



# Rapport d'élève ingénieur

## Projet de troisième année

Filière F1 : Informatique des systèmes interactifs pour  
l'embarqué, la robotique et le virtuel

---

### Etude et mise en œuvre de la solution OpenPilot sur un véhicule autonome

---

Présenté par : *Lucas PICHON et Louis LEROY*

*Responsable ISIMA* : Frédéric RENAUT

*Date de la soutenance* : 26/03/2025

*Responsables entreprise* : Romuald AUFRERE et Serge ALIZON    *Durée du projet* : 120 heures

Campus des Cézeaux, 1 rue de la Chebarde, TSA 60125, 63178 Aubière CEDEX



## Résumé

Dans le cadre de leur collaboration, l’Institut Pascal et Michelin souhaitent explorer la possibilité d’intégrer le système **OpenPilot** pour le contrôle d’un **véhicule autonome**. OpenPilot est un projet open source développé par la société **COMMA**, conçu pour permettre la conduite autonome d’un véhicule en s’appuyant sur le suivi de ligne via un dispositif, nommé **COMMA**, directement relié au **bus CAN** de la voiture.

L’objectif de ce projet est d’évaluer les performances du système et d’en identifier les limitations pour son intégration dans un processus complexe de contrôle d’un **véhicule autonome**. Afin de démontrer la possibilité d’envoyer nos propres **commandes** à la voiture via le dispositif **COMMA**, nous avons opté pour l’utilisation d’une manette de Xbox.

Nous avons débuté par l’étude du code source du projet afin d’identifier les différents modules présents, leurs rôles respectifs, et de comprendre la communication entre ces modules. Ensuite, nous nous sommes concentrés sur la partie **contrôle** pour localiser précisément l’endroit où les commandes sont envoyées au **bus CAN** du véhicule, afin d’intégrer nos propres commandes et d’effectuer quelques modifications sur le comportement du véhicule. Pour ce faire, nous avons d’abord travaillé sur une **simulation** au sein du projet OpenPilot. Une fois la **simulation** validée, nous avons implémenté notre code directement sur le dispositif **COMMA** et l’avons testé en conditions réelles.

Nous avons pu valider notre démonstration sur la **simulation** avec l’envoi de **commandes** au véhicule via une manette de Xbox. Cependant, la démonstration en conditions réelles n’a pas abouti comme prévu.

**Mots-clés** : OpenPilot, véhicule autonome, COMMA, bus CAN, contrôle, commandes, simulation

## Abstract

For their collaboration , the Pascal Institute and Michelin want to explore the possibility to integrate the system **OpenPilot** in order to control an **autonomous vehicle**. OpenPilot is an open-soucre project developed by the COMMA company , designed to enable autonomous driving of this vehicule using line tracking via the device named **COMMA** too , directly plugged into the **CAN bus** of the car.

The objective of this project to evaluate the source code of the project in order to identify its limitation for its integration into a complex process controlling a **autonomous vehicle**. So much so that we can demonstrate the possibility of using the **COMMA** device to send our own **commands** to the car. We chose to try and connect a Xbox controller to send this commands.

We started by studying the code to identify the differents modules , their respective roles and understand the communciation between those modules. Then , we focused on the **controlling** part to locate precisely in which file the commands are sent to the car’s **CAN bus** , doing so we expected to be able to send our own commands and slightly change the car’s behavior.

To do so we strated to work with the simulation available in the OpenPilot project. Once the **simulation** was working , we loaded the code on the **COMMA** device and tested our code in real conditions.

We have been able to ensure that the code was working on the **simulation** by sending commands to the vehicule and driving it through the simulated road. However the real demonstration was unsuccessfull.

**Keywords :** OpenPilot , autonomous vehicule , COMMA , CAN bus , control, commands ,simulation

## **Remerciements**

Nous tenons à exprimer notre sincère gratitude envers nos tuteurs de projet, Monsieur Romuald Aufrere et Monsieur Serge Alizon, pour leur accompagnement constant, leurs conseils avisés et leur disponibilité. Leur expertise et leur confiance ont été des atouts précieux tout au long de ce projet.

Nous souhaitons aussi remercier Monsieur Jean-Claude Kachel avec qui nous avons eu le privilège de travailler. Nous voulons aussi remercier Monsieur Malaterre Laurent avec qui nous avons pu travailler sur l'environnement de simulation Carla.

# Table des matières

Résumé . . . . .	iii
Abstract . . . . .	iii
Remerciements . . . . .	v
Table des figures . . . . .	viii
<b>Introduction</b>	<b>1</b>
<b>1 Mise en contexte</b>	<b>3</b>
1.1 Présentation du projet . . . . .	3
1.2 Les objectifs du projet . . . . .	3
1.3 Présentation du matériel utilisé . . . . .	4
1.4 Impact environnemental et sociétal . . . . .	5
1.4.1 Environnemental . . . . .	5
1.4.2 Sociétal . . . . .	5
1.5 Diagrammes de Gantts . . . . .	6
<b>2 Réalisation et conception</b>	<b>8</b>
2.1 Analyse du code source d'OpenPilot . . . . .	8
2.1.1 Analyse des différents modules . . . . .	8
2.1.2 Communication entre les différents modules . . . . .	9
2.1.3 Analyse du fichier controls.py . . . . .	12
2.2 Travail sur la simulation d'OpenPilot . . . . .	12
2.2.1 Présentation de l'environnement de simulation d'OpenPilot . . . . .	12
2.2.2 Modification du gain de direction . . . . .	14
2.2.3 Envoi de nos propres commandes via une manette . . . . .	15
2.2.3.1 Le script de la manette (joystick.py) . . . . .	15
2.2.3.2 Modification du script controls.py . . . . .	15
2.3 Intégration de notre propre code sur le dispositif Comma . . . . .	16
2.3.1 Modification de l'interface du Comma . . . . .	16
2.3.2 Intégration de la communication SSH . . . . .	17
2.3.2.1 Scripts d'envoi des commandes (SSH.py) . . . . .	17
2.3.2.2 Scripts de réception des commandes (receiver.py) . . . . .	19
2.3.3 Déploiement du code sur le dispositif Comma . . . . .	20
2.3.3.1 Installation et compilation du projet . . . . .	20
2.3.3.2 Configuration du répertoire GitHub . . . . .	21
2.3.3.3 Exécuter votre programme sur le Comma . . . . .	21
<b>3 Résultats et discussion</b>	<b>22</b>
3.1 Tests effectués . . . . .	22
3.1.1 Tests dans la simulation . . . . .	22

3.1.2	Tests en conditions réelles . . . . .	23
3.2	Discussion . . . . .	25
3.2.1	Améliorations . . . . .	25
3.2.2	Ajouts futurs . . . . .	26
3.2.2.1	Trouver la limitation de la LKS . . . . .	26
3.2.2.2	Se tourner vers un autre dispositif : le Panda . . . . .	26
<b>Conclusion</b>		<b>29</b>
<b>Bibliographie</b>		<b>ix</b>
<b>Annexes</b>		<b>x</b>

# Table des figures

1.1	Image du dispositif Comma . . . . .	4
1.2	Diagramme de Gantt Prévisionnel . . . . .	6
1.3	Diagramme de Gantt Réel . . . . .	7
2.1	Graphe représentant la communication entre les différents scripts d'OpenPilot .	10
2.2	Graphe représentant la communication entre les différents scripts de la partie contrôle . . . . .	11
2.3	Environnement de simulation d'OpenPilot . . . . .	13
2.4	Message d'alerte d'OpenPilot . . . . .	14
2.5	Commandes de la manette . . . . .	15
2.6	Ajout du champ steer max dans les menus du Comma . . . . .	17
2.7	Schéma de l'architecture de communication . . . . .	18
2.8	Configuration du Comma pour la communication SSH . . . . .	19
3.1	Contrôle du véhicule via la manette dans la simulation . . . . .	23
3.2	Installation du Comma dans la Skoda Karoq . . . . .	23
3.3	Sélection d'une nouvelle branche du projet dans le Comma . . . . .	26
3.4	Panda . . . . .	27

# Introduction

La robotisation des véhicules est aujourd’hui en pleine expansion, soutenue par des avancées technologiques significatives dans des domaines tels que l’intelligence artificielle, les capteurs, et l’implémentation d’algorithmes de commande sophistiqués. En conséquence, un nombre croissant d’industriels se tournent vers la robotique mobile afin d’analyser le potentiel de ce domaine dans leur milieu.

C’est le cas de Michelin, qui s’est tourné vers l’Institut Pascal, spécialisé dans le domaine de la robotique, afin d’analyser et de répondre à un problème majeur auquel l’entreprise est confrontée : l’automatisation des tests sur les pneus. Actuellement, les tests effectués par Michelin sur leurs pneus sont non seulement répétitifs, mais également fastidieux et peu ergonomiques pour les opérateurs humains. En robotisant un véhicule pour réaliser ces tests, Michelin pourrait améliorer l’efficacité et la précision des évaluations en garantissant une meilleure répétabilité des tests. Cette automatisation permettrait aussi de libérer les ressources humaines des tâches monotones.

Aujourd’hui, l’Institut Pascal est capable de rendre un véhicule autonome et l’a déjà réalisé avec succès. Cependant, les coûts associés à une telle opération restent relativement élevés et chaque modèle de véhicule présente des spécificités techniques propres. Afin d’envisager une robotisation du véhicule (contrôle du véhicule par un ordinateur), plusieurs solutions sont envisageables. La solution exploitée pour le moment par l’Institut Pascal est d’effectuer un reverse engineering de l’ensemble du bus de contrôle du véhicule (bus CAN) pour comprendre l’interaction entre les données circulant sur ce bus et les organes de contrôle/commande du véhicule. Cette opération est longue et fastidieuse et il n’est pas possible de réaliser la même opération pour deux modèles de véhicules différents.

Pour répondre à ce problème de non-répétabilité, une solution a été développée : le projet Openpilot, développé par la société Comma.ai. Ce projet open-source permet de contrôler via un organe externe un certain nombre de marques et modèles de véhicules. Pour cela, ils ont développé un boîtier, appelé aussi Comma, qui se connecte directement au bus CAN de la voiture. Ce boîtier transmet ensuite des commandes au véhicule en analysant les lignes blanches de la route, grâce à un programme disponible sur leur GitHub [1]. L’utilisation d’un tel dispositif permet de robotiser facilement une large gamme de véhicules. Cependant, il est souhaitable que les commandes envoyées au véhicule ne se limitent plus à celles calculées à partir du suivi de ligne, mais qu’elles soient spécifiquement adaptées aux besoins du projet envisagé. L’objectif est de pouvoir envoyer des commandes précises permettant de valider les tests réalisés sur les pneus.

Notre objectif principal pendant ce projet est de démontrer la capacité d’envoyer des commandes personnalisées au véhicule via le dispositif Comma. Pour ce faire, nous avons choisi d’utiliser une manette Xbox, avec les gâchettes pour contrôler l’accélération et les joysticks

pour l'orientation du véhicule. La première étape de notre travail a consisté en une analyse approfondie du code source d'OpenPilot afin de mieux comprendre son fonctionnement. La seconde partie de notre travail s'est concentrée sur une première limitation rencontrée lors des tests du Comma : la gestion de la direction dans des virages serrés. Nous avons ensuite travaillé sur l'intégration du contrôle du véhicule via la manette et la communication avec le boîtier. Enfin, nous avons intégré notre propre programme sur le boîtier Comma et l'avons testé dans des conditions réelles sur les pistes de Ladoux.

# 1 Mise en contexte

## 1.1 Présentation du projet

Pour notre projet de fin d'études, nous avons choisi de travailler sur le projet OpenPilot. Ce dernier a été proposé à ISIMA en octobre, suite à la récente découverte par l'Institut Pascal de la solution OpenPilot, qui répondait parfaitement à leurs besoins. Plutôt que de faire appel à un stagiaire, ils ont préféré confier ce projet aux étudiants d'ISIMA.

OpenPilot est un projet open-source développé par la société Comma.ai, visant à donner des fonctions avancées d'aide à la conduite à certains véhicules en utilisant des caméras embarquées et le bus CAN de la voiture. Ce projet de grande envergure bénéficie d'une communauté active et massive sur les réseaux, ce qui assure son évolution continue mais aussi sa complexité aux premiers abords. Les deux principaux langages de programmation utilisés sont le C++ ainsi que le Python [1].

Ce projet s'est déroulé dans le cadre d'une collaboration entre l'institut Pascal et l'entreprise Michelin Clermont-Ferrand et Michelin Espagne, afin de permettre une automatisation partielle de certains tests des pneus de voiture.

Pour nous montrer l'objectif de ce projet, nous avons eu une démonstration du travail qui avait été effectué sur une Renault Zoe par l'Institut. Cela nous a permis d'avoir une idée de ce qu'est un bus CAN et une vision plus réelle de ce qu'est une voiture robotisée.

Lorsque le projet nous a été présenté pour la première fois, il nous a été indiqué que nous étions les premiers à y travailler. Aucune autre équipe n'avait été impliquée auparavant.

## 1.2 Les objectifs du projet

Ce projet avait plusieurs objectifs successifs :

1. **Étude de la solution OpenPilot** : c'est à dire de réussir à comprendre la structure et l'organisation du code ainsi que les différents modules et leur fonctionnement de communication. Pour répondre à cet objectif, on nous a demandé d'écrire une documentation.
2. **Compréhension du fonctionnement du bus CAN** : ensuite, il fallait comprendre le fonctionnement de la communication entre le véhicule et le Comma. A savoir ce qu'est une commande CAN, sa synthaxe et sa structure de données dans le projet OpenPilot.
3. **Qualification de la solution OpenPilot sur véhicule de test** : le troisième objectif était de trouver les possibilités et les limitations qu'OpenPilot disposait dans le contexte du projet.

4. Amélioration du système pour effectuer le contrôle total du véhicule (accélération, freinage, etc.) : puis pour finir, prendre le contrôle de la voiture directement par le bus CAN sans utiliser les méthodes de détection de ligne ni les contrôleurs PID.

L’Institut Pascal est aujourd’hui capable de répondre à ces objectifs sans avoir recours à OpenPilot ou au dispositif Comma. Cependant, l’utilisation d’un tel système permettrait un gain de temps considérable dans le processus de robotisation des véhicules, ainsi qu’une réduction des coûts. En effet, le Comma est réutilisable sur une large gamme de véhicules, ce qui en fait une solution à la fois flexible et économique.

### 1.3 Présentation du matériel utilisé

Dans le cadre de ce projet, nous avons eu l’opportunité de travailler avec le dispositif Comma à plusieurs reprises, ce qui nous a permis d’intégrer notre propre programme dessus. De plus, nous avons eu l’occasion, durant une après-midi, de tester notre programme en conditions réelles sur une voiture compatible avec le dispositif Comma, sur les pistes de Michelin à Ladoux.

Avant ces tests pratiques, nous avons dû travailler dans un environnement de simulation. Cette approche nous a permis de surmonter les limitations liées à l’absence de dispositif Comma, tout en nous permettant de comprendre l’impact de nos modifications sur le comportement du véhicule. Cependant, une première limitation est rapidement apparue : l’environnement de simulation demandait beaucoup de ressources et nos ordinateurs n’étaient pas suffisamment puissants pour le faire fonctionner. Pour résoudre ce problème, l’Institut Pascal nous a mis à disposition un ordinateur plus performant sous le système d’exploitation Ubuntu pour être compatible avec OpenPilot.

Aussi nous avons pu travailler sur le dispositif Comma que vous pouvez apercevoir dans la figure 1.1. Ce dispositif se connecte au bus CAN de la voiture et fait l’interface entre le code qui y est implanté et la voiture. En d’autres termes, il envoie des consignes aux organes du véhicule, via le bus CAN, comme par exemple le régulateur de vitesse, le suivi de ligne ou encore l’assistance de freinage [1].



FIGURE 1.1 – Image du dispositif Comma

Ce dispositif possède plusieurs caractéristiques qui nous ont été utiles :

1. **Un port USB 3** : qui nous a servi pour effectuer nos premiers tests.
2. **Le Wi-Fi** : qui a permis la connexion SSH pour les tests en conditions réelles.
3. **Un port OBD-C** : c'est un port qui permet la connexion à un bus CAN par l'USB-C

## 1.4 Impact environnemental et sociétal

L'intégration d'OpenPilot dans les cycles de tests de Michelin présente des impacts significatifs sur l'environnement et la société.

### 1.4.1 Environnemental

Sur le plan environnemental, l'utilisation d'un système comme OpenPilot pourrait potentiellement contribuer à une conduite plus fluide, en limitant les accélérations et les freinages brusques, ce qui pourrait avoir un impact positif sur la consommation de carburant et, par conséquent, sur les émissions de CO<sub>2</sub>.

De plus, l'adoption de technologies comme OpenPilot pourrait améliorer la fiabilité des tests et ce faisant donner des résultats plus exploitables. Une meilleure précision des tests pourrait permettre de réduire le nombre de pneus prototypes et donc limiter le gaspillage et la production de déchets liés à ces tests.

### 1.4.2 Sociétal

D'un point de vue sociétal, avoir OpenPilot effectuant ces tests pourrait améliorer la sécurité routière. En effet, OpenPilot pourrait accélérer le développement de pneus plus fiables et résistants aux différentes conditions météorologiques. Cela bénéficierait non seulement aux conducteurs, mais aussi aux autres usagers de la route.

Ce projet présente également des avantages notables en santé et sécurité (SHS) pour les opérateurs. En effet, les tests réalisés manuellement sont souvent longs, répétitifs et monotones, ce qui peut nuire à la vigilance et au confort des techniciens. De plus, certains essais (comme les tests de freinage brusque) impliquent des efforts physiques importants et déséquilibrés, pouvant à terme provoquer des troubles musculo-squelettiques (TMS). L'automatisation partielle de ces tests via le Comma permettrait donc de réduire ces risques, tout en améliorant le confort et la sécurité des opérateurs impliqués.

Toutefois, si cette technologie venait à être utilisée, cela pourrait soulever des inquiétudes en matière d'emploi nécessitant une adaptation des compétences des travailleurs envers des tâches plus axées sur l'interprétation des données, la mise en place et l'amélioration d'algorithmes de tests.

## 1.5 Diagrammes de Gantt

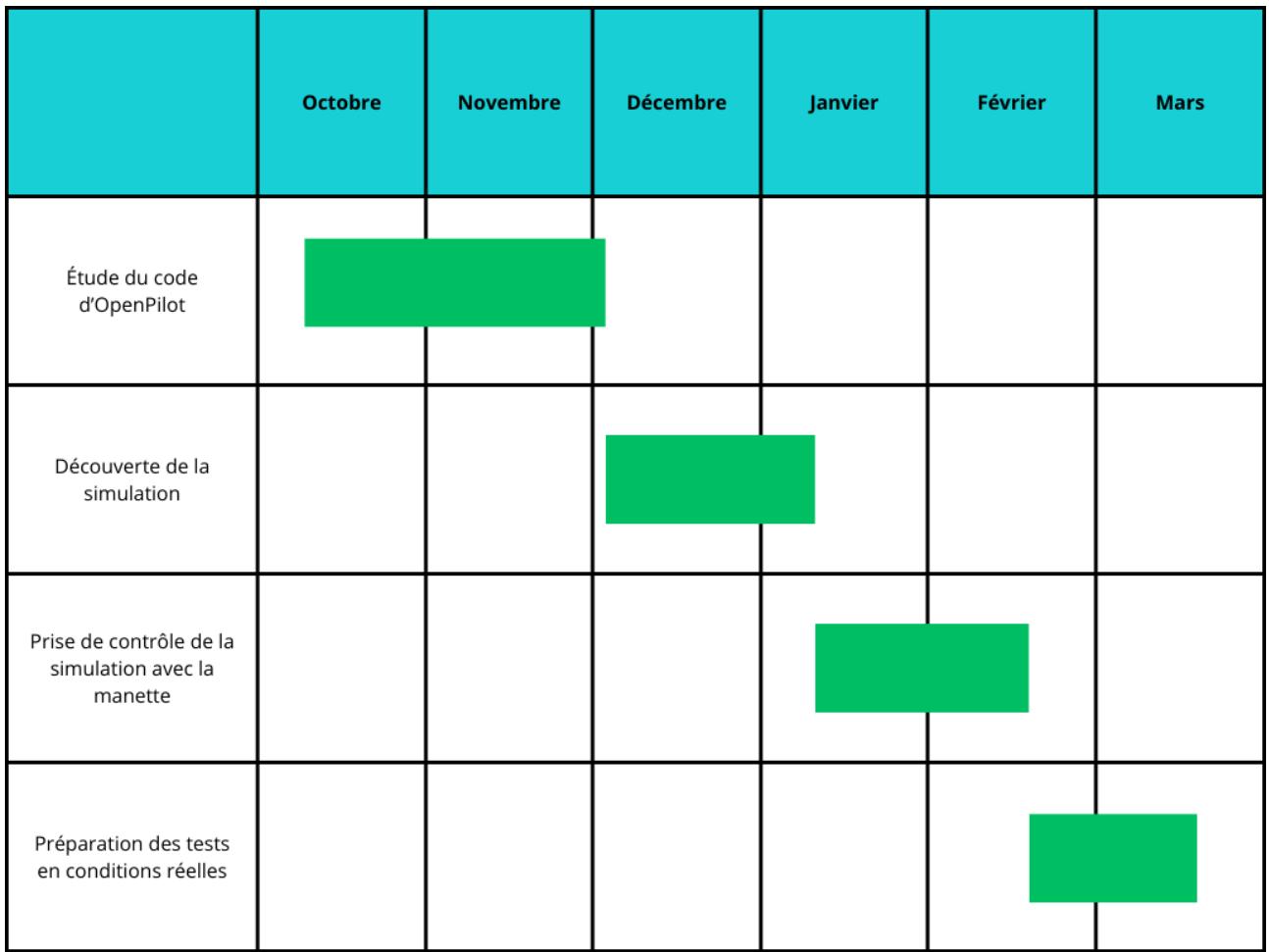


FIGURE 1.2 – Diagramme de Gantt Prévisionnel

Quand nous avons choisi ce sujet, nous avons fait un diagramme de Gantt prévisionnel avec les tâches qui nous ont paru être importantes. Nous avions identifié quatre tâches qui devraient être les tâches principales de ce projet :

1. **Étude du code OpenPilot** : l'objectif de cette tâche est de se familiariser avec la structure du code OpenPilot et d'apprendre à se repérer dans les fichiers.
2. **Découverte de la simulation** : cette tâche consiste à découvrir la simulation, trouver les codes de celle-ci sont associés et voir si ces codes associés semblent être utilisés pour les communications réelles avec le bus CAN.
3. **Prise de contrôle de la simulation avec une manette** : le but de cette tâche est de couper la communication entre les organes de prise de décision d'OpenPilot et le bus CAN afin de pouvoir insérer nos valeurs dans les trames CAN et envoyer nos propres commandes.
4. **Préparation des tests en condition réelles** : ici, nous voulons préparer tous les instruments numériques nécessaires aux tests en conditions réelles de notre projet.

	Octobre	Novembre	Décembre	Janvier	Février	Mars
Étude du code d'OpenPilot						
Découverte de la simulation						
Prise de contrôle de la simulation avec la manette						
Préparation des tests en conditions réelles						

Le diagramme de Gantt Réel montre les tâches réalisées au fil du temps. La tâche "Étude du code d'OpenPilot" a commencé en Octobre et s'est terminée en Mars. Les autres tâches ont débuté plus tard : "Découverte de la simulation" en Janvier, "Prise de contrôle de la simulation avec la manette" en Février, et "Préparation des tests en conditions réelles" en Mars.

FIGURE 1.3 – Diagramme de Gantt Réel

À la fin de notre projet, nous avons réalisé un diagramme de Gantt réel, ce qui nous a permis de nous rendre compte que nous n'avions pas réellement bien estimé la taille d'OpenPilot et la complexité de son agencement. Les modules étant étroitement liés et le GitHub OpenPilot n'explique pas vraiment où sont les choses, mais plutôt comment utiliser OpenPilot directement. Ce qui implique que l'étude du code OpenPilot a en fait été la tâche majeure de ce projet. Et nous a laissé peu de temps pour explorer les tests en conditions réelles.

# 2 Réalisation et conception

## 2.1 Analyse du code source d'OpenPilot

Le projet qui nous a été proposé en était à ses débuts lorsque nous avons été intégrés. Pour répondre aux objectifs qui nous ont été fixés, la première étape a consisté à analyser le code source afin de comprendre l'utilité des différents modules et le fonctionnement de leur communication entre eux. À la demande de nos tuteurs, nous avons également élaboré une documentation qui permet d'avoir une vue d'ensemble du projet. Vous pourrez la retrouver en annexe du rapport.

### 2.1.1 Analyse des différents modules

Au sein d'OpenPilot, les scripts peuvent être classés en cinq catégories dont nous allons brièvement évoquer le rôle [2].

1. **Capteurs** : cette catégorie regroupe un ensemble de scripts permettant de gérer les différents capteurs du projet. Parmi eux, on trouve les caméras embarquées sur le Comma qui assurent l'analyse de la route ainsi que la surveillance de l'attention du conducteur. D'autres capteurs, tels que des gyroscopes, des accéléromètres, des magnétomètres et des capteurs de lumière sont également gérés dans cette catégorie.
2. **Réseaux de neurones** : cette catégorie de scripts permet, entre autres, de prédire le chemin de conduite à partir du flux d'images des caméras de la route ainsi que des entrées de souhaits (changement de voie, etc.). Elle est également responsable de l'analyse de l'attention du conducteur afin de déterminer si ce dernier est toujours en mesure de reprendre le contrôle à tout moment. De plus, elle permet d'identifier les situations où le conducteur doit reprendre le contrôle si nécessaire.
3. **Localisation et calibration** : cette catégorie de scripts est responsable de la gestion et du traitement des données de localisation du véhicule. Elle intègre des informations provenant de plusieurs capteurs, tels que des récepteurs GNSS et des modems GPS, pour fournir des données précises sur la position, la vitesse, l'orientation et les accélérations du véhicule. En combinant ces données avec des filtres de Kalman, elle permet de déterminer la localisation exacte du véhicule dans son environnement. De plus, elle prend en charge l'estimation et l'ajustement dynamique des paramètres du véhicule en fonction des conditions de conduite, tout en calibrant le flux d'images pour assurer une précision maximale. Enfin, cette catégorie inclut des scripts pour gérer la configuration et la communication avec les récepteurs GNSS et les autres dispositifs liés à la localisation du véhicule.
4. **Contrôle** : cette catégorie de scripts est responsable de la gestion et de l'exécution des commandes de conduite du véhicule. Elle inclut plusieurs modules qui interprètent les données des capteurs (radars, caméras, etc.) pour planifier les mouvements du véhicule de manière fluide et optimisée. Le module de planification se divise en deux parties : la planification latérale (direction) et la planification longitudinale (accélérateur/frein),

utilisant un résolveur MPC pour garantir des trajectoires fluides. Les commandes issues de cette planification sont ensuite traduites en signaux spécifiques au véhicule par le module de contrôle, qui fonctionne en boucle fermée pour assurer un contrôle précis du véhicule à 100Hz. De plus, cette catégorie prend en charge la communication avec le véhicule via le bus CAN et ajuste les paramètres du système de contrôle en fonction des spécifications de chaque modèle. Elle inclut également des outils pour mesurer la précision de la direction et gérer les manœuvres longitudinales.

5. **Services systèmes, journalisation et divers** : Cette catégorie regroupe les scripts responsables de la gestion des processus système, de l'enregistrement des logs et de la communication avec les serveurs externes. Elle inclut également des modules pour la gestion de l'interface utilisateur.

La documentation en annexe contient également les explications des différents répertoires du projet, ainsi que des descriptions des scripts les plus importants.

Maintenant que nous avons une vue d'ensemble des modules et de l'architecture du projet, nous allons nous concentrer sur l'interaction entre les différents scripts. Cela nous permettra ensuite de déterminer les fichiers du code qui nous intéressent.

### 2.1.2 Communication entre les différents modules

La communication entre les scripts d'OpenPilot repose sur un système de publication et d'abonnement à des services [3]. La déclaration de tous ces services se trouve dans le fichier `openpilot/cereal/service.py`. La classe `Service` possède trois attributs :

- `should_log` : booléen indiquant si les données de ce service doivent être enregistrées.
- `frequency` : fréquence à laquelle le service fonctionne.
- `decimation` : détermine la fréquence de réduction des données (cet attribut n'est pas obligatoire).

Ensuite, un dictionnaire de services est déclaré, dans lequel chaque service est créé et associé à une chaîne de caractères représentant son nom.

Chaque script peut alors publier ou s'abonner aux différents services déclarés dans le fichier. La syntaxe utilisée dans les scripts python est la suivante (l'idée pour les scripts C reste la même) :

```
self.sm = messaging.SubMaster(['carState', 'liveTorqueParameters'])
self.pb = messaging.PubMaster(['carControl'])
```

Dans ce cas, le script est abonné aux services `carState` et `liveTorqueParameters`, et publie sur le service `carControl` où un autre script pourra venir lire les données envoyées.

Grâce à cette syntaxe répétitive, nous avons développé un script capable d'analyser l'ensemble des fichiers. Ce script récupère à la fois le nom des fichiers et la liste des services auxquels chaque fichier est abonné, ainsi que les services auxquels il souscrit. Une fois que tous les services ont été récupérés, nous avons généré un fichier `.dot`. En utilisant l'outil Graphviz, nous avons pu créer un graphe représentant la communication entre les différents scripts.

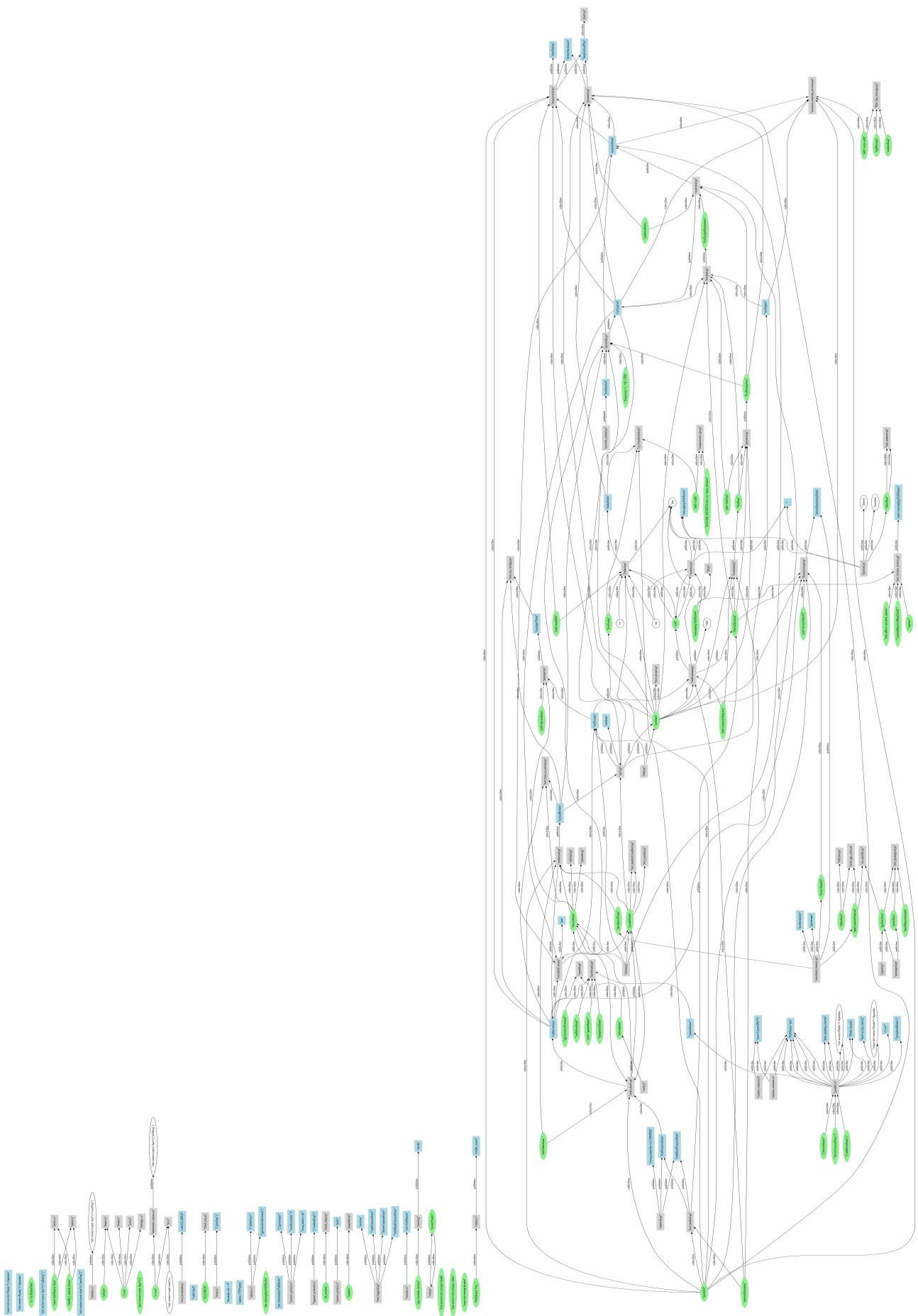


FIGURE 2.1 – Graphe représentant la communication entre les différents scripts d’OpenPilot

Le graphe sur la figure 2.1 présente la liste des scripts (en gris) ainsi que les services (en vert et en bleu). Les relations entre les scripts et les services sont représentées par des flèches, où le sens de la flèche indique si le script publie un service ou s'il y est abonné.

Comme on peut l'observer, le graphe est relativement vaste et comporte de nombreuses relations. Nous avons donc choisi d'analyser plus précisément la partie contrôle d'OpenPilot. Pour cela, nous avons isolé cette section du graphe afin d'obtenir une vue plus ciblée 2.2.

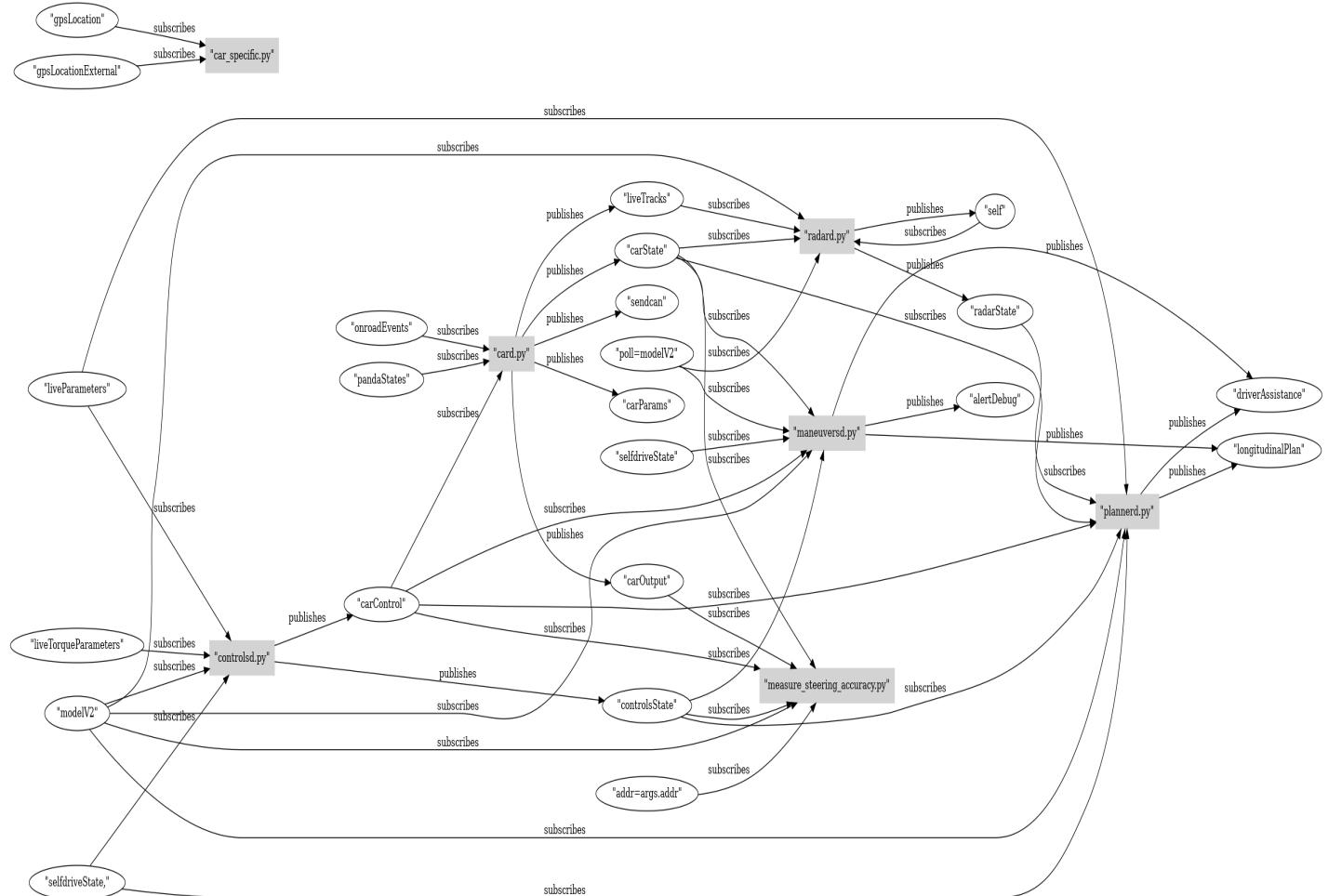


FIGURE 2.2 – Graphe représentant la communication entre les différents scripts de la partie contrôle

L'analyse du graphe nous a permis de comprendre que le contrôle du véhicule est géré par le service `carControl`, publié par le script `controlsd.py`. Le graphe révèle également que le service `carControl` est consommé par le script `card.py`, lequel publie ensuite sur un service appelé `sendcan`.

À partir de cette observation, nous avons décidé d'examiner plus en détail le fichier `controlsd.py`, qui semble, d'après le graphe, être responsable de la génération des commandes de conduite avant leur envoi sur le bus CAN du véhicule. Il s'avère en effet que le script `controlsd.py` est bien celui qui génère les commandes envoyées au véhicule. Cette analyse nous conduit ainsi à la prochaine étape : le travail sur le code de la simulation d'OpenPilot et plus précisément sur les scripts responsables du contrôle du véhicule.

### 2.1.3 Analyse du fichier controls.py

Le fichier `controls.py` est crucial dans notre projet car c'est grâce à lui que nous pourrons, par la suite, envoyer nos propres commandes à la voiture. Nous avons donc passé du temps et il est donc important d'en parler plus en détail ici.

La classe `Controls` prend en compte diverses informations en temps réel, comme l'état du véhicule et les paramètres de conduite, pour ajuster le contrôle longitudinal et latéral du véhicule. La méthode d'initialisation (`__init__`) initialise les divers modules nécessaires au contrôle du véhicule. Elle crée des instances des classes pour les paramètres du véhicule, l'interface du véhicule, ainsi que pour la gestion des communications avec les autres modules. Elle choisit également le type de contrôle latéral en fonction des paramètres du véhicule.

La mise à jour des informations du véhicule est réalisée par la méthode (`update`). Elle récupère les dernières données de calibration et de pose via la messagerie et ajuste les paramètres du calibrateur de pose et recalibre la pose du véhicule en fonction des nouvelles données.

La méthode `state_control` gère le calcul des commandes longitudinales et latérales du véhicule. Elle utilise les informations du véhicule (vitesse, direction, etc.) pour ajuster les courbures, la vitesse et les actions de contrôle en fonction des paramètres du modèle du véhicule. Elle vérifie également si les actionneurs peuvent être activés et met à jour les commandes nécessaires.

La méthode `publish` envoie les commandes de contrôle du véhicule à travers le réseau de messages. Elle envoie en particulier des informations de contrôle longitudinal et latéral ainsi que les informations de planification longitudinale.

La méthode `run` exécute la boucle principale du contrôle du véhicule. Elle appelle les méthodes `update`, `state_control` et `publish` de manière itérative. Elle utilise une instance de `Ratekeeper` pour gérer la fréquence d'exécution.

Les choses les plus importantes à retenir ici sont surtout la fonction `state_control` qui gère l'activation des actionneurs pour le contrôle du véhicule. Dans cette fonction, on s'est intéressé en particulier à une structure nommée `actuators` qui possède deux attributs qui nous intéressent : `accel` et `steeringAngleDeg`. Ces attributs sont essentiels dans notre cas d'usage car `accel` détermine l'accélération longitudinale du véhicule, ce qui permet de gérer la vitesse du véhicule en fonction du plan de conduite, en accélérant ou en freinant selon les conditions de conduite. D'autre part, `steeringAngleDeg` contrôle l'angle de direction du véhicule pour ajuster la trajectoire du véhicule et maintenir sa stabilité dans les virages ou lors des changements de voie. Ces deux attributs sont donc les éléments clés pour la commande du véhicule. Nous reviendrons plus en détail sur ces paramètres un peu plus tard [3].

## 2.2 Travail sur la simulation d'OpenPilot

### 2.2.1 Présentation de l'environnement de simulation d'OpenPilot

Lors de nos recherches sur OpenPilot nous avions vu des vidéos de la simulation et avions compris que cette simulation s'effectuait sous Carla, une plateforme OpenSource de simulation avec des actifs numériques à accès libre comme des voitures ou des bâtiments. Le problème de ce logiciel est qu'il est assez gourmand en ressources. C'est pourquoi nous avions demandé de

l'aide à l'Institut.

Lors de tests à l'Institut, nous nous sommes rendus compte que le simulateur n'était plus Carla mais que, le projet d'OpenPilot propose un simulateur permettant de simuler une voiture dans un environnement virtuel, tout en étant connectée au dispositif Comma.

Cette simulation s'est avérée particulièrement utile, notamment parce que nous n'avons pas eu accès immédiatement au Comma, et qu'une voiture compatible était également requise. Voici un aperçu de l'environnement de simulation 2.3 :

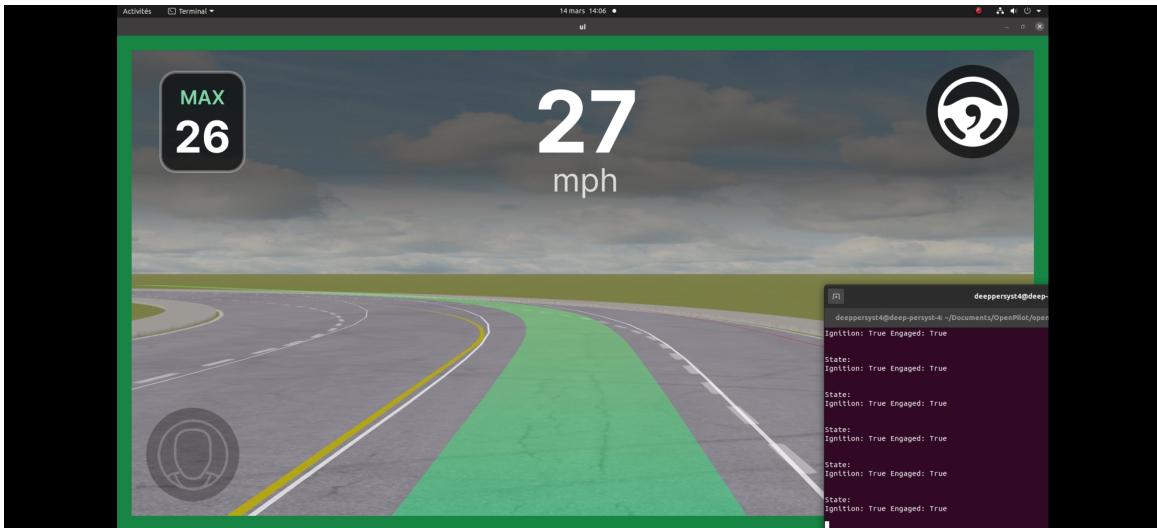


FIGURE 2.3 – Environnement de simulation d'OpenPilot

La simulation est un environnement virtuel représentant une piste de test en forme d'anneau oblong à quatre voies, similaire à un hippodrome, dans lequel le véhicule tourne en continu dans le même sens, principalement vers la gauche. Le tracé vert visible à l'écran correspond au suivi de ligne, c'est-à-dire la trajectoire que le véhicule doit suivre. La vitesse maximale autorisée est affichée en haut à gauche de l'interface et détermine la vitesse cible à atteindre. La vitesse actuelle du véhicule est indiquée au centre en haut de l'écran.

Pour exécuter l'environnement de simulation, il faut se rendre dans le répertoire `tools/sim` et suivre les étapes suivantes [3] :

- Dans un premier terminal, exécuter le script `launch_openpilot.sh`. Cela lance l'interface utilisateur d'OpenPilot, similaire à celle que l'on retrouve sur un dispositif Comma.
- Dans un second terminal, exécuter le script `run_bridge.py`. Ce script initialise l'environnement de simulation et l'affiche sur l'interface précédemment lancée. La vue obtenue est alors identique à celle du Comma, mais avec une route simulée.

Une fois la simulation lancée, plusieurs commandes clavier permettent d'interagir avec le véhicule simulé et de contrôler différents aspects de la simulation. Voici un aperçu des principales commandes disponibles [3] :

- **1** : relance le régulateur de vitesse ou augmente la vitesse de consigne.
- **2** : définit la vitesse de croisière ou diminue la vitesse de consigne.
- **3** : désactive le régulateur de vitesse.

- **r** : réinitialise la simulation.
- **i** : active ou désactive l'allumage du véhicule.
- **q** : ferme la simulation et quitte l'environnement.
- **w, a, s, d** : permettent de contrôler manuellement le véhicule lorsque le Comma est désactivé (commande 3).

Grâce à cette simulation, nous avons pu modifier le code source du projet et d'en observer les modifications afin de répondre à deux objectifs principaux.

### 2.2.2 Modification du gain de direction

Le premier objectif de cette simulation est permettre à la voiture de prendre des virages plus serrés, par exemple en essayant d'augmenter les gains de la commande de direction , ou en levant d'éventuelles limitations relatives à cette commande. Actuellement, OpenPilot impose des limites sur l'actionnement du volant pour garantir la sécurité du véhicule, en régulant l'accélération latérale et la variation de celle-ci. Selon les normes en vigueur, ces limites latérales ne doivent pas permettre un dépassement de plus de 1 mètre en moins de 0,9 seconde sous actionnement maximal [3]. Cette contrainte représente un frein pour les tests que Michelin souhaite réaliser, car elle empêche le véhicule de prendre des virages plus serrés et limite la capacité à tester des comportements plus extrêmes du véhicule.

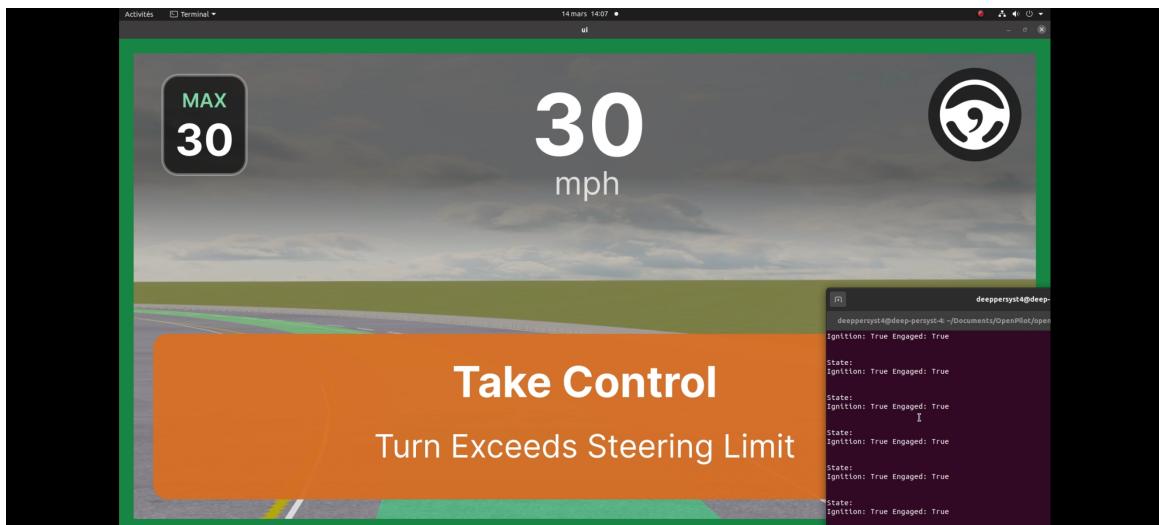


FIGURE 2.4 – Message d'alerte d'OpenPilot

Lorsque le Comma détecte que la voiture s'apprête à prendre un virage trop serré, ce message d'alerte apparaît à l'écran, indiquant à l'utilisateur de reprendre le contrôle immédiatement. Afin de comprendre cette limitation, nous avons effectué des recherches dans les fichiers du projet. Après quelques investigations, nous avons trouvé que la variable définissant l'action maximale sur le volant se situe dans le fichier `latcontrol.py` situé dans le répertoire `openpilot/selfdrive/lib`.

La variable se nomme `steer_max` et est utilisée dans la fonction `_check_saturation`. Cette fonction vérifie si l'action du volant atteint la limite de saturation, en fonction de l'état du véhicule (véhicule dans un virage, véhicule à telle vitesse...) et d'une condition sur un booléen calculé à partir de `steer_max`.

En augmentant la valeur de `steer_max`, nous avons amélioré la maniabilité du véhicule en simulation, permettant d'effectuer des virages plus serrés et plus rapides. Initialement, la simulation affichait un message d'erreur lorsque la voiture prenait un virage à environ 22mph (35km/h). Grâce à cette modification, il est désormais possible d'atteindre environ 75km/h dans les virages sans déclencher d'erreur.

### 2.2.3 Envoi de nos propres commandes via une manette

Le deuxième objectif que nous avions à réaliser avec la simulation est d'envoyer nos propres commandes au véhicule et donc qu'il ne suive plus la route comme il le faisait jusqu'à maintenant. Comme évoqué précédemment, nous avons décidé d'utiliser une manette de Xbox afin de prendre le contrôle de la voiture.

#### 2.2.3.1 Le script de la manette (joystick.py)

Afin de lire les boutons de la manette connectée à l'ordinateur, nous avons utilisé le module `pygame` de Python, qui permet de gérer l'interface utilisateur et d'interagir avec les périphériques d'entrée comme les manettes, claviers et souris. Grâce à lui, nous pouvons récupérer l'état des boutons et des axes de la manette afin de mettre à jour la commande à envoyer au véhicule. La figure 2.5 présente les boutons que nous utilisons pour effectuer les différentes actions de contrôle.

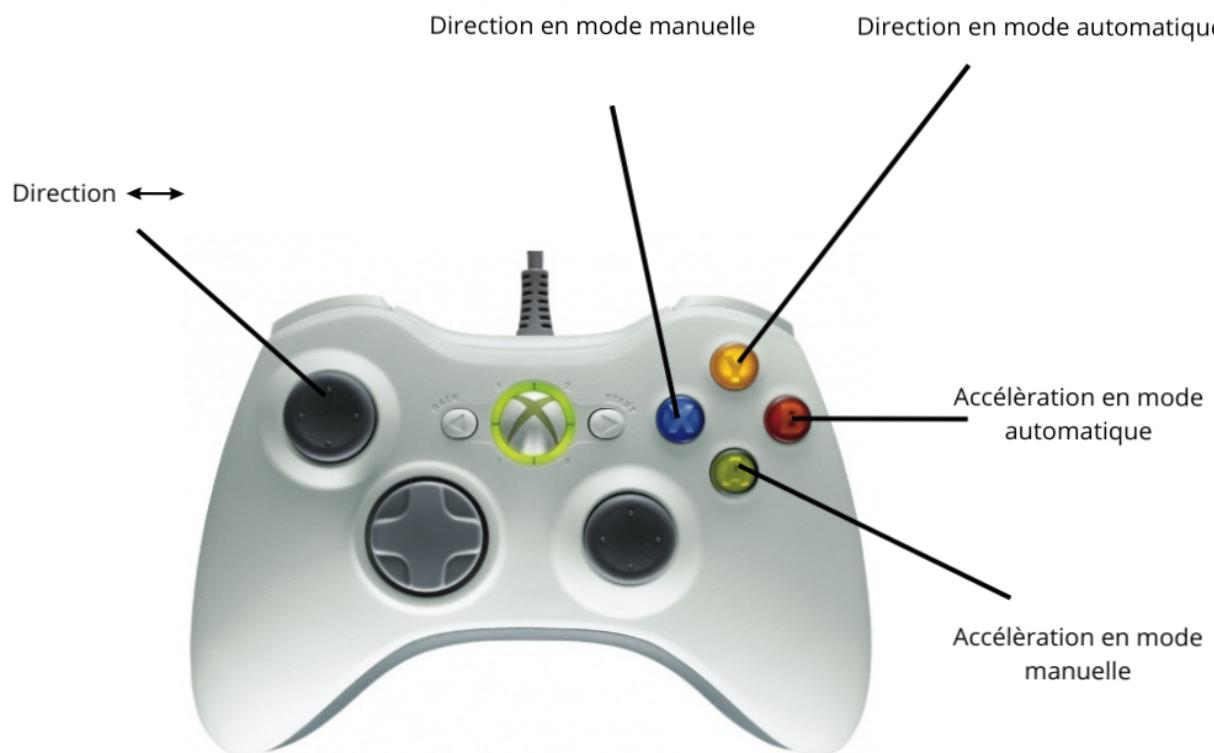


FIGURE 2.5 – Commandes de la manette

#### 2.2.3.2 Modification du script controls.py

Dans cette section du code, nous avons modifié les valeurs de `actuators.accel` et `actuators.steeringAngleDeg` afin d'envoyer nos propres commandes au véhicule. Plus précisément, la variable `actuators.accel` est mise à jour en fonction des valeurs des gâchettes de

la manette. La gâchette de gauche est utilisée pour freiner, tandis que celle de droite permet d'accélérer. Les valeurs lues par notre programme se situent dans l'intervalle  $[-1, 1]$ .

Afin d'obtenir le comportement souhaité, nous avons affecté à `actuators.accel` la formule suivante :

$$\text{actuators.accel} = (\text{RT} + 1) - (\text{LT} + 1)$$

Ainsi, lorsque aucune gâchette n'est pressée ( $\text{RT} = -1$  et  $\text{LT} = -1$ ), l'accélération appliquée est nulle. En fonction de la pression exercée sur les gâchettes, le véhicule peut alors accélérer ou freiner. Il est possible qu'un facteur multiplicatif doive être ajouté afin d'obtenir des accélérations et des freinages plus puissants.

De même, nous avons ajusté `actuators.steeringAngleDeg` afin de commander directement l'angle de direction du véhicule en utilisant la valeur en X du joystick gauche de la manette. Cette valeur étant également comprise entre  $[-1, 1]$ , nous avons appliqué un facteur d'échelle supplémentaire pour obtenir une direction adaptée au contrôle du véhicule. Grâce à ces modifications, nous sommes désormais capables de piloter le véhicule dans la simulation avec la manette.

Maintenant que l'envoi de commandes fonctionne dans la simulation, nous sommes passés à l'étape suivante, qui consiste à déployer notre propre code sur le dispositif Comma. L'objectif est que ce dernier puisse recevoir les commandes de la manette et les transmettre ensuite au bus CAN de la voiture.

## 2.3 Intégration de notre propre code sur le dispositif comma

### 2.3.1 Modification de l'interface du Comma

Dans un premier temps, nous avons travaillé sur l'ajout d'un élément dans l'interface du menu du Comma afin de vérifier que nos modifications avaient bien un impact sur ce dernier. De plus, l'idée de base était de pouvoir modifier la valeur maximale de direction dans les virages grâce à l'ajout de cet élément.

Pour ce faire, nous avons travaillé dans le répertoire UI situé au chemin `selfdrive/ui`. À l'intérieur, un répertoire nous a particulièrement intéressé. Il s'agit du répertoire `qt/offroad`, qui gère les écrans et fenêtres affichées lorsque le véhicule est à l'arrêt (mode *offroad*). Dans ce sous-répertoire, le fichier responsable de la logique et des interactions pour le menu est situé dans `settings.cc` et `settings.h`.

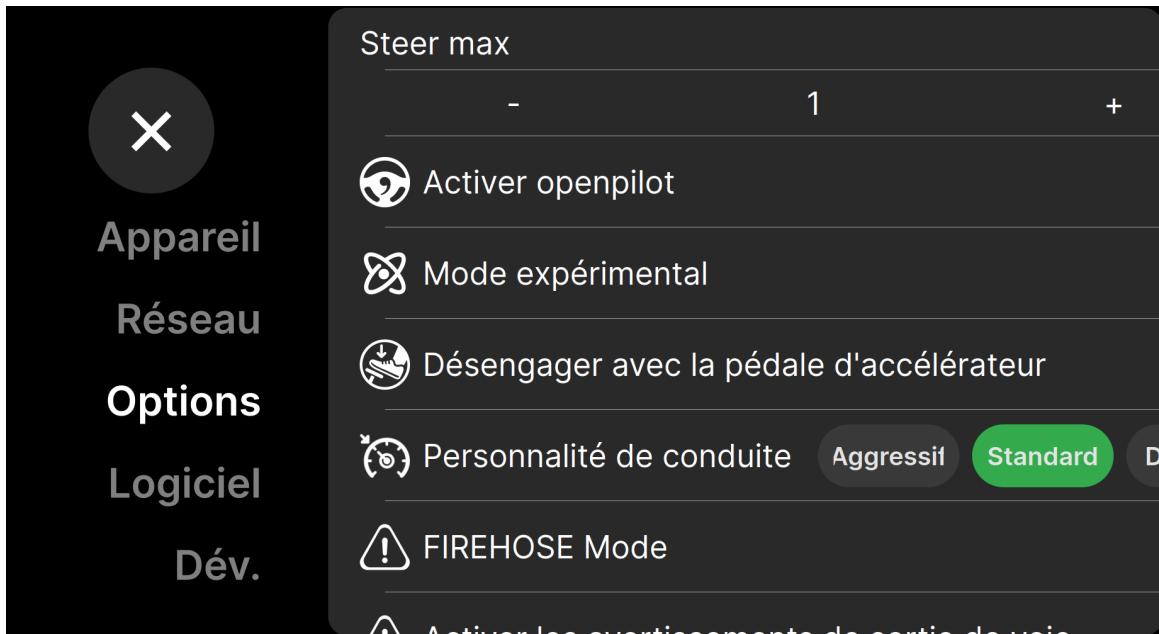


FIGURE 2.6 – Ajout du champ steer max dans les menus du Comma

Sur la figure précédente 2.6, nous avons ajouté un nouveau champ (le premier que l'on peut voir sur la figure 2.6). Il se situe dans la catégorie option et se nomme steer max. Il est possible d'augmenter la valeur ou de la diminuer avec les boutons + et -. Nous n'avons cependant pas approfondi davantage cette idée au cours du projet, car nous avons finalement choisi de fixer la valeur en dur directement dans le code (comme cela est fait dans le projet initial), en définissant une valeur suffisamment élevée pour observer une différence notable dans les virages.

### 2.3.2 Intégration de la communication SSH

Dans la section précédente, la communication avec le dispositif `comma` s'effectuait via une liaison série, la manette étant directement reliée à l'ordinateur. Cependant, dans un contexte réel, cette approche n'est pas envisageable, car l'appareil ne dispose d'aucun port permettant de connecter un périphérique externe.

La solution que nous avons retenue consiste à établir une communication avec le dispositif via une connexion SSH. Autrement dit, la manette est connectée à un ordinateur qui communique en SSH avec le Comma et transmet en continu les données de la manette. De son côté, le Comma récupère en permanence ces valeurs afin de mettre à jour les variables essentielles mentionnées précédemment, à savoir `actuators.accel` et `actuators.steeringAngleDeg`.

Il est important de préciser que le Comma et l'ordinateur doivent être connectés sous le même réseau Wi-Fi afin que les prochaines étapes fonctionnent.

Afin de mettre en place cette communication, nous avons développé deux scripts principaux, en plus de celui qui permet de lire les valeurs de la manette.

#### 2.3.2.1 Scripts d'envoie des commandes (SSH.py)

Dans cette section, nous décrivons la mise en place d'une communication entre un ordinateur et un dispositif Comma via une connexion SSH. Ce script permet de lire les valeurs de la manette de jeu, de les envoyer en temps réel au Comma, et de récupérer les réponses du dispositif pour le débogage. Il doit être exécuté sur un ordinateur sur lequel une manette est connectée.

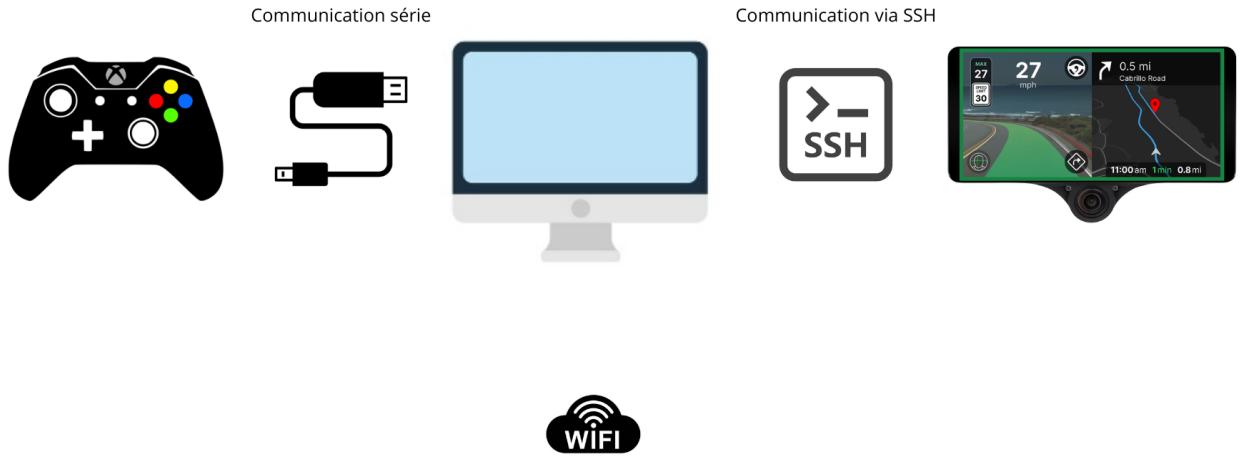


FIGURE 2.7 – Schéma de l'architecture de communication

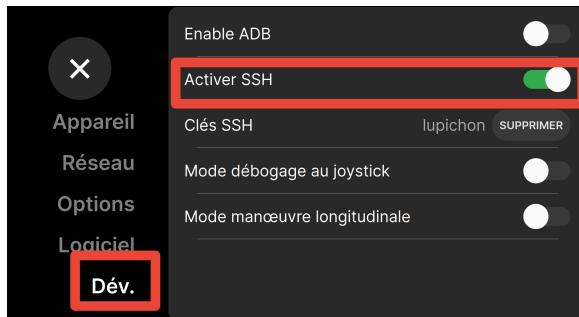
Le script commence par initialiser la manette de jeu en appelant la fonction `init_joystick`, qui permet de préparer les entrées de la manette pour la lecture des données. Ensuite, il initialise la connexion SSH avec le dispositif Comma en utilisant son adresse IP, le nom d'utilisateur (ici `comma`) et un mot de passe (vide dans ce cas) pour établir une communication avec le dispositif [3].

```
COMMA_IP = "192.168.*.*"
USERNAME = "comma"
PASSWORD = ""
```

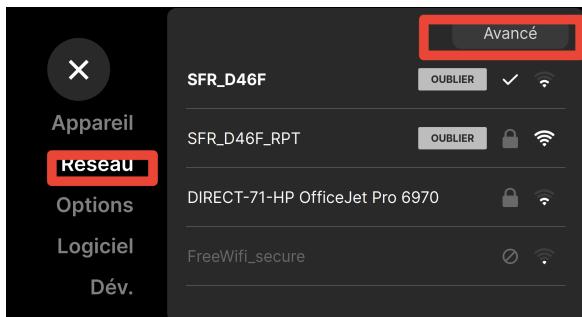
L'adresse IP du dispositif Comma peut être facilement obtenue dans la section avancée des paramètres réseau du menu du Comma. Il ne faut pas oublier aussi d'autoriser la connexion SSH dans les paramètres du Comma. Le script utilise ensuite la bibliothèque `paramiko` pour établir une connexion SSH, suivie de l'ouverture d'un canal de communication sécurisé.

Une fois la connexion établie, le script envoie la commande suivante pour exécuter un script Python directement sur le Comma (script qui sera détaillé dans la section suivante).

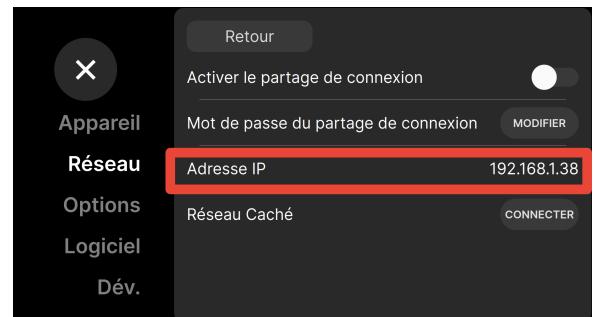
Le script utilise la fonction `read_joystick` pour récupérer les valeurs des axes et des gâchettes de la manette. Les valeurs lues sont formatées en chaîne de caractères avec une précision de deux décimales et envoyées au Comma. Le programme peut être interrompu par l'utilisateur avec un `Ctrl+C`. Dans ce cas, un message de fin est envoyé au Comma, et la connexion SSH est fermée proprement.



(a) Autorisation de la connexion SSH dans la Comma



(b) Emplacement de l'adresse IP du Comma



(c) Adresse IP du Comma

FIGURE 2.8 – Configuration du Comma pour la communication SSH

### 2.3.2.2 Scripts de réception des commandes (receiver.py)

Ce script, que l'on a intégré dans le répertoire `/data/openpilot/selfdrive/controls/receiver.py` du projet OpenPilot, est conçu pour être exécuté sur le Comma. Cependant, ce il n'est pas lancé automatiquement au démarrage du Comma. Il est uniquement activé lorsqu'il reçoit la commande correspondante, envoyée par le script précédent via la connexion SSH. Voici le fonctionnement détaillé du script `receiver.py` :

Le script commence par initialiser plusieurs variables globales, représentant les valeurs lues de la manette et les états des actionneurs (accélérateur et volant).

Le script, une fois lancé, appelle la fonction `start_receiver()`. Il attend que des données soient envoyées via l'entrée standard (`stdin`) et les passe à la fonction `update_values()` pour mettre à jour les variables. Si les données reçues sont incorrectes, une erreur est affichée. Sinon, les valeurs des axes et des boutons sont extraites et les actions appropriées sont appliquées aux états des actionneurs. Cette fonction gère également l'arrêt du script lorsqu'une interruption clavier est effectuée du côté de l'ordinateur. Lorsqu'une telle commande est reçue, le script interrompt l'attente des données et termine proprement son exécution.

La fonction `update_values()` prend en entrée une chaîne de caractères `data`, représentant les valeurs envoyées par le script `SSH.py`. Elle divise cette chaîne en une liste de valeurs flottantes et les affecte aux variables globales. Les boutons de la manette (A, B, X, Y) permettent de sélectionner les commandes utilisées par le Comma. Par défaut, les commandes utilisées par le Comma sont celles de base, c'est-à-dire que le contrôle du véhicule est en mode automatique. Cependant, il est possible de modifier ces commandes grâce à l'utilisation des boutons :

- A : active le contrôle manuelle de l'accélération via la manette

- B : désactive le contrôle manuel de l'accélération (passage en automatique)
- X : active le contrôle manuelle de la direction via la manette
- Y : désactive le contrôle manuelle de la direction (passage en automatique)

L'avantage d'intégrer les boutons est qu'il est désormais possible de sélectionner de manière totalement indépendante le choix entre le contrôle automatique ou manuel de l'accélération et de l'orientation.

La fonction `start_receiver()` gère la réception des données. Elle attend indéfiniment des entrées via `stdin`, les traite à l'aide de la fonction `update_values()` et met à jour les variables correspondantes. Si la commande "STOP" est reçue, le processus d'attente est arrêté et le script termine son exécution.

Ainsi, grâce à ces deux scripts, nous avons pu réaliser une connexion SSH entre un ordinateur et le dispositif Comma afin d'envoyer les commandes de la manette. Il faut maintenant passer à l'étape suivante qui consiste à implémenter notre propre code sur le dispositif Comma.

### 2.3.3 Déploiement du code sur le dispositif Comma

Cette section détaille les étapes nécessaires que nous avons réalisées pour déployer un projet OpenPilot personnalisé sur le dispositif Comma.

#### 2.3.3.1 Installation et compilation du projet

OpenPilot s'installe de préférence sur Ubuntu 24.04, avec une compatibilité partielle sur macOS et via WSL sous Windows. Les autres systèmes nécessitent des adaptations spécifiques.

Pour commencer, il est nécessaire de cloner le dépôt OpenPilot . Pour réaliser un clonage partiel du dépôt, il faut exécuter la commande suivante [2] :

```
git clone --filter=blob:none --recurse-submodules --also-filter-submodules https://github.com/commiaai/openpilot.git
```

Ou si l'on souhaite réaliser un clonage complet, il faut exécuter :

```
git clone --recurse-submodules https://github.com/commiaai/openpilot.git
```

Une fois le dépôt cloné, il faut accéder au répertoire `openpilot` et exécuter le script d'installation avec la commande :

```
tools/op.sh setup
```

Après l'installation, il faut récupérer les fichiers gérés par `Git LFS` avec la commande suivante :

```
git lfs pull
```

Ensuite, il faut activer l'environnement virtuel Python en utilisant la commande :

```
source .venv/bin/activate
```

Enfin, pour compiler le projet, il faut utiliser la commande **scons** suivante :

```
scons -u -j$(nproc)
```

À l'issue de cette procédure, nous sommes en mesure de lancer OpenPilot en exécutant le script `launch_openpilot.sh` situé à la racine du projet. La prochaine étape consiste à configurer un dépôt Github afin d'y pousser notre propre version d'OpenPilot.

### 2.3.3.2 Configuration du répertoire GitHub

Après avoir effectué un fork sur GitHub du projet OpenPilot de Comma.ai, il faut pousser les modifications et configurer le dépôt distant avec les commandes suivantes :

```
git remote rm origin
git remote add origin git@github.com:<votre-nom-d'utilisateur>/openpilot.git
git add .
git commit -m "premier commit"
git push --set-upstream origin master
```

### 2.3.3.3 Exécuter votre programme sur le Comma

Afin d'utiliser notre clone sur le Comma, il faut réinitialiser le Comma si une installation est déjà présente, puis choisir l'option "custom software" et d'entrer l'URL suivante : `installer.comma.ai/<nom-d'utilisateur-GitHub>/master`.

À la fin de ces étapes, nous sommes en mesure d'établir une communication avec le Comma, en lui envoyant les données de la manette.

Pour cela, il faut suivre les étapes suivantes :

1. **Récupérer l'adresse IP du Comma** et l'inscrire dans le code à l'emplacement prévu.
2. **Connecter une manette** à l'ordinateur.
3. **Exécuter le script** de communication.

Si tout fonctionne correctement, un message doit s'afficher dans la console du Comma sous la forme suivante :

```
Mis à jour : LX= ... , LY= ... , RX= ... , RY= ... , LT= ... , RT= ...
```

Avec à la place des ... les valeurs des variables mises à jour en temps réel en fonction des actions effectuées sur la manette.

# 3 Résultats et discussion

## 3.1 Tests effectués

### 3.1.1 Tests dans la simulation

Les tests réalisés en simulation se sont révélés concluants. Avant d'envoyer des commandes via la manette, nous avons d'abord effectué plusieurs essais dans l'environnement simulé afin de nous assurer que nous avions correctement identifié les instructions nécessaires. Ces tests nous ont également permis d'observer l'impact des modifications apportées sur le comportement du véhicule.

Par exemple, l'un des premiers tests que nous avons réalisés consistait à faire rouler normalement le véhicule avec le Comma activé, puis, au bout de 10 secondes, à le forcer à s'arrêter. Autrement dit, nous appliquons une valeur de 0 au champ `actuators.accel`. Nous avons également effectué un test similaire sur l'orientation : après 10 secondes, nous empêchions le véhicule de tourner en forçant la valeur de `actuators.steeringAngleDeg` à 0. Cela nous permettait de vérifier que nous avions bien identifié et contrôlé les valeurs influençant la trajectoire du véhicule.

Ces premiers tests s'étant révélés concluants, nous avons ensuite connecté la manette. Les tests réalisés avec celle-ci consistaient à pousser le véhicule dans des conditions extrêmes, c'est-à-dire à des vitesses élevées, avec des freinages brusques et en prenant des virages rapides. Les résultats observés ont montré que nous étions désormais capables de prendre entièrement le contrôle du véhicule en ajustant son accélération et son orientation. Vous pouvez voir dans la figure 3.1 que le véhicule est à une vitesse élevée et au milieu de deux voies, montrant bien que c'est nous qui le contrôlons.

Nous observons dans la simulation que, lorsque nous souhaitons freiner, le véhicule ralentit effectivement. Cependant, nous ne sommes pas certains que ce ralentissement provienne de l'activation du frein lui-même, malgré l'importance de la décélération constatée.



FIGURE 3.1 – Contrôle du véhicule via la manette dans la simulation

Le suivi de ligne fonctionne toujours. C'est la raison pour laquelle il y a toujours le tracé vert sur l'écran. Cependant, le véhicule ne le suit plus comme nous le commandons avec la manette. Il restait maintenant à tester notre programme en conditions réelles, ce qui a été bien plus difficile.

### 3.1.2 Tests en conditions réelles

Le mercredi 5 mars, nous nous sommes rendus sur les pistes de Ladoux, chez Michelin, afin de réaliser les tests précédemment effectués en simulation, mais cette fois-ci dans des conditions réelles. Le véhicule sur lequel le dispositif Comma a été installé est une Skoda Karoq.



FIGURE 3.2 – Installation du Comma dans la Skoda Karoq

Nous avons commencé par nous rendre sur le circuit, puis avons entrepris les tests que nous étions venus réaliser, à savoir, dans un premier temps, tester le contrôle de la voiture via la manette.

Cependant, très rapidement, nous avons rencontré un premier problème. Lorsqu'on activait le Comma, le voyant sur le tableau de bord du véhicule responsable de prévenir d'un problème venant du LKAS (Lane Keeping Assist System) s'allumait, et le message d'erreur suivant apparaissait sur le Comma : "LKAS fault : Restart car to engage". Le véhicule lui-même était en défaut.

Nous avons alors décidé de vérifier, dans un premier temps, que le problème ne venait pas du véhicule, mais bien du code. En conséquence, nous avons réinstallé la version originale d'OpenPilot sur le Comma. En plus d'un temps de téléchargement extrêmement long, nous avons constaté qu'à un certain moment, le Comma perdait la connexion Wi-Fi durant l'installation. Après une certaine attente, nous avons finalement réussi à installer la version originale, et cette version fonctionnait correctement. Le problème semblait donc provenir de notre clone du projet. Nous avons ensuite réinstallé notre clone et restauré le fichier `controls.py` à sa version initiale (c'est-à-dire sans les modifications liées au contrôle de la manette) afin de vérifier si le dysfonctionnement était dû aux commandes que nous envoyions au véhicule, ou s'il s'agissait d'un problème plus général lié à notre version du code. Là encore, le code fonctionnait normalement. Nous en avons donc conclu que le problème serait vraisemblablement causé par les commandes que nous envoyions au véhicule. Nous pensons également que le message d'erreur observé apparaissait spécifiquement à cause de ces nouvelles commandes.

La perte de Wi-Fi s'explique par le changement d'OS effectué avant le changement de version d'OpenPilot (entre les versions 0.9.7 et 0.9.9). Dans ce cas, le processus de mise à jour d'OpenPilot se déroule en deux étapes : d'abord le téléchargement et le changement du système d'exploitation, ce qui entraîne une perte de connexion Wi-Fi, puis il est nécessaire de reconfigurer le Wi-Fi et de redemander le téléchargement d'OpenPilot, qui s'installe ensuite sans problème.

Un autre problème nous a également fait perdre énormément de temps. En effet, après avoir réinstallé notre version du projet et voulu effectuer des modifications dans le code, le Comma détectait bien qu'une mise à jour du programme était disponible, la téléchargeait puis procédait à son installation. Cependant, lors du redémarrage, le code du commit n'était pas celui de la dernière version de notre répertoire GitHub, mais bien celui du commit précédent. Le Comma n'arrivait plus à installer les mises à jour que nous apportions au programme. Nous pensons que ce problème est survenu à cause de la perte de connexion Wi-Fi lors de la réinstallation de la version originale d'OpenPilot. Nous avons donc été contraints de réinitialiser le Comma, ce qui a résolu le problème.

Il semblerait que la cause de ce problème provienne d'un changement dans la version du système d'exploitation du Comma, et que cette nouvelle version n'ait pas encore été pleinement validée. À ce moment-là, nous utilisions la version 0.9.9, tandis que lors de la réinstallation de la version originale d'OpenPilot le lendemain, la version installée était la 0.9.7. Ce changement de version aurait également été à l'origine de problèmes de compilation sur le Comma en fin de journée (alors que sur notre ordinateur la compilation fonctionnait), ce qui nous a empêchés de tester la prise de virage plus serrée, la deuxième fonctionnalité que nous avions modifiée dans le code.

Nous supposons donc que le message d'erreur "LKAS fault : Restart car to engage" apparaissait en raison des nouvelles commandes envoyées, et nous pensons également que la commande initiale transmise par le Comma au véhicule pouvait en être la cause. Dans un premier temps, nous avons tenté d'ajouter un délai au démarrage du Comma dans la section de contrôle, mais cette solution ne s'est pas révélée efficace. Une deuxième approche que nous avons mise en œuvre consistait à ajouter des boutons permettant de basculer entre un contrôle manuel et un contrôle automatique du véhicule, pour l'accélération et la direction de manière indépendante. Au démarrage, le Comma aurait ainsi envoyé les commandes "normales" qu'il transmet habituellement au véhicule, en se basant sur le suivi des lignes blanches de la route. Cela aurait

permis au système de s'initialiser correctement avant que nous ne reprenions la main manuellement, à l'aide des boutons de la manette. Cependant, cette solution n'a pas non plus fonctionné.

## 3.2 Discussion

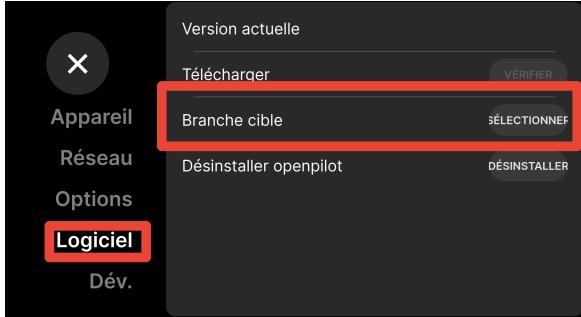
### 3.2.1 Améliorations

Compte tenu des problèmes rencontrés lors des tests en conditions réelles et afin de simplifier ces tests, il a été décidé de développer plusieurs versions de notre programme. Cela permet de tester différentes fonctionnalités de manière plus efficace tout en minimisant les risques de dysfonctionnements ou de conflits. Pour ce faire, nous avons créé un total de quatre branches sur notre dépôt GitHub : **master**, K5, K10 et J. Chacune de ces branches possède une spécialisation particulière détaillée ci-dessous :

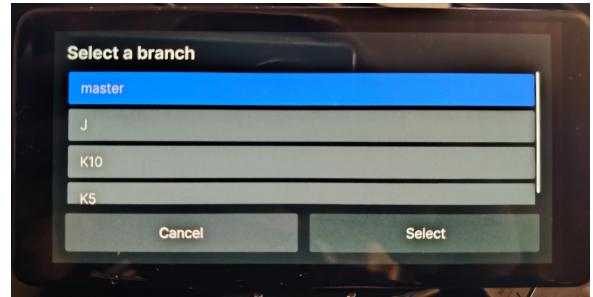
1. **master** : il s'agit de la branche principale du projet. Elle ne comporte aucune modification par rapport au code d'origine d'OpenPilot et permet donc de conduire la voiture de manière autonome. Cette branche sert notamment de référence pour observer les limitations d'OpenPilot, en particulier lors des virages serrés pris à grande vitesse.
2. **K5** : cette branche apporte des améliorations permettant au véhicule de négocier les virages à une vitesse plus élevée que la branche **master**, tout en conservant une conduite entièrement automatique. Dans cette branche, nous avons donc modifié la variable **steer\_max** en la modifiant sa valeur de 1 à 5.
3. **K10** : similaire à la branche K5, cette version optimise encore davantage la prise de virages serrés à une vitesse plus élevée. Ici, **steer\_max** vaut 10.
4. **J** : cette branche intègre le code permettant de contrôler manuellement le véhicule.

L'avantage d'une telle organisation est qu'au cours de la prochaine phase de tests, il sera plus simple et plus fiable de valider les différents objectifs, tels que le passage de virages serrés à grande vitesse ou encore le contrôle du véhicule à l'aide de la manette. De plus, cette approche permet de minimiser les modifications de code à effectuer ainsi que le risque d'erreurs lors du déploiement sur le dispositif Comma. En effet, chaque mise à jour du code nécessite de le télécharger sur le Comma, avec la possibilité qu'une erreur survienne, obligeant alors à réinitialiser complètement l'appareil avant de réinstaller le projet. Cette opération est particulièrement longue, comme mentionné précédemment.

Pour changer de branche, il suffit de se rendre dans la section **Software** du menu du Comma et de sélectionner la branche souhaitée dans le champ **Target Branch**, comme illustré dans la figure 3.3 :



(a) Menu permettant de choisir la branche



(b) Choix parmi les branches existantes

FIGURE 3.3 – Sélection d'une nouvelle branche du projet dans le Comma

### 3.2.2 Ajouts futurs

#### 3.2.2.1 Trouver la limitation de la LKS

La prochaine étape du projet pourrait être de trouver l'origine du message d'erreur lié au système LKS. Nous avons plusieurs hypothèses liées à ce problème.

La première est qu'il est possible que ce message d'erreur apparaisse en raison d'une comparaison incorrecte entre les valeurs de l'orientation calculée par le modèle et les valeurs envoyées lors de l'exécution de la commande. Le système LKS repose sur une estimation continue de l'orientation du véhicule en fonction des lignes blanches de la route et utilise cette information pour ajuster le contrôle de la direction afin de maintenir la voiture dans sa voie. Comme l'orientation calculée par le modèle et les commandes que l'on envoie via la manette diffèrent, cela peut créer un conflit et amener au problème rencontré.

La deuxième hypothèse que nous avons est que le défaut LKAS viendrait du fait que nous avons initialement travaillé sur une version plus récente du dépôt d'OpenPilot (la 0.9.9), alors que le Comma utilise par défaut la version 0.9.7. Ce décalage de version pourrait expliquer l'apparition du défaut. L'objectif serait de tirer notre dépôt, basé cette fois sur la version 0.9.7, d'utiliser la branche master puis la branche K5, simplement pour observer si le défaut LKAS persiste malgré tout.

#### 3.2.2.2 Se tourner vers un autre dispositif : le Panda

Le Panda est un dongle open-source développé aussi par la société Comma.ai qui permet aux utilisateurs d'interagir avec les systèmes électroniques des véhicules, principalement via le bus CAN. Il s'agit d'un dispositif matériel compact qui se branche sur le port OBD-II du véhicule et offre une interface pour lire, enregistrer, et envoyer des messages CAN, permettant ainsi de diagnostiquer, analyser, et même contrôler certains systèmes du véhicule [1].



FIGURE 3.4 – Panda

Ainsi, ce dispositif pourrait potentiellement remplacer complètement le Comma, car sa principale fonction consiste à lire et écrire des données sur le bus CAN. Grâce au Panda, il est probable que nous ne serons plus limités par le système LKS, puisque le suivi de ligne n'est plus impliqué. De plus, le Panda dispose d'une connectivité Wi-Fi, ce qui permettrait d'envoyer des données de la même manière que nous l'avons fait avec le Comma. Comme le Comma, le Panda est aussi un projet open-source.

Finalement, les fonctionnalités du Panda sont intégrées dans le Comma, permettant ainsi de communiquer avec le bus CAN. Utiliser uniquement le Panda pourrait donc permettre de se débarrasser de toutes les fonctionnalités non nécessaires pour notre cas d'usage, telles que le suivi de ligne, en conservant uniquement la partie contrôle du véhicule via la communication avec le bus CAN.

Dans le projet OpenPilot, il existe un répertoire Panda, qui contient le code et les outils nécessaires pour interagir avec le matériel Panda. Ce répertoire inclut des pilotes permettant de communiquer avec le bus CAN, ainsi que des bibliothèques en Python et C++ pour faciliter l'intégration de Panda dans des systèmes tels qu'OpenPilot. Il contient également des tests pour vérifier le bon fonctionnement des fonctionnalités. Voici un aperçu des répertoires principaux du projet Panda [3] :

- **board** : ce répertoire contient le code qui s'exécute directement sur les microcontrôleurs STM32. Il gère les communications sur le bus CAN, le contrôle des dispositifs de sécurité et la gestion des messages CAN.
- **drivers** : ce répertoire regroupe les pilotes nécessaires au bon fonctionnement du Panda.
- **python** : ce répertoire fournit les fonctions nécessaires pour envoyer et recevoir des messages CAN, ainsi que pour interagir avec le bus CAN.
- **tests** : ce répertoire contient des tests pour vérifier l'efficacité du Panda.

Le README du projet explique aussi comment utiliser la bibliothèque Panda pour interagir avec le bus CAN. Il décrit en particulier des exemples de code permettant de recevoir ou d'envoyer des messages sur le bus CAN. Voici un exemple de code permettant de recevoir des messages [3] :

```
from panda import Panda
panda = Panda()
panda.can_recv()
```

Cet exemple montre comment instancier un objet Panda et utiliser la méthode `can_recv()` pour recevoir des messages sur le bus CAN.

De même, voici un exemple de script pour envoyer un message sur le bus CAN :

```
from opendbc.car.structs import CarParams
panda.set_safety_mode(CarParams.SafetyModel.allOutput)
panda.can_send(0x1aa, b'message', 0)
```

Dans cet exemple, un message, avec un identifiant `0x1aa`, est envoyé sur le bus 0, après avoir défini le mode de sécurité du véhicule avec `set_safety_mode()`. Au vu du code présenté ici, l'utilisation du Panda pourrait être une bonne alternative à celle du Comma, puisqu'il est très simple et ne garde que l'essentiel : la communication avec le bus CAN.

# Conclusion

Les objectifs du projet étaient l'étude de la solution OpenPilot, la compréhension du fonctionnement du bus CAN, la qualification de la solution OpenPilot sur véhicule de test et l'amélioration du système pour effectuer le contrôle total du véhicule (accélération, freinage etc.).

Pendant la première phase du projet, nous avons pris le temps de découvrir OpenPilot et d'explorer son code source. Étant donné l'ampleur du projet, nous avons rédigé une documentation afin de localiser et de comprendre les différents modules du projet. Cette documentation nous a ensuite permis de nous concentrer sur la partie contrôle du véhicule et de l'envoi des commandes, limitant ainsi la taille du projet que nous considérions.

Au cours de ce projet, nous avons réussi à montrer, en simulation, que la communication entre le bus CAN de la voiture et le dispositif Comma pouvait être utilisée afin d'envoyer nos propres commandes. Nous avons aussi montré, dans la simulation, la possibilité de lever la contrainte de la vitesse dans des virages serrés. Bien que le jour de la tentative en tests réels ait été un échec, nous avons pu découvrir d'autres problèmes liés à la voiture réelle et aux sécurités qui y sont implantées.

Pour la suite, il serait intéressant de refaire des tests sur un véhicule réel en repartant d'une version antérieure d'OpenPilot. Ensuite, de poursuivre les recherches sur le système LKS afin de déterminer l'origine du problème et d'examiner si les sécurités peuvent être désactivées sur un véhicule de test (physiquement ou sur OpenPilot). Une deuxième piste à explorer serait l'utilisation du dispositif Panda, qui pourrait offrir une alternative intéressante.

Ce projet nous a également permis d'explorer des domaines de l'informatique que nous n'avions pas encore abordés, tels que la communication SSH entre l'ordinateur connecté à la manette et le dispositif Comma, le fonctionnement du bus de communication CAN, ainsi que la compréhension d'un système embarqué complexe dans le domaine de l'automobile.

# Bibliographie

- [1] COMMA.AI. *Site officiel*. 2016. URL : <https://comma.ai/> (visité le 16/03/2025).
- [2] COMMA.AI. *Comma.ai Blog*. 2016. URL : <https://blog.comma.ai> (visité le 01/11/2024).
- [3] COMMA.AI. *Répertoire OpenPilot GitHub*. 2016. URL : <https://github.com/commaai/openpilot> (visité le 16/03/2025).

# Annexes

Voici les liens vers nos répertoires GitHub :

- Clone d'OpenPilot sur GitHub
- Documentation du projet et scripts pour la connexion SSH sur GitHub