawn_model`.

Illustration de la simulation

On montre ci-dessous l'aspect visuel de la simulation, telle que développée sous Gazebo. Plusieurs éléments apparaissent ici : les cylindres correspondent aux obstacles, la roue représente la voiture, le cône de chantier symbolise le point de départ et le panneau stop, l'arrivée. Ces objets ont été placés de façon arbitraire dans un but d'illustration uniquement.

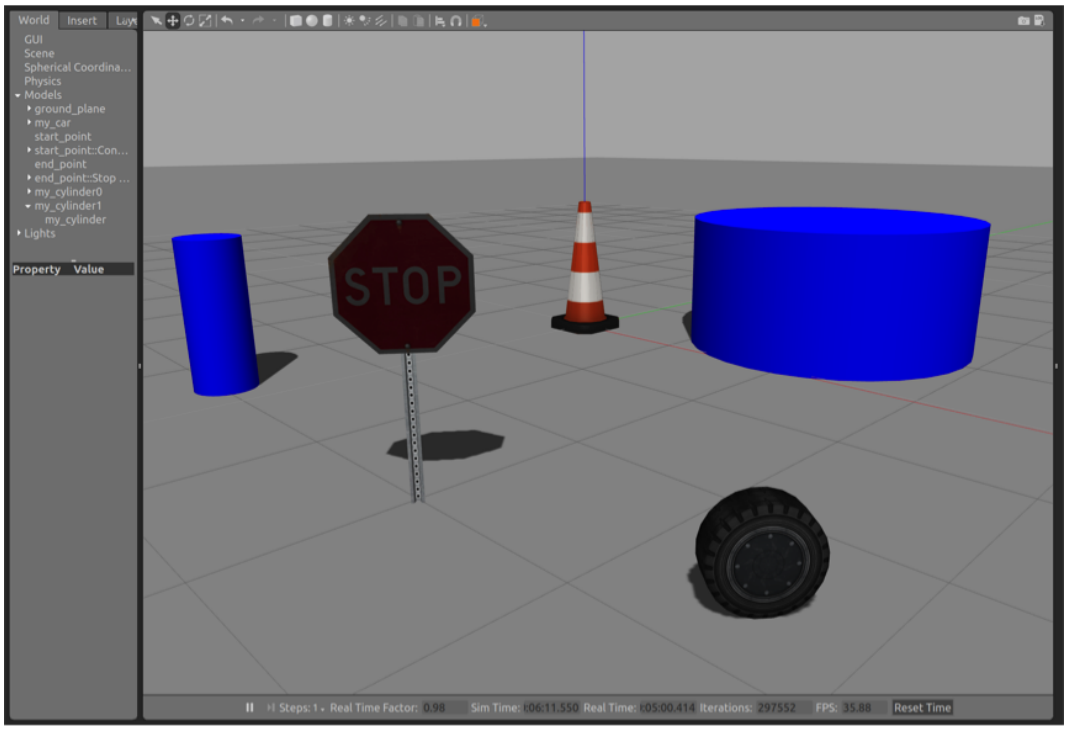

FIGURE 2.8 – Notre simulation sous Gazebo

2.4 Implémentation

2.4.1 Architecture haut niveau

On illustre notre logiciel par le schéma suivant, qui représente les blocs majeurs de l'implémentation. On représente en rouge le fichier contenant les obstacles, en bleu la simulation, en jaune la génération de trajectoire, et en vert les problématiques liées à la résolution du contrôle optimal.

De plus, on met en avant les deux phases de l'algorithme : à gauche, la partie hors-ligne (construction du réseau de neurones), et à droite, la partie en ligne (génération de trajectoire et simulation).

2.4.2 Architecture ROS

ROS est un logiciel qui propose une large gamme de fonctions pour la communication entre différents morceaux de codes, indépendamment du langage utilisé (utile dans notre cas, car nous avons utilisé Python et C++). Il met à notre disposition trois modes de communications que nous utilisons pour des cas de figures distincts :

- un `topic` : permet de publier une information sur un canal de communication, à une fréquence donnée. Typiquement, on utilisera des `topics` pour publier à intervalles de temps donnés la commande pour la voiture, ainsi que publier l'état de la voiture depuis le simulateur.
- un `service` : basé sur le schéma client/server. Un premier noeud *client* effectue une requête à un autre noeud sur lequel tourne en continu un *server* de service. Le premier noeud reçoit une réponse synchrone, c'est-à-dire qu'il est bloquant

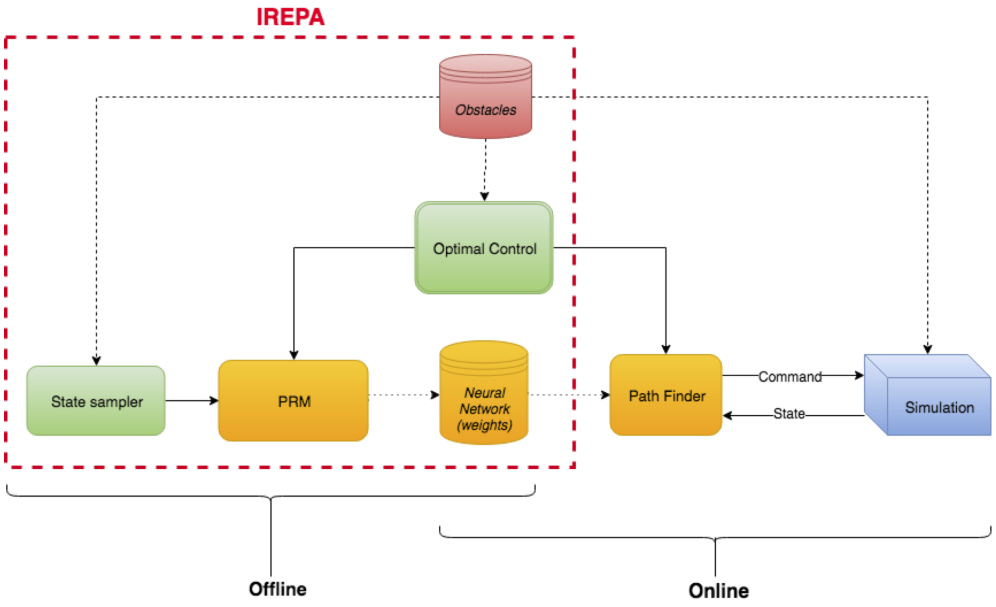

FIGURE 2.9 – Architecture haut-niveau du démonstrateur

en attendant la réponse. On utilisera les `services` pour des opérations courtes pouvant se faire de manière synchrone sans ralentir le reste du démonstrateur. On peut par exemple citer la requête permettant de récupérer la liste d'obstacles.

- une `action` : basée sur les topics, cette surcouche permet des communications asynchrones de type client/server. Un client envoie une requête à un server d'action et peut choisir d'attendre sa réponse où de continuer à effectuer d'autres tâches. Dans les deux cas, il est possible de traiter des informations quant à l'avancement de la tâche demandée pendant son exécution. Ce mode est adaptée aux taches longues comme le calcul du contrôle optimal en ligne.

NB : Il est possible d'obtenir un graph de l'ensemble des topics et services en fonctionnement après avoir lancé une application ROS, en utilisant la commande suivante dans un nouveau terminal : `rosrun rqt_graph rqt_graph`.

2.4.3 Répertoire et code source

Organisation du répertoire

L'organisation du répertoire des sources a été modifiée à plusieurs reprises pendant le développement. Il est néanmoins possible de noter une constante : dans chaque architecture du répertoire, nous avons séparé en différents dossiers les fonctions indépendantes du logiciel. Par exemple, on forme des dossiers contenant :

- le contrôle optimal → *opt_ control*
- la simulation → *display*
- l'algorithme IREPA → *roadmap*
- la gestion des obstacles → *obstacles*
- les fichiers de lancement → *demo_ launch*

Code source

Le code source est disponible sur GitHub, à l'adresse suivante :

`https://github.com/PIE86/MemoryEnhancedPredictiveControl`

Il contient tous les documents nécessaires pour l'installation et la manipulation du logiciel du projet de cette année (2018), ainsi que les projets des années précédentes (2016, 2017).

2.5 Résultats

Dans cette section, nous présentons les résultats de l'implémentation d'IREPA pour la modélisation simple d'une voiture. Nous nous intéresserons d'abord aux preuves de l'applicabilité de la méthode à ce système, puis aux trajectoires générées par le contrôle optimal et enfin à la question de l'implémentation d'un contrôleur on-line.

2.5.1 Construction de la PRM et apprentissage du réseau

Comme expliqué précédemment, chaque itération d'IREPA se déroule en trois étapes.

- **Expansion** : on essaie de connecter l'ensemble des noeuds non connectés de la PRM deux à deux. À l'initialisation (itération 0), l'estimateur n'est pas entraîné. Le contrôleur optimal utilise donc son initialisation par défaut, ou le plus court chemin déjà existant dans le graphe ($\mathcal{A}^\star$).
- **Apprentissage** : on entraîne l'estimateur (les réseaux de neurones) en utilisant les arêtes enregistrées dans le graphe.
- **Amélioration** : chaque arête pour laquelle on trouve une meilleure estimation avec le réseau de neurone est améliorée en initialisant le contrôle optimal avec cette estimation. Si la trajectoire trouvée n'est pas elle aussi bonne que la trajectoire déjà présente, elle est abandonnée.

Une des questions de ce PIE était de savoir si l'algorithme était applicable aux calcul de trajectoire pour une voiture. On peut le vérifier si on parvient à montrer qu'il est possible de construire un estimateur permettant un démarrage à chaud du contrôleur optimal (i.e. un estimateur tel que, lorsque le contrôleur optimal est initialisé avec l'estimation, la trajectoire trouvée est meilleure que sans intiialisation).

Les figures 2.10 et 2.11 en sont l'illustration. Sur la première, les courbes bleue, verte et orange sont respectivement le nombre de connections réussies en initialisant avec le plus court chemin ($\mathcal{A}^\star$), l'estimateur et le nombre total d'essais à chaque itération. Le deuxième graphes représente quant à lui l'évolution du nombre de connections à l'issue

de chacune des itérations d'IREPA. La ligne en pointillés correspond au nombre maximal de connections possibles ($C = n(n-1)$ où n est le nombre de noeuds).

Ces figures montrent donc que l'estimateur remplit sa tâche : il permet de connecter de plus en plus de noeuds au fur et à mesure de son apprentissage, jusqu'à un graphe quasiment complètement connecté.

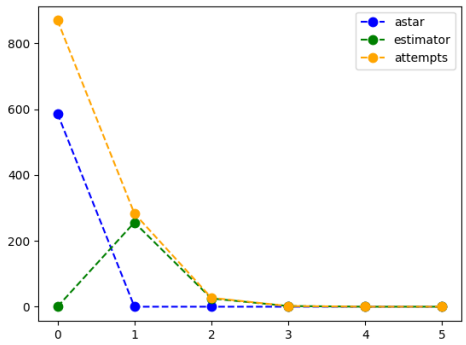

FIGURE 2.10 – Nombre de connections effectuées à chaque itération

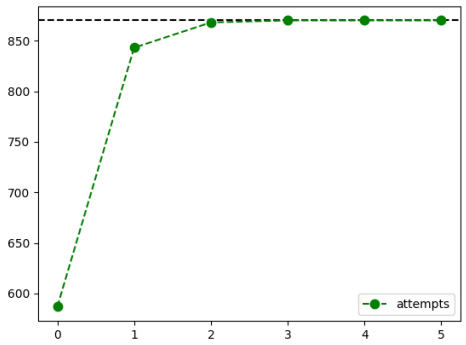

FIGURE 2.11 – Nombre de connections à l'issue de chaque itérations

Il est également intéressant d'observer l'évolution du coût total, qui est censé baisser du fait de l'étape d'amélioration. On rappelle que nous avons choisi d'utiliser le temps de la trajectoire comme fonction de coût. Sur la figure 2.12, on a en gris le coût total des arêtes du graphe à chaque itération (après l'étape d'expansion) avant amélioration des arêtes, et en vert le coût après améliorations. On observe que les différences de coûts sont très importantes sur les 3 premières itérations et deviennent faibles par la suite.

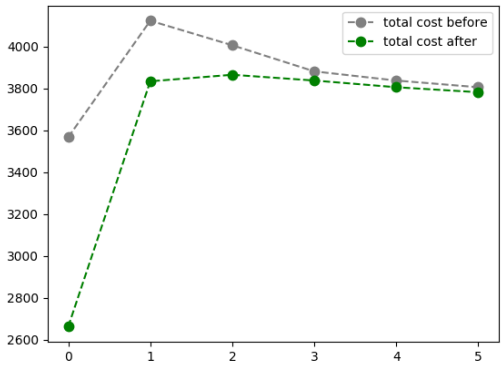

FIGURE 2.12 – Coût total des arêtes avant et après amélioration

2.5.2 Génération de trajectoires

Après avoir montré que l'apprentissage est possible avec ce modèle, nous allons étudier les trajectoires obtenues par contrôle optimal initialisé avec l'estimateur. Nous avons sélectionné deux trajectoires représentatives de l'apport de l'initialisation avec l'estimateur sur le calcul du contrôle optimal.

Trajectoire simple

- départ : $x = 2,\ y = 2,\ \theta = 0$
- arrivée : $x = 12,\ y = 15,\ \theta = 0$

Cette trajectoire paraît relativement simple à déterminer : elle ne comporte pas de rotation trop importante, n'y d'obstacles à proximité. La figure 2.13 représente les graphes de l'angle θ (en noir) et des contrôles en vitesses linéaire et angulaire v et w (en bleu), en fonction du temps. Les graphes du dessus correspondent à la trajectoire calculée avec l'initialisation par défaut d'ACADO, et ceux du dessous à celle avec l'initialisation de l'estimateur entraîné. Sur la figure 2.14, on observe en vu de dessus les différentes trajectoires calculées.

- En rose : trajectoire calculée par l'estimateur
- En orange : trajectoire calculée par le contrôle optimale avec initialisation par défaut
- En bleu clair : trajectoire calculée par le contrôle optimal avec initialisation de l'estimateur

Les trajectoires suivent des chemins assez proches en 2D mais le temps de trajet sans initialisation de l'estimateur est de 6.86s contre 2.1s avec. Cela s'explique par le fait que les contrôles sont dans le deuxième cas saturés. Par exemple, pour le contrôle en vitesse angulaire w, on a d'abord une valeur de 1 rad/s correspondant à sa borne supérieure, puis 0 et enfin -1 rad/s, la borne inférieure. La vitesse linéaire et durant tout le long à la borne supérieure 5 m/s.

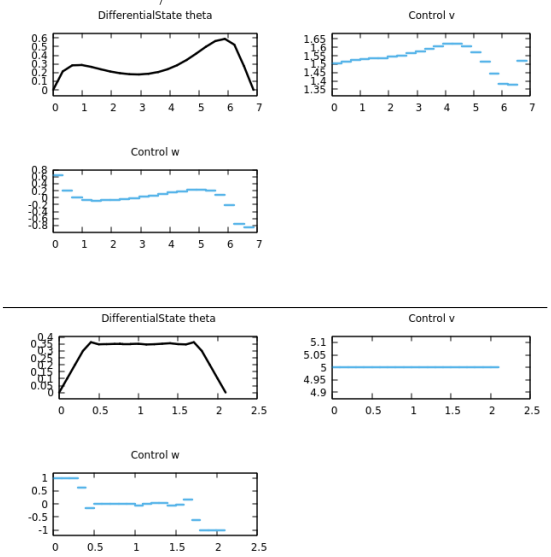

FIGURE 2.13 – Angle, contrôle en vitesses linéaire et angulaire en fonction du temps. Haut/bas : sans/avec initialisation de l'estimateur

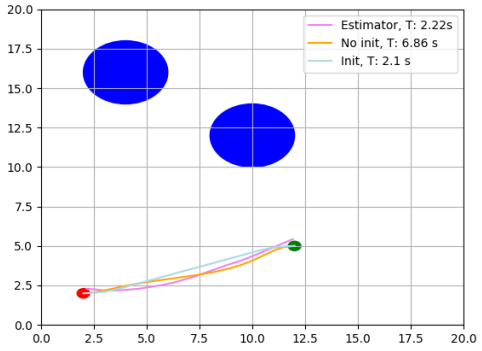

FIGURE 2.14 – Visualisation de la trajectoire et des obstacles. Point rouge/vert : départ/arrivée

Contournement d'obstacle

- depart : $x = 7,\ y = 1,\ \theta = -1$
- arrivée : $x = 14,\ y = 14,\ \theta = 1.5$

Observons maintenant le cas où la voiture doit contourner un obstacle. Le contrôle optimal sans initialisation donne une mauvaise trajectoire durant de 10.1 contre 2.5s pour la trajectoire bien initialisée. Les deux trajectoire calculées correspondent à des choix différents : dans le premier cas, l'obstacle est contourné par la gauche, la voiture reculant d'abord jusqu'au point d'inflexion, puis allant de l'avant à pleine vitesse, puis reculant encore. Dans le deuxième cas, l'obstacle est contourné par la droite. Encore une fois, on a une saturation des contrôles dans le cas initialisé, notamment la vitesse angulaire qui reste constamment égale à sa borne supérieure 1.

NB : Nous avons choisi de ne pas énumérer un trop grand nombre d'exemples de trajectoires afin de ne pas alourdir ce rapport. On peut cependant évoquer le fait que pour certains calculs non initialisés, le contrôle optimal n'était pas capable de trouver une solution contrairement au cas initialisé. Certains autres résultats ont été présentés lors de la soutenance et sont disponibles dans notre document de présentation.

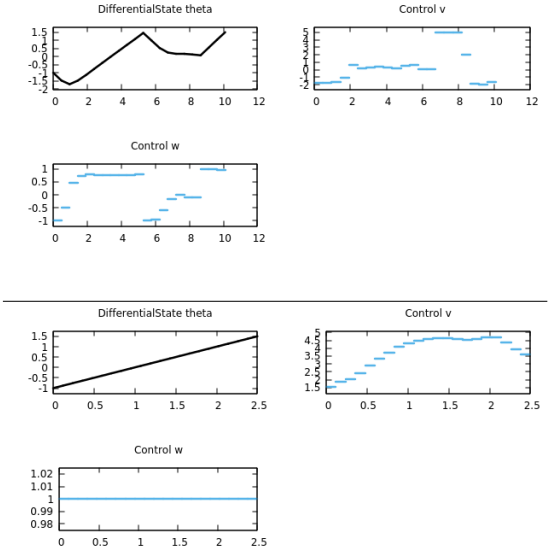

FIGURE 2.15 – Angle, contrôle en vitesses linéaire et angulaire en fonction du temps. Haut/bas : sans/avec initialisation de l'estimateur

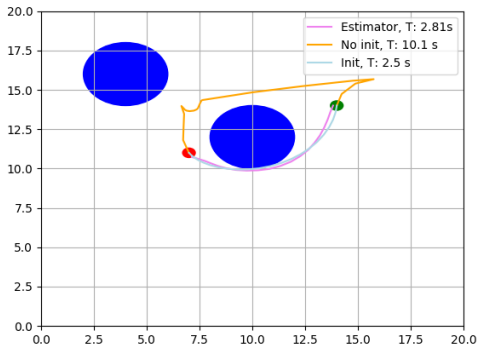

FIGURE 2.16 – Visualisation de la trajectoire et des obstacles. Point rouge/vert : départ/arrivée

2.5.3 Contrôle en ligne

Les trajectoires générées par le contrôle optimal initialisé par l'estimateur semblent donc être optimales et capables d'éviter les obstacles. Théoriquement, il devrait donc être possible d'implémenter un contrôleur online basé sur ces trajectoires.

Cependant, le facteur temps de calcul est très limitant en ligne, et il nous faut étudier plus en profondeur les possibles relations entre temps de calcul et "difficulté" de la trajectoire. Pour ce faire, calculons les trajectoires partant de $N = 400$ points aléatoirement échantillonnés dans l'espace admissible, vers un même point d'arrivée ($x = 12,\ y = 4,\ \theta = 0$). Pour chacune de ces trajectoires, on récupère la distance euclidienne entre les deux points considérés, le temps de calcul et le temps que dure la trajectoire. On comparera les performances des trajectoires calculées sans et avec initialisation (respectivement à

gauche et à droite, à chaque fois).

Temps de calcul et longueur de la trajectoire. Un premier graphe intéressant est le tracé du temps de calcul en fonction de la distance euclidienne entre les deux points. On observe que les trajectoires mettent de l'ordre de 0.25s à être calculées avec estimateur, et plutôt 0.10s sans. Il est possible de dégager une légère corrélation positive entre la distance euclidienne et le temps de calcul. Cependant les nombreux cas particuliers et la forte dispersion que l'on observe (calculs entre 0.5s et 2s) rendent difficile l'utilisation de cette métrique comme estimation de la complexité d'une trajectoire.

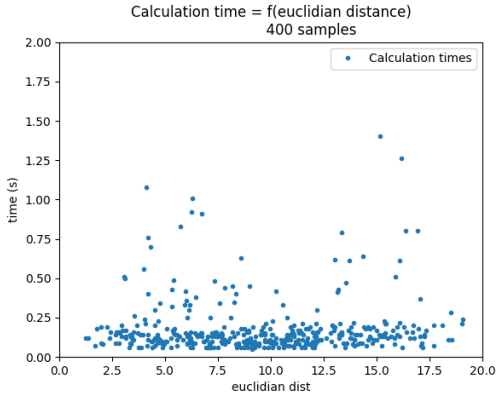

FIGURE 2.17 – Temps de calcul en fonction de la distance euclidienne - sans estimateur.

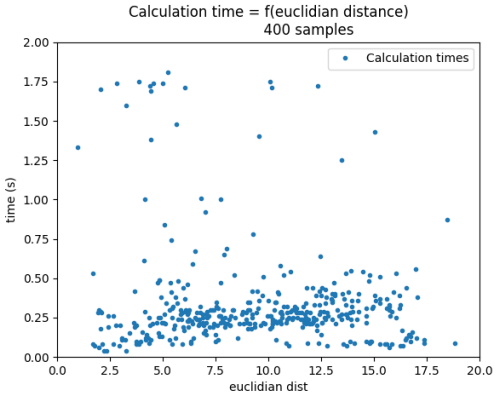

FIGURE 2.18 – Temps de calcul en fonction de la distance euclidienne - avec estimateur.

Temps de calcul et durée de la trajectoire. Examinons maintenant la relation entre le temps de calcul et la durée des trajectoires. On n'observe pas de corrélation entre les deux valeurs. Avec estimateur, les trajectoires sont réparties autour d'un même cluster. On note la présence d'un certain nombre de cas particuliers où l'une des valeurs est anormalement élevée et l'autre non.

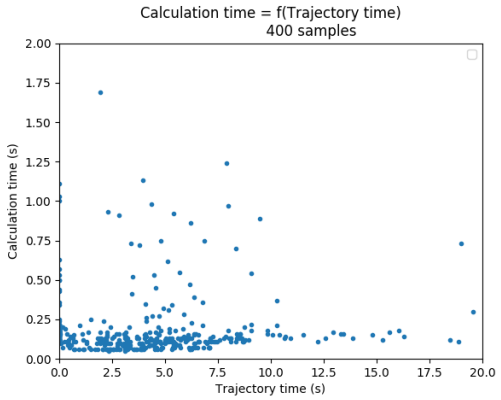

FIGURE 2.19 – Durée trajectoire en fonction du temps de calcul - sans estimateur

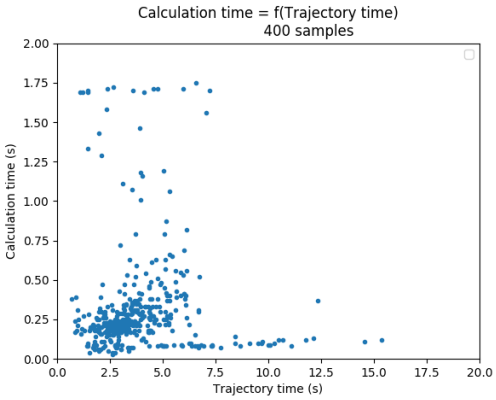

FIGURE 2.20 – Durée trajectoire en fonction du temps de calcul - avec estimateur

Longueur de la trajectoire et durée de la trajectoire. Enfin examinons la relation entre la distance euclidienne et la durée des trajectoires. On a tracé en bleu les durées des trajectoires et en orange les durées des trajectoires, moins leur temps de calcul. Cette fois, on observe une corrélation positive entre les deux valeurs. On note également de nombreuses valeurs nulles ou négatives pour le cas sans estimateur. Celles-ci correspondent aux cas où le solver n'a pas réussi à converger vers une solution respectant les contraintes du problème. On a aussi certaines valeurs négatives avec l'estimateur, correspondant aux cas où le temps de calcul est supérieur à la durée de la trajectoire calculée. Ces valeurs sont présentes dans le cas où les points sont proches au sens de la distance euclidienne. Ainsi pour les points trop proches, il semble impossible de contrôler le système de cette manière. Une solution pourrait être de se contenter de trajectoire sous optimales pour les points trop proches, de façon à abaisser le temps de calcul nécessaire.

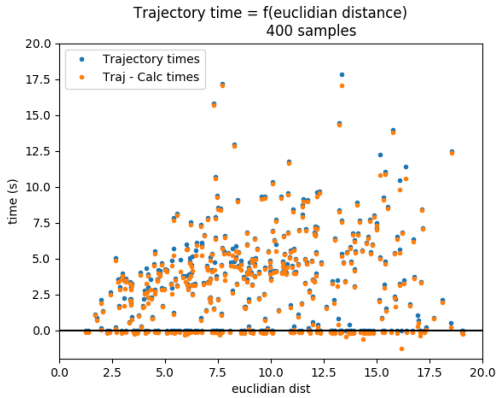

FIGURE 2.21 – Temps de calcul = f(Distance euclidienne) - sans estimateur

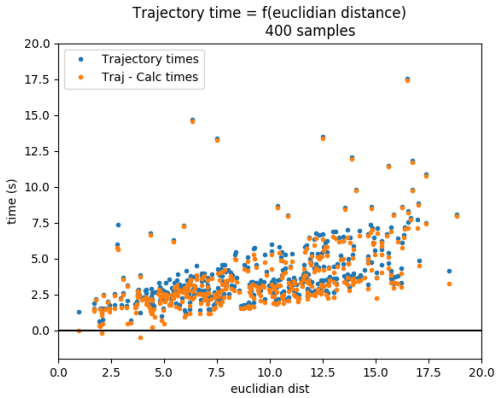

FIGURE 2.22 – Temps de calcul = f(Distance euclidienne) - avec estimateur

La grande incertitude sur le temps de calcul nécessaire à chaque mise à jour de la trajectoire optimale est l'une des difficultés à laquelle nous nous sommes heurtées. Le problème est qu'il est quasiment impossible de prévoir le temps de calcul, et on sait que la voiture aura progressé dans la trajectoire pendant que le contrôleur s'efforce de calculer les nouveaux contrôles... qui ne sont, du coup, plus totalement valables. La principale difficulté que nous avons rencontrée est donc de trouver une fréquence de calcul satisfaisante. Trop grande, le module contrôle optimal reçoit des requêtes de calcul pour de nouvelles trajectoires alors qu'il n'a pas fini de calculer les précédentes – trop faible, le contrôle sera en retard et de mauvaise qualité.

Nous avons finalement choisi de manière empirique de faire ce calcul à une fréquence de 3Hz.

Chapitre 3

Conclusion, perspectives

3.1 Synthèse

Le travail présenté a permis, au travers d'un démonstrateur développer sous ROS et Gazebo, de montrer que l'algorithme IREPA est applicable à la modélisation d'une voiture. On peut notamment mettre en avant les réalisations suivantes :

Modélisation de l'environnement et simulation à l'aide de Gazebo. Nous avons pu décrire un environnement (surface plane sur laquelle on trouve des obstacles cylindriques) modulaire, qu'il est possible de simuler avec l'outil Gazebo. C'est dans cet environnement que vit la voiture autonome que nous avons implémentée par la suite. Son contrôle est réalisé à l'aide d'une interaction entre ROS et Gazebo, au travers des plugins C++ proposés par Gazebo.

Contrôle optimal. Nous avons mis en oeuvre un outil de résolution de problèmes de contrôle optimal : ACADO Toolkit, afin de pouvoir déterminer la *meilleure* trajectoire (dans notre cas : la plus courte) entre deux points de l'espace d'état de la voiture. Ce calcul prend en compte les obstacles présents dans l'environnement.

Approche IREPA. Enfin, nous avons développé du code réalisant un contrôle grâce à l'approche IREPA, développée au LAAS. Et on a montré la pertinence de cette approche dans le cas de la voiture. En effet, l'apprentissage de l'estimateur permet d'améliorer la PRM en rendant possible la création de nouvelles connexions, et en diminuant le coût de celles existantes. Nous avons pu vérifier que les trajectoires obtenues était satisfaisantes et étaient bien optimales en temps, contrairement à celles obtenues sans initialisation. La fonction de coût temps entraîne des contrôles brusques pour lesquels les commandes sont la plupart du temps saturées.

Gestion de projet. Nous avons pu, au travers de ce travail, avoir une première vision de la gestion de projet telle qu'elle peut être appliquée dans un contexte industriel. Ce projet étant quasiment purement orienté développement logiciel, nous n'avons pas appliqué toutes les notions de gestion de projet vues en cours, mais celles que nous avons utilisées nous ont bien guidé et ont permis de faire avancer le projet tout au long de l'année.

3.2 Extensions possibles

On donne ici des pistes pour d'éventuelles extensions et améliorations qu'il serait intéressant d'étudier, dans le cas où le projet devait être repris. Elles sont données dans un ordre arbitraire.

Interface interactive. On pourrait imaginer implémenter des fonctions qui rendent la manipulation du démonstrateur plus agréable pour l'utilisateur : clics pour définir les points de départ et d'arrivée, visualisation des trajectoires calculées, etc.

Obstacles mouvants. Nous nous sommes limités à des obstacles fixes : il faudrait maintenant étendre la méthode à des obstacles mouvants. Plusieurs solutions semblent possibles, parmi lesquelles nous proposons : utiliser pour les obstacles des paramètres relatifs à la voiture et mesurés à tout temps (ex. par Lidar : angle et distance), ajouter ces paramètres dans l'état de manière dynamique (selon les obstacles visibles). Éventuellement, ajouter une prédiction de la dynamique des obstacles.

Capteurs et simulation réaliste. Une hypothèse forte de notre projet étaient la connaissance exacte de l'environnement de la voiture (obstacles, dimensions...). Il faudrait maintenant étendre cet algorithme en utilisant un point de vue plus réaliste : en utilisant les données de capteurs, bruitées par exemple, comme seules informations sur l'environnement.

Modèle de voiture complet. Ce point comprend deux éléments, qui sont : d'une part, l'implémentation d'un modèle complet de voiture dans le calcul optimal (sous forme d'équations différentielles) ; d'autres part, l'implémentation d'un modèle visuel de voiture dans la simulation.

Fonction coût dans le contrôle optimal. La fonction de coût temps minimal donne des trajectoires brusques. Il pourrait être intéressant d'explorer d'autres fonctions, minimisant par exemple les à-coups (de la commande et/ou des états : ajout des dérivées successives dans la fonction coût), voire une fonction multi-critères.

IREPA. On pourra, sous la direction de Nicolas Mansard, tenter d'améliorer l'approche IREPA de plusieurs façons : utiliser un même réseau de neurones pour estimer états, contrôles et fonction coût, ajuster les hyper-paramètres du modèle (structure des réseaux), etc.

Contrôleur en ligne. Le contrôleur implémenté pour la voiture n'est pas tout à fait satisfaisant – probablement en partie à cause de la fonction coût utilisée, qui rend le contrôle peu stable. En effet, le calcul de la trajectoire de contrôle étant long à exécuter par rapport à la cadence de contrôle de la voiture, apparaît une contrainte de synchronisation : le calcul du contrôle est lancé lorsque la voiture est à une position donnée, mais se finit lorsque la voiture est à une position suivante – il n'est donc plus entièrement valable. Il serait intéressant d'étudier plus en détails ce mécanisme, et d'implémenter un contrôleur en ligne robuste et efficace.

Bibliographie

[1] Andrea Willige. When will there be a self-driving car in your driveway ? https ://www.weforum.org/, 2017.

[2] Chris Woodyard. Self-driving car sales will explode. https ://www.usatoday.com/, 2014.

[3] CNRS-LAAS. www.laas.fr, 2018.

[4] Nicolas Mansard, Del Prete, Mathieu Geisert, Steve Tonneau, and Olivier Stasse. Using a memory of motion to efficiently warm-start a nonlinear predictive controller. *HAL Archives Ouvertes*, Rapport LAAS n ° 17347, 2017.

[5] ROS. Powering the world's robots. http ://www.ros.org/, 2018.

[6] Gazebo. Robot simulation made easy. http ://gazebosim.org/, 2018.

[7] ACADO. Toolkit for automatic control and dynamic optimization. http ://acado.github.io/.

[8] GNU. Coding standards. https ://www.gnu.org/prep/standards/.

[9] PEP8. Style guide for python code. https ://www.python.org/dev/peps/pep-0008/.

[10] Travis. Test and deploy with confidence. https ://travis-ci.org/.

[11] Mattermost. Open source, private cloud messaging. https ://about.mattermost.com/.

[12] Yang Wang and Stephen Boyd. Fast model predictive control using online optimization. *IEEE Transactions on control systems technology*, 18(2) :267–278, 2010.

[13] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv :1509.02971*, 2015.

[14] Igor Mordatch and Emo Todorov. Combining the benefits of function approximation and trajectory optimization. *Robotics : Science and Systems*, 2014.

Glossaire des sigles et acronymes

PIE	*Projet Ingénierie et Entrepreneuriat*
ROS	*Robot Operating System*
IREPA	*Iterative Roadmap Extension and Policy Approximation*
GUI	*Graphical User Interface*
WBS	*Work Breakdown Structure*
PBS	*Product Breakdown Structure*
MPC	*Model Predictive Control*
PRM	*Probabilistic RoadMap*
RN	*Réseau de Neurones*
OCP(S)	*Optimal Control Problem (Solver)*

Annexe A

Tableaux des tâches

Comme évoqué à la fin du chapitre de gestion de projet, on propose ici deux tableaux récapitulant les tâches liées au projet. On les présente sous la forme de deux tableaux : l'un présentant les tâches par jalon (A.1) et l'autre les présentant par lot de travail (A.2).

Jalo	Lot de travail	Nom de la tâche	Volume horaire (pp	Nb personn	Total heures	Total heures lot de travail	Total heures jalon
J0	ALL	Installations diverses (ACADO, ROS, Gazebo, IntelTeleop2017)	3	5	15		
	ALL	Mise en place outils (GitHub, Framateam, Travis)	1	2	2		
	ALL	Définir les normes et méthodes (codage, Git)	2	5	10	27	
							27
J1	ALL	Prendre en main l'architecture ROS (tutoriel)	6	5	30		
	ALL	Spécifier la conception du code	3	5	15		
	ALL	Comprendre la conception du code de N.Mansard	3	5	15	60	
	IREPA	Comprendre l'implémentation du RN (avec TensorFlow)	2	2	4		
	IREPA	Comprendre l'implémentation de la PRM	3	3	9	13	
	CO	Comprendre l'implémentation du contrôle optimal (ACADO)	3	2	6		
	CO	Prendre en main le format de modèle dynamique	5	2	10		
	CO	Implémenter et valider modèle P(x,y)	3	1	3	19	
	SV	Créer le monde Gazebo	3	1	3		
	SV	Créer les obstacles	3	1	3		
	SV	Créer la voiture	3	1	3		
	SV	Lier ROS et Gazebo (service)	3	2	6	15	
	ALL	Intégrer & valider	5	5	25	25	
							132
J2	IREPA	Adapter la PRM au modèle P(x,y,theta)	5	2	10		
	IREPA	Adapter (et valider) le RN (I/O) au modèle P(x,y,theta)	3	2	6		
	IREPA	Implémenter et valider IREPA	10	2	20	36	
	CO	Implémenter un modèle P(x,y,theta)	3	1	3		
	CO	Proposer et implémenter un modèle pour J3	3	2	6	9	
	SV	Lier ROS et Gazebo (plugin)	5	2	10	10	
	ALL	Intégrer & valider	5	5	25	25	
							80
J3	IREPA	Adapter IREPA au modèle J3	5	2	10		
	IREPA	Évaluer les possibilités d'optimisation du RN	10	2	20	30	
	CO	Proposer et implémenter un modèle pour J4	5	1	5	5	
	SV	Définir d'autres géométries d'obstacles	6	2	12	12	
	ALL	Intégrer & valider	5	5	25	25	
							72
J4	IREPA	Adapter IREPA au modèle J4	5	2	10	10	
	CO	Valider modèle J4	3	1	3	3	
	ALL	Intégrer & valider	5	5	25		
	ALL	Développer outils visu (graphes, trajectoires)	10	5	50	50	
							63

FIGURE A.1 – Tâches par jalon

Lot de travail	Jalon	Nom de la tâche	Volume horaire (pp	Nb personnes	Total heures	Total heures par jalo
ALL	J0	Installations diverses (ACADO, ROS, Gazebo, IntelTeleop2017)	3	5	15	
	J0	Mise en place outils (GitHub, Framateam, Travis)	1	2	2	
	J0	Définir les normes et méthodes (codage, Git)	2	5	10	27
	J1	Prendre en main l'architecture ROS (tutoriel)	6	5	30	
	J1	Spécifier la conception du code	3	5	15	
	J1	Comprendre la conception du code de N.Mansard	3	5	15	
	J1	Intégrer & valider	5	5	25	85
	J2	Intégrer & valider	5	5	25	25
	J3	Intégrer & valider	5	5	25	25
	J4	Développer outils visu (graphes, trajectoires)	10	5	50	
	J4	Intégrer & valider	5	5	25	75
						237
IREPA	J1	Comprendre l'implémentation du RN (avec TensorFlow)	2	2	4	
	J1	Comprendre l'implémentation de la PRM	3	3	9	13
	J2	Adapter la PRM au modèle P(x,y,theta)	5	2	10	
	J2	Adapter (et valider) le RN (I/O) au modèle P(x,y,theta)	3	2	6	
	J2	Implémenter et valider IREPA	10	2	20	
	J3	Adapter IREPA au modèle J3	5	2	10	46
	J3	Évaluer les possibilités d'optimisation du RN	10	2	20	20
	J4	Adapter IREPA au modèle J4	5	2	10	10
						89
CO	J1	Comprendre l'implémentation du contrôle optimal (ACADO)	3	2	6	
	J1	Prendre en main le format de modèle dynamique	5	2	10	
	J1	Implémenter et valider modèle P(x,y)	3	1	3	19
	J2	Proposer et implémenter un modèle pour J3	3	2	6	6
	J3	Proposer et implémenter un modèle pour J4	5	1	5	5
	J4	Valider modèle J4	3	1	3	3
						33
SV	J1	Créer le monde Gazebo	3	1	3	
	J1	Créer les obstacles	3	1	3	
	J1	Créer la voiture	3	1	3	
	J1	Lier ROS et Gazebo (service)	3	2	6	15
	J2	Lier ROS et Gazebo (plugin)	5	2	10	10
	J3	Définir d'autres géométries d'obstacles	6	2	12	12
						37

FIGURE A.2 – Tâches par lot de travail