

Rapport de Projet de Fin d'Étude

Camille, Ulrich

ulrich.camille@etu.u-bordeaux.fr

Gaspard, Clément

clement.gaspard@u-bordeaux.fr

Rodrigues, Nicolas

nicolas.rodrigues.1@etu.u-bordeaux.fr

March 2022

Table des matières

1	Introduction	3
2	Besoins et objectifs du projet	3
3	Gestion de projet	4
3.1	Kanban depuis le logiciel : Shortcut	4
3.2	Architecture fonctionnelle : ROS(2)	5
3.3	Méthode Agile	6
3.4	UML	8
3.4.1	ROS(2)	8
3.4.2	ROS(1)	9
3.5	Différence avec ROS et ROS2	9
4	Développement technique	12
4.1	Perception image	12
4.1.1	YOLO	12
4.1.2	Deep learning	13
4.1.3	Entraînement du réseau de neurones	13
4.1.4	De la detection à la position	15
4.2	Calcul commande moteurs	17
4.2.1	PX4 (Pixhawk)	17
4.2.2	MAVROS	17
5	Simulation	20
5.1	Simulation (Gazebo)	20
5.2	Simulation (Python et RVIZ)	21

6 Application à un système réel - Cozmo	24
6.1 Présentation du système	24
6.2 Mise en place du projet	25
6.2.1 Présentation globale	25
6.2.2 Calibration de la caméra	26
6.2.3 Changement de repère de la caméra vers celui du robot Cozmo	27
6.2.4 Estimation position du véhicule dans le repère robot	28
6.2.5 Contrôle des moteurs de Cozmo	30
6.3 Tests unitaires	30
6.3.1 Estimation de la position d'un point sur le sol vis à vis de Cozmo	30
6.4 Tests haut-niveau	32
7 Améliorations	34
8 Conclusion	34

1 Introduction

Dans le cadre de notre dernière année de Master Informatique (ASPIC), il nous a été proposé un projet de fin d'étude afin de pouvoir appliquer nos connaissances et différentes compétences acquises tout au long de notre formation ; notre responsable de formation, Mr. Serge Chaumette, nous a laissé le choix entre deux projets :

- Suivi d'un véhicule terrestre (radiocommandé) avec un drone
- accrochage / libération d'un drone à une zone d'accrochage au plafond (électroaimant)

Notre choix s'est porté sur le premier sujet.

Un sujet qui a sûrement déjà été traité une multitude de fois. Voici un exemple de travail très récent sur le sujet : *article*. Dans ces travaux nous associons les données de circulation routière au suivi des véhicules. Nous n'irons pas aussi loin dans la démarche par manque de temps et de données mais cet article peut être une piste pour une éventuelle amélioration.

2 Besoins et objectifs du projet

Il faut aussi comprendre les enjeux à coder un tel système. De plus, ce genre de système possède une multitude d'applications :

- Application civile : suivi d'un sujet lors d'un tournage de cinéma/publicité avec un drone,
- Application de défense : espionnage, surveillance

Sur le plan physique, nous allons rester sur les besoins d'un drone classique (autonomie, composant,...).

En ce qui concerne le logiciel, celui-ci doit pouvoir :

3 GESTION DE PROJET

- Communiquer la position de la voiture
- Changer le schéma de suivi à tout moment
- Pouvoir l'implémenter sur n'importe quel drone.

3 Gestion de projet

3.1 Kanban depuis le logiciel : Shortcut

Tout le fonctionnement est basé sur les étiquettes, qui sont destinées à optimiser la production et la collaboration entre les équipes. Une étiquette (Kanban, en japonais) voit le jour dès lors qu'une commande est passée. Elle indique une tâche à réaliser. Les tâches sont réparties dans un tableau en fonction de leur état et peuvent passer d'une colonne à l'autre au fur et à mesure du processus : par exemple à l'étude, à faire, en cours, fait. Chaque membre de l'équipe sait ainsi ce qu'il a à faire et où en est le travail des autres. Outre une meilleure communication, cette technique basée sur le principe des *post-it* permet de visualiser de manière claire l'ensemble de la chaîne de production et l'avancement des tâches, ce qui permet de repérer facilement les blocages et les urgences.

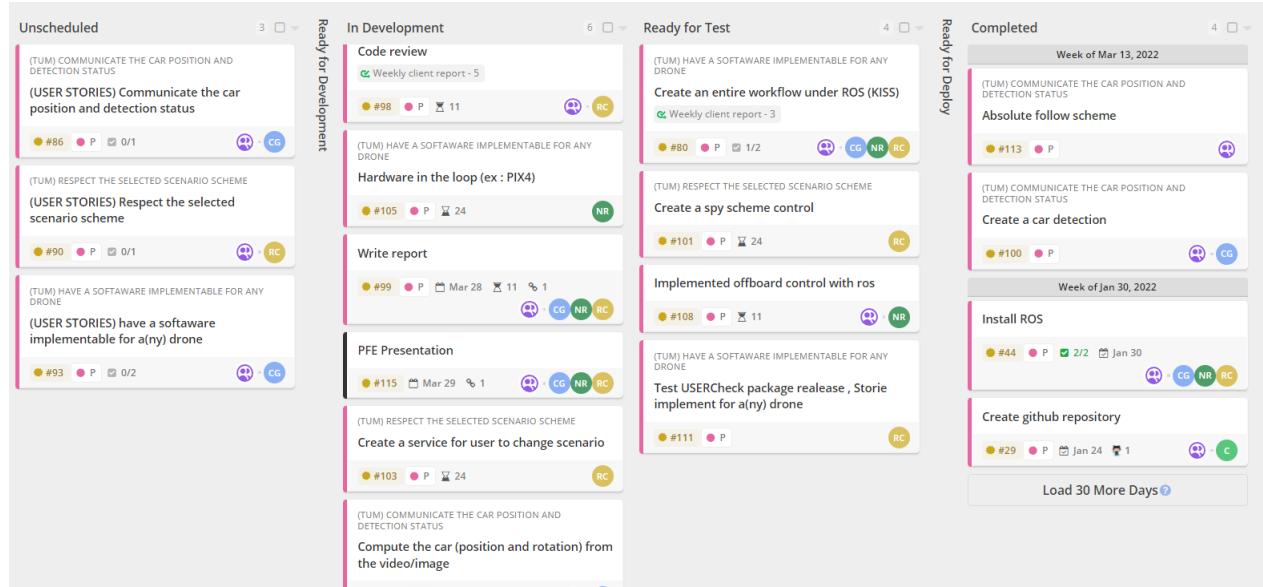


FIGURE 1 – Kanban

Le kanban ci-dessus représente l'avancement de nos tâches. On peut donc remarquer dans la colonne "Completed" que seul le schéma de suivi absolu de la voiture et la détection voiture a été terminé et testé. Ces tâches ont été validées par le même test que nous verrons en détail plus tard dans le chapitre des tests. Il y a aussi la colonne "Ready for Test" les tâches terminées mais non validées

3 GESTION DE PROJET

par des tests, pour la plupart soit le test n'est pas encore validé soit il n'est pas validé.

On peut aussi remarquer certaines tâches qui ont été assignées à toute l'équipe, car elles nécessitaient un avancement rapide. Nous parlons notamment de l'installation de ROS qui n'était pas anodine surtout pour les membres de l'équipe peu familiers avec ce logiciel. Ou encore "*Create an entire workflow under ROS*" qui nécessitait l'avancement de toute l'équipe afin de pouvoir enfin commencer les premières itérations sur l'avancement des *packages* ou autre et donc ainsi pouvoir directement l'implémenter/intégrer dans nos futurs travaux, quel que soit l'avancement des autres membres du groupe.

#	ID	Name	Stories	Points	Team	Owners	State	Progress
	16	(TUM) Communicate the car position and detection status	4	-	👤	CG	In Progress	<div style="width: 50%;"><div style="width: 100%;">50%</div></div>
	58	(TUM) Respect the selected scenario scheme	3	-	👤	NR	In Progress	<div style="width: 100%;"><div style="width: 100%;">0%</div></div>
	59	(TUM) have a software implementable for any drone	4	-	👤	RC	In Progress	<div style="width: 100%;"><div style="width: 100%;">0%</div></div>

FIGURE 2 – Users stories progression avec Shortcut

La figure 2 représente l'avancement des "*User stories*" car les tâches du kanban découlent de ces "*User stories*" ce qui nous permet d'avoir un suivi sur la progression de celle-ci.

On remarquera donc ici que la partie communication de la position de la voiture et la détection de la voiture ont été partiellement accomplies, le reste est encore en développement. Cette progression n'est pas fixe car certaines tâches peuvent s'ajouter aux "*Users stories*" et donc changer l'état de progression.

3.2 Architecture fonctionnelle : ROS(2)

R.O.S. qui signifie Robot Operating System est un *middleware* et donc un outil permettant de faire la jonction entre le hardware et software car celui-ci présente les outils nécessaires pour s'occuper de la partie matérielle mais aussi logicielle (IA, interface, ...).

Les systèmes robotiques font appel à de nombreuses compétences telles que la mécanique, l'électrotechnique, le contrôle, la vision par ordinateur, et de nombreux pans de l'informatique comme l'informatique temps-réel, le parallélisme, et le réseau. Souvent les avancées en robotique dépendent de verrous technologiques résolus dans ces champs scientifiques. Un système robotique peut donc être complexe et faire intervenir de nombreuses personnes. Afin de pouvoir être efficace dans la conception de son système robotique.

R.O.S. possède des aspects pratiques à plusieurs niveaux : conception d'architecture plus simple, outils de debug, multiples langages de programmation possible (c++, python, java, ...).

3 GESTION DE PROJET

En ce qui concerne les *nodes* la plupart ont été codé en Python, notamment pour la bibliothèque Pytorch et Numpy qui nous faciliteront le travail sur le traitement d'image, cependant le coder en C++ permettrait d'être plus efficace.

Nous pouvons séparer ces *nodes* par des *packages*. Chaque *package* peuvent avoir un type de codage différent (c++, python) et possède plusieurs dossiers/-fichiers :

- Un dossier du même nom que le *package* comportant tous les fichiers sources
- Un dossier " params" avec un fichier params renseignant tous les paramètres, variables pour chaque noeud de ce *package*. Celui-ci permet de pouvoir changer certaines variables sans recompiler le programme. Il est aussi important afin de renseigner des chemins de fichier.
- Un dossier " launch", possédant un fichier launch en python qui viendra localement exécuter les nodes du *package* en tenant compte des fichiers paramètre.
- Un fichier " CmakeList" si on compile un programme en c++ par exemple ou un fichier setup.py si l'on possède du code en python. C'est dans ces fichiers que l'on renseigne les fichiers params en argument pour les *nodes* mais aussi que l'on définit quels sont les fichiers exécutables et/ou bibliothèques locaux (fichier .h, ...)
- enfin un fichier package.xml, qui lors de la cross-compilation du projet renseignera, les dépendances/bibliothèques/*packages* présente sur la machine locale ou dans le projet en tant que .dll ou .so (bibliothèque objet) afin de rendre le projet exécutable sur une autre machine indépendamment de la configuration de l'OS.

3.3 Méthode Agile

L'outil R.O.S est souvent au service de la méthode Agile, car il permet de toujours avoir une bonne vision du projet, de l'architecture du projet de mieux le moduler donc de mieux concevoir pour le client de manière itérative.

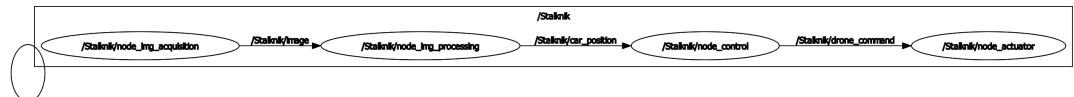


FIGURE 3 – Architecture fonctionnelle depuis l'outil RQT : itération 1

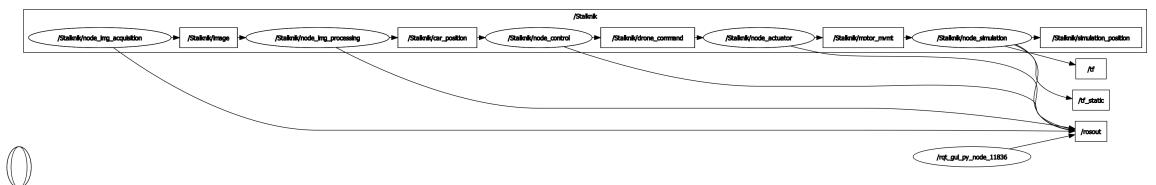


FIGURE 4 – Architecture fonctionnelle depuis l'outil RQT : itération 2

3 GESTION DE PROJET

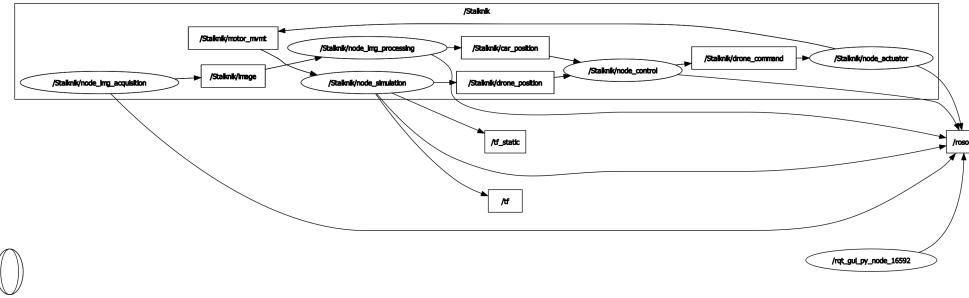


FIGURE 5 – Architecture fonctionnelle depuis l’outil RQT : itération 3

Les figures 1,2,3 sont des graphes visibles via l'outil de ROS appelé RQT, celui-ci permet de visualiser les connections entre les *nodes*, ces connections sont aussi appelées *topics* car elles représentent les données que chaque noeud écrit ou lit.

À la première itération, nous avons suivi le "KISS" (*Keep It Simple and Stupid*). Ainsi la chaîne permettant de passer d'une image à une commande drone a été réalisée. On a ici une chaîne plutôt directe, à partir d'un flux d'image, on en déduit une position de voiture, d'une position voiture on en déduit une position de drone voulu et enfin d'une position de drone voulu on en déduit une commande moteur. Ce qui nous donne la figure 3 Ces fonctions renvoyaient chacune des valeurs arbitraires voir aléatoires, mais la structure était présente. Dès le début, nous avions compris que le traitement d'image était la tâche la plus compliquée. Ainsi nous avions réorganisé les tâches afin de trouver rapidement une solution ensemble au problème et ensuite retourner à nos tâches respectives.

Dans une deuxième itération, nous avons implémenté des simulations. Deux simulations ont été implémentées : Une simulation python et une simulation Gazebo avec PX4. N'ayant aucun élément physique du projet, une simulation était nécessaire.

La simulation python permet d'avoir sous forme d'équation la physique du drone et donc potentiellement pouvoir créer un filtre de Kalmann pour un contrôle plus précis si nécessaire, car le Kalmann nécessite les équations de mouvement/physique du système.

Sur la figure 4 seulement la simulation python est visible car la simulation sur Gazebo est fonctionnelle seulement sur ROS1 et pas encore sur ROS2.

Dans une troisième itération, nous avons amélioré les *packages* déjà existant. En effet les relations entre *packages* deviennent plus complexe et non seulement le traitement d'image, mais aussi les méthodes de suivi ont été amélioré. Ce que l'on remarque sur la figure 5

Une deuxième simulation a aussi été développée. Pourquoi 2 simulations ? La simulation Gazebo permet d'avoir une simulation beaucoup plus réaliste car les effets de collision y sont calculés et il est possible d'obtenir un flux vidéo de

3 GESTION DE PROJET

l'environnement ou autre. Elle permet l'implémentation direct d'une solution à la commande moteur : PX4 Une fois que nous aurons le matériel, toutes ces simulations seront encore utiles dans une démarche de "*hardware in the loop*" et permettra de tester la caméra, le drone et d'autres éléments un par un.

3.4 UML

3.4.1 ROS(2)

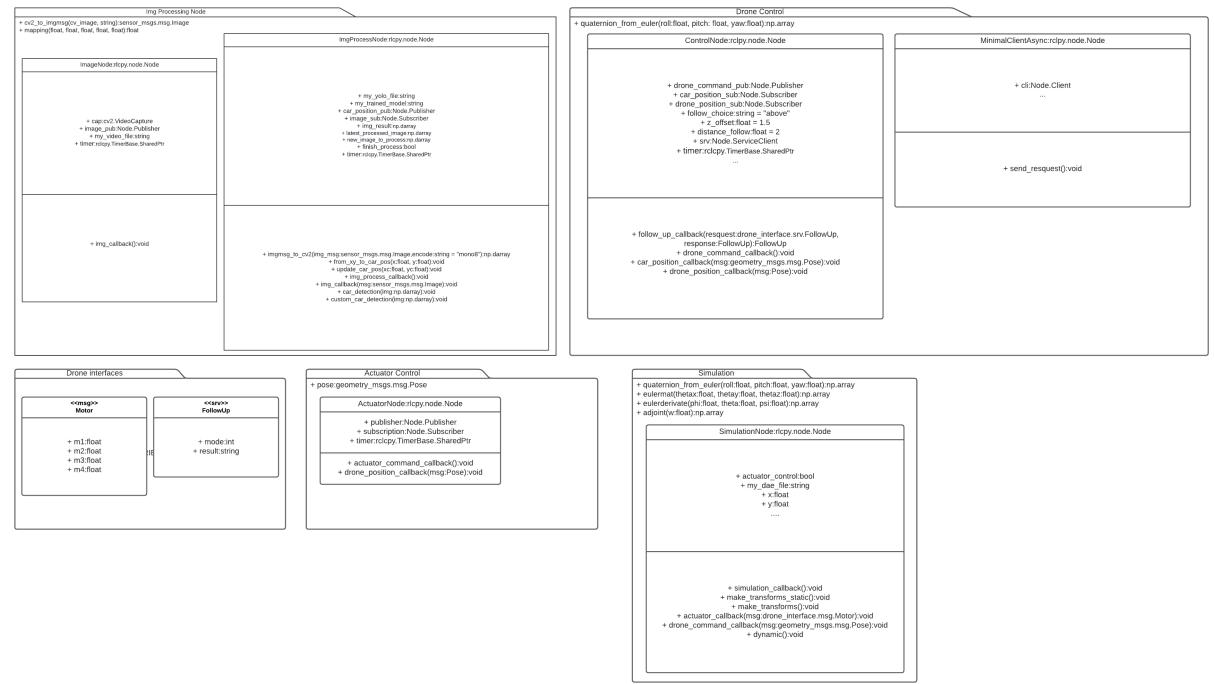


FIGURE 6 – Diagramme de classes ROS2

3 GESTION DE PROJET

3.4.2 ROS(1)

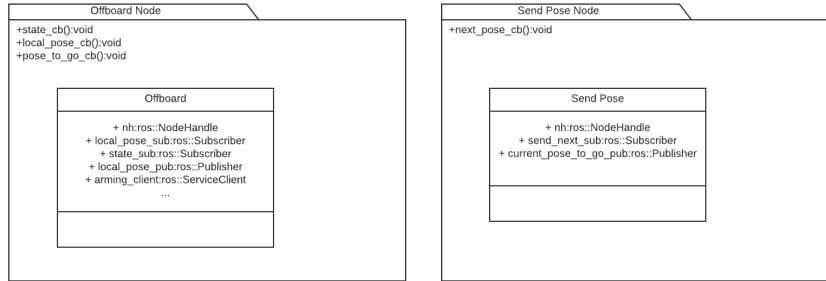


FIGURE 7 – Diagramme de classes ROS1

3.5 Différence avec ROS et ROS2

Ce projet était une première expérience à ROS2 qui succédera à ROS1. ROS(1) est disponible seulement sur une distribution unix (ubuntu, debian,...) il existe aussi de nombreuses versions, les dernières étant : Noetic et Melodic. Noetic devant être sollicité de préférence car c'est la première version à utiliser python3 au lieu de python2.

ROS(2) possède actuellement deux distributions LTS (*Long Term Support*) : Foxy et Galactic. Nous avons utilisé ici la plus récente, c'est à dire Galactic. Un avantage de ROS2 est la compatibilité avec quasiment n'importe quelle OS (Operating System) soit Windows, Mac, Linux, Android,...

D'autres différences fondamentales résident dans la bibliothèque rospy remplacé par la bibliothèque rclpy.

Ci-dessous un exemple de noeud avec rospy (ROS1) :

```
1 #!/usr/bin/env python
2 # license removed for brevity
3 import rospy
4 from std_msgs.msg import String
5
6 def talker():
7     pub = rospy.Publisher('chatter', String, queue_size=10)
8     rospy.init_node('talker', anonymous=True)
9     rate = rospy.Rate(10) # 10hz
10    while not rospy.is_shutdown():
11        hello_str = "hello world %s" % rospy.get_time()
12        rospy.loginfo(hello_str)
13        pub.publish(hello_str)
14        rate.sleep()
15
16 if __name__ == '__main__':
17     try:
18         talker()
19     except rospy.ROSInterruptException:
```

3 GESTION DE PROJET

```
20     pass

Ci-dessous un exemple de noeud avec rclpy (ROS2)

1 import rclpy
2 from rclpy.node import Node
3
4 from std_msgs.msg import String
5
6
7 class MinimalPublisher(Node):
8
9     def __init__(self):
10         super().__init__('minimal_publisher')
11         self.publisher_ = self.create_publisher(String, 'topic',
12                                              10)
12         timer_period = 0.5 # seconds
13         self.timer = self.create_timer(timer_period, self.
14                                         timer_callback)
14         self.i = 0
15
16     def timer_callback(self):
17         msg = String()
18         msg.data = 'Hello World: %d' % self.i
19         self.publisher_.publish(msg)
20         self.get_logger().info('Publishing: "%s"' % msg.data)
21         self.i += 1
22
23
24 def main(args=None):
25     rclpy.init(args=args)
26
27     minimal_publisher = MinimalPublisher()
28
29     rclpy.spin(minimal_publisher)
30
31     # Destroy the node explicitly
32     # (optional - otherwise it will be done automatically
33     # when the garbage collector destroys the node object)
34     minimal_publisher.destroy_node()
35     rclpy.shutdown()
36
37
38 if __name__ == '__main__':
39     main()
```

ROS2 propose un meilleur contrôle des appels synchrones ou asynchrones des fonctions. Le format est aussi différent, les fonctions de callback sont retenues en tant que méthode dans la classe objet (ici nommée *MinimalPublisher*) qui hérite de l'objet *Node*. Ensuite soit la méthode peut être dans un *thread* soit être appelé dans un *timer* via rclpy, là où ROS1 était limité à l'écriture de l'init() et d'un main() répété en boucle dans un *thread*.

ROS2 propose aussi un fichier launch en python au lieu d'un fichier en xml. Ce qui permet d'avoir une création de noeud plus complexe et automatisée selon les paramètres d'entrées qui eux sont stockés en général dans les fichiers yaml.

3 GESTION DE PROJET

Dans la structure nous avons créé des fichiers launch locaux pour chaque *package*, ce qui permet de vérifier l'exécution *package* par *package*, et enfin un fichier launch à plus haut niveau qui viendra juste appeler les fichiers launch locaux dans chaque *package*.

Distro	Release date	Poster	Tuturtle, turtle in tutorial	EOL date
ROS Noetic Ninjermys <i>(Recommended)</i>	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)

FIGURE 8 – Historique ROS selon ROS.org

Distro	Release date	Logo	EOL date
Humble Hawksbill	May 23rd, 2022		
Galactic Geochelone	May 23rd, 2021		November 2022
Foxy Fitzroy	June 5th, 2020		May 2023

FIGURE 9 – Historique ROS2 selon ROS.org

4 Développement technique

4.1 Perception image

La perception est nécessaire dans notre projet afin de répondre au besoin de détecter une voiture. La perception ne se limite pas à la vision par caméra, cela peut être une vision infrarouge, une carte de profondeur, ou une image hyperspectrale, ... Cependant nous avons jugé que l'acquisition d'image par caméra était la méthode la plus accessible voir même la plus efficace.

Il existe différents types de reconnaissance d'objets :

- La classification d'images consiste à prédire la classe d'un objet dans une image. Entrée : une image avec un seul objet. Sortie : une étiquette de classe. Ceci est intéressant afin d'être sûr que l'on détecte une voiture cependant on aura aucune information sur sa position (ce qui important pour la commande de suivi).
- La localisation d'objets consiste à identifier l'emplacement d'un ou plusieurs objets dans une image et à dessiner un cadre de délimitation autour de leur étendue. Entrée : une image avec un ou plusieurs objets. Sortie : un ou plusieurs cadres de délimitation. La localisation est intéressante pour avoir la position d'un objet mais comment être sûr que l'objet en question est la cible ? est-ce que l'on suit la position d'une voiture ou d'un oiseau.
- La détection d'objets combine ces deux tâches et dessine un cadre de délimitation autour de chaque objet dans l'image et leur attribue une classe. Entrée : une image avec un ou plusieurs objets. Sortie : un ou plusieurs cadres de délimitation et une étiquette de classe pour chaque cadre de délimitation. Ainsi avec cette méthode on pourra s'assurer d'avoir et le véhicule et "sa position"

"Sa position" car nous n'avons pas réellement sa position nous avons sa "*bounding box*" qui détient des informations sur la position sur l'image et la taille, à partir de ces deux informations il faut en déduire une position dans l'espace. Une étape qui nécessite de connaître la FOV (Field Of View) de notre caméra.

Lors de la première itération, nous avons développé une détection d'image via un réseau de neurone déjà construit aussi appelé le YOLO (You Only Look Once) et par la suite nous avions rédigé notre propre réseau par entraînement d'image à poids négatif ou positif et labélisation.

4.1.1 YOLO

Comment fonctionne le YOLO ? Les systèmes de détection antérieurs réutilisent les classificateurs ou les localiseurs pour effectuer la détection. Ils appliquent le modèle YOLO à une image à plusieurs endroits et échelles. Les régions à score élevé de l'image sont considérées comme des détections.

Le YOLO une approche totalement différente. Il applique un seul réseau de neurones à l'image complète. Ce réseau divise l'image en régions et prédit

4 DÉVELOPPEMENT TECHNIQUE

les cadres de délimitation et les probabilités pour chaque région. Ces boîtes englobantes sont pondérées par les probabilités prédictes.

Pour plus d'informations, nous vous invitons à regarder le lien ci-contre : YOLO article

```
1     detector = ObjectDetection()
2     detector.setModelTypeAsYOLOv3()
3     detector.setModelPath(self.my_yolo_file)
4     detector.loadModel()
5     returned_image, detections, extracted_objects = detector.
detectObjectsFromImage(input_image=img, input_type="array",
output_type="array", extract_detected_objects=True,
minimum_percentage_probability=20)
```

Lors de travaux dirigés avec M. CAUBET, nous avions réalisé un exercice similaire avec l'outil YOLO. Par ces lignes, on initialise le CNN dans l'objet "detector" et simplement par la méthode "detectObjectsFromImage" on obtient une liste d'objet retenu dans "extracted_objects".

On pourra aussi obtenir les informations sur les "bounding" box cependant nous ne sommes pas aller plus loin avec cette méthode.

4.1.2 Deep learning

4.1.3 Entraînement du réseau de neurones

Afin d'entraîner notre réseau de neurone, nous avons utilisé 791 images. (Elles se trouvent dans "Stalknik_MK2/ Cozmo_follow_car/trained_carmodel/datasets/Cozmo-cars/images").

Grâce au logiciel "Label-studio" (Figure 10), nous avons étiqueté chaque image une par une en définissant l'emplacement de la voiture dans l'image. Nous avons utilisé le ratio préconisé par YoloV5 soit 90% d'images avec la voiture présente et 10% d'images sans la voiture.

4 DÉVELOPPEMENT TECHNIQUE

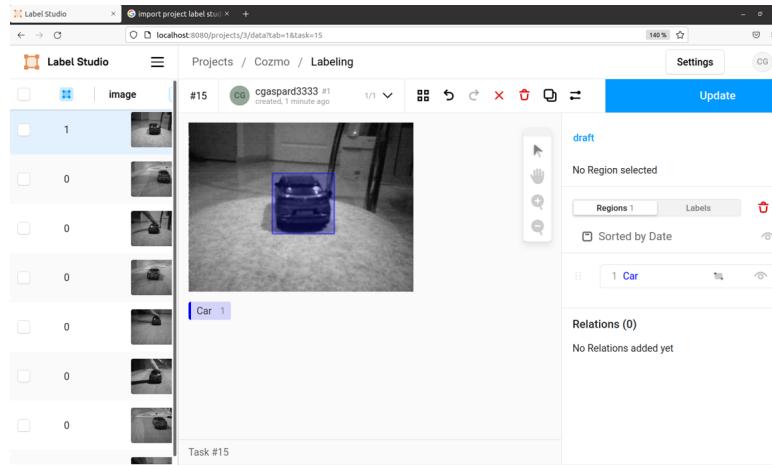


FIGURE 10 – Interface Label Studio

Nous avons par la suite exporté les images étiquetées vers YoloV5 en lançant l'entraînement de notre réseau de neurone grâce à la commande suivante :

```
python train.py --rect --img 320 --batch 16 --epochs 100  
--data cozmo-cars.yaml --weights yolov5s.pt
```

En effet, celle-ci permet, à partir d'un réseau de neurone pré-entraîné YoloV5 small (`--weights yolov5s.pt`), d'entraîner notre propre réseau de neurone à partir des images étiquetées rectangulaire de largeur 320 pixels (`--rect --img 320`). En utilisant un batch de 16 (le maximum possible pour notre GPU sans dépasser la taille maximum de 2Go de VRAM) et 100 epochs (ce qui nous a paru raisonnable pour avoir une détection correcte de la voiture).

Grâce au module Python "Weights Biases", compatible avec YoloV5, nous avons pu suivre l'avancement de l'apprentissage ainsi que les résultats finaux de celui-ci. Ce module est une sorte de serveur Web qui permet de consulter les données du réseau de neurone depuis n'importe où à travers internet, il permet de consulter aussi les résultats de validation à partir d'un nouveau set de données, mais aussi de voir les différences d'inférence en fonction du nombre d'epochs retenus : Figure 11.

Wandb.ai, permet aussi de générer des graphiques permettant de voir l'évolution des fonctions de coût du réseau de neuronal, mais aussi l'évolution du *mAP*, de la *precision* et du *recall*, des grandeurs intrinsèques au *Deeplearning* et qui ne seront pas expliquées dans le cadre de ce projet.

Ces résultats sont publics et consultables à l'adresse suivante :
<https://wandb.ai/cgaspard3333/train?workspace=user->

4 DÉVELOPPEMENT TECHNIQUE

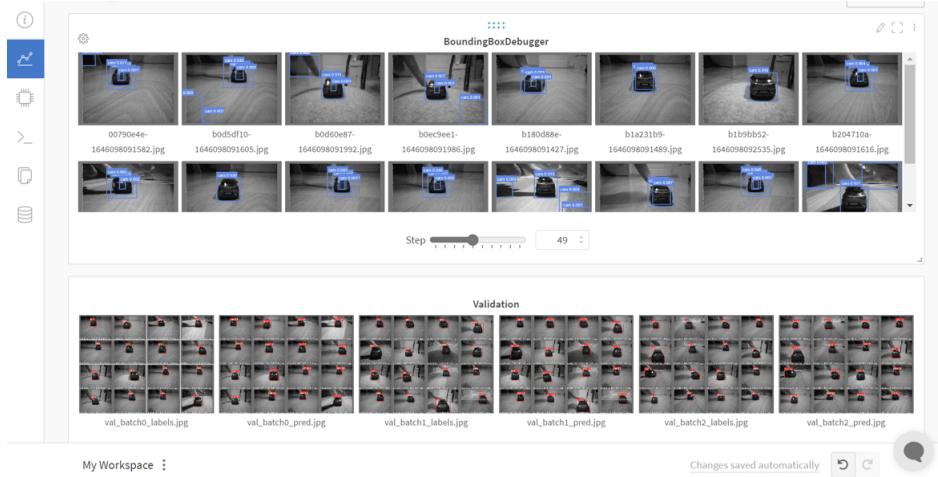


FIGURE 11 – Interfaçce de Wandb.ai | Module de débugage nombre d'epochs et de tests de validation du réseau neuronal

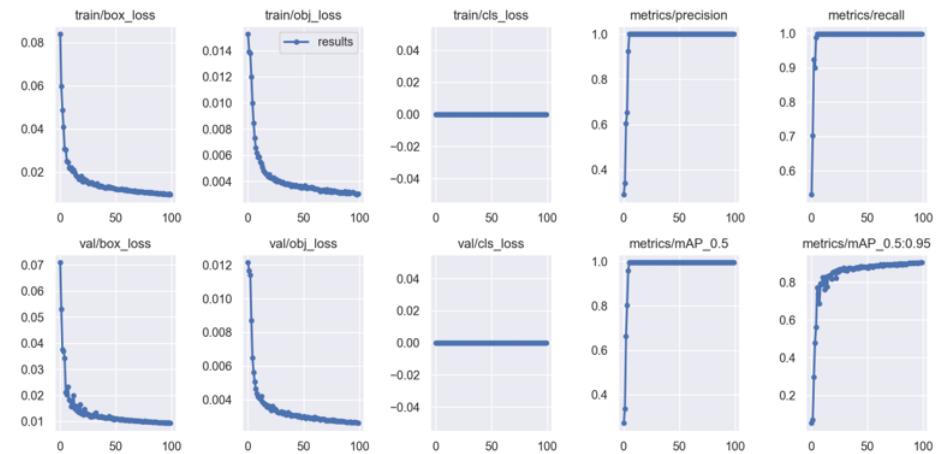


FIGURE 12 – Graphiques de l'évolution des fonctions de coût du réseau de neuronal (aussi *mAP*, *precision* et *recall*) en fonction du nombre d'epochs

Nous pouvons donc à la suite de cela utiliser notre réseau de neurone entraîné afin d'encadrer et localiser la voiture dans notre image. (voir code : Cozmo_follow_car/Detection_IA.py)

4.1.4 De la detection à la position

Après les travaux de détection, nous avons travaillé sur la relation entre un couple (x,y) sur l'image et la position dans l'espace par rapport à la caméra.

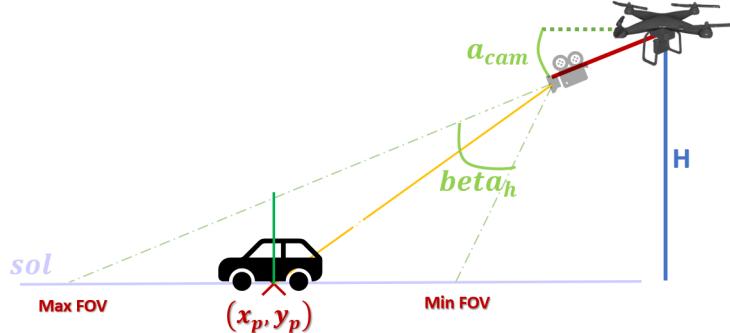


FIGURE 13 – schéma FOV

Dans ce cas-là, nous sommes partis avec l'hypothèse d'un drone volant à une hauteur H et d'une caméra rotée d'un angle α_{cam} vers l'avant du drone donc l'axe x du drone La FOV de la caméra donne l'information sur l'angle "d'ouverture" sur l'axe verticale et horizontale (car elles peuvent être différentes, généralement l'angle d'ouverture est plus grand horizontalement). Ces angles seront appelés : β_h et β_v respectivement l'angle d'ouverture horizontal et vertical.

Ainsi posons :

$$\begin{aligned}\Omega_1 &= \alpha_{cam} - \beta_h/2 \\ \Omega_2 &= \alpha_{cam} + \beta_h/2\end{aligned}\quad (1)$$

L'intersection entre la droite d'angle Ω_1 et le sol représente le pixel le plus bas sur l'image aussi représenté sur la figure 13 par le point min_{FOV} . De même que pour l'intersection entre la droite d'angle Ω_2 et le sol représente le pixel le plus haut sur l'image aussi représenté sur la figure 13 par le point max_{FOV} .

Ensuite on peut déterminer la position dans l'espace de max_{FOV} et min_{FOV} par rapport au drone et donc selon l'axe x du drone (l'avant du drone). Elle peut être déterminé comme ceci grâce à des relations trigonométriques :

$$\begin{aligned}distance_{ex_min_{FOV}} &= \frac{H}{\tan(\Omega_1)} \\ distance_{ex_max_{FOV}} &= \frac{H}{\tan(\Omega_2)}\end{aligned}\quad (2)$$

le véhicule (si jamais il est détecté) est quelque part entre min_{FOV} et max_{FOV} or ce que nous avons c'est la position sur l'image que l'on va appeler P_x et P_y Ensuite afin de passer d'une position en pixel à une distance il suffira de faire une relation de proportionnalité en tenant compte de la résolution sur l'axe horizontal et vertical que l'on va appeler respectivement Res_x , Res_y . On en déduit donc :

$$distance_voiture_x = distance_{ex_min_{FOV}} + \frac{P_x}{Res_x} * (distance_{ex_max_{FOV}} - distance_{ex_min_{FOV}}) \quad (3)$$

4 DÉVELOPPEMENT TECHNIQUE

Ceci est le raisonnement pour l'axe verticale et donc la distance par rapport à l'axe x du drone. Cependant le raisonnement sur l'axe horizontal n'est pas exactement la même et suit l'équation suivante :

$$\begin{aligned} \text{distance_abs_voiture} &= \sqrt{H^2 + \text{distance_voiture_x}^2} \\ \text{distance_voiture_y} &= \text{distance_abs_voiture} * \tan(\beta_v) * \frac{Py}{Resy} \end{aligned} \quad (4)$$

ainsi connaissant la position du drone, on peut déterminer la position de la voiture qui est juste celle du drone plus le vecteur dans le repère du drone de composante ($\text{distance_voiture_x}$, $\text{distance_voiture_y}$, $-H$). Cette méthode a été créée lors des premières itérations, par la suite cette méthode qui reste rudimentaire a été améliorée

4.2 Calcul commande moteurs

Un fois l'image récupéré et les différentes informations calculées, position véhicule, distance véhicule..., il fallait convertir ces informations en des commandes moteurs; c'est à dire des messages que le drone pourrait comprendre et ainsi interpréter en fonction des informations récupérées et calculées. Pour ce faire, après des recherches, nous avons choisi de s'appuyer sur PX4.

4.2.1 PX4 (Pixhawk)

PX4 est un système de pilote automatique open source. Développé par des développeurs de classe mondiale de l'industrie et du milieu universitaire, et soutenu par une communauté mondiale active, il alimente toutes sortes de véhicules, des drones aux véhicules terrestres et submersibles. Nous avons pu, de plus, en avoir un aperçu en situation réelle lors d'une UE montage de drone durant laquelle on a pu concevoir notre propre drone auquel nous avons attaché un contrôleur de vol pixhawk 4 utilisant ce système.

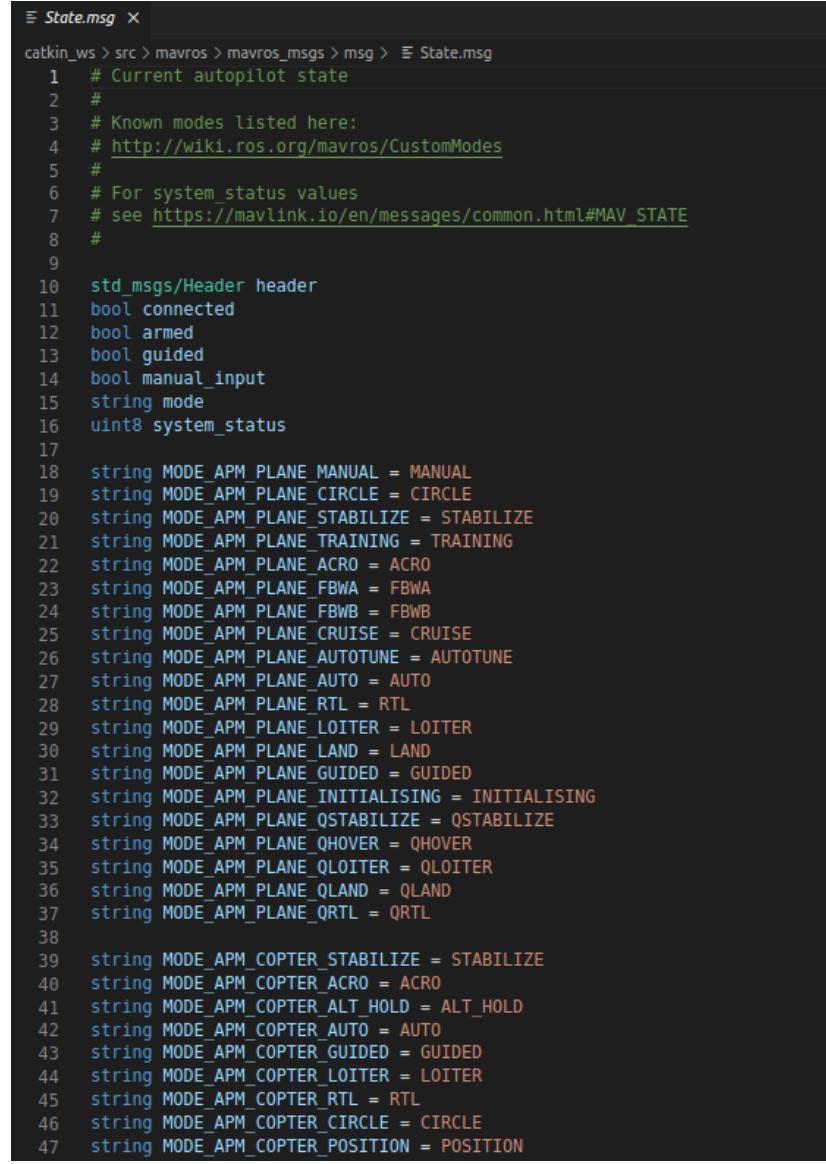
Un des avantages qui nous a amené également à utiliser PX4 c'est que le système est compatible avec ROS que l'on avait déjà choisis d'utiliser en amont. Grâce à PX4 on peut donc communiquer avec notre système (drones), et ainsi lui demander de passer en mode "OFFBOARD". Ce mode permet de passer le drone en mode pilote automatique, ainsi il nous suffit d'envoyer des positions au drone pour qu'il les atteigne de lui-même. Mais la grande force de PX4 c'est qu'il permet de pouvoir également récupérer toute sorte de message sur l'état du drone, de la batterie à son odométrie, PX4 permet d'utiliser la méthode de "Subscriber", "Publisher" de ROS facilement via des *topics* qui correspondent à des états ou des actions possibles du drone. L'ensemble de ses *topics* sont décrit dans le package "MAVROS".

4.2.2 MAVROS

MAVROS est un *package* qui fournit des noeuds de communication MavLink qui permettent de communiquer avec le drone via ROS. C'est ce *package*

4 DÉVELOPPEMENT TECHNIQUE

qui nous permet donc de communiquer avec le drone et de pouvoir récupérer l'ensemble des informations qu'envoie le drone et de pouvoir lui envoyer des commandes. Le *package* met à disposition des fichiers de type .msg ou .srv qui permettent de d'envoyer des messages compréhensibles par PX4 et qui les rendent compréhensibles par le drone.



```
catkin_ws > src > mavros > mavros_msgs > msg > State.msg
  1 # Current autopilot state
  2 #
  3 # Known modes listed here:
  4 # http://wiki.ros.org/mavros/CustomModes
  5 #
  6 # For system_status values
  7 # see https://mavlink.io/en/messages/common.html#MAV\_STATE
  8 #
  9
10 std_msgs/Header header
11 bool connected
12 bool armed
13 bool guided
14 bool manual_input
15 string mode
16 uint8 system_status
17
18 string MODE_APM_PLANE_MANUAL = MANUAL
19 string MODE_APM_PLANE_CIRCLE = CIRCLE
20 string MODE_APM_PLANE_STABILIZE = STABILIZE
21 string MODE_APM_PLANE_TRAINING = TRAINING
22 string MODE_APM_PLANE_ACRO = ACRO
23 string MODE_APM_PLANE_FBWA = FBWA
24 string MODE_APM_PLANE_FBWB = FBWB
25 string MODE_APM_PLANE_CRUISE = CRUISE
26 string MODE_APM_PLANE_AUTOTUNE = AUTOTUNE
27 string MODE_APM_PLANE_AUTO = AUTO
28 string MODE_APM_PLANE_RTL = RTL
29 string MODE_APM_PLANE_LOITER = LOITER
30 string MODE_APM_PLANE_LAND = LAND
31 string MODE_APM_PLANE_GUIDED = GUIDED
32 string MODE_APM_PLANE_INITIALISING = INITIALISING
33 string MODE_APM_PLANE_QSTABILIZE = QSTABILIZE
34 string MODE_APM_PLANE_QHOVER = QHOVER
35 string MODE_APM_PLANE_QLOITER = QLOITER
36 string MODE_APM_PLANE_QLAND = QLAND
37 string MODE_APM_PLANE_QRTL = QRTL
38
39 string MODE_APM_COPTER_STABILIZE = STABILIZE
40 string MODE_APM_COPTER_ACRO = ACRO
41 string MODE_APM_COPTER_ALT_HOLD = ALT_HOLD
42 string MODE_APM_COPTER_AUTO = AUTO
43 string MODE_APM_COPTER_GUIDED = GUIDED
44 string MODE_APM_COPTER_LOITER = LOITER
45 string MODE_APM_COPTER_RTL = RTL
46 string MODE_APM_COPTER_CIRCLE = CIRCLE
47 string MODE_APM_COPTER_POSITION = POSITION
```

FIGURE 14 – Exemple d'un message mavros (State)

4 DÉVELOPPEMENT TECHNIQUE

```
≡ SetMode.srv ×
catkin_ws > src > mavros > mavros_msgs > srv > ≡ SetMode.srv
1  # set FCU mode
2  #
3  # Known custom modes listed here:
4  # http://wiki.ros.org/mavros/CustomModes
5
6  # basic modes from MAV_MODE
7  uint8 MAV_MODE_PREFLIGHT      = 0
8  uint8 MAV_MODE_STABILIZE_DISARMED = 80
9  uint8 MAV_MODE_STABILIZE_ARMED   = 208
10 uint8 MAV_MODE_MANUAL_DISARMED = 64
11 uint8 MAV_MODE_MANUAL_ARMED    = 192
12 uint8 MAV_MODE_GUIDED_DISARMED = 88
13 uint8 MAV_MODE_GUIDED_ARMED   = 216
14 uint8 MAV_MODE_AUTO_DISARMED  = 92
15 uint8 MAV_MODE_AUTO_ARMED     = 220
16 uint8 MAV_MODE_TEST_DISARMED  = 66
17 uint8 MAV_MODE_TEST_ARMED    = 194
18
19 uint8 base_mode   # filled by MAV_MODE enum value or 0 if custom_mode != ''
20 string custom_mode # string mode representation or integer
21 ---
22 bool mode_sent   # Mode known/parsed correctly and SET_MODE are sent
23
```

FIGURE 15 – Exemple d'un service mavros (SetMode)

Évidemment ce travail a été possible car PX4 est accompagnée d'une documentation très riche. Malheureusement nous avons pas réussis à faire marcher, en simulation, l'exemple de mise en "OFFBOARD" du drone sous ROS2, ce qui nous a obligé à utiliser ROS1 pour contrôler le drone. Mais le mode de fonctionnement est le même entre ROS2 et ROS1.

```
nico@nico-Lenovo-Y520-15IKBM:~$ rostopic list | column -s " "
/diagnostics                                         /mavros/global_position/rel_alt          /mavros/local_position/velocity_local      /mavros/setpoint_position/local
/mavlink/from                                         /mavros/global_position/set_gp_origin   /mavros/log_transfer/raw_log_data        /mavros/setpoint_raw/attitude
/mavlink/gcs_ip                                       /mavros/gps_input/gps_input             /mavros/log_transfer/raw_log_entry       /mavros/setpoint_raw/global
/mavlink/to                                           /mavros/gps_rtk/rtk_baseline            /mavros/nag_calibration/report         /mavros/setpoint_raw/local
/mavros/actuator_control                           /mavros/gpsstatus/gps1/rtk              /mavros/manual_control/status          /mavros/setpoint_raw/target_attitude
/mavros/adsb/send                                    /mavros/gpsstatus/gps1/raw              /mavros/manual_control/control        /mavros/setpoint_raw/target_global
/mavros/adsb/vehicle                                /mavros/gpsstatus/gps1/rtk              /mavros/manual_control/send          /mavros/setpoint_raw/target_local
/mavros/altitude                                     /mavros/gpsstatus/gps2/rtk              /mavros/mission/reached            /mavros/setpoint_trajectory/desired
/mavros/battery                                      /mavros/gpsstatus/gps2/raw              /mavros/mission/waypoints           /mavros/setpoint_trajectory/local
/mavros/battery2                                     /mavros/hil/actuator_controls         /mavros/noac/pose                     /mavros/setpoint_velocity/cmd_vel
/mavros/camera_imu/jnc/cam_imu_stamp               /mavros/hil/hil_controls              /mavros/noac/control/command        /mavros/setpoint_velocity/cmd_vel_unstamped
/mavros/camera/image_captured                      /mavros/hil/imu/gyro                /mavros/mount_controller/command     /mavros/state
/mavros/companion_process/status                   /mavros/hil/imu/accel               /mavros/mount_controller/status      /mavros/statustext/recv
/mavros/debug_value/debug                          /mavros/hil/rc_inputs               /mavros/nav_controller/send         /mavros/statustext/send
/mavros/debug_value/debug_vector                  /mavros/hil/state                 /mavros/odometry/in                 /mavros/target_actuator_control
/mavros/debug_value/named_value_float            /mavros/home_position/home          /mavros/odometry/out                /mavros/terrain/report
/mavros/debug_value/named_value_int             /mavros/position/position_set       /mavros/obstacle/computer/status    /mavros/time_reference
/mavros/debug_value/send                         /mavros/imu/data                  /mavros/param/param_value           /mavros/time_stamps
/mavros/esc_info                                    /mavros/imu/data_raw              /mavros/play_tune                  /mavros/trajectory/desired
/mavros/esc_status                                 /mavros/imu/diff_pressure         /mavros/pxflow/ground_distance     /mavros/trajectory/generated
/mavros/esc_telemetry                             /mavros/imu/mag                  /mavros/pxflow/raw/optical_flow_rad /mavros/trajectory/path
/mavros/estimator_status                          /mavros/imu/static_pressure       /mavros/pxflow/raw/send            /mavros/tunnel/in
/mavros/extended_state                           /mavros/ins/temperature_baro     /mavros/pxflow/temperature         /mavros/tunnel/out
/mavros/fake_gps/noacap/tf                       /mavros/ins/temperature_baro     /mavros/rally_point/status         /mavros/vision_pose/pose
/mavros/global_position/compass_fixes            /mavros/landing/target/lt_marker   /mavros/rally_point/waypoints      /mavros/vision_pose/pose_cov
/mavros/global_position/compass_hdg             /mavros/landing/target/pose       /mavros/rc/in                      /mavros/vision_speed/speed_twist_cov
/mavros/global_position/global                  /mavros/landing/target/pose_in    /mavros/rc/out                     /mavros/wind_estimation
/mavros/global_position/gp_lp_offset            /mavros/local_position/accel     /mavros/rc/override                /rosout
/mavros/global_position/gp_origin               /mavros/local_position/odom      /mavros/setpoint_accel/accel       /rosout_agg
/mavros/global_position/home                   /mavros/local_position/pose      /mavros/setpoint_attitude/cmd_vel  /tf
/mavros/global_position/local                  /mavros/local_position/pose_cov  /mavros/setpoint_attitude/thrust   /tf_static
/mavros/global_position/local_px                /mavros/local_position/velocity_body /mavros/setpoint_position/global_to_local
/mavros/global_position/raw/gps_vel            /mavros/local_position/velocity_body_cov /mavros/setpoint_position/global
```

FIGURE 16 – Liste des *topics* mise à disposition par mavros

Vous pouvez voir sur la figure 16 la liste exhaustive mise à disposition par mavros et qui permet de souscrire et publier des messages au système. Plus concrètement dans le projet nous utilisons les *topics* suivants :

5 SIMULATION

- "mavros/global_position/local" qui permet de récupérer en directe la position du drone dans son espace à lui.
- "mavros/state" qui permet de récupérer l'état du drone notamment pour savoir dans quel mode le drone se trouve mais aussi s'il est armé ou pas, pour savoir si l'on peut décoller.
- "mavros/setpoint_position/local" qui permet d'envoyer des positions au drone qu'il atteindra une fois passer en mode "OFFBOARD".
- "mavros/cmd/arming" qui permet d'envoyer une commande pour armer le drone avant décollage.
- "mavros/set_mode" qui permet de changer le mode du drone.

Ainsi à l'aide de ses *topics* on arrive à contrôler un drone en pilote automatique.

5 Simulation

Comme expliqué précédemment étant donné que nous n'avions pas accès à un système réel nous avons réalisé l'ensemble des tests et développements (pour le côté drone) via des simulateurs.

5.1 Simulation (Gazebo)

Gazebo est un puissant simulateur 3D orienté pour les systèmes autonomes et très utilisé notamment pour tester la vision par ordinateur et visualiser le comportement des systèmes en situation. Nous avons utilisé Gazebo car c'est un simulateur qui est très utilisé, notamment par PX4 et ROS qui rendent l'utilisation du simulateur très abordable.

PX4 communique avec le simulateur pour recevoir les données des capteurs depuis le monde simulé et envoyer des commandes moteurs; afin d'avoir des données télémétriques et odométriques précises on peut communiquer via QGroundControl.

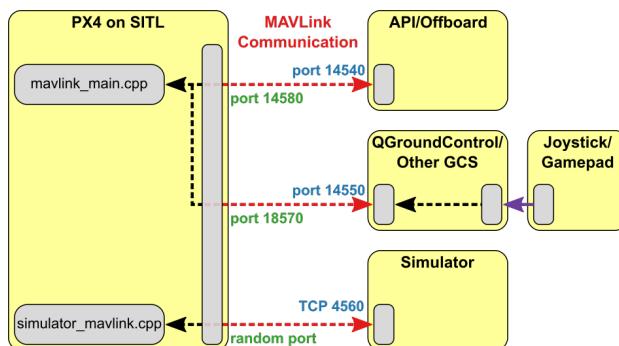


FIGURE 17 – Schéma de communication entre PX4-ROS-Gazebo

Ici ROS fait office d'API/Offboard et Gazebo de simulateur. Ainsi Gazebo nous a permis de simuler un quadrirotor, plus précisément le 3rd iris.

5 SIMULATION

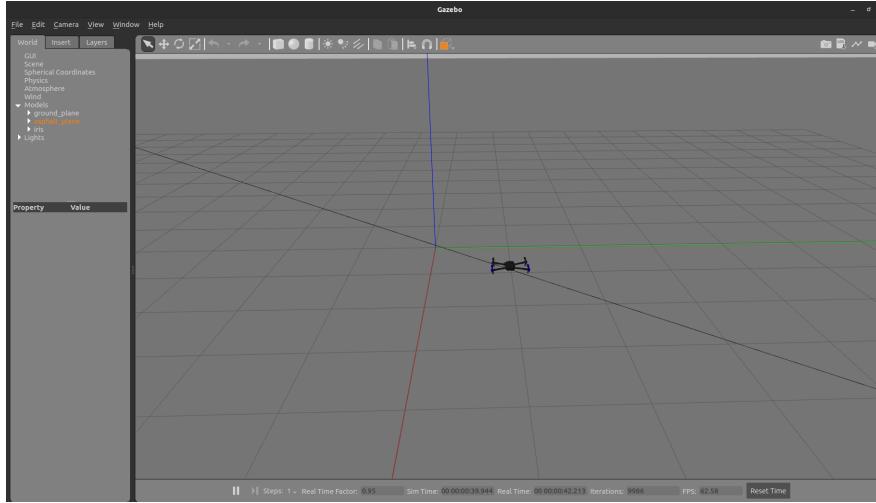


FIGURE 18 – Gazebo 3dr Iris

5.2 Simulation (Python et RVIZ)

Une partie simulation avec Python et RVIZ a été réalisée. Celle-ci permet de simuler le drone avec une consigne pour chaque moteur, u_1, u_2, u_3, u_4 . Afin de créer un contrôle plus précis de la position du drone, il peut être nécessaire de déterminer les équations qui régissent la physique du drone, une fonction sûrement non-linéaire donc il sera difficile d'en déduire une solution, d'autant plus qu'il peut y avoir plusieurs solutions afin d'atteindre l'objectif. Heureusement beaucoup d'outils existent à ce jour afin de réaliser ceci dont le Kalman. Nous avions commencé à travailler sur cette piste de réflexion, cependant par la suite, l'utilisation de PX4 a permis de régler le problème de la commande d'un drone.

Comment déterminer les équations physiques du drone ? Tout d'abord nous pouvons essayer une approche cinématique. Ce qui nous mène à l'équation suivante :

avec $g(\text{gravity})$, $\text{actionneur}_x \in \mathbb{R}$

$$\begin{cases} z'' = \text{actionneur}_1 + \text{actionneur}_2 + \text{actionneur}_3 + \text{actionneur}_4 + g \\ \theta_x'' = \text{actionneur}_2 - \text{actionneur}_4 \\ \theta_y'' = \text{actionneur}_1 - \text{actionneur}_3 \\ \theta_z'' = (\text{actionneur}_1 + \text{actionneur}_3) - (\text{actionneur}_2 + \text{actionneur}_4) \end{cases}$$

à partir de ces équations par relation d'intégration on en déduit

$$\begin{cases} z'_{drone} = \int z'' dt \\ \theta'_x_{drone} = \int \theta''_x dt \\ \theta'_y_{drone} = \int \theta''_y dt \\ \theta'_z_{drone} = \int \theta''_z dt \end{cases}$$

Il faut bien être conscient qu'ici z est représenté dans le repère du drone, de même pour x , et y . Ce changement de repère est le produit des trois angles de rotation ce que l'on peut appeler une matrice d'Euler. Ainsi on a par exemple : $v = \text{Eulermat} * v_{drone}$ où la matrice d'Euler s'écrit comme ci-dessous

$$\begin{aligned} R &= R_{z''}(\gamma) \cdot R_{y'}(\beta) \cdot R_z(\alpha) \\ &= \begin{pmatrix} c\gamma & s\gamma & 0 \\ -s\gamma & c\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} c\beta & 0 & -s\beta \\ 0 & 1 & 0 \\ s\beta & 0 & c\beta \end{pmatrix} \cdot \begin{pmatrix} c\alpha & s\alpha & 0 \\ -s\alpha & c\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} c\gamma c\beta c\alpha - s\gamma s\alpha & c\gamma c\beta s\alpha + s\gamma c\alpha & -c\gamma s\beta \\ -s\gamma c\beta c\alpha - c\gamma s\alpha & -s\gamma c\beta s\alpha + c\gamma c\alpha & s\gamma s\beta \\ s\beta c\alpha & s\beta s\alpha & c\beta \end{pmatrix} \end{aligned}$$

FIGURE 19 – Matrice d'Euler

ainsi si on dérive v on obtient : $v' = \text{EulerMat}' * v_{drone} + \text{EulerMat} * acc_{drone}$ avec le vecteur acc_{drone} qui est l'accélération dans le repère du drone et qui est connu grâce au système d'équations précédent. On réalise ensuite le même raisonnement pour la rotation, sans oublier la loi de transport. A partir de ces équations, on peut donc déterminer la position du drone par les relations d'intégration suivante :

$$\begin{cases} x = \int z' dt \\ y = \int y' dt \\ z = \int y' dt \\ \theta_x = \int \theta'_x dt \\ \theta_y = \int \theta'_y dt \\ \theta_z = \int \theta'_z dt \end{cases}$$

Lors d'une simulation informatique l'intégration est remplacée par définition de l'intégrale à l'aide de séries avec un pas dt qui tend vers 0 si possible.

On sélectionne ces 4 variables à commander : z , θ_x , θ_y et θ_z . Nous avons aussi 4 actionneurs, donc par les lois de l'automatisme, comme nous avons autant de d'actionneurs différents et 4 variables que nous voulons commander. Soit, plus précisément, les équations d'états et d'observation suivantes :

$$\begin{cases} \dot{x} = A x + B u, \\ y = C x + D u \end{cases}$$

5 SIMULATION

Si le rang de ce critère de commandabilité : $\text{rang} [B \ AB \ \dots \ A^{n-1}B] = n$ est égal ou supérieur au nombre de variable que l'on veut commander alors notre système est commandable.

5.2) le système que nous avons est commandable plus précisément : $z, \theta_x, \theta_y, \theta_z$ sont commandables. Bien sûr nous voudrons commander x et y mais ceux-ci sont dépendant de θ_x, θ_y , ainsi il suffira normalement juste d'exploiter cette relation. Ainsi afin d'avoir une seule opération du type :

$$X(t+1) = X(t) + dt*X(t)'$$

où X serait une matrice et X' sa dérivé comportant à eux deux toutes les variables d'états apparaissent voir plus, ceci est le cas si on a une relation différentielle. Sur le drone nous avons une relation différentielle du deuxième ordre donc afin de réussir à tout avoir en 1 seule matrice X avec $X(t+1) = X(t) + dt*X(t)$ il suffit de poser $X = (x, y, z, \theta_x, \theta_y, \theta_z, x', y', z', \theta_x', \theta_y', \theta_z')$ Cependant tout ceci n'est pas suffisant. On comprend par la suite que beaucoup de donnée manque à l'appel tels que l'inertie du système, la masse, et d'autres mesure. C'est pourquoi un vrai PFD (Principe Fondamental de la Dynamique) est nécessaire afin d'exprimer l'état du système en fonction des actionneurs, car les actionneurs exercent une force résultante sur le drone. De plus l'avantage du PFD est d'intégrer toute force physique, ce qui nous mène aux équations suivantes :

$$\begin{cases} \sum \vec{F}_i = m\vec{a} \\ C_{ext} = J_{\Delta}\alpha \end{cases}$$

avec J_{Δ} une matrice d'inertie, m la masse et α l'accélération angulaire

Pour plus de détails merci de consulter le code sur le GitHub ci-contre : [dynamic_sim.py](#)

6 Application à un système réel - Cozmo

6.1 Présentation du système



FIGURE 20 – Cozmo et le véhicule à suivre

Dans cette partie, nous allons nous intéresser plus particulièrement à une application sur un système réel des principes énoncés précédemment. C'est à dire, suivre une voiture grâce à un drone qui sera, ici, terrestre. En effet, pour cela, nous avons décidé d'utiliser les moyens que nous avions à notre disposition soit : le drone terrestre Cozmo et une voiture miniature.

Cozmo est un robot éducatif se programmant soit grâce à du codage par blocs depuis un téléphone/tablette Android soit avec Python depuis un Ordinateur au travers d'une API fournie par le constructeur. Ce qui nous a permis de déduire qu'il serait un bon outil pour le projet est que tous les calculs et analyses sont déportées sur l'ordinateur, ce qui permet de faire de l'inférence avec un réseau de neurone sans être limité par la puissance de calcul embarquée dans le robot.

Comme vous pouvez le voir Figure 21, Cozmo est composé de 3 groupes de circuits reliés à la carte mère : un premier dédié aux mouvements de ses roues, de sa tête et de son bras ; un second dédié à la vision ; et un 3ème dédié à la connectivité et à l'interprétation des commandes reçues.

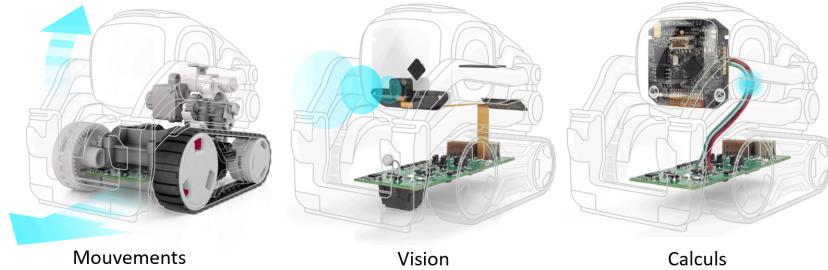


FIGURE 21 – Vues éclatées Cozmo

6.2 Mise en place du projet

6.2.1 Présentation globale

Afin de répondre à la problématique du projet grâce à ce matériel, nous avons donc décider de procéder de la manière suivante :

1. Nous allons entraîner un réseau de neurone grâce à PyTorch et YoloV5 à partir d'images prises depuis Cozmo
2. Nous allons récupérer l'image de la caméra au travers de l'API Python de Cozmo (Image 320x240 N&B compressée pour passer à travers le réseau)
3. Nous allons appliquer un algorithme d'inférence permettant de délimiter la zone où se trouve la voiture dans l'image
4. Grâce à des changements de bases, nous allons déterminer où se trouve la voiture dans le repère du robot Cozmo
5. Nous utiliserons ces informations de position de la voiture afin de contrôler les roues de Cozmo et faire en sorte qu'il suive la voiture.

Le fonctionnement de notre système sera celui présenté en Figure 22. En effet, le robot Cozmo pour fonctionner doit être relié par Wifi à un téléphone portable Android. Celui-ci, à travers le mode "Débogueur" d'Android transmet les données reçues du robot par USB à l'ordinateur grâce à "Android Debug Bridge". Ensuite, nous utiliserons l'API Python Cozmo afin d'échanger des données avec le robot (récupérer les données de la caméra et contrôler ses moteurs).

Le traitement d'image sera réalisé avec PyTorch et les coeurs CUDA de la carte NVIDIA (GTX 960M) de l'ordinateur afin d'utiliser l'accélération matérielle et ainsi détecter la voiture en temps réel (Impossible de réaliser la même opération en temps réel uniquement avec le CPU Intel i7 de la machine)

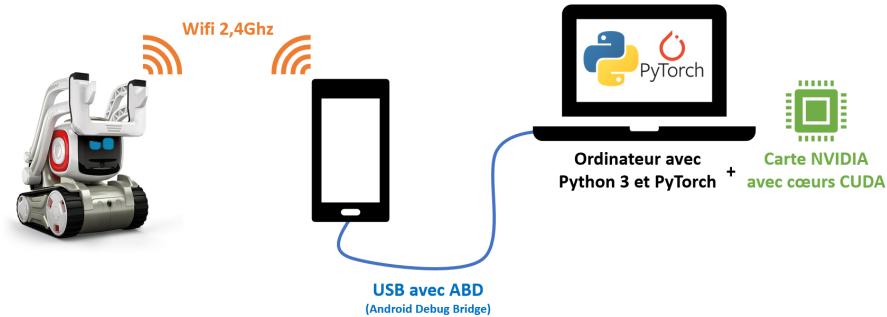


FIGURE 22 – Schéma fonctionnement système

6.2.2 Calibration de la caméra

Afin de réaliser la calibration de la Caméra, nous avons utilisé des images prises depuis la caméra du Cozmo avec le damier d'OpenCV (Figure 23 et une adaptation du code (Cozmo_follow_car/Calibration_camera.py) fourni par la documentation OpenCV afin d'obtenir la matrice de la caméra du robot Cozmo (Figure 24).

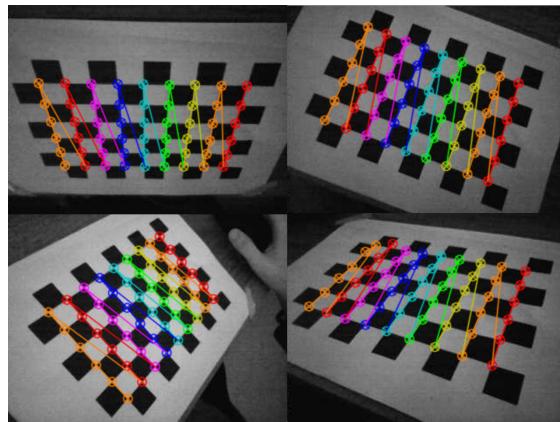


FIGURE 23 – Calibration de la camera à partir du damier OpenCV

6 APPLICATION À UN SYSTÈME RÉEL - COZMO

```
# Matrix of the Cozmo camera (pinhole model) given by "Calibration_Camera.py"
MatrixCamera = np.array([[291.41193986, 0., 170.58829057],
                        [0., 291.0430028, 108.7210315],
                        [0., 0., 1.]])

# fx and fy are the focal lengths expressed in pixel units
fx = MatrixCamera[0][0]
fy = MatrixCamera[1][1]

# (cx,cy) is the image center
cx = MatrixCamera[0][2]
cy = MatrixCamera[1][2]
```

FIGURE 24 – Matrice de la caméra obtenue par la Calibration et paramètres de la caméra extraits de cette matrice

Nous obtenons ainsi les longueurs focales F_x et F_y de la caméra, ainsi que C_x et C_y , les coordonnées du centre optique de la caméra. Nous utiliserons par la suite ces valeurs afin d'estimer la position du véhicule dans le repère du robot.

6.2.3 Changement de repère de la caméra vers celui du robot Cozmo

Afin d'obtenir la position de la voiture dans le repère du robot Cozmo, nous avions besoin de faire un changement de repère (ou encore changement de base) afin de trouver la matrice de transformation nous permettant de passer du repère de la caméra vers celui du robot.

En Figure 25 se trouvent donc le schéma annoté et la modélisation des deux repères (caméra et robot) selon les différentes dimensions du robot Cozmo.

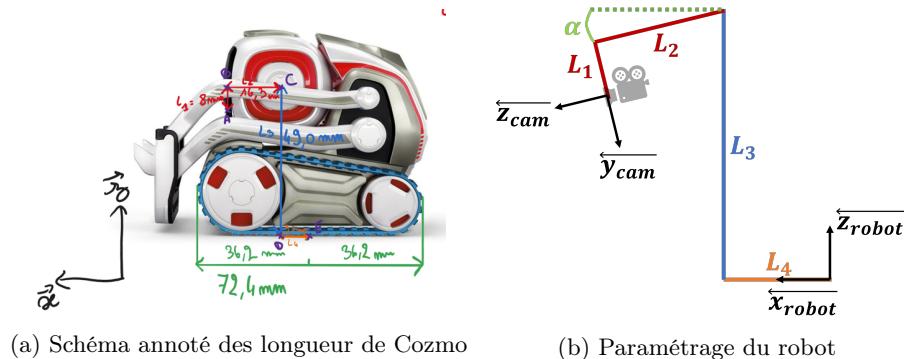


FIGURE 25 – Dimensions du robot Cozmo

Le programme présenté en Figure 26 nous permet donc grâce à 4 translations et une rotation et des paramètres représentant les différentes longueurs d'obtenir la matrice de transformation permettant de passer du repère de la caméra à celui du robot.

6 APPLICATION À UN SYSTÈME RÉEL - COZMO

```

def translation(vector):
    return np.array([[1, 0, 0, vector[0]],
                   [0, 1, 0, vector[1]],
                   [0, 0, 1, vector[2]],
                   [0, 0, 0, 1]])

def Rx(alpha):
    return np.array([[1, 0, 0, 0],
                   [0, np.cos(alpha), -np.sin(alpha), 0],
                   [0, np.sin(alpha), np.cos(alpha), 0],
                   [0, 0, 0, 1]])

def Ry(alpha):
    return np.array([[np.cos(alpha), 0, np.sin(alpha), 0],
                   [0, 1, 0, 0],
                   [-np.sin(alpha), 0, np.cos(alpha), 0],
                   [0, 0, 0, 1]])

def Rz(alpha):
    return np.array([[np.cos(alpha), -np.sin(alpha), 0, 0],
                   [np.sin(alpha), np.cos(alpha), 0, 0],
                   [0, 0, 1, 0],
                   [0, 0, 0, 1]])

L1 = 8
L2 = 16.3
L3 = 49
L4 = 9.2
alpha = np.deg2rad(25)

T_rc = translation([L4, 0, L3]) @ Ry(alpha) @ translation([-L2, 0, -L1])

T_rc = T_rc @ ([[0, 0, 1, 0],
                 [-1, 0, 0, 0],
                 [0, -1, 0, 0],
                 [0, 0, 0, 1]])

T_rc = T_rc @ np.array ([0, 0, 0, 1])
print (T_rc)

```

FIGURE 26 – Code Matrice de transformation du changement de base caméra → robot

6.2.4 Estimation position du véhicule dans le repère robot

Pour pouvoir estimer la position de la voiture dans le repère robot nous allons utiliser plusieurs éléments définis dans les parties précédentes :

1. La Matrice de la caméra qui nous donne ses paramètres de longueur focale (F_x et F_y) et de centre optique (C_x , C_y)
2. Les coordonnées de la *Bounding box* données grâce à la localisation calculée par notre réseau de neurone englobant la voiture sur l'image.
3. La Matrice de transformation permettant de passer du repère de la caméra à celui du robot : T_{rc} .

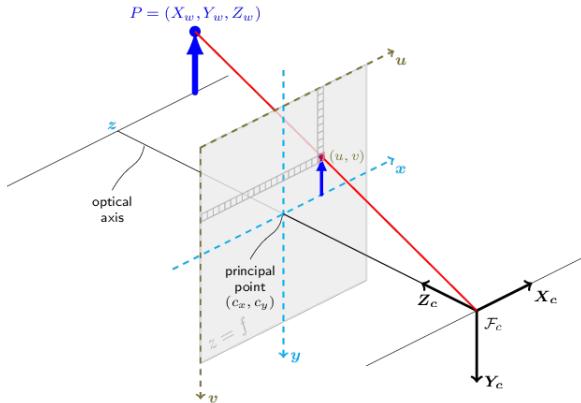


FIGURE 27 – Modèle mathématique reliant les coordonnées d'un point dans l'espace tridimensionnel et sa projection sur le plan image d'un sténopé idéal

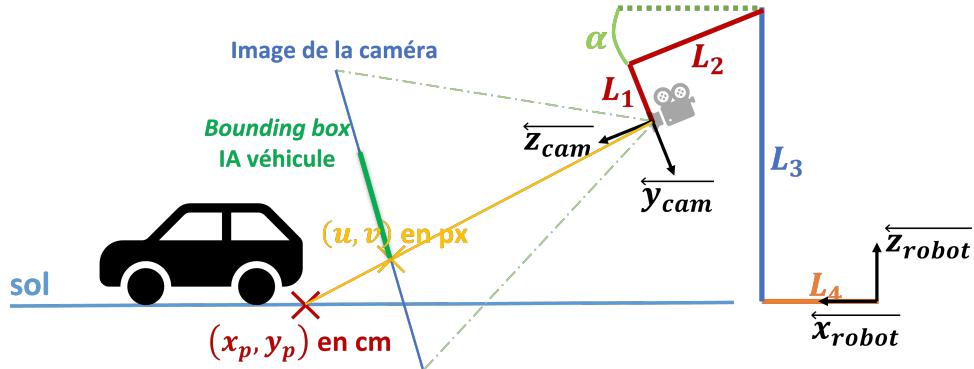


FIGURE 28 – Schéma représentant la méthode d'estimation de la position de la voiture dans le repère robot

Grâce aux 3 éléments cités précédemment, au Schéma présenté en Figure 28 et au modèle *pinhole* (Figure 27) nous allons pouvoir calculer la position du véhicule à suivre.

Premièrement, nous voulons estimer les coordonnées d'un point (en cm) dans le repère de la caméra grâce aux coordonnées de ce même point (en pixel) dans l'image capturée par la caméra. Pour cela, nous calculons les coordonnées (x_p, y_p) de l'intersection entre le plan défini par le sol et le prolongement du segment partant de la caméra et passant les coordonnées (u, v) du point dans l'image. (Voir Figure 28). Cela revient à résoudre le système d'équation égalisant le plan défini par le sol et le vecteur prolongé partant de la caméra et passant par le point (u, v) dans l'image.

Dans un second temps, nous définissons le point de coordonnées (u, v) comme étant le point milieu du segment inférieur de la *bounding box* retournée par notre inférence.

En effet, en faisant cela, nous obtiendrons dans l'image les coordonnées en pixels d'un point du monde réel que nous supposons sur le sol et étant milieu du segment représentant l'intersection entre un rectangle collé à l'arrière de la voiture et le plan du sol.

Nous obtenons donc les coordonnées représentant en simplifiant, le centre arrière du véhicule (au niveau du sol) dans le repère de la caméra.

Nous utilisons donc la matrice de Transformation T_{rc} afin de passer ces coordonnées du repère de la caméra vers celui du robot.

Enfin, nous avons alors les coordonnées (x_p, y_p) représentant le centre arrière du véhicule dans le repère du robot. Il suffirait alors par la suite d'ajouter la moitié de la longueur du véhicule à la coordonnée x_p de ce point afin d'obtenir les coordonnées du centre du véhicule.

6.2.5 Contrôle des moteurs de Cozmo

Le contrôle des moteurs des chenilles de Cozmo se fait alors simplement de la manière suivante :

$$right_rot_wheel = C \times line_speed + K \times rot_speed$$

$$left_rot_wheel = C \times line_speed - K \times rot_speed$$

Avec $line_speed$ étant la coordonnée x_p et rot_speed la coordonnée y_p . K et C sont alors simplement des coefficients proportionnels déterminés de manière empirique afin d'ajuster la commande moteur (en mm/s)

6.3 Tests unitaires

Des tests unitaires ont été écrits, ils sont importants et assurent le fonctionnement "local" de certaines méthodes, afin de s'assurer de certains critères logiques ou mathématiques.

Par exemple pour tester les commandes drones, étant donné que nous utilisions ROS, nous avons testé si les différents "Subscriber" et "Publisher" publient bien ce que l'on voulait. La simulation et les *topics* que mets à disposition mavros a permis de tester si le drone se déplace bien aux positions que nous lui envoyions étant donné que nous pouvions voir en temps réel l'odométrie du drone et même récupérer sa position dans le repère monde.

6.3.1 Estimation de la position d'un point sur le sol vis à vis de Cozmo

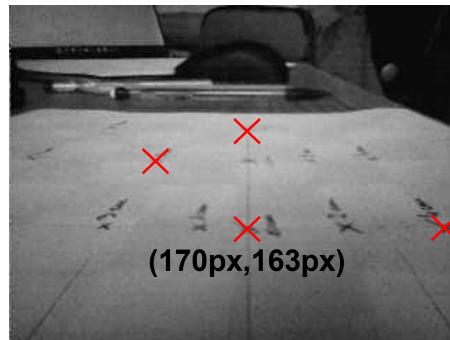


FIGURE 29 – Image de la feuille de test capturée par la caméra de Cozmo

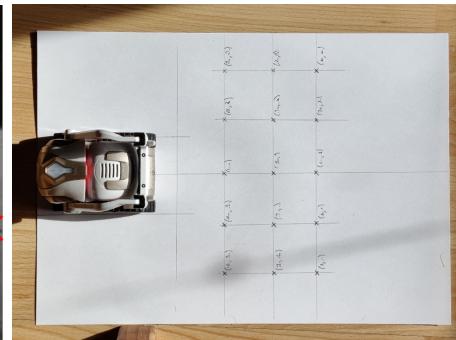


FIGURE 30 – Photo de Cozmo positionné sur la feuille de test

Afin de tester l'estimation de la position d'un point dans le repère robot à partir d'une image de la caméra de Cozmo, nous avons créé la feuille de tests représentée Figure 29. Nous avons la vue de cette même feuille à travers la caméra de Cozmo Figure 30. Il nous faut donc récupérer les coordonnées en pixels dans

6 APPLICATION À UN SYSTÈME RÉEL - COZMO

l'image de chacun des points et vérifier l'erreur entre la position réelle du point et celle calculée par notre programme (`Cozmo_follow_car/Verif_distance_point_robot.py`).

Nous prenons comme exemple trois points de tests A, B et C nous obtenons les résultats répertoriés dans le tableau suivant à partir du programme de test présenté en Figure 31 :

	A	B	C
Coordonnées dans l'image (en px)	(170,163)	(170,115.5)	(170,93)
Coordonnées réels (en mm)	(100,0)	(150,0)	(200,0)
Coordonnées calculés (en mm)	(100.34,0.17)	(149.31,0.27)	(198.58,0.36)
Erreur absolue sur x (en mm)	0.34	-0.69	-1.42
Erreur absolue sur y (en mm)	0.17	0.27	0.36

FIGURE 31 – Code Tests unitaire vision Cozmo

Avec une caméra d'une résolution de seulement 320x240 pixels, nous obtenons des erreurs maximales d'à peine plus d'1mm sur l'axe x à 20cm du robot. Nous considérons donc que cette erreur est négligeable vis à vis de l'utilisation de la valeur qui ne nous sert qu'à contrôler la vitesse des chenilles du robot Cozmo de manière proportionnelle à la distance qui le sépare du véhicule. Nous pouvons donc dire que ce test est validé.

6.4 Tests haut-niveau

Ces tests haut-niveaux permettent de valider ou non un critère/objectif utilisateur. Rappelons les 3 critères :

- Implémentable sur tout type d'OS, drone terrestre/volant, ...
- Communiquer la position de la voiture.
- Sélectionner le schéma de suivi à tout moment

L'avantage de la compilation par ROS est de pouvoir créer des exécutables sans avoir besoin des fichiers sources. Cependant lorsque les exécutables sont construits, des bibliothèques objets sont aussi apportées afin de pouvoir appeler des fonctions tels que numpy, matplotlib, pytorch. Si ces bibliothèques objets ne sont pas exportées dans le dossier build, alors lors de l'exécution celui-ci essaiera de chercher les bibliothèques manquantes sur la machine locale. Or, le drone ne possède pas nécessairement ces bibliothèques.

C'est pourquoi nous avons créé un docker. Avec la technologie Docker, nous pouvons traiter les conteneurs comme des machines virtuelles très légères et modulaires. En outre, ces conteneurs offrent une grande flexibilité : créer, déployer, copier et déplacer d'un environnement à un autre. Ainsi nous sommes partis d'un conteneur Unix avec ROS2 pré-installé. Si la compilation s'exécute normalement alors le test est vérifié, sinon le test a échoué, ce qui nous mènera donc à changer les fichiers Cmakelist.txt et package.xml.

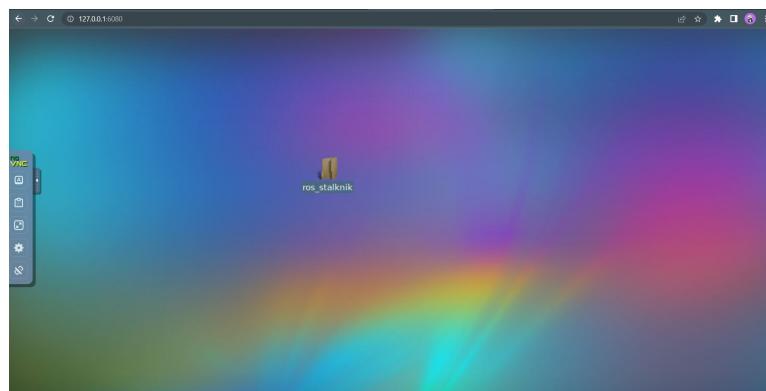


FIGURE 32 – Docker environnement

6 APPLICATION À UN SYSTÈME RÉEL - COZMO

FIGURE 33 – Execution sous Docker

Un des critères de validation de ce test serait par exemple l'exécution sans erreur de commande simple :

- start [node_name].exe & start [node2_name].exe

des fichiers qui se trouve dans le dossier "Install" et donc la partie compilée du projet.

Ce test est pour l'instant non validée pour plusieurs raisons : un fichier ".exe" n'est pas exécutable depuis une distribution UNIX, or le projet a d'abord été cross- compilé sous ROS2 pour une machine Windows. Ce qui pose le problème de comment changer la cross-compilation pour un autre OS. Ceci est une question qui se posait moins avec ROS1 car seul les distributions unix peuvent utiliser ROS1, donc tant que toutes les bibliothèques ".so", ... étaient bien exportées dans le dossier "build", alors le projet pouvait très bien fonctionner sur une autre machine.

Un test permettant de communiquer la position de la voiture a été réalisé. Celle-ci teste en outre non seulement le flux entier du système car plusieurs *nodes* sont impliqués dans le processus mais aussi la communication de la position de la voiture qui se traduira pour simplement par des logs sur ROS.

Nous avons vu précédemment que nous avions des tests unitaires pour la détection d'une voiture, cependant, dans notre cas de test haut-niveau on souhaite vérifier que le besoin utilisateur est rempli. Or, lors du traitement d'image nous avons besoin d'informations tels que le FOV de la caméra, ce que nous n'avons pas forcément sur les photos des test unitaires du traitement d'image.

Nous sommes partis d'une vidéo enregistrée via le Cozmo, car nous avons pu déterminer la FOV de la caméra, ainsi comme nous avions mesuré grossièrement la position de la voiture dans le temps, il nous suffit de vérifier que le programme nous renvoie des valeurs dans cette ordre de grandeur.

Un test qui a été validé avec succès car notre voiture se déplace d'environ 20cm ce que l'on retrouve par le traitement d'image et le calcul.

Enfin un test pour la sélection du schéma de suivi. Il permet de vérifier que selon la mission, il exécute bien ses déplacements, un test non validé pour l'instant car la partie service-client afin de changer le mode de suivi n'est pas encore fonctionnelle.

7 Améliorations

Les améliorations sont des pistes que nous aurions pu développer lors de nouvelles itérations du projet :

Une amélioration possible serait de passer la partie "Commande moteur" avec le passage dans le mode "OFFBOARD" du drone, vers ROS2 ; en effet nous n'avons pas réussi à faire fonctionner ROS2 avec PX4 pour passer le drone en autopilote, du moins en simulation, bien que pendant l'apprentissage de PX4 on a réussi à faire décoller et récupérer des données du drone depuis ROS2, le mode "OFFBOARD" ne voulait jamais s'enclencher.

Une autre amélioration au niveau de la perception, aurait été de réaliser la détection d'image jusqu'à détecter une voiture et ensuite utiliser uniquement de la localisation d'objet par segmentation des déplacements de POI (Point Of Interest). Ce genre de solution permet non seulement de libérer le CPU pour d'autres tâches et par extension diminuer la consommation du drone et donc augmenter son autonomie, un critère important pour un système embarqué.

8 Conclusion

Le but de ce projet était d'apporter une solution au sujet "suivi de véhicule à l'aide d'un drone" ; le sujet étant assez ouvert cela nous a laissé un libre choix sur la technologie à utiliser pour le développement de celui-ci. ROS s'est trouvé être un bon choix étant donné qu'il était utilisable avec PX4. Malheureusement le manque de moyens techniques réels nous a obligé à tester nos implantations pour des drones de type volant uniquement sur des espaces simulés. La partie reconnaissance d'image étant, quant à elle, l'élément-clé de ce projet, nous avons réussi à entraîner un réseau de neurone et développer le programme associé permettant (par calculs déportés) une utilisation en temps réel sur un drone terrestre (Cozmo) afin de détecter et retourner la position du véhicule suivi.

Hélas, comme expliqué dans la partie améliorations, nous n'avons pas réussi à faire fonctionner l'ensemble de notre *workflow* ROS en simultané étant donné qu'un morceau est uniquement compatible avec ROS(1) et le reste du projet a été développé sur ROS(2). Un bridge ROS1/ROS2 ou résoudre le blocage du mode OFFBOARD sous ROS2 pourraient être des solutions à notre problème que nous n'avons pu, à l'heure actuelle, implémenter.

Cependant, dans l'ensemble, ce projet nous a permis de découvrir des technolo-

8 CONCLUSION

gies qui étaient pour certains membres du groupe inconnues ou jamais utilisées auparavant et qui sont de plus en complète relation avec notre formation et qui nous seront sûrement utiles dans notre futur professionnel.

Nous remercions aussi nos encadrants M.Serge CHAUMETTE et M.Pascal DESBARATS qui ont su nous accompagner tout au long de ce projet.

RÉFÉRENCES

Références

- [1] https://en.wikipedia.org/wiki/Robot_Operating_System (accès en mars 2022)
- [2] https://docs.px4.io/master/en/simulation/ros_interface.html(accès en mars 2022)
- [3] https://docs.px4.io/master/en/ros/mavros_offboard.html(accès en mars 2022)
- [4] <http://wiki.ros.org/mavros> (accès en mars 2022)
- [5] <https://easyspin.org/easyspin/documentation/eulerangles.html> (accès en mars 2022)
- [6] https://github.com/camillul/Stalknik_MK2.git