UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Suivi d'une cible mobile par des drones autonomes

Mémoire présenté par
Julien GÉRARDY , Félicien SCHILTZ

pour l'obtention du grade de master en
Ingénieur civil électromécanicien et en mathématiques appliquées

Promoteurs
Julien HENDRICKX , François WIELANT

Lecteurs
François BAUDART, Jean-Charles DELVENNE

Année académique 2015-2016

Remerciements

Nous tenons à remercier le superviseur de notre mémoire, le professeur Julien Hendrickx, pour nous avoir conseillés et orientés tout au long de l'année ainsi que pour nous avoir donné l'opportunité de travailler sur ce sujet de mémoire.

Pour sa présence et sa disponibilité durant toute l'année ainsi que pour ses conseils avisés, nous tenons aussi à remercier François Wielant.

Nous remercions également Arnaud Jacques et Alexandre Leclère, l'autre groupe de mémorants avec qui nous avons eu le plaisir de travailler pour une partie de ce mémoire. Leur aide nous a été d'une grande utilité durant toute l'année.

Pour terminer, nous remercions l'École Polytechnique de Louvain pour l'infrastructure mise à notre disposition ainsi que nos familles et amis pour le soutien et les nombreuses relectures de ce mémoire.

Table des matières

1 Introduction **1**
- 1.1 Contexte et mise en situation . . . 1
- 1.2 Travaux réalisés antérieurement . . . 2
- 1.3 Objectifs . . . 4
  - 1.3.1 Objectifs communs aux deux groupes de mémorants . . . 4
  - 1.3.2 Objectifs de ce mémoire . . . 6

2 Parrot AR.Drone **9**

3 Etat de l'art **13**
- 3.1 L'utilisation des drones aujourd'hui . . . 13
- 3.2 Les drones et le suivi de cibles mobiles . . . 15
- 3.3 L'utilisation de ROS pour les drones . . . 18
  - 3.3.1 Lily . . . 18
  - 3.3.2 Bird MURI . . . 19
  - 3.3.3 PTAM et *ethzasl_ptam* . . . 20
  - 3.3.4 PTAM, la TUM et son contrôle en position . . . 21
- 3.4 Les algorithmes d'exploration . . . 22
  - 3.4.1 Un exemple de drone utilisant un algorithme d'exploration . . . 22
  - 3.4.2 Algorithmes d'exploration à un seul robot . . . 23
  - 3.4.3 Algorithmes d'exploration à plusieurs robots . . . 24

4 ROS : Le choix de notre système d'exploitation **25**
- 4.1 Introduction . . . 25
- 4.2 Le SDK de Parrot . . . 25
- 4.3 ROS . . . 26
  - 4.3.1 Description de ROS . . . 26
  - 4.3.2 Fonctionnement de ROS . . . 27
  - 4.3.3 Librairies externes . . . 28
- 4.4 ardrone_autonomy . . . 28

5 Premiers pas **31**
- 5.1 Le code de la TUM comme point de départ . . . 31
- 5.2 L'utilisation de PTAM de l'ETHZASL comme point de départ . . . 32

5.3 Le code des années précédentes comme point de départ 33
5.4 Conclusion sur le choix du point de départ 33

6 Intelligence artificielle et régulation 35
6.1 Nos objectifs 35
6.2 Vue d'ensemble du code 36
6.3 Multi-Strategy 37
6.4 Stratégie 39
6.4.1 Stratégie de notre implémentation de la mission des années précédentes 40
6.4.2 Stratégie de notre mission 40
6.5 Planification de trajectoire et exploration 43
6.5.1 Rôle et description de ce module 43
6.5.2 Algorithme d'exploration 44
6.6 Régulation 50
6.6.1 Introduction 51
6.6.2 Régulation en Altitude 52
6.6.3 Régulation en RotZ 56
6.6.4 Régulation en X et Y 60

7 Validation 65
7.1 Environnement de vol 65
7.2 Résultat de la mission 67
7.3 Facteurs d'échec de la mission 71

8 Amélioration du matériel 73
8.1 Ajout d'une caméra 73
8.2 Un routeur pour l'utilisation simultanée de plusieurs drones 74

9 Conclusion 77
9.1 Difficultés liées à ce mémoire 78
9.2 Améliorations possibles 78
9.2.1 Implémentation d'un calcul de trajectoire 78
9.2.2 Remplacement de l'oracle par une fonction de détection des murs . 79
9.2.3 Implémentation d'un algorithme d'exploration multi-drones 79
9.3 Notre avis sur la réalisation d'une application de cartographie multi-drones autonomes 80

A Tableau comparatif des caméras 85

B Structure du code 87

Bibliographie 91

Comment lire ce mémoire

Ce rapport de mémoire est constitué d'une septantaine de pages. Ces dernières retracent notre parcours depuis le début de l'année, le résultat de nos recherches ainsi que de nos différentes implémentations.

Le lecteur qui est déjà familier avec l'univers des drones n'est pas obligé de lire en détail les chapitres 2 et 3 pour comprendre la suite du rapport, plus axée sur la programmation et l'objectif principal de notre mémoire.

De même, le lecteur qui connait déjà le *Robot Operating System*, plus connu sous le nom de ROS ne sera pas gêné pour comprendre notre mémoire s'il ne lit pas le chapitre 4. Il est par contre essentiel que chacun lise attentivement les chapitres 6 et 7 où sont décrits en détails nos différents objectifs et accomplissements.

Nous vous souhaitons une bonne lecture.

Chapitre 1

Introduction

Dans cette introduction, nous commençons par présenter l'utilité croissante des drones dans la société actuelle (1.1). Ensuite, nous décrivons le travail qui a déjà été accompli par les étudiants mémorants des années précédentes (1.2). Enfin, nous parlons des objectifs que nous nous sommes fixés en début d'année avec l'autre groupe d'étudiants mémorants et la manière dont nous avons réparti la charge de travail (1.3).

1.1 Contexte et mise en situation

Aujourd'hui, le marché des drones est en pleine croissance. Auparavant uniquement utilisés dans le domaine militaire, ces véhicules aériens motorisés ont aujourd'hui tendance à séduire un large public de par leur coté ludique et leur prix de plus en plus démocratiques. De nombreux drones sont également utilisés dans divers secteurs tels que le cinéma, la sécurité, la construction, le transport, la cartographie... Pour se faire une idée de l'ampleur du phénomène, on peut noter que le secteur des drones en France, qui représentait 62 millions d'euros de chiffre d'affaire en 2012 en représentait déjà 288 millions en 2015. Il a généré 150 000 emplois en 5 ans et cela devrait continuer à augmenter [29].

Un exemple concret d'utilisation est l'univers d'Hollywood, semblant avoir adopté les drones pour filmer certaines vues aériennes, ceux-ci étant moins chers, moins dangereux et plus maniables que des hélicoptères [1]. On peut également observer les drones de la police qui scrutent la foule lors de manifestations ou de gros rassemblements. Il est aussi fréquent que des usines fassent appel à des drones pour inspecter l'état de certaines conduites difficilement accessibles sans devoir louer des élévateurs bien plus onéreux.

Bref, les drones sont partout et leurs utilisations sont variées. Malheureusement, la plupart d'entre eux demandent la présence d'un pilote expérimenté afin de réaliser un vol en toute sécurité. Non seulement cela demande une main d'œuvre qualifiée, mais aussi cela contraint fortement la zone géographique dans laquelle les drones peuvent voler. En effet, en plein air, une distance maximale d'une trentaine de mètres peut séparer un drone

grand public "low-cost" de son pilote. Pour plus de performance, le prix grimpe rapidement jusqu'à quelques dizaines de milliers d'euros.

Le but de ce mémoire est de rendre autonomes et intelligents des drones low-cost vendus à un large public. De cette manière, un drone basique pourrait facilement être reconfiguré pour effectuer des tâches compliquées sans aucune aide extérieure, qu'elle soit humaine ou matérielle.

1.2 Travaux réalisés antérieurement

Cela fait déjà 4 ans que des étudiants travaillent avec des drones dans le cadre de leur travail de fin d'études à l'EPL. L'objectif à long terme de ces mémoires est de développer un programme permettant à des drones, "low-cost" dans un premier temps, d'explorer un environnement *indoor*, de créer une carte de celui-ci, de trouver et de suivre des cibles tout en profitant de la communication inter-drones.

La première année, les deux étudiants, Nicolas Beghin et Thibault Martin, ont mis en place un système de localisation et de cartographie simultanées (SLAM). Les drones étaient alors capables de voler dans un environnement inconnu tout en créant une carte virtuelle.

Cette carte se construisait au fur et à mesure en indiquant l'emplacement de repères visuels appelés *pucks*. Ces derniers, qui étaient en fait des CDs recouverts d'une peinture rouge disposés sur le sol, étaient détectés par la caméra ventrale du drone qui ajoutait alors leur position à sa carte. Le drone pouvait dès lors se déplacer autour de ce point connu afin de trouver un autre puck. La carte se construisait petit à petit avec le drone se déplaçant de puck en puck, en ayant toujours au moins un puck dans son champ de vision afin de ne pas se perdre. L'usage de cette carte était double puisqu'elle servait aussi bien à donner un aperçu de l'environnement (ici la position des CD rouges) à l'utilisateur mais aussi à permettre au drone de se repérer dans ce lieu préalablement inconnu en comparant la disposition des pucks observés autour de lui avec ceux présents dans la carte virtuelle. Cette comparaison lui permettait d'estimer sa position dans la carte. Cette dernière est illustrée sur la FIGURE 1.1.

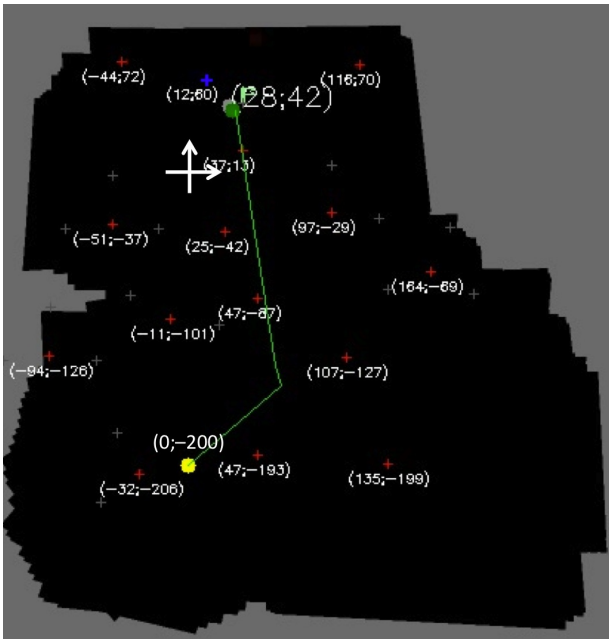


FIGURE 1.1 – Représentation de la carte créée par le drone avec les pucks [12]

En 2013-2014, l'étudiant qui a repris le flambeau, Jean Herman, a décidé d'améliorer le code précédent en y ajoutant un traitement d'images plus évolué [14]. De cette façon, la carte virtuelle ne reposait plus uniquement sur la présence de pucks mais bien sur la présence de n'importe quel objet qui contrastait avec le sol (e.g. une farde à rabat d'une couleur vive sur le sol terne). De cette façon, le drone pouvait se repérer dans une pièce "en désordre" sans que l'on ait à disposer des pucks sur le sol au préalable. Cette amélioration rendait la mission du drone plus flexible. Pour y parvenir J. Herman a implémenté un algorithme de **SIFT**. Une représentation de ce que voit le drone après les différents algorithmes de traitement d'images est disponible à la FIGURE 1.2.

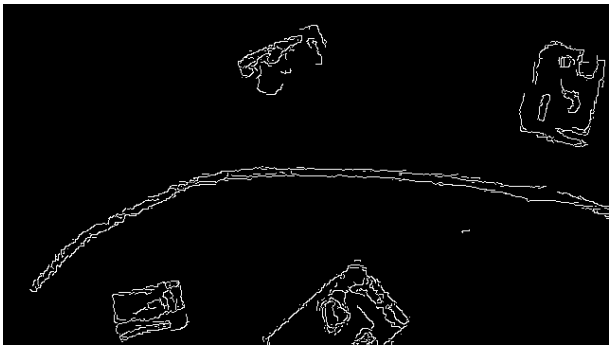


FIGURE 1.2 – Aperçu de la vision du drone après traitement d'images [14]

L'année dernière, les deux mémorants, Brieuc de Radigues et Florent Van Hijfte, se sont fixés comme objectif d'établir une communication entre les drones. La mission principale

était donc de partir à la recherche d'une cible connue dans une pièce tout en dressant la carte des endroits parcourus. Une fois la cible trouvée, le premier drone envoyait un message au deuxième avec les coordonnées de la cible afin que ce dernier vienne le rejoindre. En plus de cette communication entre drones, l'équipe de l'an passé a amélioré les différents algorithmes utilisés afin de rendre les temps de calculs plus courts et le drone plus précis en terme de positionnement.

Un exemple d'une carte créée lors d'une mission qui s'est déroulée parfaitement est disponible à la FIGURE 1.3. On peut y voir la trajectoire du premier drone qui a exploré la zone avant de repérer sa cible ainsi que la trajectoire du deuxième drone qui est plus directe. Ceci montre que le premier drone a bien renseigné le deuxième drone de la position à laquelle il devait se rendre.

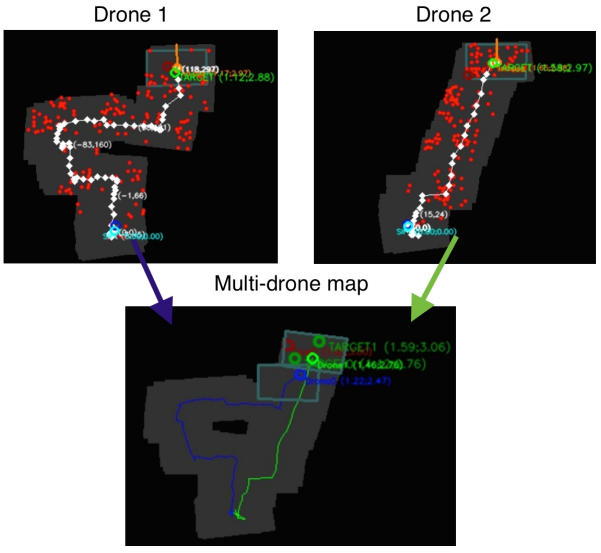


FIGURE 1.3 – Illustration de la carte à la fin de la mission. [9]

1.3 Objectifs

1.3.1 Objectifs communs aux deux groupes de mémorants

Cette année, quelques nouveautés sont venues interrompre l'évolution régulière observée jusqu'à maintenant. Tout d'abord, ce sujet de mémoire a été attribué à quatre étudiants composant deux groupes de deux. Chacun de ces groupes est composé d'un étudiant en mécatronique et d'un étudiant en mathématiques appliquées. Le but étant de combiner les domaines de connaissance de chacun.

Ensuite, pour des raisons qui sont expliquées plus en détails dans le chapitre 4, nous n'avons pas continué à améliorer le code développé lors des années précédentes. En effet,

l'utilisation d'un *framework* plus efficace et plus adapté à nos besoins nous paraissait judicieuse.

Nous avons donc, avec l'aide d'Arnaud Jacques et Alexandre Leclère, les deux étudiants de l'autre groupe, ré-implémenté la mission résultant du travail des années précédentes à l'aide d'un nouveau *framework* connu sous le nom de *Robot Operating System*. Commencer par cette étape avant de réaliser nos objectifs personnels nous a permis de ne pas perdre tout ce qui avait été fait lors des années précédentes. De plus, ce premier objectif nous a permis de mieux comprendre le fonctionnement de ROS et offrait une structure solide à notre code avant de continuer avec notre objectif principal par la suite.

Afin de pouvoir commencer au plus vite à implémenter notre propre application relative au mémoire, le second groupe d'étudiants ainsi que le notre avons décidé de travailler en collaboration afin de ne pas passer trop de temps sur l'implémentation de la mission réalisée lors des années précédentes. Chaque groupe a alors ré-implémenté les parties de l'ancien code dont il allait avoir le plus besoin pour sa partie individuelle.

Notre groupe s'est alors lancé dans la programmation du régulateur, de la planification de trajectoire et de la stratégie (celles-ci servant d'intelligence artificielle au drone). L'autre groupe, quant à lui, s'est penché sur l'implémentation de la reconnaissance d'images et de l'estimation de la position du drone. Une fois la mise en commun faite, la mission de l'année passée était opérationnelle et les deux groupes ont chacun pris une voie différente pour la suite de leur mémoire respectif.

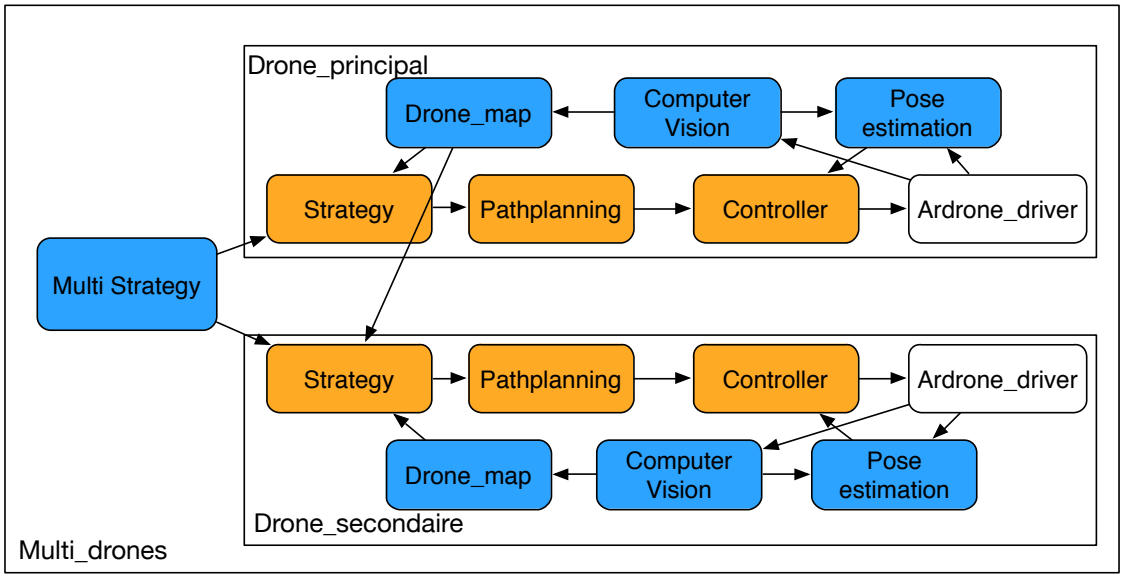


FIGURE 1.4 – Architecture du code pour accomplir la mission de l'année passée

A la FIGURE 1.4 se trouve l'architecture du code de la mission de 2015. On peut y voir deux ensembles distincts possédant un seul module en commun, le module "*multi_strategy*". Ce dernier est comparable à une fonction d'initialisation comptant le nombre de drones connectés et leur attribuant un rôle différent. Dans ce cas-ci, le code du drone qui va explorer son environnement à la recherche de la cible est représenté par la partie supérieure de l'architecture. Celui du drone qui vient le rejoindre est quant-à lui représenté par la partie inférieure. Les modules en orange ont été réalisés par notre groupe et ceux en bleu par Arnaud Jacques et Alexandre Leclère. Le module sans couleur a été repris d'un package qui sera expliqué plus en détail à la section 4.4.

Voici *infra* une brève description de chacun des modules utilisés lors de cette première mission :

- La stratégie est l'intelligence de notre drone, c'est elle qui donne le comportement général de celui-ci en fonction de son rôle et de son état.
- La planification de trajectoire reçoit les informations venant de la stratégie et les interprète en donnant au *controller* les coordonnées auxquelles le drone doit se rendre. Elle est notamment responsable de l'algorithme d'exploration, de recherche de la cible.
- Le *controller* reçoit une position de référence imposée par la planification de trajectoire. Il va tenter de s'y rendre en se régulant sur base des estimations de la position reçue.
- Le *driver* envoie au drone les commandes de vitesses calculées dans la régulation. Il fournit également des informations de position à la partie *Computer Vision* et *pose estimation*.
- La partie *Computer Vision* analyse le flux vidéo de la caméra ventrale ainsi que les angles d'inclinaison du drone afin de fournir une estimation du déplacement et de détecter des objets.
- La *pose estimation* va utiliser les données reçues par le driver et par la *Computer Vision* pour fournir une estimation de la position du drone au *controller*.
- La partie *drone_map* sert à construire la carte sur base des informations reçues par le module *Computer Vision*. C'est également celle-ci qui prévient la stratégie lorsque la cible a été aperçue. On peut également remarquer que lorsque le drone principal détecte la cible, cette fonction envoie également un message au deuxième drone pour lui signaler qu'il peut démarrer.

1.3.2 Objectifs de ce mémoire

Une fois arrivés au même stade que l'année passée, nous avons décidé des nouveaux objectifs à atteindre pour effectuer la partie personnelle de notre mémoire. Nous avons décidé d'améliorer la technique d'exploration du drone afin de faire en sorte qu'il trouve de manière certaine la cible dans une pièce inexplorée. Nous avons également décidé d'im-

plémenter le suivi d'une cible mobile afin que le drone soit capable de repérer une cible préalablement enregistrée, décrite physiquement et de la suivre en se maintenant juste au-dessus d'elle. Concrètement, cette application peut servir dans de nombreux domaines tels que l'espionnage, le cinéma ou encore le sport. De fait, de plus en plus de drones disponibles sur le marché offrent la possibilité de suivre leur propriétaire lors de la pratique de sports extrêmes afin de filmer le sportif en mouvement [4] [31]. Finalement, comme le plus gros inconvénient des drones est souvent leur autonomie, nous avons implémenté un système de remplacement des drones en activité, afin qu'ils puissent être remplacés lorsque leur niveau de charge descend en-dessous d'un certain point. De cette manière, un drone "fatigué" appelle un drone chargé qui était resté à la base. Lorsque ce dernier drone arrive à proximité du premier drone, le drone "fatigué" retourne à la base où il pourra être rechargé tandis que le nouveau drone reprend la mission.

Afin de mettre en œuvre cette nouvelle mission, nous nous sommes basés sur notre version du code de la mission de 2015. Cela nous a demandé de modifier la plupart des modules afin de permettre au drone d'accomplir les nouveaux objectifs. Nous avons également adapté notre code en gardant en tête les modifications de l'autre groupe sur ses propres modules afin que les deux mémoires restent compatibles. De cette façon, il est aujourd'hui possible de faire tourner les deux codes en parallèle sur un seul drone et ainsi de profiter des plus-values des deux mémoires.

Les modules dont l'adaptation était nécessaire pour notre nouvelle mission sont les modules de la planification de trajectoire et celui de la stratégie. Le module qui a du être modifié pour rester compatible avec l'autre groupe est celui qui s'occupe de la régulation. Ce dernier était trop impulsif que pour permettre aux algorithmes de traitement d'images d'estimer correctement la position du drone. Nous avons donc été dans l'obligation d'adapter les gains de notre régulateur et d'y ajouter des éléments de saturation pour ralentir la dynamique et diminuer la nervosité du drone.

Chapitre 2

Parrot AR.Drone

Le drone utilisé dans le cadre de notre mémoire est un AR.Drone 2.0 commercialisé par l'entreprise Parrot. Il s'agit d'un quadricoptère orienté loisirs que l'on peut trouver dans la plupart des magasins multimédia pour un prix avoisinant les 300 euros. A priori construit uniquement pour servir de jouet à un large public, l'AR.Drone 2.0 a très vite suscité la curiosité du monde scientifique. Ce dernier y voyait un hardware relativement peu cher qui était idéal pour des activités de recherche et développement dans le domaine du drone et du traitement d'images. De plus, l'AR.Drone 2.0 est facilement contrôlable via des implémentations "non-officielles" grâce au *Software Development Kit* proposé par Parrot (cfr. section 4.2). Quelques unes des utilisations scientifiques de ces drones sont reprises dans la section 3.

Comme dit *supra*, l'AR.Drone 2.0 est un quadricoptère. Sa structure mécanique se compose de 4 rotors accrochés aux extrémités d'une croix en plastique (un peu trop fragile) à laquelle sont également attachées l'électronique embarquée et la batterie. Chaque paire de moteurs diamétralement opposés tournent dans le même sens. Une paire tourne dans le sens horlogique, l'autre dans le sens antihorlogique. Un schéma est présenté à la FIGURE 2.1.

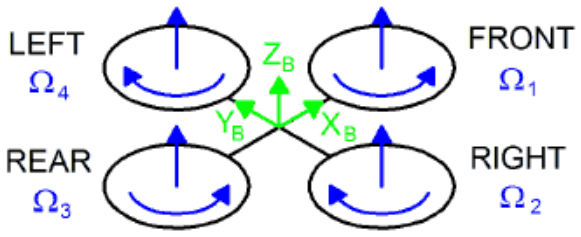


FIGURE 2.1 – Schéma du sens de rotation des hélices de l'AR.Drone [10]

Lorsque les moteurs de gauche et de droite tournent de manière opposée, cela se traduit par un angle de roulis. Pour obtenir un angle de tangage, il suffit de faire aller les mo-

teurs avant et arrière à des vitesses différentes. Une combinaison de roulis et de tangage permet au drone d'avancer ou de reculer dans la direction de sa caméra frontale. Enfin, pour pivoter vers la gauche ou vers la droite, il faut faire tourner de manière différente une paire de moteurs diamétralement opposés l'un par rapport à l'autre.

Bien que dans notre cas le drone vole toujours en intérieur et principalement dans la DroneZone conçue à cet effet dans le bâtiment du pôle Génie Civil, il existe également une configuration matérielle pour voler à l'extérieur.
Même si la carène extérieure protège mieux les hélices de certains obstacles et de certains doigts, celle-ci est plus lourde et plus encline aux perturbations liées au vent. C'est pourquoi Parrot vend également ses drones avec une carène plus légère et ergonomique comme on peut le voir à la FIGURE 2.2

FIGURE 2.2 – Différentes carènes pour vol intérieur (gauche) et extérieur (droite) [10]

L'AR.Drone 2.0 est équipé de 4 moteurs brushless à courant triphasé. Ces derniers sont contrôlés à l'aide d'un micro-contrôleur qui détecte automatiquement les moteurs branchés et ajuste la régulation. Les moteurs sont dotés de capteurs indiquant s'ils sont en mouvement ou pas. Le micro-contrôleur peut alors basculer le drone en état d'urgence lorsqu'une des hélices est bloquée afin d'éviter des dégâts matériels ou des blessures.

De base, le drone est livré avec une batterie de 1000 mAh. Cette dernière ne durant pas longtemps en vol [1], nous utilisons plus régulièrement des batteries de 1500 mAh, également fabriquées par Parrot. La tension de la batterie oscille de 12.5 V lorsqu'elle est pleine à 9 V lorsqu'elle est vide. Lorsque le drone en utilisation observe que la batterie est basse, celui-ci prévient l'utilisateur et atterrit. Si elle est vraiment trop basse, le drone s'éteint afin d'éviter tout comportement inattendu.

L'AR.Drone 2.0 dispose d'un panel de capteurs de mouvements. On y retrouve une unité de mesure inertielle qui analyse le déplacement dans les 6 degrés de liberté. C'est elle qui fournit au software les angles de roulement, de tangage et de lacet. C'est sur ces valeurs que se basent la régulation automatique du drone ainsi que le contrôle du déplacement. Un capteur à ultrasons se situe sur le coté ventral du drone. Il permet de calculer la

1. Les batteries de 1000 mAh durent moins de 10 minutes en vol.

hauteur par rapport au sol à laquelle vole le drone et d'assurer une bonne stabilisation en altitude ainsi qu'une bonne gestion de la vitesse verticale. On remarque également sur le ventre du drone une petite caméra QVGA (320x240). Cette dernière est chargée de filmer le déplacement du drone par rapport au sol. Ceci permet un meilleur contrôle en position par rapport à l'utilisation de capteurs inertiels, seuls. Depuis la deuxième version de l'AR.Drone, Parrot a rajouté un magnétomètre 3 axes qui permet un mode de contrôle absolu. Cette mise-à-jour fait également gagner un capteur de pression afin de pouvoir connaître l'altitude du drone quelle que soit sa hauteur.

Destinée à la partie plus ludique, l'AR.Drone 2.0 dispose d'une caméra frontale. Celle-ci propose une résolution 720p (1280x720) nettement meilleure que la caméra ventrale. Cette caméra donne la possibilité à l'utilisateur d'enregistrer des vidéos sur une clé USB branchée au drone ou bien d'effectuer un streaming directement sur un smartphone ou une tablette. Quelques images appelées *Tags visuels* ont également été préenregistrées afin d'avoir des fonctions dans différents jeux pour smartphones proposés par Parrot. Par exemple, lorsqu'un drone détecte ces *tags*, il considère qu'il s'agit d'un drone ennemi et la réalité augmentée permet dès lors l'envoi de missiles virtuels.

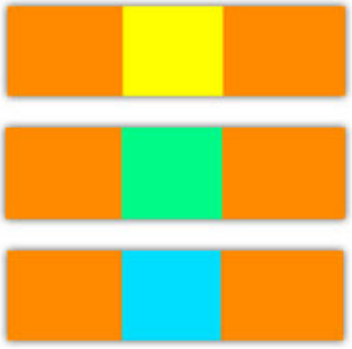


FIGURE 2.3 – Exemples de tags "préenregistrés" par le drone pour la détection lors de jeux [10]

L'AR.Drone 2.0 peut être contrôlé depuis n'importe quel appareil pouvant faire office de client wifi. Le drone crée un réseau wifi avec un ESSID habituellement appelé $adrone2_xxx$. Il s'alloue également automatiquement une adresse IP qui est généralement *192.168.1.1*.

Afin de communiquer avec le drone, l'utilisateur doit connecter son appareil au réseau ESSID du drone. L'appareil demande alors une adresse IP au serveur DHCP du drone. Ce dernier lui renvoie une adresse qui correspond à sa propre adresse IP incrémentée d'un chiffre entre 1 et 4. Une fois ces opérations effectuées, l'utilisateur peut commencer à envoyer des requêtes à l'adresse IP du drone et à ses ports de services.

Le port qui s'occupe du contrôle du drone communique avec l'utilisateur à une fréquence de 30 Hz. Il s'agit du port 5556. Le port 5554, quant à lui, est chargé de transmettre

les informations de vol telles que la vitesse, la position, l'altitude, etc. à l'utilisateur. Ce port fonctionne à une vitesse de 15 Hz en mode démo ou de 200 Hz en mode debug. Le port 5555 sert à envoyer le flux video de l'une des deux caméras embarquées.

Chapitre 3

Etat de l'art

3.1 L'utilisation des drones aujourd'hui

Comme annoncé brièvement dans l'introduction de ce rapport, les drones deviennent de plus en plus importants dans la société d'aujourd'hui. Ils permettent d'améliorer le quotidien de l'homme ou de réaliser des choses qui étaient impensables il y a quelques années.

En Amérique, par exemple, il est maintenant coutume que les agriculteurs utilisent des drones pour voir à quel stade en sont leurs récoltes sans pour autant les piétiner ou les abimer. Les drones permettent également d'aller observer si des fruits sont disponibles à la cime des arbres et de juger de leur état.

Drone utilisé en agriculture. [6]

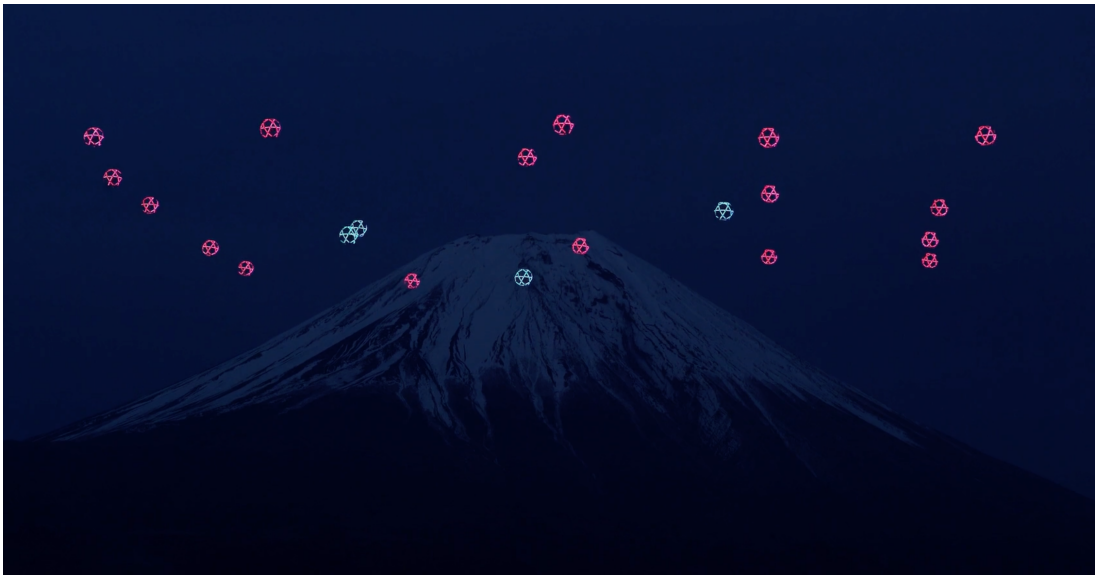
Toujours dans cette optique d'inspection, les drones sont fortement utilisés pour vérifier l'état de certaines lignes à hautes tensions. En à peine une dizaine de minutes, un drone peut avoir parcouru quelques centaines de mètres de câbles. Il peut dès lors filmer les éventuels dommages que des pylônes ou des câbles pourraient avoir subis. Le principal avantage de cette nouvelle méthode est la diminution nette des risques encourus par le technicien ainsi que le coût lié à la location de l'hélicoptère [37].

Les drones peuvent également jouer des rôles cruciaux lors de gros incendies et feux de forêts. En effet, il arrive de plus en plus souvent que les pompiers utilisent des drones munis d'une caméra infrarouge pour déterminer l'emplacement des différents foyers de flammes à travers la fumée ou les murs d'une maison. Les informations recueillies par le

Drone Aibotix inspectant une ligne haute tension.[8]

drone permettent donc aux pompiers de savoir où et quand agir pour être efficaces.

Dans le domaine du spectacle, certaines sociétés sont capables de recréer une sorte de feu d'artifice à l'aide de drones équipés de lumières. Il s'agit d'une centaine de drones préprogrammés se déplaçant les uns par rapport aux autres dans le but de créer un jeu de lumières qui rappelle l'ambiance d'un feu d'artifice. Utiliser des drones permet d'éviter les problèmes d'incendie pouvant parfois se produire avec la poudre. De plus, les drones permettent de répartir le coup d'achat des effets lumineux sur plusieurs représentations.



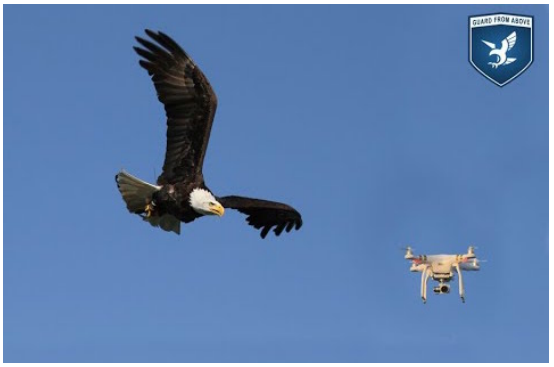
Spectacle de drones.[25]

Certaines société de livraison commencent également à se tourner vers les drones pour livrer quelques-uns de leurs colis. On peut, par exemple, voir depuis quelques temps des vidéos sur internet d'un drone qui livre un colis Amazon à peine quelques heures après sa commande sur internet. A ce jour, cette vidéo semble un petit peu illusoire. Pourtant,

plusieurs autres marques ont l'air de suivre le pas. C'est le cas de Domino's Pizza qui a lancé un nouveau spot publicitaire où l'on peut voir 2 pizzas livrées à domicile en moins de 10 minutes à l'aide d'un drone.

Domino's pizza teste la livraison par drone.[16]

Comme le montrent ces quelques exemples, les drones ont un potentiel énorme qui ne demande qu'à être développé. Redoutant cela, la police hollandaise a vu naître une nouvelle section en son sein appelée "Guard from above". Cette nouvelle filière de la police a pour mission d'entrainer des aigles à intercepter des drones en plein vol. Cette initiative a pour but d'éviter l'utilisation illégale de drones près de lieux interdits (casernes militaires, aéroports ...) ou encore lors de grosses manifestations.

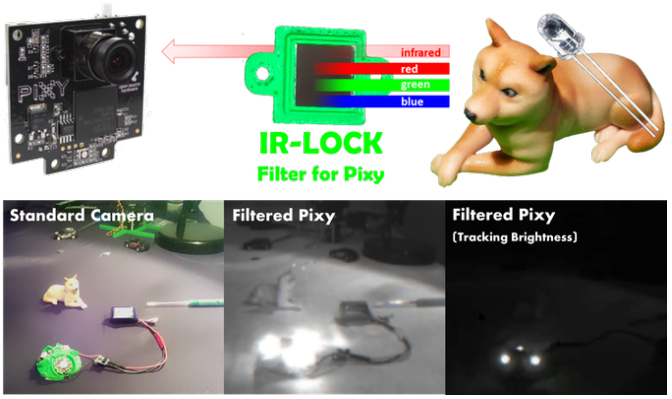


Aigle dressé en train de saisir un drone.[13]

3.2 Les drones et le suivi de cibles mobiles

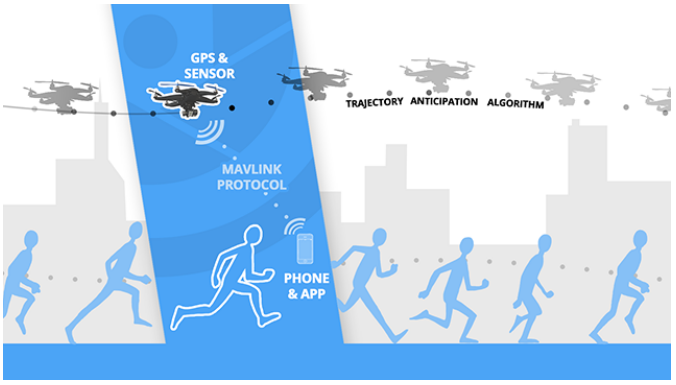
Un des mots clés de ce mémoire est le suivi de cibles mobiles. Nous allons donc passer en revue les différentes façons de suivre un objet qui existent déjà ailleurs dans le monde. Après quelques recherches, il est apparu qu'il existait deux grandes méthodes pour permettre à un drone ou même un robot quelconque de suivre une cible. La première méthode, qui est la plus utilisée dans le commerce de par sa fiabilité et sa facilité à mettre en œuvre, repose sur l'ajout d'un module extérieur au drone que l'on vient fixer sur la cible. Ce module est généralement actif et émet donc un signal qui sera perçu par le drone

suiveur. C'est en travaillant sur ce principe que la start-up IR-Lock a vu le jour.
Cette jeune société commercialise des ampoules spéciales ainsi qu'une caméra à infrarouges afin de déterminer l'endroit où se trouve la cible. Il suffit donc de mettre un petit dispositif reprenant une ampoule à infrarouges et une batterie sur l'objet que l'on veut suivre et d'allumer le drone pour que sa caméra repère la source d'infrarouges.



Caméra et ampoule infrarouges.[17]

Toujours dans cette première méthode de suivi de cible, la société Hexo+ commercialise des drones capables de suivre leur propriétaire et de les filmer selon différents angles facilement programmables via un smartphone. Le drone s'exécute alors et enregistre de magnifiques vidéos reprenant le plus souvent les acrobaties sportives de son propriétaire. Afin de savoir où se trouve sa cible et à quelle vitesse elle se déplace, le drone d'Hexo+ est en permanence connecté au smartphone de l'utilisateur qui lui transmet ses coordonnées GPS.

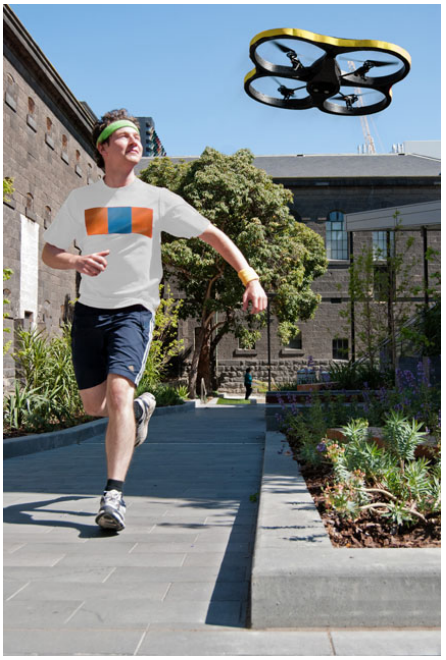


Drone Hexoplus qui suit un joggeur[44]

Dans les deux exemples cités ci-dessus, que ce soit une lampe à infrarouges ou un smartphone doté d'une puce GPS, le drone a toujours besoin d'une source extérieure pour pouvoir suivre sa cible. Ce qui ne rend pas le drone totalement autonome. Dans le but de combler cette dépendance extérieure, certaines personnes ont décidé de se baser

uniquement sur le traitement du flux vidéo pour suivre une cible. Nous verrons dans la suite de cette section des méthodes plus et moins complexes qui permettent à un drone de suivre un objet ou une personne sans que ceux-ci n'aient besoin d'un quelconque émetteur.

Il est pertinent de parler de la façon dont les créateurs de Joggobot ont implémenté leur drone pour le rendre capable de suivre une personne. Joggobot est un AR.Drone 2.0 de chez Parrot identique à ceux que nous utilisons pour notre mémoire. Il est destiné à motiver les sportifs en herbe à courir grâce à sa présence. Son fonctionnement est simple, le drone se positionne en face du coureur et pointe sa caméra sur lui. Lorsque le coureur démarre, le drone recule pour toujours rester à la même distance de celui-ci. Afin de localiser sa cible et de pouvoir reculer à la même allure qu'elle, le joggeur est obligé de mettre un tshirt comportant les tags visuels que nous avions décrits à la FIGURE 2.3 et dont la détection est implémentée par Parrot. Cet algorithme n'est donc pas très développé et ses résultats ne semblent pas exceptionnels, le drone ne pouvant précéder le joggeur que si celui-ci court en ligne droite. Ce concept semble néanmoins beaucoup faire parler de lui sur la toile.



Joggeur avec un tshirt taggué [15]

Une autre méthode, un peu plus complexe, a été développée à l'Université de Salford à Manchester dans le centre de recherches en robotique avancée. Cette méthode consiste à repérer un carré d'une couleur vive, le rouge dans ce cas-ci, et à le suivre. Il s'agit donc d'une méthode ressemblant à la précédente sauf qu'ici, n'importe quelle couleur peut être utilisée. On ne doit pas se servir d'un tag visuel pré-implémenté par Parrot. Afin d'identifier ce carré de couleur et de connaître sa position, l'Université de Salford utilise une fonction d'OpenCV. Cette fonction permet de détecter n'importe quelle forme d'une couleur prédéfinie que l'on peut facilement modifier. Elle y indique sa position en 2 dimensions dans l'image. Sur base de cette position et de la taille du carré rouge, le robot peut alors estimer la distance qui le sépare de sa cible et s'en rapprocher ou s'en éloigner.

Enfin, une dernière méthode bien plus complexe a retenu notre attention. Des chercheurs de Telecom ParisTech ont mis sur pied le suivi autonome d'une personne à l'aide d'un AR.Drone 2.0. Pour ce faire, ils ont implémenté un algorithme d'histogramme de gradient (HOG) permettant de détecter n'importe quel piéton et de le suivre. Cet algo-

rithme se base sur le fait que notre structure corporelle générale est la même quels que soient notre genre et notre origine ethnique. En effet, la plupart des êtres humains ont la même morphologie générale (une tête, un torse, deux jambes et deux bras). C'est ce que cet algorithme essaie de détecter lors de son exécution. La technique consiste en le calcul des histogrammes locaux de l'orientation du gradient sur une grille dense, c'est-à-dire sur des zones régulièrement réparties sur l'image. Elle possède des points communs avec SIFT mais en diffère notamment par l'utilisation d'une grille dense. L'idée principale est que l'apparence et la forme locale d'un objet dans une image peuvent être décrites par la direction des contours. Il suffit de décomposer l'image en petites cellules pour y analyser les orientations des contours pour les pixels à l'intérieur de la cellule. La combinaison des histogrammes forme alors le descripteur HOG [43]. Cette méthode comporte de nombreux avantages. Elle permet, entre autres, de détecter une forme générale plutôt qu'un élément précis comme c'était par exemple le cas pour le Joggobot. Ainsi, si l'on recherche une voiture abandonnée dans un désert, il n'est pas nécessaire de connaître le modèle du véhicule ni sa couleur, une simple information sur sa forme suffit pour la détecter.

Images et leurs descripteurs HOG [3]

3.3 L'utilisation de ROS pour les drones

ROS est le framework qui nous a permis de programmer notre application. Pour plus d'informations sur ce dont il s'agit exactement et la manière dont il fonctionne, le lecteur est invité à consulter le chapitre suivant. Nous allons ici présenter quelques exemples de drones à la pointe de la technologie utilisés soit dans le secteur commercial, soit pour la recherche, utilisant ce framework.

3.3.1 Lily

"*Throw your Lily in the air like you just don't care*". Tel est l'un des slogans de la compagnie *Lily robotics*, qui a développé ce drone [31]. Celui-ci, qui sera disponible à l'achat durant l'été 2016, et déjà pré-commandable, aurait également pu être décrit dans la section précédente. En effet, il s'agit d'un drone capable de suivre une personne de manière autonome afin de faire des films et des photos pendant le suivi. Ce suivi est effectué grâce à un "dispositif de pistage", que l'utilisateur devra porter à tout moment pour que Lily puisse le suivre. Lily utilise la combinaison entre les données GPS du dispositif de pistage et les informations transmises par ses caméras, lui permettant ainsi de suivre l'utilisateur.

Figure 3.1 Lily et son dispositif de pistage

De nombreux drones e ectuant ce genre de missions existent déjà (voir section précédente), mais aucun ne semble avoir des performances et une adaptabilité à l'environnement aussi bonnes que Lily. Un atout majeur de Lily est sa facilité d'utilisation. Il su t de l'allumer puis de le lancer pour que le drone vous suive. ROS est grandement utilisé a n de suivre son propriétaire en utilisant l'ensemble des informations à sa disposition [5]. Il a été utilisé en particulier pour la transmission des messages entre le drone et le dispositif de pistage. Grâce à la structure de ROS, il devenait facile de faire pro ter de toutes les informations au drone, le dispositif et celui-ci semblant agir comme une même entité grâce à l'architecture en n÷uds causée par ROS.

ROS a également été sollicité dans la phase de développement et de tests du robot. Les programmeurs ont utilisé l'outil de visualisation 3D "Rviz" de ROS pour simuler les mouvements de Lily, pour e ectuer des tests en temps réel et pouvoir rejouer ces tests a n d'e ectuer le débogage.

Cet exemple témoigne de l'utilité de ROS dans le monde de l'industrie. En e et, les drones commerciaux dernier cri qui e ectuent du suivi de cible mobile l'utilisent. Ci-dessous sont présentés quelques exemples de l'utilisation de ROS dans la recherche.

3.3.2 Bird MURI

Le but du projet Bird Multi University Research Initiative (MURI) est de développer des technologies capables de permettre des vols rapides et sûrs pour des drones dans des environnements "encombrés" (comme une forêt, une foule ...). Ils programment des AR.Drones 2.0 avec ROS et parviennent grâce à un algorithme de machine learning à faire en sorte qu'un drone puisse parcourir de longues distances dans les bois. (Voir [26] pour une vidéo des performances du drone.)

Dans cette application, le drone apprend les mouvements qu'il doit e ectuer en fonction du contexte grâce à un algorithme d'imitation [35]. Celui-ci se base sur une série de vols e ectués par un expert a n d'appréhender les actions que lui même doit e ectuer. Il

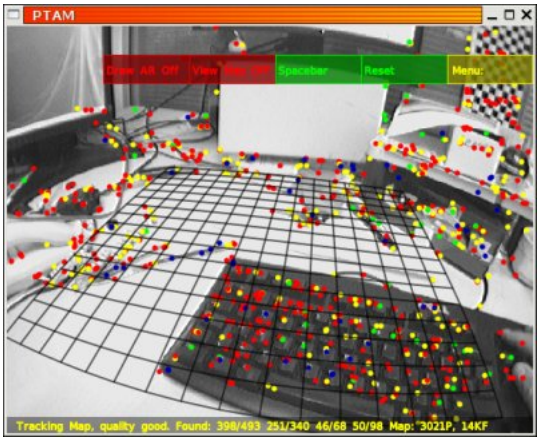


FIGURE 3.3 – PTAM en fonctionnement avec les *landmarks* affichés

FIGURE 3.4 – Comparaison des estimations de la trajectoire d'un drone entre ses données GPS et PTAM avec intégration des capteurs.

3.3.4 PTAM, la TUM et son contrôle en position

A la TUM, ou *Technische Univeristät München*, une équipe de chercheurs est allée encore plus loin que celle de l'ETHZASL. En utilisant ROS, ils ont créé leur propre package utilisant PTAM afin qu'un ARdrone 2.0 puisse se localiser dans son environnement. Afin de se positionner de la manière la plus précise possible, le drone utilise l'estimation de PTAM ainsi que celle des ses capteurs, fusionnant les deux à l'aide d'un filtre de Kalman. L'avantage par rapport au cas précédent est que, ici, le programme fait en sorte que le drone se stabilise dans l'espace à une position indiquée par l'utilisateur à l'aide d'une interface graphique. Cette interface envoie alors au drone une série de coordonnées auxquelles il doit se rendre. Le fonctionnement de ce programme est expliqué dans [18], [20], [21] et [19].

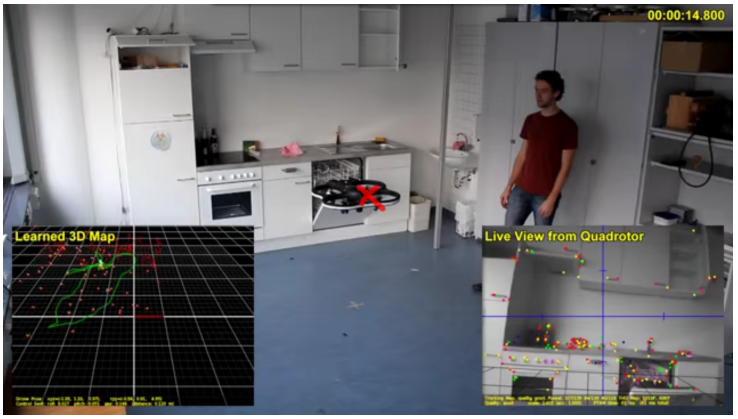


FIGURE 3.5 – Un ARdrone 2.0 avec le programme de la TUM restant en place (croix rouge) malgré les perturbations. On remarque les *keyframes* sur la map construite en bas à gauche et les *landmarks* sur l'image de la caméra en bas à droite.

Ce code, qui est open source, était le premier point de départ que nous avons envisagé pour l'implémentation de notre application. Cependant, comme pour le précédent, nous avons du abandonner l'idée pour des raisons que nous expliquerons dans le chapitre suivant. Il a cependant été une grande source d'inspiration pour notre propre code et grâce à lui, nous avions un bon exemple d'utilisation de ROS pour créer notre application. De plus, la TUM offre un cours gratuit que nous avons suivi en ligne et qui permet d'en apprendre plus sur les drones, l'estimation de leur position et leur commande [22].

3.4 Les algorithmes d'exploration

Une partie de notre travail consistait en la création d'une stratégie d'exploration pour notre drone. C'est pourquoi il nous semble important de résumer ce qui se fait déjà en robotique de nos jours à ce niveau-là.

3.4.1 Un exemple de drone utilisant un algorithme d'exploration

Depuis des décennies de nombreux algorithmes d'exploration autonome existent et ont été implémentés sur des robots. Cependant, il est difficile de trouver des exemples d'implémentation sur des drones. Le meilleur que nous ayons pu trouver a été implémenté à l'université de Pennsylvanie. Un drone avec une grande quantité de capteurs embarqués est capable d'explorer un bâtiment avec plusieurs pièces, couloirs et escaliers de façon autonome et de créer une carte de celui-ci [39]. Cependant, la stratégie d'exploration n'est pas expliquée, l'accent étant mis sur la création de la carte. La figure 3.6 montre que leur stratégie fonctionne pour l'exploration *indoor*. Afin de se repérer dans son environnement, le drone utilise des sonars pointant dans toutes les directions, ce qui simplifie grandement la construction de la carte.

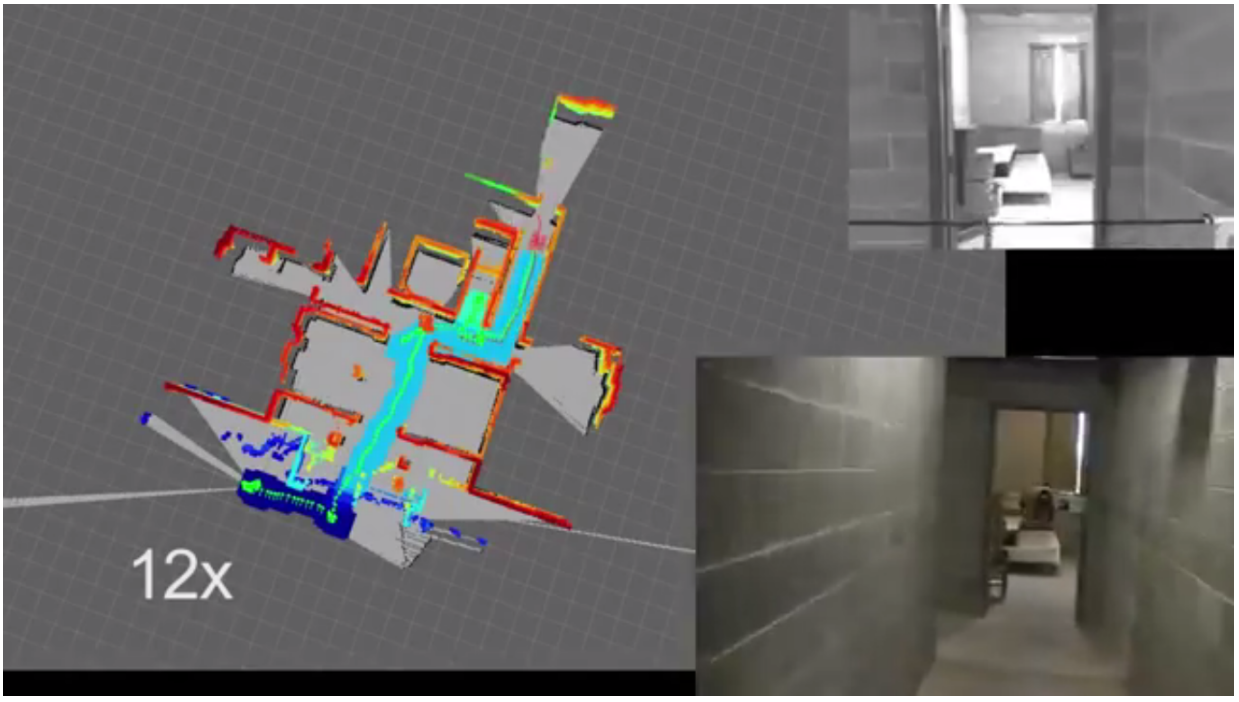


FIGURE 3.6 – Capture d'écran d'une vidéo d'un drone explorant un environnement *indoor* de manière autonome [38]. On voit la carte créée ainsi que la vue de deux de ses caméras.

3.4.2 Algorithmes d'exploration à un seul robot

Au niveau des articles décrivant les stratégies optimales pour l'exploration avec des robots, on trouve par exemple [11] et [40], qui ont été les principaux à inspirer notre propre implémentation. Pour le premier, il s'agit de la description d'un algorithme optimisant le temps que met un robot avec une vue périphérique pour explorer une pièce. Un exemple d'exploration avec cet algorithme est présenté à la FIGURE 3.7. Le problème résolu ressemble fortement au notre, puisque le drone voit un rectangle en dessous de lui.

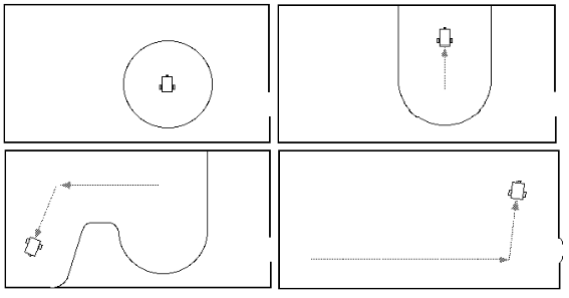


FIGURE 3.7 – Exemple d'exploration d'une pièce par un robot autonome. La frontière entre la zone explorée et non explorée est représentée par le trait fin.

Pour le second, il s'agit d'un article modélisant l'optimisation de l'exploration d'une pièce en découpant celle-ci selon un quadrillage abstrait. Notre implémentation expliquée à la section 6.5 s'inspire de ce découpage. Un exemple de découpage de l'environnement est donné à la FIGURE 3.8.

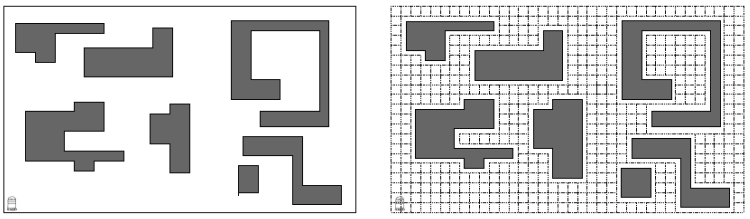


FIGURE 3.8 – Exemple de découpe en grille d'un environnement *indoor* avec des obstacles.

3.4.3 Algorithmes d'exploration à plusieurs robots

D'autres articles très intéressants (*cfr.*[7]) travaillent quant à eux sur l'exploration avec plusieurs robots et présentent des algorithmes qui pourraient être utilisés dans les années futures si l'exploration avec plusieurs drones devenait l'un des objectifs de ce mémoire. Un exemple d'exploration est fourni à la FIGURE 6.3. Les résultats de ces algorithmes permettent d'explorer beaucoup plus rapidement un même environnement. Le temps d'exploration n'est cependant pas inversement proportionnel au nombre de drones. Dans ce rapport, nous ne nous étendons pas plus sur ce type d'algorithmes car cela sort du cadre de ce mémoire.

FIGURE 3.9 – Exploration d'une pièce avec un algorithme utilisant 3 robots simultanément.

Chapitre 4

ROS : Le choix de notre système d'exploitation

4.1 Introduction

Lorsque nous avons commencé notre mémoire, nous avons dû nous poser la question du choix du *framework* (interface de programmation et système d'exploitation) que nous allions utiliser pour le fonctionnement des drones. En effet, le groupe de l'année précédente avait donné comme piste d'amélioration de changer celui-ci au profit de ROS. L'interface de programmation utilisé par ce groupe était le SDK, ou *Software Development Kit*, fourni par Parrot. L'utilisation de ce SDK entraînait toutes sortes de limitations pour la suite du projet, que nous allons évoquer. Nous allons commencer par expliquer ce qu'est ce SDK (4.2), nous expliquerons ensuite ce qu'est ROS (4.3) et ce qu'il permet d'améliorer. Nous terminerons ensuite par expliquer comment il peut être utilisé avec l'*ARdrone 2.0* à l'aide du package *ardrone_autonomy* (4.4).

4.2 Le SDK de Parrot

Le SDK est un programme qui permet de faire l'interface entre l'AR.Drone 2.0 et un autre programme écrit par un humain. Même si nous n'avons pas utilisé directement ce SDK, nous avons utilisé un nœud ROS qui encapsule celui-ci (nous y reviendrons plus tard). Il est donc intéressant de décrire le fonctionnement général de ce SDK.

La première chose intéressante à savoir est que le SDK est fourni en **C** avec son guide du développeur, qui donne toutes sortes d'informations sur les AR.Drones ainsi qu'un descriptif complet de ce que fait le SDK ainsi que des fonctions et informations qu'il permet d'utiliser [10]. Dans ce guide, de nombreuses fonctionnalités sont expliquées. Celles qui nous intéressent sont les suivantes :

- Un *thread* de management des commandes envoyées au drone, qui collecte les commandes de tous les autres *threads*, les transforme en commandes compréhensibles

pour celui-ci et les lui envoie,
- Un *thread* de management de *navdata*, qui recueille les données de navigation du drone, les décode et fournit au programme des données de navigation utilisables grâce à une fonction "*call-back*",
- Un *thread* de management des données vidéos, qui reçoit les images des caméras telles qu'envoyées par les drones (sous forme segmentées, voir [10] pour plus d'informations) et fournit à l'utilisateur des images prêtes à être utilisées via une fonction "*call-back*".

Tous ces *threads* sont créés grâce à une fonction *main* déjà fournie dans un fichier séparé, dans lequel l'utilisateur peut travailler. Cette fonction main s'occupe également d'établir la connexion entre le drone et l'ordinateur.

Comme on peut le constater, ce SDK nous permet de faire beaucoup de choses, puisqu'il représente un bon interface entre l'ordinateur et le drone. Cependant, utiliser ROS présente bien des avantages, décrits dans la section suivante.

4.3 ROS

4.3.1 Description de ROS

ROS, ou *Robot Operating System*, est un framework permettant de créer des programmes pour les robots [34]. Celui-ci agit à la fois comme une interface de programmation et comme un système d'exploitation. C'est-à-dire qu'il va gérer le fonctionnement du programme créé par l'utilisateur et qu'il fournit toute une série de librairies et d'outils afin de faciliter la tâche de création de ce programme.

Un des avantages principaux de ROS est qu'il privilégie un développement collaboratif. En effet, celui-ci est open source. Des milliers d'utilisateurs à travers le monde [32] l'utilisent et contribuent à son amélioration. Un très grand nombre de *packages* résolvant divers problèmes liés aux robots sont disponibles gratuitement, plus de 22.000 pages de wiki existent déjà et une communauté très active répond aux diverses questions sur le forum de ROS (plus de 13.000 questions posées avec 70% de réponses)[33]. Tout cela nous permet d'éviter de résoudre des problèmes ayant déjà été résolus par d'autres, que ce soit en réutilisant leurs *packages*, en consultant des forums ou en y posant des questions appropriées.

Un autre avantage de ce *framework* est son architecture. Celle-ci sera décrite dans la sous-section suivante, mais nous pouvons déjà mentionner le fait qu'elle encourage une grande modularité grâce à ses différents nœuds, qu'elle correspond bien à l'abstraction dont nous avons besoin et que la gestion des différents *threads* du programme est presque automatique. En effet, il suffit de "laisser la main" à ROS pour qu'il décide de manière autonome quelle partie du programme devra être exécutée.

4.3.2 Fonctionnement de ROS

La principale caractéristique du fonctionnement de ROS est son architecture en nœuds. Chacun de ceux-ci est en fait un processus, dont ROS assure la facilité de création, destruction ainsi que la facilité de communication avec les autres. Afin de pouvoir utiliser les fonctions et structures de données de ROS, ces processus doivent être programmés en **C++** ou en Python. Dans le cas de notre application, chaque nœud va représenter une partie concrète de notre programme (un pour le nœud de régulation, un autre pour la planification de trajectoire,...) et chacun peut fonctionner indépendamment des autres, puisqu'il est un processus à part entière, tout en ayant la possibilité de communiquer avec les autres (et même en ayant l'obligation de la faire, afin d'avoir une application qui fonctionne !).

La communication entre les nœuds peut être faite de deux manières différentes :

- Par message : l'utilisateur peut à partir de n'importe quel nœud accéder à différents topics, qui sont des buffers pouvant contenir ces messages. N'importe quel nœud peut devenir un "*publisher*" ou un "*subscriber*" de n'importe quel topic créé ultérieurement, afin de pouvoir publier dans ce topic ou de lire son contenu respectivement. Chaque topic contient un certain type de message. Lorsqu'un *publisher* publie dedans, il doit donc respecter scrupuleusement l'architecture des messages du topic. La lecture des messages est gérée par ROS. Celle-ci se fait grâce à la création par l'utilisateur d'une fonction par nœud et par topic qui sera appelée lorsqu'un message est publié et qui prendra ce message en argument.
  Ce type de communication est anonyme et asynchrone. C'est-à-dire que lorsqu'un message est lu par un nœud, celui-ci ne sait pas d'où il provient ni quand il a été publié. Cela présente quelques avantages, comme par exemple le fait que plusieurs nœuds peuvent lire les mêmes messages sans qu'ils soient envoyés plusieurs fois, ou encore que les données publiées peuvent être aisément enregistrées. C'est le type de communication que nous avons le plus utilisé dans notre programme.
- Par service : bien que les messages soient un mode de communication approprié pour notre application, il se peut qu'un nœud ait besoin d'avoir une interaction où un autre nœud répondrait à une de ses requêtes. Les services remplissent ce rôle et fonctionnent comme un appel à une procédure présente dans un autre nœud. Pour utiliser un service, un nœud envoie un message de requête à un autre nœud et attend le message de réponse avant de continuer son exécution.
  Ce type de communication est nominatif et synchrone, contrairement au précédent.

Pour ce qui est du fonctionnement pratique de ROS, des ses topics etc., l'utilisateur doit d'abord lancer le cœur de ROS dans un terminal Linux et peut ensuite ajouter à sa guise tous les nœuds qu'il désire, par exemple via un fichier *launch* qui va automatiser ce processus. La création des nœuds, des topics et la publication des différents messages via ceux-ci est alors prise en main par ROS.

ROS offre bien d'autres fonctionnalités. Par exemple, des structures adaptées aux besoins de la robotique en général sont déjà créées, ainsi que les messages standards utilisés dans la plupart des applications. Il existe déjà des structures et messages servant à décrire des concepts géométriques tels que la pose ou d'autres encore pour contenir des vecteurs de vitesse, d'accélération, ou encore des données renvoyées par des capteurs d'altitude ou d'accélération, avec par exemple une place pour le *timestamp*, des structures servant à contenir des durées, etc. Une autre fonctionnalité que nous avons utilisée est l'outil *rqt*, qui nous permet d'afficher en temps réel des graphes de l'état du robot en s'abonnant à des topics. Cet état qui peut être affiché comprend par exemple l'altitude, la position du drone, le nombre de keypoints détectés dans la reconnaissance d'image, etc. Il existe encore beaucoup d'autres outils sur ROS que nous n'avons pas utilisés, principalement parce qu'ils sont surtout utiles pour des robots avec des parties humanoïdes (*rviz*, Robot Geometry Library, etc.).

4.3.3 Librairies externes

En plus de ses multiples librairies et packages open sources, ROS permet aussi d'intégrer facilement plusieurs librairies externes. Celles qui ont été utilisées dans notre programme sont OpenCV et Point Cloud Library.

OpenCV, dont le nom fait référence à "*Open Computer Vision*", est une libraire open source implémentant toutes sortes de fonctions d'analyse d'images et de *machine learning*. Ces différentes fonctions vont de la recherche d'objets dans une image jusqu'à la production d'une image de plus haute qualité à partir de plusieurs photos d'un même sujet. Dans le cadre de notre mémoire, OpenCV a été utilisé par l'autre équipe d'étudiants mémorants afin de faire le suivi de la cible dans une image. Pour plus d'informations sur OpenCV, le lecteur est invité à consulter [27].

PCL, ou *Point Cloud Library* est une bibliothèque qui se concentre sur la manipulation d'images et de cartes en 3D [30]. Celle-ci fournit des algorithmes permettant de travailler avec ces cartes, y compris le filtrage, le repérage de points d'intérêt, l'enregistrement, la visualisation, etc. Elle est également facilement intégrable dans tout programme ROS.

4.4 ardrone_autonomy

Le package ROS ardrone_autonomy est un driver pour les AR.Drones 1.0 et 2.0 développé à la Simon Fraser University, au Canada [28]. Celui-ci est open source et utilise le SDK de Parrot pour fournir un nœud ROS donnant une interface de programmation permettant de contrôler les drones et de collecter toutes sortes d'informations.

Les informations données par ce nœud et publiées dans divers topics sont les suivantes : le pourcentage de batterie restant, l'état du drone (en vol, en train d'atterrir...) la valeur

des angles du drone (roulis, tangage et lacet), son altitude, la valeur du champ magnétique dans toutes les directions, la vitesse et l'accélération linéaire dans chaque direction ainsi qu'un *timestamp*.
Les messages contenant les vitesses et accélérations peuvent provenir de plusieurs topics différents. L'un contient l'estimation des mouvements calculée par les capteurs inertiels et l'autre celle par l'odométrie, ou l'estimation du mouvement grâce au flux vidéo [1]. Dans ce dernier topic, on trouve également l'estimation de la position grâce à cette même odométrie, qui nous a été très utile pour développer notre application. Il est intéressant d'ajouter que les données de vitesse et d'accélération sont fournies avec leurs matrices de covariances, qui pourraient être utilisées à l'avenir pour l'implémentation de filtre de Kalman.
La dernière donnée publiée est simplement le flux vidéo d'une des caméras, frontale ou ventrale, selon le choix de l'utilisateur.

Au niveau des commandes que l'on peut envoyer au drone via ardrone_autonomy, certaines sont envoyées en publiant des messages dans des topics et d'autres via des services.
Celles qui sont envoyées via des topics sont les suivantes :

- L'utilisateur peut envoyer un message vide dans les topics *ardone/takeoff*, *ardrone/land* et *ardrone/reset* afin que le drone décolle, atterrisse ou change de mode [2] respectivement.
- La publication d'un message dans le topic *cmd_vel* est utilisée afin d'envoyer des commandes de vitesses linéaires au drone selon les axes de son repère propre. Les vitesses selon chaque axe doivent être précisées dans le message. Un mode "hover" existe également, pour lequel le drone reste sur place même lorsqu'il est soumis à des perturbations extérieures.

Pour celles qui sont envoyées via des services, seules deux ont été utilisées :

- *togglecam* et *setcamchannel*, permettant de choisir la caméra dont le flux vidéo est envoyé.
- *flattrim*, qui permet de recalibrer les capteurs d'accélération lorsque le drone est à l'arrêt et sur un sol plat.

On voit donc que l'interface de programmation que nous fournit ardrone_autonomy est à elle seule plus riche que le SDK de Parrot sans ROS. On bénéficie en effet d'informations supplémentaires sur les données de navigation calculées dans le driver ainsi que d'un mode de communication simplifié, où il suffit de publier et de s'inscrire à des topics.

1. Cette estimation grâce au flux vidéo ne se fait pas grâce à la création d'une carte, mais en comparant plusieurs images consécutives et en estimant le déplacement entre plusieurs positions de la caméra. Le fait de ne pas utiliser de carte engendre un drift, c'est-à-dire qu'en attendant suffisamment longtemps, la position du drone sera complètement erronée.
2. Changer de mode signifie simplement passer du mode "emergency" où le drone ne peut plus faire bouger ses moteurs au mode normal et vice-versa

Chapitre 5

Premiers pas

Lorsque nous avons voulu commencer à implémenter notre application sous ROS, nous voulions profiter du travail que d'autres équipes avaient fait afin de ne pas devoir commencer à partir de zéro. En effet, nous ne sommes pas les premiers à programmer l'AR.Drone 2.0 en utilisant ROS.

Après avoir cherché ce qui se faisait déjà dans la littérature, nous avons remarqué que plusieurs options s'offraient à nous. Soit partir de l'application de la TUM (voir 3.3.4), soit utiliser l'intégration de PTAM par l'ETH Zurich (voir 3.3.3), soit partir du code de nos prédécesseurs et le porter vers ROS.

5.1 Le code de la TUM comme point de départ

Lors de notre recherche sur ROS et les drones programmés avec ce système, nous sommes tombés sur le projet de la TUM, université de Münich. Les résultats obtenus là-bas sont très impressionnants. Le drone est très stable et parvient à se déplacer d'un point à l'autre de façon précise. De plus le code est open source et le robot est programmé avec ROS. Tous ces critères nous ont poussé à vouloir partir de ce code et l'adapter pour effectuer la mission de l'année passée afin de servir de point de départ pour la suite. Partir de ce code aurait en effet présenté plusieurs avantages conséquents : l'architecture générale pour ROS était déjà construite, même si elle aurait du être modifiée pour être adaptée à notre mission (absence de stratégie). De plus plusieurs modules étaient déjà présents (filtre de Kalman pour l'estimation de la position, contrôle du drone en position ...). Nous nous sommes donc naturellement d'abord orientés vers cette possibilité.

La première étape était d'installer cette implémention de la TUM et d'apprendre à s'en servir, ce qui n'a pas trop posé de problèmes. Ensuite, avant de commencer à programmer, nous avons dû nous plonger dans le code et comprendre celui-ci. Cette étape s'est révélée beaucoup moins aisée en raison du manque de documentation de ce code. Nous avons cependant fini par comprendre que celui-ci n'était pas compatible avec l'utilisation que nous voulions en faire pour plusieurs raisons.

Tout d'abord, dans cette version du code, pour des raisons que nous ne connaissons pas, après avoir trouvé un ensemble de *landmarks* (les points de repères du drone sur l'image), le drone n'en cherche plus et est incapable d'en retrouver des nouveaux. Cela est étonnant puisque PTAM, l'algorithme utilisé dans le code, ne présente pas cette contrainte. Nous ne savons pas pourquoi elle existe dans cette implémentation. Étant donné que le drone ne peut pas se baser sur de nouveaux repères visuels pour se déplacer, cela rend l'exploration impossible.

Ensuite, l'architecture de ce code est sensiblement différente de ce que nous cherchons à faire. En effet, il n'y a pas de stratégie d'exploration. Le drone se contente de tenir sa position. Cela ne correspond donc pas à nos objectifs et les changements à opérer dans l'architecture du code sont importants.

Pour terminer, une troisième difficulté résidait dans le fait que le code de la TUM est créé pour fonctionner avec la caméra frontale de l'ARdrone, ce qui ne serait pas dans la continuité du travail des années précédentes. Modifier cela posait 2 problèmes : la caméra ventrale donne des images trop floues pour trouver suffisamment de landmarks et de plus, les mouvements détectés par les capteurs ne correspondaient pas aux mouvements de la caméra. Pour régler ces problèmes, nous avons mené des expériences avec un drone dont la caméra frontale pointait vers le bas et nous avons implémenté une matrice de rotation dans le code afin de faire correspondre le mouvement des caméras aux mouvements détectés par les capteurs. Un dernier problème était l'adaptation d'un estimateur d'échelle de la carte, qui est nécessaire au bon fonctionnement de PTAM, et nécessite un mouvement précis de la caméra. Un article y est consacré [20].

Pour toutes ces raisons et après plusieurs semaines de recherches et tentatives diverses pour régler ces problèmes, nous avons décidé que nous ne pouvions pas nous servir de ce code comme point de départ. Nous avons donc cherché autre chose.

5.2 L'utilisation de PTAM de l'ETHZASL comme point de départ

Le "Autonomous System Lab" de l'ETH Zurich proposait aussi une adaptation de PTAM pour ROS ainsi qu'un package pour réaliser la fusion des données des capteurs d'accélération et des caméras à l'aide d'un Extended Kalman Filter. L'avantage de leur implémentation était que la fusion de capteurs est totalement découplée du SLAM, ce qui la rend modulaire. Si l'on avait désiré utiliser un autre algorithme de SLAM que PTAM, cela aurait théoriquement été possible. Par contre, la pose corrigée par la fusion de capteurs n'était pas du tout utilisée par le SLAM et donc la carte construite ne la prend pas du tout en compte. Cette constatation nous laissait penser que ce genre d'approche nous obligerait à construire une autre carte qui contiendrait l'information spécifique à la navigation.

L'utilisation de cette application comme point de départ ne présentait pas autant d'avantages que le code de la TUM. En effet, la seule chose dont nous aurions bénéficié était une estimation de la position du drone plus facilement implémentable. Cependant, comme la carte n'était jamais mise à jour avec les nouvelles informations des capteurs du drone, il nous semblait possible d'obtenir de meilleurs résultats. Nous avons donc choisi de ne pas nous orienter vers cette solution non plus. Cependant, étant donné la modularité de notre code, nous pourrions, si nous le désirions, remplacer notre estimation de la position du drone par celle-ci sans avoir à repartir de zéro et en gardant intacts tous les modules qui ne sont pas concernés par cette estimation.

5.3 Le code des années précédentes comme point de départ

La dernière possibilité envisageable était de partir simplement du code des années précédentes et de le porter vers ROS. C'est une possibilité que nous voulions éviter car elle comportait de nombreux désavantages. En effet, l'architecture de ROS est foncièrement différente de celle du code des années précédentes et celui-ci n'était pas très bien documenté. Porter le code en ROS revenait presque à recommencer à zéro. Nous avons cependant opté pour cette dernière option. Le code de l'an passé nous a tout de même servi d'exemples pour réaliser notre régulateur de position.

5.4 Conclusion sur le choix du point de départ

Après avoir longtemps cherché un point de départ potentiel pour l'implémentation de notre application, nous avons finalement choisi de partir de zéro en nous inspirant du code des années précédentes. Cependant, tout le travail fourni jusque là n'était pas inutile. En effet, faire des recherches sur les autres points de départs potentiels nous a permis de comprendre beaucoup de choses sur ROS et son utilisation ainsi que ce qu'il est possible de faire avec des drones. Nous nous sommes également inspirés du code de la TUM pour implémenter notre programme.

Chapitre 6

Intelligence artificielle et régulation

Dans cette section, nous commençons par expliquer clairement la mission que nous souhaitons réaliser cette année (6.1). Ensuite, nous rappelons brièvement la structure du code qui est sensiblement restée la même depuis la ré-implémentation de la mission des années précédentes (6.2). Finalement, le code sera séparé en plusieurs modules qui seront expliqués en détails.

6.1 Nos objectifs

Cette année, nous avons décidé d'orienter notre mémoire sur le suivi autonome d'une cible mobile au sol. Plus concrètement, une mission classique se compose de 3 grosses parties.

Premièrement, lorsque nous démarrons la mission, un premier drone décolle et explore la pièce dans laquelle il se trouve à la recherche d'une cible dont il connaît l'apparence. Une photo est fournie au programme de détection d'image. Dans notre cas, cette cible est une voiture téléguidée orange recouverte d'autocollants. Tant que le drone n'a pas trouvé cette cible, il continue d'explorer les parties de la pièce où il n'est pas encore allé.

Deuxièmement, une fois la cible détectée, le drone arrête son exploration et commence à suivre cette cible en se positionnant juste au-dessus d'elle. Cette étape dure jusqu'à ce que le niveau de la batterie du drone atteigne un seuil critique. Dès lors, nous entrons dans la troisième partie de la mission où un deuxième drone entre en jeu.

Dans cette dernière étape, le premier drone en appelle un second qui était resté à la base avec sa batterie pleine. Celui-ci décolle et va rejoindre le premier dans le but de continuer à suivre la cible. Le point clé de cette étape repose sur la transition entre les drones. Afin de ne pas perdre la cible, le premier drone attend que le deuxième drone soit suffisamment proche de lui pour passer le relais et retourner à la base. Afin d'éviter tout contact entre les deux drones, le premier drone s'élève d'un mètre supplémentaire afin que le nouveau drone puisse passer en dessous dans le cas où leurs chemins se croiseraient.

Cette technique permet au deuxième drone de détecter la cible même si le premier drone est toujours au-dessus d'elle. Une fois arrivé au dessus de la base, le premier drone se pose et nous pouvons récupérer sa batterie pour la recharger.

Afin d'accomplir cette mission et partant de notre propre implémentation de la mission des années précédentes, nous avons principalement dû modifier le code de la stratégie ainsi que du *pathplanning*. Un autre changement essentiel à la réussite de notre mission a également été effectué dans le cadre du mémoire de l'autre groupe. Cette amélioration consiste à donner une estimation de la position du drone correcte même lorsqu'il suit la cible mobile afin que le deuxième drone puisse le rejoindre pour le remplacer.

6.2 Vue d'ensemble du code

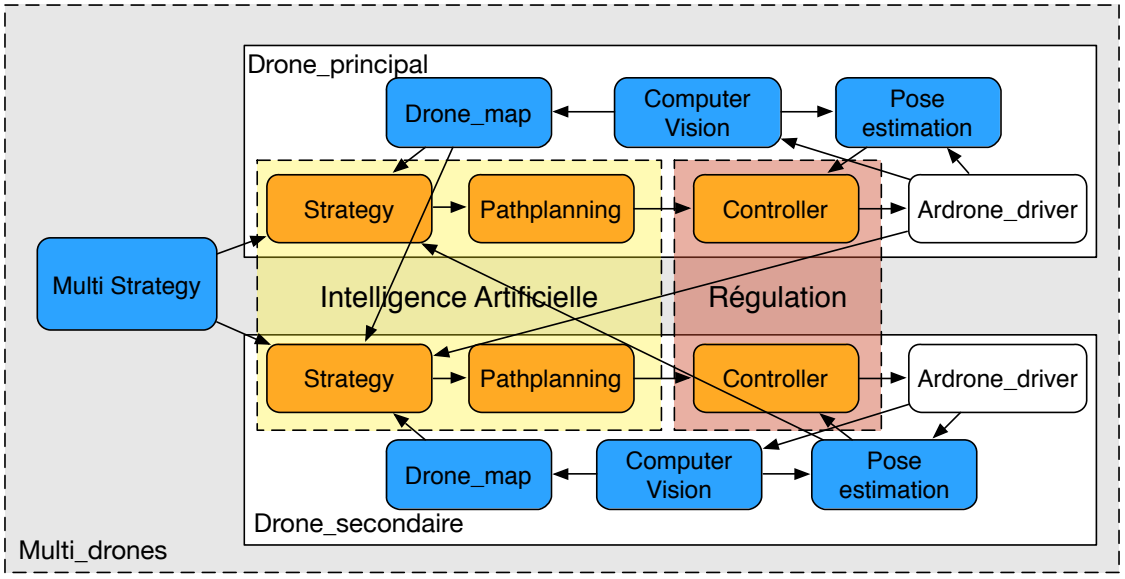


FIGURE 6.1 – Schéma de l'architecture du code

Comme nous pouvons le voir à la FIGURE 6.1 l'architecture de notre code n'a pas changé depuis la reconstitution de la mission de l'an passé présentée dans l'introduction. Nous y retrouvons toujours les nœuds implémentés par nos soins en orange et les nœuds implémentés par l'autre groupe en bleu. Le nœud en blanc fait quant à lui partie du package *ardrone_autonomy* expliqué à la section 4.4. Nous observons cependant plus de communications entre les noeuds de drones différents que lors de la mission précédente. Ceux-ci permettent notamment d'échanger plus d'informations entre les drones. On remarque sur le schéma que notre code est composé de 4 grosses parties :

— La partie **multi_drones**, en gris, permet de travailler avec plusieurs drones en même temps en leur attribuant des tâches différentes.

— La section **Intelligence Artificielle (IA)**, en jaune, qui indiquera au drone ce qu'il doit faire et où il doit aller en fonction de la tâche qui lui a été attribuée et des informations reçues par ses capteurs.
— La partie **Régulation**, en rouge, où sont implémentés les régulateurs en altitude, en position et en orientation du drone.
— Enfin, la partie estimation de la pose et construction de la carte qui est le sujet du mémoire de l'autre groupe et qui fournit à notre IA des informations sur l'objet trouvé. Cette partie donne également la position du drone à notre régulateur.

Les prochaines sections de ce chapitre ont pour objectif de décrire en détail l'implémentation et le fonctionnement des trois premières parties.

6.3 Multi-Strategy

Ce nœud ROS a été implémenté par l'autre groupe d'étudiants mémorants mais comme son fonctionnement a une grande importance pour la compréhension du reste de la stratégie des drones et que celui-ci a été longuement discuté entre nos deux groupes avant son implémentation, nous allons le décrire en quelques paragraphes.

Dans une application où de nombreux drones sont utilisés, plusieurs implémentations sont possibles pour implémenter le moyen de s'échanger de l'information. Soit tous les drones communiquent entre eux, soit un drone prend le rôle de maître, ou "*master*". Ce dernier est alors le seul à pouvoir communiquer avec les autres et c'est lui qui attribue les diverses tâches pour la durée de la mission. Une illustration de ces différents modes de communication est présentée à la FIGURE 6.2.

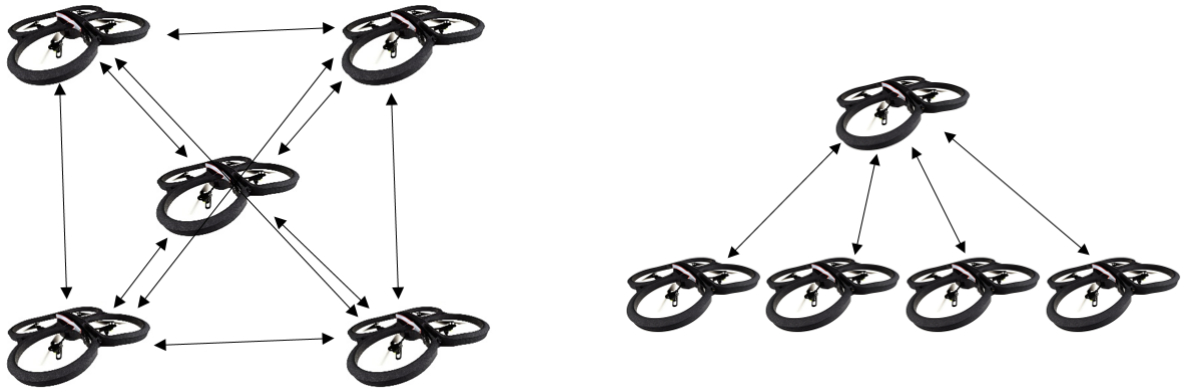


FIGURE 6.2 – A gauche, tous les drones communiquent entre eux. A droite, un drone *master* est le seul à communiquer avec les autres.

Chacune de ces situations présente divers avantages et inconvénients. Les avantages de la seconde méthode par rapport à la première sont que le nombre de réseaux de communi-

cation est moindre (un entre chaque drone et le *master* au lieu d'un entre chaque couple de drones) et que l'on y retrouve une information centralisée qui peut faciliter l'implémentation de diverses fonctions. Par exemple, la construction d'une carte sera plus facile puisqu'une seule map principale se servira des informations de tous les drones pour se construire alors que dans le cas précédent, les drones n'ont d'autre choix que de comparer leurs informations deux à deux afin de mettre leurs cartes à jour. Toujours concernant la seconde méthode, un premier inconvénient est que les drones doivent en permanence pouvoir rester en contact avec le *master* (comment, par exemple, éviter une collision entre deux drones si le *master* n'est pas là pour les prévenir de leur proximité ?). Cela peut être très handicapant dans le cadre de l'exploration *indoor*. Un second inconvénient est qu'un seul drone doit pouvoir se charger d'un beaucoup plus grand nombre d'opérations que les autres. A cause de ces inconvénients, nous avons choisi de nous orienter vers la première situation, celle où tous les drones peuvent communiquer entre eux.

Dans le cadre de notre mémoire, la situation est un peu particulière par rapport aux deux situations présentées *supra*. En effet, tout le programme que nous avons implémenté n'est pas embarqué sur les drones. Il fonctionne sur un ordinateur qui communique avec ceux-ci. Théoriquement, l'ordinateur est donc omniscient par rapport à l'état de tous les drones et pourrait agir comme *master* mais, en pratique, notre application a été implémentée de façon à ce que cela puisse ne pas être le cas. En effet, le programme propre à chaque drone fonctionne de façon indépendante par rapport à celui des autres. De cette façon, notre programme pourrait fonctionner sur l'ordinateur embarqué de chacun des drones si leur hardware le permettait. Ceci est en effet un des objectifs de notre mémoire que nous ne pouvons perdre de vue.

C'est dans cette optique que nous avons créé l'intelligence artificielle gouvernant le comportement des drones : chacun doit pouvoir agir indépendamment des autres tout en étant apte à communiquer avec chacun d'entre eux sans avoir à passer par un *master*. Dans notre mission, tout comme dans celle développée l'année passée, deux drones sont utilisés. Chacun d'eux obtient un rôle différent lors de l'initialisation du programme. En fonction des rôles attribués, nous les avons appelés *slave* et *master* et ils continueront d'être appelés ainsi dans la suite du mémoire. Le drone *master* est celui qui décolle en premier et qui cherche la cible tandis que le *slave* est celui qui le remplace lorsqu'il n'a plus de batterie. C'est l'utilisateur qui choisit quel drone a quel rôle dans le fichier de lancement du programme. Attention à ne pas confondre le nom *master* avec celui du drone qui centralisait les communications comme expliqué *supra*. Nous avons décidé d'appeler ce drone *master* car il est le seul à explorer, le premier à suivre la cible et le seul à appeler l'autre lorsqu'il n'a plus de batterie.

Dans le cadre de notre mission, afin que chaque drone puisse prendre connaissance de son rôle et du nom des autres drones, une décision initiale et centralisée doit être prise. Celle-ci est faite dans le nœud ROS *Multi Strategy*. Lors du lancement du programme, le nœud de la stratégie interne à chaque drone s'inscrit à un *topic* créé par le nœud *Multi*

Strategy et partagé entre tous les drones. Ce dernier nœud de son côté publie à intervalle régulier dans ce *topic* la liste des noms de chaque drone avec leur rôle respectif. Ainsi, comme tous les drones ont accès à cette liste, chacun est informé de son rôle et peut communiquer avec tous les autres grâce à la liste reprenant leurs noms respectifs, ceux-ci servant d'adressage pour les communications entre les nœuds de ROS. Une fois qu'un drone a reçu cette liste, il devient indépendant du nœud central. Cependant, si pour une application il devenait nécessaire de recourir à ce nœud central durant le déroulement de la mission, par exemple pour changer le rôle d'un drone et en informer les autres, cela serait possible puisque cette liste continue d'être publiée et peut être mise à jour. L'objectif initial qui est de rendre chaque drone au maximum indépendant est donc bien atteint.

6.4 Stratégie

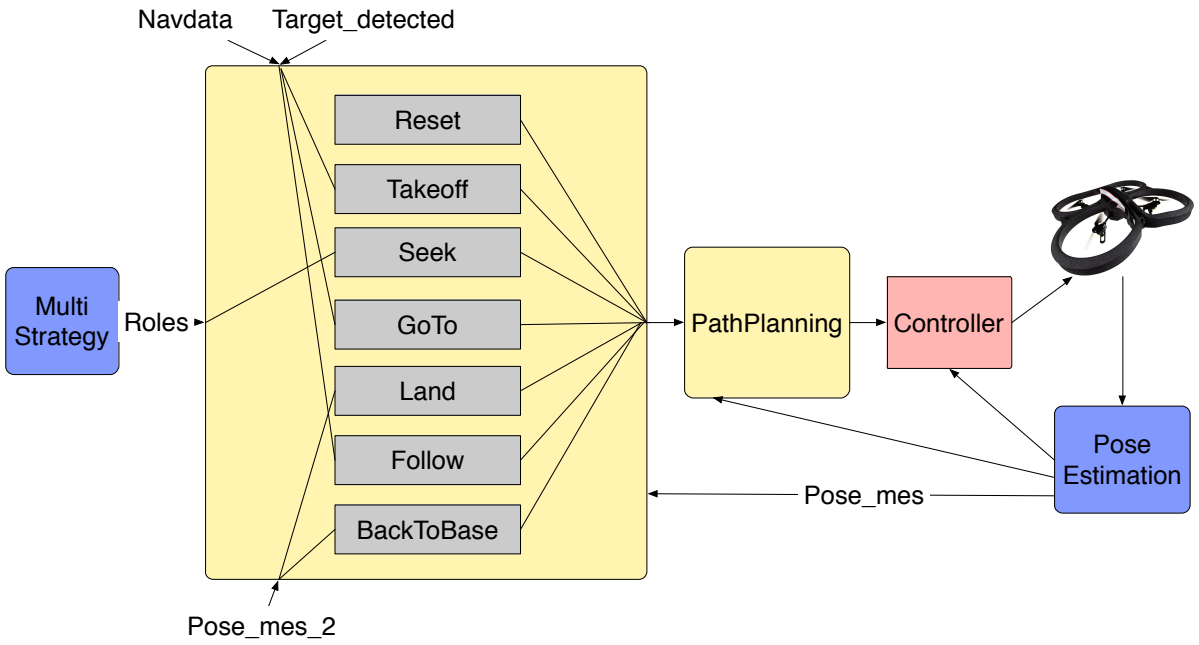


FIGURE 6.3 – Schéma bloc du nœud de stratégie et de la connexion aux autres nœuds via des topics

Le module *Strategy* est le nœud ROS qui décide du comportement général du drone en fonction des informations données par les autres nœuds. Un schéma bloc de son inclusion dans l'ensemble du code et son fonctionnement sont présentés à la figure 6.3. La stratégie choisit parmi une liste de 7 états différents lequel d'entre eux sera attribué au drone. Pour ce faire, la stratégie envoie le numéro de l'état choisi au *pathplanning* via un topic. La liste des différents états possibles est la suivante :

- RESET. Il est utilisé en début de mission et sert à initialiser certaines variables utilisées dans la suite du programme.
- TAKE OFF. Le drone est en phase de décollage.
- SEEK. Le drone explore son environnement et cherche la cible.
- GO TO. Le drone a pour mission de se rendre à un point donné. Dans notre mission il s'agit de l'endroit où se trouve la cible.
- LAND. Le drone est en phase d'atterrissage.
- FOLLOW. Le drone a pour mission de suivre la cible et de se positionner au dessus d'elle. Cet état ne sera utilisé qu'une fois la cible repérée.
- BACK TO BASE. Cet état indique au drone qu'il doit retourner à la base, son point de départ qui est aussi l'origine du repère de la carte et atterrir une fois qu'il en est suffisamment proche.

Nous allons maintenant expliquer brièvement les stratégies utilisées pour notre implémentation de la mission des années précédentes ainsi que celle de notre propre mission.

6.4.1 Stratégie de notre implémentation de la mission des années précédentes

Le schéma bloc présentant la stratégie de notre implémentation de la mission des années précédentes est présenté à la FIGURE 6.4. On y remarque que chacun des drones, qu'il s'agisse du *master* ou du *slave* commence par effectuer un *reset* afin d'initialiser les variables qu'il utilisera pour la suite du programme.

- Pour ce qui est du drone *master*, une fois qu'il a connaissance de son rôle, il décolle, cherche la cible et atterrit lorsqu'il la trouve.
- Concernant le drone *slave*, une fois qu'il a connaissance de son rôle, il attend que la cible soit trouvée par le *master*, décolle, se rend à l'endroit où se trouve la cible et atterrit lorsqu'il la trouve.

6.4.2 Stratégie de notre mission

Le schéma bloc présentant la stratégie de notre mission est présenté à la FIGURE 6.5. Celui-ci est déjà un peu plus complexe. Comme dit précédemment, chacun des drones, commence par effectuer un *reset* et attend de recevoir son rôle. Décrivons maintenant la stratégie de chacun des drones.

- Pour ce qui est du *master*, une fois qu'il a reçu son rôle, il décolle puis se met à chercher la cible. Il rentre ensuite dans une boucle dans laquelle il regarde si la cible a été trouvée ou non. Si elle n'a pas été trouvée, il regarde si l'environnement a été complètement exploré ou non. Si oui, il retourne à la base. Soit la cible n'est pas dans l'environnement, soit elle n'a pas été trouvée. Si la cible a été trouvée, il se met à la suivre et ce jusqu'à ce que son niveau de batterie atteigne un seuil critique. A ce moment, le drone *slave* devrait décoller et le rejoindre. Il continuera

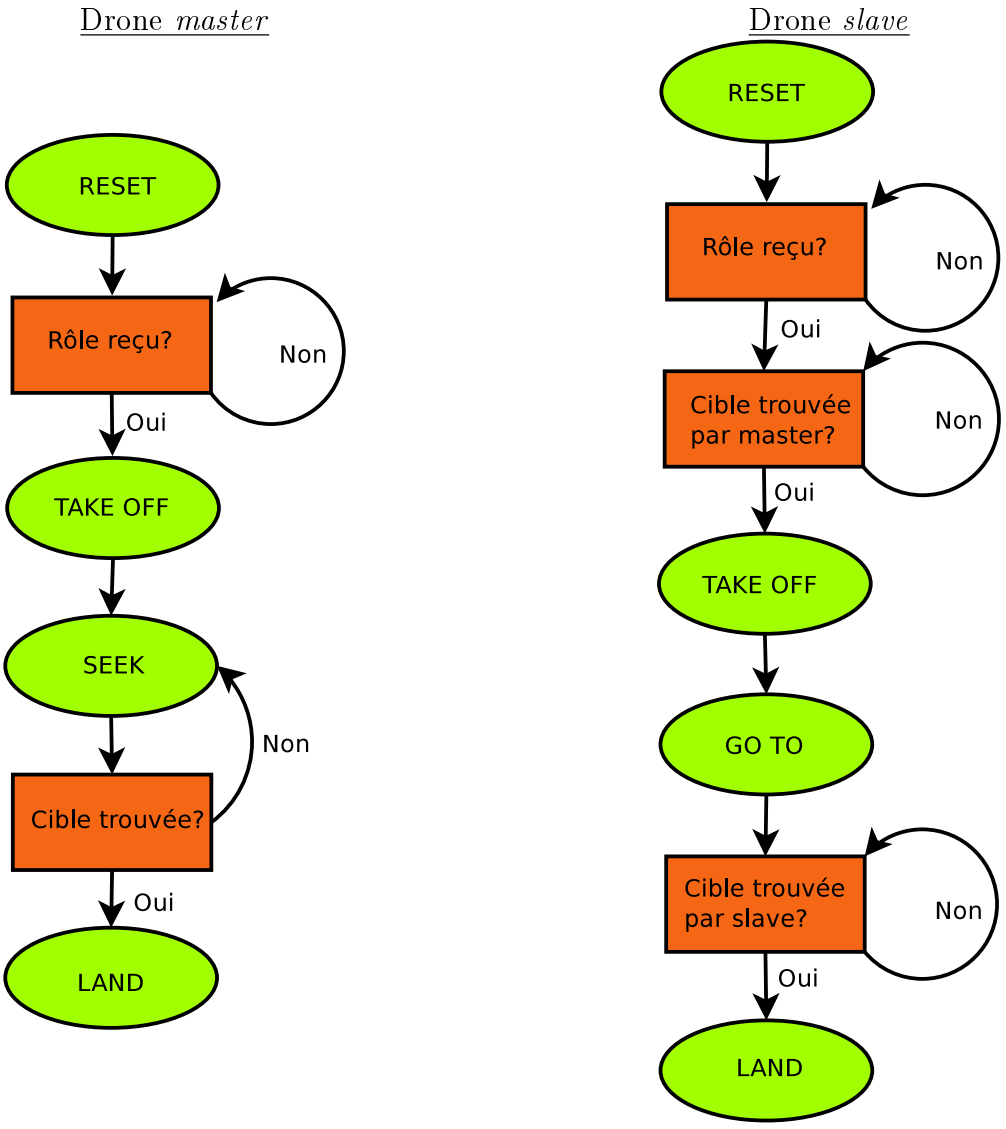


FIGURE 6.4 – Stratégies des drones *master* et *slave* dans notre ré-implémentation de la mission des années précédentes

de suivre la cible jusqu'à ce que celui-ci l'ait rejoint. Une fois que le drone *slave* est suffisamment proche de la cible, le drone *master* retourne à la base et atterrit.

- Concernant le drone *slave*, sa stratégie est fort semblable à celle qu'il avait dans notre implémentation de la mission de l'année précédente. Les différences notables sont : le fait que le drone master attende que son niveau de batterie descende en-dessous d'un seuil avant d'appeler le *slave* ainsi que la mise à jour des coordonnées auxquelles le drone doit se rendre jusqu'à ce qu'il détecte la cible. Cette dernière est en effet mobile dans notre cas. Cela est symbolisé dans notre diagramme d'états par la flèche "non" sortant du bloc "cible trouvée par *slave* ?" qui retourne jusqu'à l'état "GO TO" afin de mettre à jour la destination, contrairement à l'ancien

diagramme, où la flèche indiquait une simple boucle sur le bloc "cible trouvée par *slave* ?". La dernière différence est que le drone *slave* suit la cible une fois qu'il l'a trouvée plutôt que d'atterrir.

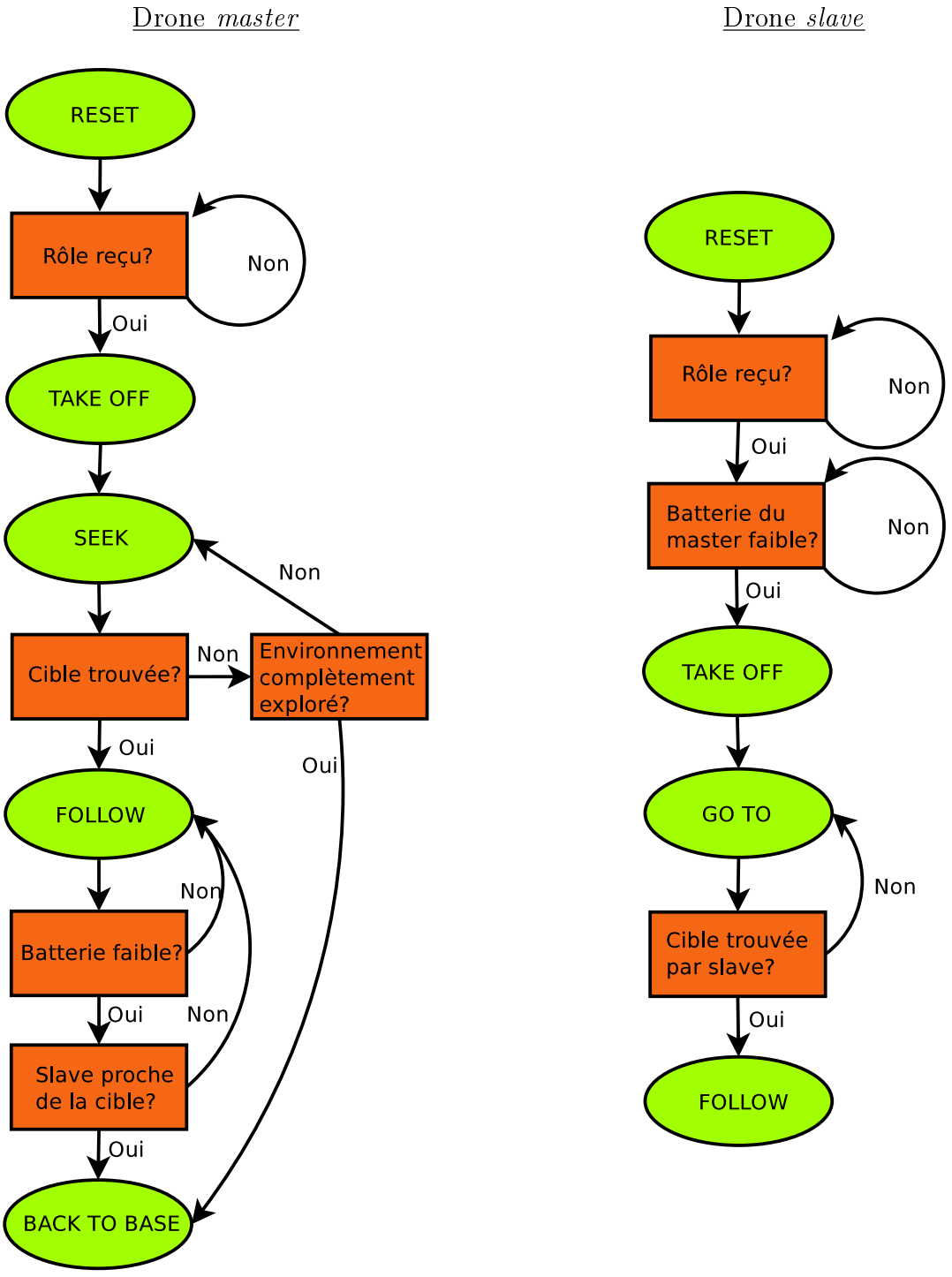


FIGURE 6.5 – Stratégies des drones *master* et *slave* dans notre mission

6.5 Planification de trajectoire et exploration

6.5.1 Rôle et description de ce module

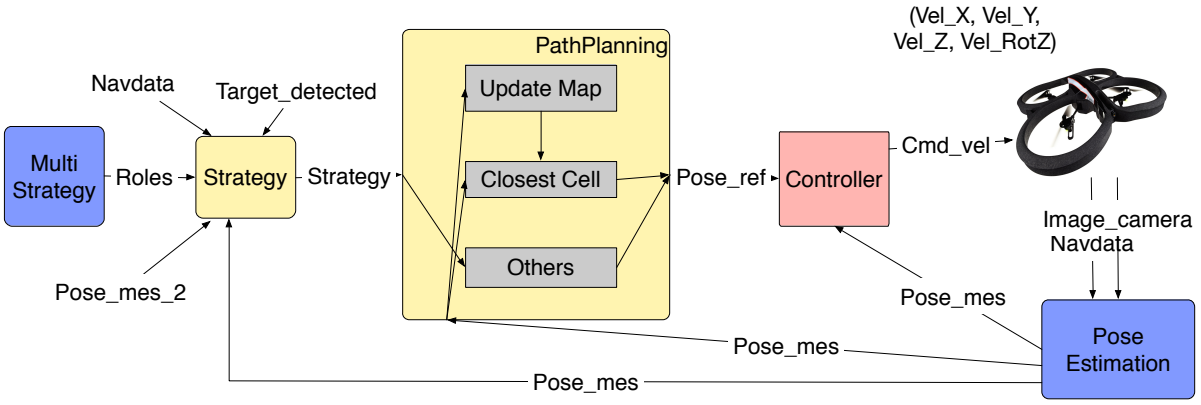


FIGURE 6.6 – Schéma bloc du nœud de planification de trajectoire et de la connexion aux autres nœuds via des topics

Le module *Pathplanning* est un nœud ROS qui fait le lien entre la stratégie et la correction de pose. C'est-à-dire qu'en fonction de l'état actuel du drone, qui lui est communiqué par la stratégie, il va indiquer au régulateur l'action à effectuer ou le prochain *setpoint* auquel le drone doit se rendre. Un schéma bloc de son inclusion dans l'ensemble du code et son fonctionnement sont présentés à la figure 6.6. Il est à noter que le comportement de ce nœud ROS est indépendant du rôle du drone (*master* ou *slave*). En fonction des différents états, les actions effectuées par le module seront les suivantes :

- RESET. Le nœud ne fait rien, les actions à effectuer dans cet état ne sont pas prises en charge par cet état.
- TAKE OFF. Le nœud indique au régulateur que le drone doit décoller.
- SEEK. Le nœud va donner une suite de *setpoints* au contrôleur afin que le drone explore son environnement. Celle-ci va dépendre de la mission effectuée (la nôtre ou notre ré-implémentation de celle de l'an passé). La façon d'obtenir cette suite de *setpoints* est expliquée dans la sous-section suivante.
- GO TO. Le nœud va simplement transmettre au régulateur le *setpoint* auquel le drone doit se rendre, qui lui a été communiqué par la stratégie au préalable.
- LAND. Le nœud indique au nœud *controller* que le drone doit atterrir.
- FOLLOW. Du point de vue du module *Pathplanning*, cet état est équivalent à GO TO. En effet, il communique simplement au contrôleur l'endroit où le drone doit se rendre, qui lui est communiqué par la stratégie et qui est le dernier endroit où la cible a été détectée. Cependant, ce comportement pourrait être modifié si cela s'avérait nécessaire en envoyant par exemple une consigne de vitesse et non plus une succession de points.

- BACK TO BASE. Le nœud envoie simplement l'origine du repère comme *setpoint* du drone et lui demande d'atterrir lorsqu'il en est suffisamment proche.

6.5.2 Algorithme d'exploration

a) Introduction

L'algorithme d'exploration dans le cadre de notre mission est l'algorithme calculant la trajectoire que le drone cherchera à suivre afin d'explorer au mieux son environnement et de trouver la cible au plus vite. Plusieurs articles présentant les stratégies d'exploration pour les drones et autres robots sont présentés dans la section 3.4. Les objectifs que nous nous sommes fixés pour le calcul de cette trajectoire sont les suivants : le drone doit parcourir son environnement afin que celui-ci ait été fouillé entièrement à l'issue de la mission dans le cas où la cible n'aurait pas été trouvée et cela doit être fait le plus rapidement possible.

Lorsque nous avons ré-implémenté la mission de l'année précédente, nous avons choisi un algorithme d'exploration simple, préférant ne pas nous concentrer sur cette partie et désirant aller à l'essentiel. Pour chercher la cible, le drone effectuait un parcours en forme de serpentin (voir FIGURE 6.7) dont l'espace entre les lignes parallèles étaient calculé de manière à ne pas laisser de zones non explorées. Nous avons décidé d'améliorer cela, l'un des objectifs de notre application étant la cartographie d'un environnement *indoor*[1]. Nous avons donc implémenté un algorithme capable de cartographier l'environnement en entier et de trouver la cible "à tous les coups" si celle-ci est statique.

b) Description de l'algorithme

La première chose importante à savoir est que notre algorithme se base sur un oracle interne pour savoir quelle est la zone à laquelle il peut accéder et donc quelle est la zone qu'il peut explorer. Cet oracle est une fonction que nous avons implémentée et dont les spécifications sont qu'on lui passe en argument des coordonnées dans le repère absolu de la carte et qui retourne **true** si cet endroit est accessible et **false** si il ne l'est pas. C'est donc cet oracle qui va décider de l'architecture de la pièce explorée par le drone. Celle-ci peut prendre de nombreuses formes, sous certaines conditions sur lesquelles nous reviendrons.

Il est à noter que notre algorithme d'exploration ne peut utiliser cet oracle que sur des coordonnées qui sont dans le champ visuel de la caméra ventrale du drone. Cela sert à représenter le fait que le drone voit qu'il y a un obstacle à cet endroit, comme un mur ou une colonne, et qu'il ne peut y accéder. L'objectif est de faire en sorte que cet oracle puisse être remplacé par nos successeurs par une fonction de *computer vision* permettant d'indiquer la présence d'un mur.

1. Objectif dont s'occupe l'autre groupe d'étudiants mémorants pour ce qui est de la cartographie en elle-même.

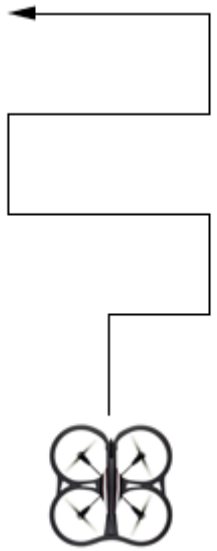


FIGURE 6.7 – La trajectoire effectuée par le drone dans notre ré-implémentation de la mission de l'année précédente

Pour l'instant, notre drone connaît à tous moments sa position et peut demander à un oracle si il y a un mur pour chacun des points présents dans le champ de vision de sa caméra ventrale (qui est un rectangle dont les dimensions dépendent de la hauteur du drone). Il nous faut donc créer un algorithme faisant en sorte que le drone sache quels sont les endroits qui ont déjà été explorés et permettant d'explorer les autres, quelles que soient la forme et les dimensions de la pièce définie par l'oracle.

L'approche que nous avons utilisée est la suivante : le repère absolu dans lequel le drone se déplace est divisé en cases, dont les tailles doivent être inférieures à celle de la cible afin d'être sur qu'elle soit trouvée si elle est immobile. Il ne faut cependant pas que ces cases soient trop petites car elles augmenteraient le temps de calcul et diminueraient donc les performances de notre drone. Nous avons choisi des cases carrées de 10 centimètres de côté. Ces cases forment un grillage enregistré dans notre code comme un tableau d'entiers. La valeur de chaque élément de ce tableau correspond à l'état de la case correspondante. Ces états sont les suivants :

- 0 : La case est inexplorée.
- 1 : La case est explorée et accessible.
- 2 : La case est explorée, accessible et est une frontière avec une région non-explorée.
- 3 : La case est explorée et inaccessible. A noter que ces deux qualificatifs ne sont pas contradictoires. La case a été explorée car le drone l'a eue dans son champ visuel sans pour autant qu'il n'ait eu à se rendre jusqu'à elle. Il peut s'agir par exemple d'un mur.

Au début de notre algorithme, avant que le drone ne décolle, toutes les cases sont à l'état "inexplorée". Une fois que le drone décolle, il commence à utiliser les 2 fonctions qui sont le cœur de l'algorithme. L'une permet de mettre à jour la carte en fonction de son état et de la position du drone, l'autre permet de calculer quel est le meilleur endroit où le drone devrait aller pour optimiser son exploration. Décrivons maintenant ces fonctions.

La fonction de mise à jour de la carte fonctionne de la manière suivante. A chaque fois qu'elle est appelée, elle va mettre à jour l'état de l'ensemble des cases qui sont dans le champ visuel du drone. L'état des cases peut être modifié des manières suivantes :

- Une case inexplorée devient explorée et inaccessible si l'oracle renvoie **false** pour le point en son centre. Sinon, elle devient explorée, accessible et frontière si elle est à l'extrémité du champ visuel du drone et simplement explorée et accessible dans le cas contraire.
- Une case explorée et accessible reste explorée et accessible.
- Une case qui est explorée, accessible et frontière devient simplement explorée et accessible si elle est n'est plus à la frontière du champ visuel du drone et reste frontière sinon.
- Une case explorée et inaccessible reste explorée et inaccessible.

Cette manière de fonctionner permet à la frontière entre la zone explorée et la zone inexplorée par le drone d'être composée de cases frontières et de cases inaccessibles en permanence.

La fonction choisissant le meilleur endroit auquel le drone doit se rendre sélectionne en fait la meilleure case. Celle-ci sera toujours une case frontière. Lorsque le drone se rend sur une case frontière, il augmente la taille de la zone déjà explorée. Afin de choisir quelle est la meilleure case frontière, il compare simplement leur proximité par rapport à sa position actuelle et se rend à celle dont le centre est le plus proche.

Cet algorithme se termine lorsque la pièce est explorée en entier. En effet, à ce moment, il n'y a plus aucune case frontière et accessible puisqu'elles auront toutes été remplacées par des cases explorées et inaccessibles, correspondant aux murs ou par des cases explorées et accessibles. A ce moment, le drone a pour instruction de retourner à son point de départ, d'où il pourra être relancé afin de tenter à nouveau de trouver la cible. En effet, celle-ci a sans doute eu l'occasion de se déplacer pendant l'exploration du drone et aura ainsi pu échapper à la vue de celui-ci, mais la pièce aura été cartographiée entièrement.

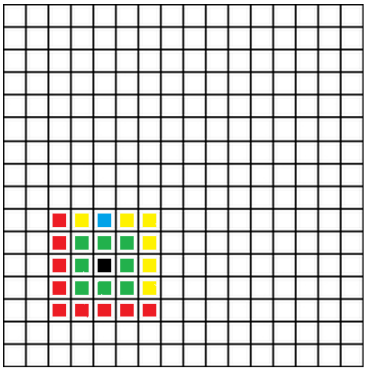
Les deux fonctions décrites ci-dessus sont utilisées tour à tour en permanence. Il n'est pas nécessaire que le drone atteigne la nouvelle case qui est son objectif avant de mettre sa grille à jour et de recalculer la nouvelle case objectif la plus proche. Cela permet d'adapter le parcours du drone aux imprévus dans la trajectoire et d'améliorer la vitesse d'exploration (un nouvelle case frontière plus proche peut être trouvée en se rendant vers la précédente si la trajectoire du drone a été perturbée). Un exemple de fonctionnement théorique de l'algorithme est présenté sur la FIGURE 6.8.

Cet algorithme fonctionne quelle que soit la forme de la pièce, tant que celle-ci est convexe. C'est-à-dire que le drone doit pouvoir se rendre de n'importe quel point à n'importe quel autre de celle-ci en ligne droite sans rentrer en collision avec un mur, puisque c'est toujours ainsi que le drone se déplace. Afin de contourner cette contrainte, il faudrait implémenter un algorithme A* [41] ou de Dijkstra [42] afin de trouver la meilleure trajectoire d'une case à une autre. Cette amélioration est laissée au soin des futurs mémorants.

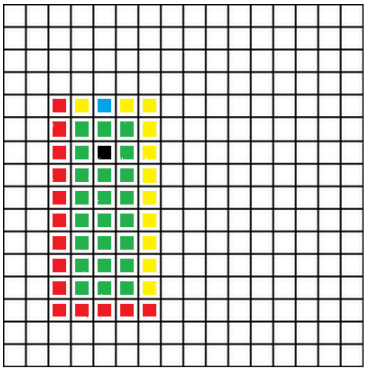
c) Résultats

La FIGURE 6.9 présente les résultats obtenus lors de l'exploration d'une pièce de $3 * 3$ mètres par le drone. Celle-ci dure 5 minutes. Pour ce test, le drone volait à une altitude de 1 mètre et voyait donc à une distance de 30 centimètres en avant et arrière et de 40 centimètres sur ses flancs. On constate donc que les objectifs fixés sont bien atteints : le drone a exploré la pièce entièrement (l'entièreté du sol est passé dans son champ de vision) et il n'a pas traversé les murs.

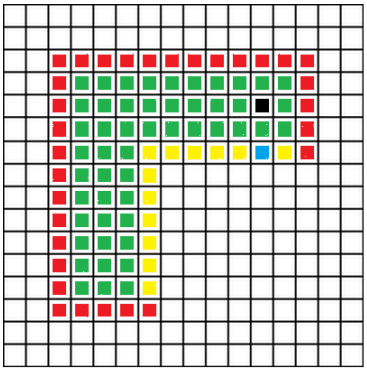
1)



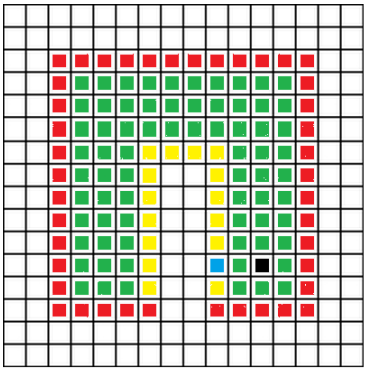
2)



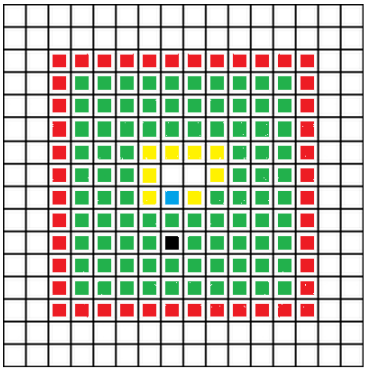
3)



4)



5)



6)

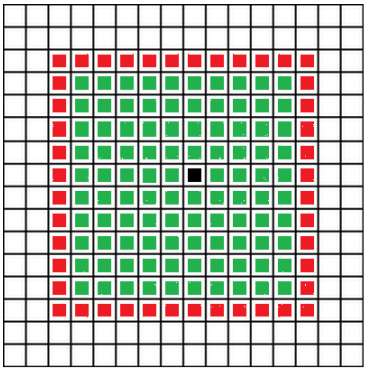


FIGURE 6.8 – Exemple de fonctionnement de l'algorithme d'exploration. En blanc, les cases inexplorées, en rouge, les cases inaccessibles, en jaune les cases frontières accessibles, en vert les cases explorées et accessibles, en noir la position du drone et en bleu, la case la plus proche du drone où il désire se rendre. La taille des cases a été exagérée.

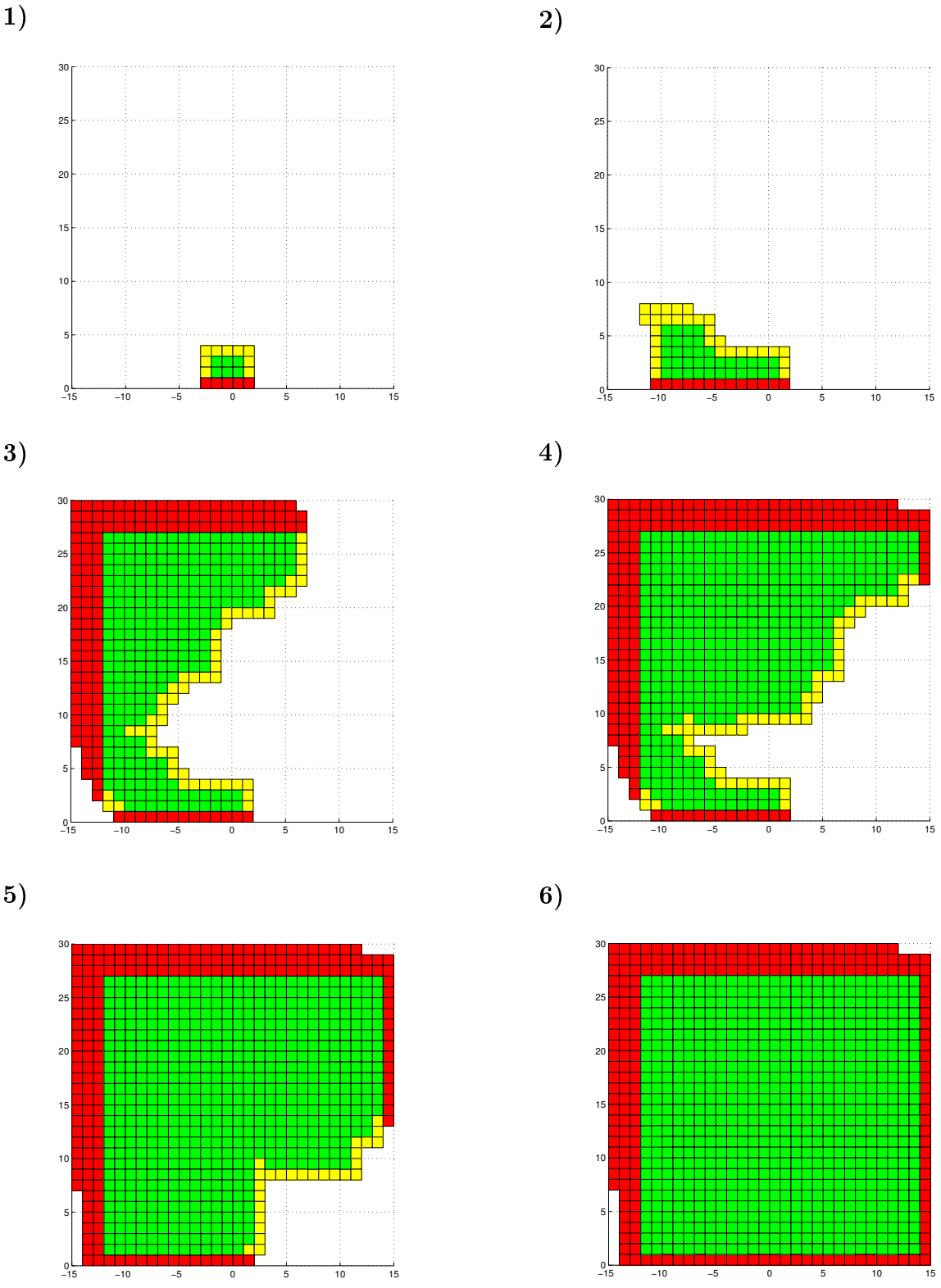


FIGURE 6.9 – Résultat obtenu lors de l'exploration d'une pièce de $9m^2$ par le drone. En rouge, les cases inaccessibles, en jaune les cases frontières accessibles et en vert les cases explorées.

6.6 Régulation

Une fois l'Intelligence Artificielle implémentée sur nos drones, ceux-ci sont capables d'analyser une situation et de savoir comment réagir. Dans la quasi totalité des cas, celle-ci demande à notre drone de se rendre à un endroit spécifique. Il est donc important de créer un régulateur qui pourra au mieux contrôler la vitesse du drone pour qu'il atteigne précisément sa position de référence.

L'AR.Drone 2.0 dispose déjà d'un régulateur interne, implémenté par Parrot, qui lui permet de rester stable en cas de turbulences (vent, effets de sol) ou d'accidents (bousculades, collisions). Ce régulateur, dont nous ne connaissons ni la nature ni la valeur des gains, repose sur l'intégration des capteurs inertiels, du flux vidéo, de l'altimètre, de l'ultrason et du magnétomètre tri-axial. Cependant, celui-ci, en plus de présenter quelques grosses dérives, détaillées dans les sous-sections suivantes, n'est pas suffisant pour se déplacer d'un point A à un point B. Ainsi, il est important d'implémenter notre propre régulateur qui donne des consignes de vitesses au drone et qui encapsule son régulateur interne.

Afin d'implémenter cette boucle de régulation, nous avons décidé de nous baser sur un modèle existant et d'ensuite en ajuster les paramètres. Ce modèle, MAVwork, a été créé par l'*"Australian Research Centre for Aerospace Automation"* et régule en parallèle l'altitude, la longitude, la latitude ainsi que l'orientation autour de l'axe z. C'est donc de cette façon que nous avons compartimenté notre boucle de régulation. La raison pour laquelle nous avons décidé d'utiliser ce modèle est détaillée *infra*.

6.6.1 Introduction

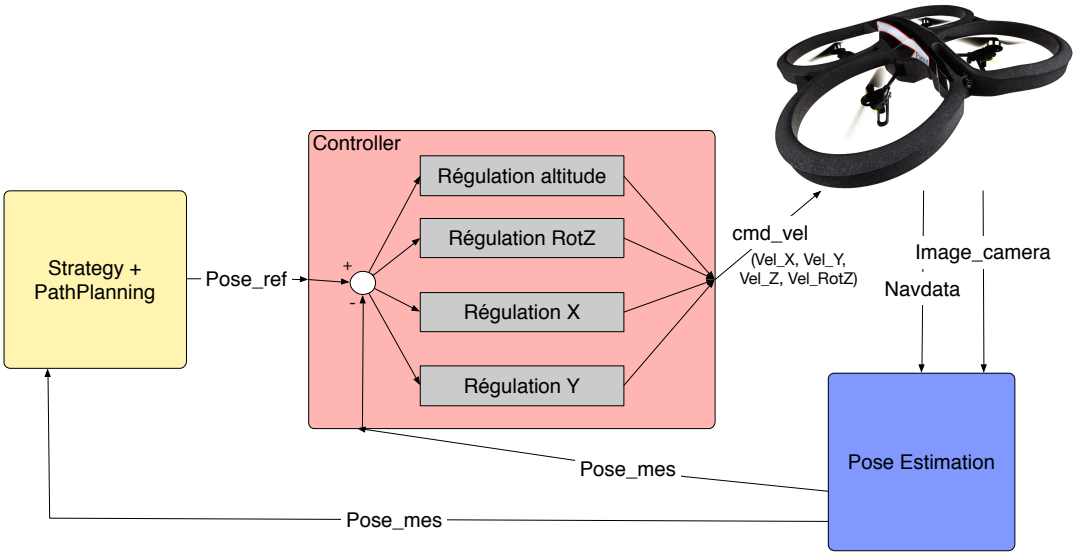


FIGURE 6.10 – Schéma bloc du nœud de régulation et de la connexion aux autres noeuds via des topics

On peut voir sur la FIGURE 6.10 que notre nœud de régulation (toujours en rouge) reçoit des informations de l'intelligence artificielle (en jaune) et du nœud de la *pose estimation* (en bleu). L'intelligence artificielle envoie une position de référence au régulateur via le topic *Pose_ref*. Ce topic contient des messages de type "Pose3D" dont la structure leur permet de communiquer une valeur pour chacun des 6 degrés de libertés (3 translations + 3 rotations) ainsi que leur vitesse. Un autre topic appelé *Pose_mes* permet au nœud *Pose Estimation* d'envoyer la position actuelle du drone au *controller*. Les messages y sont aussi de type "Pose3D".

Notre régulateur, quant à lui, envoie des messages au drone via le topic *cmd_vel*. Celui-ci contient des messages de type "geometry_msgs : :Twist" *[sic]* qui nous permettent de renseigner la vitesse à attribuer à chacun des degrés de liberté. C'est ensuite le code interne du drone qui interprètera cette commande pour donner des consignes aux 4 moteurs. Cette interprétation se faisant dans le secret du code interne de Parrot.

Étant donné que les noeuds *PathPlanning* et *Pose Estimation* publient des nouveaux messages à une fréquence de 20 Hz, nous avons choisi cette même fréquence pour faire tourner notre régulateur. De cette façon, notre régulateur peut s'adapter à chacun des messages de l'estimation de la pose et reçoit directement une réponse. Ces deux nœuds, bien qu'étant asynchrones, fonctionnent donc au même rythme. Une fois ces deux messages reçus, le noeud *controller* les transmet à ses 4 fonctions internes qui vont chacune contrôler un degré de liberté.

Le fait de scinder la régulation en plusieurs branches nous permet d'avoir un code plus modulaire et plus facilement adaptable. En effet, si quelqu'un souhaite utiliser notre code sur un autre drone, il peut facilement tester chaque partie du régulateur une par une et ajuster les paramètres pour que notre régulateur fonctionne avec son drone. De même, si quelqu'un considère que notre régulateur pour tel ou tel degré de liberté n'est pas assez performant, il pourra le modifier plus aisément en laissant les autres parties intactes. Pour le moment, notre contrôle fonctionne sur base d'une régulation sur des termes *proportionnels, intégraux et dérivés*. Un des avantages d'un régulateur *PID* est qu'il est simple à comprendre et à mettre en place. De plus, il est très utilisé ce qui rend notre code plus attractif et ouvert à tous. Et comme nous le verrons par la suite, ses performances sont tout à fait suffisantes pour notre application.

Nous aurions pu trouver un régulateur plus élaboré dont les paramètres étaient auto-ajustables ou bien encore développer un modèle prédictif afin d'anticiper les erreurs et non plus seulement les corriger. Cependant, cela nous est impossible en raison du fait que notre correction de pose englobe déjà une régulation embarquée dont nous ne connaissons pas le fonctionnement.

6.6.2 Régulation en Altitude

Maintenant que le type d'intégrateur choisi a été expliqué, commençons par décrire le fonctionnement de la régulation en altitude. Dans le cadre de notre mémoire, l'altitude est très importante dans le nœud en charge de l'exploration de la pièce. En effet, la hauteur du drone par rapport au sol fait varier la taille de la zone qui est vue par la caméra ventrale. Il est donc primordial de voler à l'altitude désirée afin d'explorer l'environnement dans les meilleures conditions.

Dans cette sous-section, nous commençons par décrire en détail notre fonction qui régule l'altitude (a)). Ensuite, nous expliquons des problèmes rémanents introduits par la régulation interne de Parrot (b)). Enfin, nous montrons les résultats obtenus lors de nos tests (c)).

a) Fonction de régulation en altitude

Afin de corriger l'altitude, nous avons d'abord commencé par créer un simple régulateur proportionnel. Après quelques ajustages du coefficient, nous obtenions toujours des oscillations de hauteur autour de l'altitude désirée. Celles-ci se voyaient aussi bien sur nos écrans qu'en regardant la trajectoire du drone. Nous avons alors décidé d'insérer un terme dérivé. Celui-ci a eu pour conséquence de lisser les oscillations et nous avons alors eu un résultat plus que satisfaisant. Le drone se place rapidement à la bonne hauteur et y reste. Le schéma bloc de notre régulateur est disponible à la FIGURE 6.11. On remarque qu'après notre régulation, la vitesse que l'on envoie au drone est saturée entre les valeurs +1 et -1. Cette saturation est implémentée dans *ardrone_autonomy* pour faire correspondre notre commande avec l'intervalle d'envoi autorisé. Cependant, il semblerait que

cette saturation soit inutile et due à une mauvaise compréhension du code de Parrot par les auteurs de l'*ardrone_ autonomy*. En effet, la valeur contenue dans *cmd_ vel.z* exprime une vitesse verticale en mètres par seconde et pas un coefficient qui multiplie la vitesse verticale maximale [23].

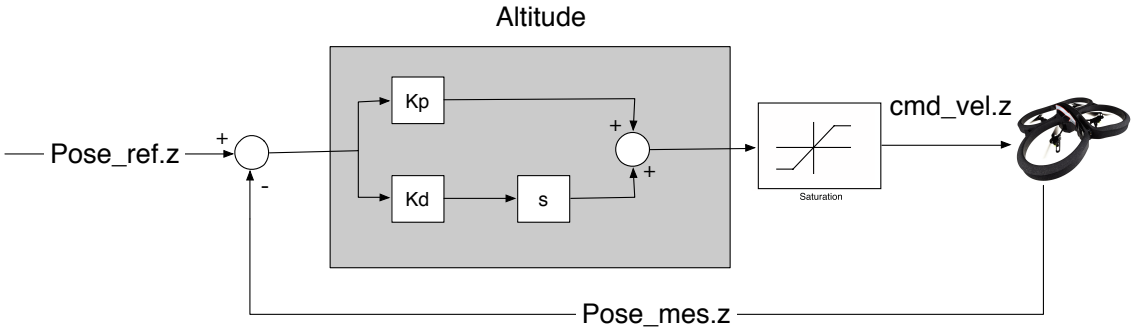


FIGURE 6.11 – Schéma bloc de notre fonction qui régule la position en altitude

b) Problèmes liés au régulateur interne

Comme nous l'avons déjà expliqué *supra*, les AR.Drones 2.0 sont déjà équipés d'un régulateur en altitude. Celui-ci fonctionne sur un programme interne qui ne nous est ni détaillé ni accessible. Pour rappel, ce régulateur est efficace lorsqu'il s'agit de maintenir le drone en place. Par contre, il peut induire de larges erreurs lorsque le drone se déplace. Pour l'altitude, une erreur typique est illustrée à la FIGURE 6.12. Le schéma que l'on y retrouve est séparé en 6 situations différentes. Il est important de préciser que pendant toute la durée du vol illustré sur ce schéma, le drone est censé voler à une hauteur constante qui est *Z0*. Dans les situations *S0* et *S5*, le drone vole sur une surface plane. Dans la situation *S1*, le drone passe au dessus d'une falaise verticale d'une hauteur h. Dans les autres situations, le drone vole sur des surfaces pentues, qui montent ou descendent de manière lente et progressive.

Malgré le fait que l'on ne demande pas au drone de faire varier son altitude, on remarque que sa hauteur finale est plus basse que sa hauteur initiale. Ce phénomène s'explique au moyen de la régulation interne au drone. En effet, lorsque le drone perçoit un changement rapide, abrupt dans sa hauteur, le code interne considère qu'il s'agit d'un objet momentané qui passe sous le drone et ce dernier indique qu'il est toujours à la même altitude afin que le drone ne monte pas soudainement. Cette situation correspond à la zone *S1* sur notre schéma. Le drone passe alors de l'altitude relative *Z0* à l'altitude relative *Z0-h*. On remarque que cette régulation est plutôt bien pensée car le drone reste à la même hauteur absolue et ainsi, il ne devra pas redescendre une fois l'objet au sol dépassé.
Cependant, on peut remarquer que lorsque la distance entre le drone et le sol varie de manière continue, i.e. quand il s'agit d'une pente au lieu d'une transition nette, le régu-

lateur interne croit que c'est le drone qui dévie et décide donc de s'adapter en modifiant sa commande en altitude. C'est le cas des situations *S2,S3,S4*. Même si les situations *S2* et *S3* se compensent, la situation *S4*, quant à elle introduit un offset d'une valeur de *-h*. Ceci est du au fait que le drone n'a pas pris en compte le changement soudain de distance verticale entre lui et le sol mais régule les changements légers de *S4*.

Ce schéma nous résume donc un problème de hauteur dont le drone est victime à chaque fois qu'il passe au dessus d'objets qui ont une pente raide et une pente lisse. C'est le cas lorsque le drone passe, par exemple, au dessus d'un classeur posé au sol.

Dans le cadre de notre mémoire, ce problème peut difficilement être résolu. Cependant, il est possible de remédier à cela en prenant compte de ces problèmes dans l'estimation de la pose. La pose pourrait, par exemple, se baser sur le flux vidéo et sur l'intégration des capteurs inertiels et à ultrasons pour détecter qu'il y a eu un saut et qu'après cela, le drone n'est pas en train de bouger mais que le sol retourne progressivement à sa hauteur originale.

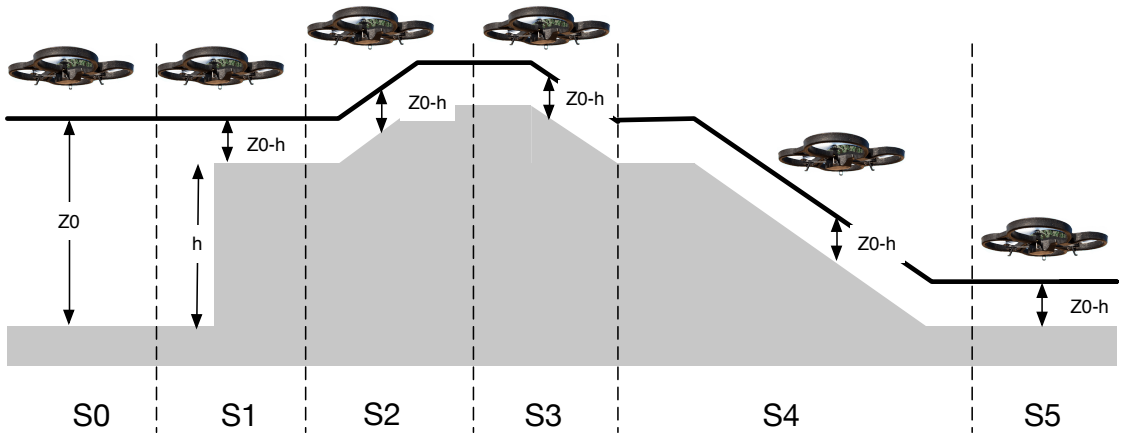


FIGURE 6.12 – Schéma mettant en scène la problématique introduite par le régulateur d'altitude interne

c) Résultats

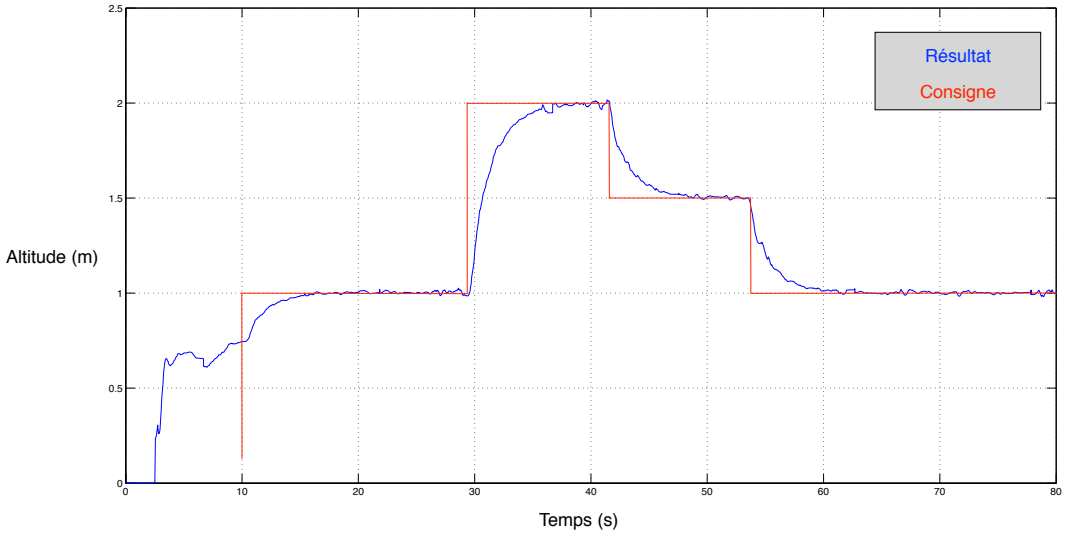


FIGURE 6.13 – Résultats de la réaction du drone face à différentes consignes en altitude.

On peut observer à la FIGURE 6.13 la réaction du drone lorsqu'on lui applique différentes consignes en altitude. Tout d'abord, on peut remarquer un changement drastique en altitude aux alentours de 3 secondes. Celui-ci correspond au moment où le drone décolle. Le programme interne du drone essaie alors de le stabiliser à 0.7 mètre du sol. Il reste à cette position de référence (interne au drone) pendant 7 secondes.

10 secondes après le lancement de notre programme, nous envoyons la première consigne au drone. Celui-ci doit se placer à une altitude de 1 mètre pendant environ 20 secondes. On remarque que la phase de transition entre la première et la deuxième altitude prend plus ou moins 5 secondes. Une fois arrivée à sa nouvelle position de référence, le drone se stabilise et seulement de faibles oscillations apparaissent. On peut cependant remarquer que ces oscillations sont bien moins importantes que celles observées lors des 10 premières secondes. Notre régulation en altitude est donc plus lente mais plus précise et plus stable que la régulation de Parrot qui est interne au drone.

Un peu avant la 30ème seconde, alors que le drone est toujours stable, ce dernier reçoit une nouvelle consigne lui indiquant de monter à une altitude de 2 mètres. Une fois de plus, le drone met environ 5 secondes pour atteindre son altitude de référence et s'y stabilise. Ceci montre que notre régulateur est robuste et constant quelle que soit l'altitude à laquelle se trouve le drone et quelle que soit la nouvelle altitude demandée.

12 secondes après le début de la consigne de 2 mètres, une nouvelle consigne demande, cette fois, au drone de descendre. Le drone doit se stabiliser à une nouvelle altitude de 1,50

mètres. Une fois de plus, le drone rejoint sa consigne en 5 secondes. Notre implémentation du régulateur fonctionne donc aussi bien pour monter que pour descendre. Enfin, notre drone reçoit une dernière consigne qui lui demande de rester à une altitude de 1 mètre jusqu'à la fin de cette expérience.

On peut donc conclure que notre régulateur fonctionne selon nos attentes. Nous avons privilégié un système précis et plus lent plutôt qu'un système dynamique et plus instable. En effet, dans le cadre de notre mémoire, nous fonctionnons avec des drones "low-cost" et des mouvements brusques entrainent souvent des erreurs dans les capteurs. Il est donc préférable de garder le drone stable plutôt que de le rendre trop dynamique. Nous pouvons voir sur le graphe que la présence d'un terme intégral n'était pas utile. En effet, notre drone ne semble pas présenter d'erreur statique sur son altitude.

6.6.3 Régulation en RotZ

Continuons à explorer la régulation de notre drone en analysant la deuxième fonction qui la compose : le contrôle en orientation. Dans le cadre de notre mémoire, la régulation en orientation n'est pas essentielle. Dans notre mission, nous l'utilisons simplement pour faire en sorte que le drone pointe toujours dans la même direction. Celle-ci pourrait tout à fait fonctionner même sans cette régulation. Cependant il est utile de l'implémenter pour deux raisons.

Premièrement, il existe deux grandes façons pour un drone de se rendre d'un point A à un point B. Dans la première approche, le drone peut s'orienter vers le point B puis avancer jusqu'à y arriver. Dans la deuxième approche, celle que nous avons choisie, le drone peut se déplacer en combinant des mouvement latéraux et longitudinaux afin d'arriver au point B sans avoir à pivoter. Même si nous avons opté pour cette dernière méthode, nous avons préféré implémenter la régulation en orientation afin de permettre de rapidement passer d'une méthode de déplacement à une autre en cas de besoin.

Deuxièmement, il est intéressant de pouvoir choisir l'orientation du drone si l'on utilise la caméra frontale. Que ce soit pour détecter des obstacles, s'orienter ou même analyser l'espace qui nous entoure, cette dernière pourrait nécessiter le besoin de pointer dans une certaine direction afin d'en fournir des images.

Dans cette sous-section, nous commençons d'abord par décrire le fonctionnement de notre contrôle d'orientation (a)). Ensuite, nous évoquerons la présence d'une erreur liée au régulateur interne (b)). Enfin, nous discuterons des résultats obtenus lors des différents essais (c)).

a) Fonction de régulation en orientation

Lorsque le drone s'allume, il crée un repère absolu dont les axes sont parallèles à son propre repère, i.e. le repère relatif. Alors que ce dernier se déplacera avec lui (pour rappel,

l'axe x pointant dans la même direction que la caméra frontale), le repère absolu restera en place. Par conséquent, lorsque le drone tournera sur lui-même, son repère ne sera plus aligné avec le repère absolu. La différence d'angle entre ces deux repères correspond à la valeur que nous souhaitons réguler. Celle-ci se situe dans un créneau compris entre $-\pi$ et π, 0 signifiant que le drone est orienté de la même façon que lors de son démarrage et π voulant dire qu'il a fait un demi tour sur lui même.

Afin de réguler cette valeur, nous avons décidé d'utiliser un régulateur proportionnel et dérivé pour les mêmes raisons que pour l'altitude. Nous souhaitions arriver précisément au bon angle tout en évitant les oscillations autour de la valeur de référence. Le schéma bloc de cette régulation est disponible à la FIGURE 6.14.

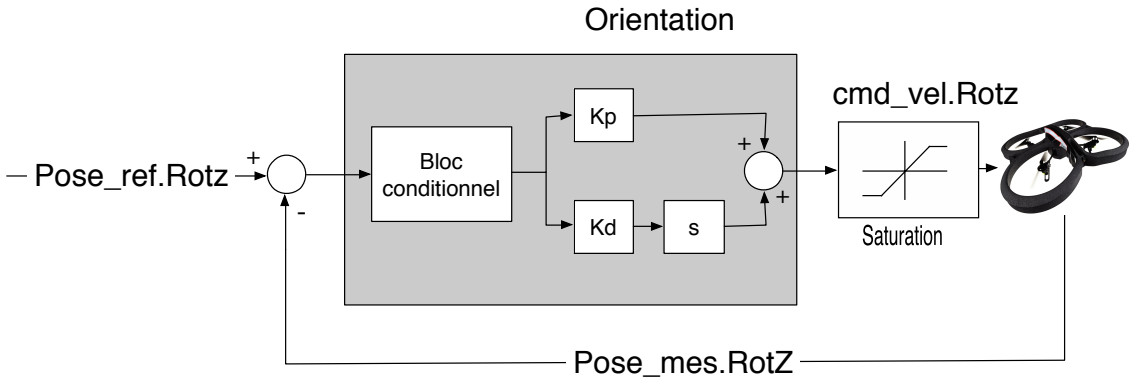


FIGURE 6.14 – Schéma mettant en scène la problématique introduite par le régulateur d'altitude interne

On peut remarquer que la fonction orientation contient un bloc conditionnel. Celui-ci permet au drone de se rendre plus rapidement de la zone $[-\frac{\pi}{2}; -\pi]$ à la zone $[+\frac{\pi}{2}; +\pi]$. En effet, si ce bloc conditionnel n'était pas présent et que le drone était orienté à -179°, il devrait faire tout le tour en repassant par un angle de 0° pour aller à +179°. Afin d'éviter une perte de temps et une perte d'énergie, nous regardons si la différence entre l'angle actuel et l'angle de référence n'est pas plus grande en valeur absolue qu'un demi tour (π). Si c'est le cas, nous soustrayons (ou ajoutons dans le cas d'une valeur négative) un tour complet (2π) à la différence des deux angles afin que le drone passe par le plus petit chemin. Toujours dans l'optique d'avoir un drone qui ne se perd pas, nous avons introduit un saturateur pour limiter la vitesse de pivotement de celui-ci. De cette façon, il est moins susceptible de perdre ses repères au sol lorsqu'il change d'orientation.

b) Erreur interne

Tout comme pour l'altitude, l'implémentation interne de Parrot peut parfois nous poser quelques soucis. Tout d'abord, il semblerait que le drone n'utilise pas le flux vidéo pour calculer ses variations d'orientation. Il ne prendrait en compte que son magnétomètre et

ses capteurs inertiels. Il arrive alors bien souvent que le drone se mette à tourner lentement sur lui-même. Ceci est du à l'intégration des capteurs qui donnent parfois des valeurs erronées ou pas assez précises. Le code interne de Parrot croit détecter une rotation du drone sur lui-même selon l'axe Z et réagit en tournant dans l'autre sens. Nous en sommes venus à cette conclusion car les informations de navigation reçues par le drone nous indiquent qu'il se considère toujours comme fixe alors qu'il tourne.

Ces rotations, survenant de manière aléatoire et à des amplitudes très différentes, suffisent presque toujours à mettre la mission en péril. En effet, le fait que le drone ne soit pas en phase avec le repère absolu introduit une erreur dans le déplacement de celui-ci. Alors qu'il doit par exemple avancer dans l'axe des X, le drone qui a maintenant pivoté d'un quart de tour avance en suivant l'axe des Y. Ceci a pour conséquence directe de perdre le drone qui sort de la zone qu'il est censé explorer. De plus, si par chance il arrive tout de même à trouver sa cible, les informations concernant sa position absolue sont erronées et le deuxième drone ne parvient jamais à venir le remplacer lorsque sa batterie faiblit.

Un autre problème que nous avons rencontré mais qui a pu être réglé est l'apparition d'un léger *drift* au décollage. En effet, il arrive que le drone détecte qu'il a pivoté lorsqu'il effectue un décollage alors que ce n'est pas le cas. Afin de remédier à ce problème, nous avons décidé avec l'aide de l'autre groupe de réinitialiser le repère absolu une fois que le drone avait déjà décollé. De cette façon, même si une petite erreur survient au décollage, le drone réinitialise son repère absolu en fonction de sa position une fois qu'il est stable dans les airs.

c) Résultats

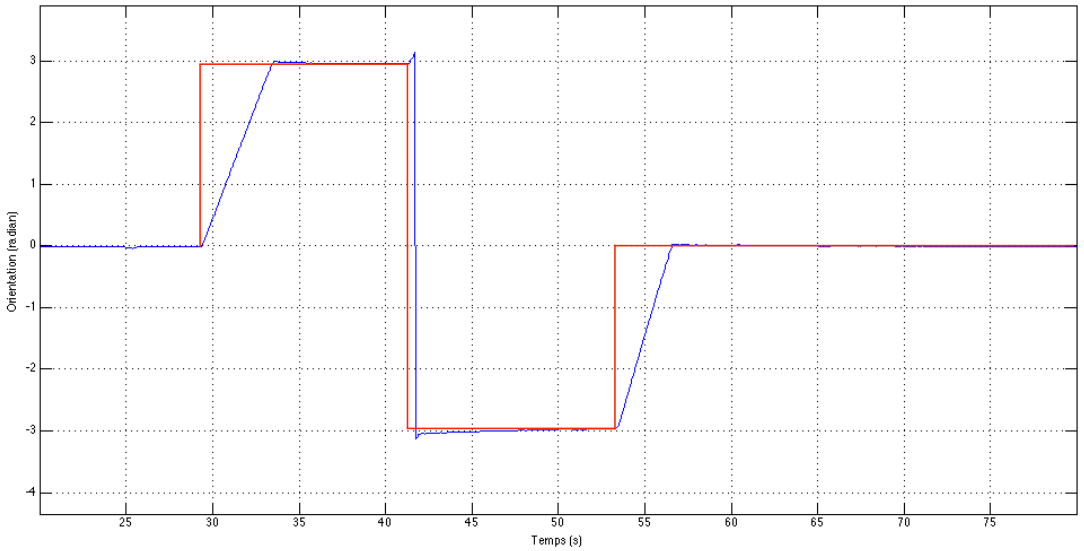


FIGURE 6.15 – Résultats de la réaction du drone lors de l'application de consignes d'orientation.

La FIGURE 6.15 montre la réponse du drone face à une consigne d'orientation. Elle nous permet de distinguer 4 phases successives. La première phase s'étend jusqu'à approximativement 29 secondes et correspond à l'alignement du drone avec le repère absolu après son décollage. Après cette phase d'initialisation, on observe que la consigne demande au drone de pivoter pour s'orienter à un angle de 2.9 radians. Celui-ci va alors mettre 4 secondes pour pivoter et s'aligner avec la consigne. On peut remarquer que la rotation du drone s'effectue à vitesse constante. Cela est dû au fait que le drone tourne à sa valeur maximale imposée à $\frac{\pi}{2}[rad/s]$ par le saturateur comme expliqué à la fin de la section a). De cette façon, on comprend bien pourquoi il faut effectivement un peu moins de 4 secondes à notre drone pour passer de 0 à 2.9 radians.

La troisième phase débute aux alentours des 41 secondes. Cette fois-ci, le drone doit s'orienter à -2.9 radians. On observe bien sur le graphe que le drone utilise le chemin le plus court pour y accéder. En effet, il passe par la position π radians afin de se rendre du 2ème quadrant vers le 3ème. Cela se traduit, sur le schéma, par la montée de la courbe bleue jusqu'à sa nouvelle orientation de référence. La ligne verticale signifiant que l'on a bien changé de quadrant.

Enfin, après s'être aligné dans la bonne direction, le drone reçoit une nouvelle consigne lui demandant de se remettre en phase avec le repère absolu, i.e. s'orienter en rotZ = 0. Une fois de plus, le drone met un peu moins de 4 secondes pour atteindre son objectif et le maintient correctement jusqu'à la fin de l'expérience, 23 secondes plus tard.

Nous pouvons donc conclure que notre régulation en rotZ fonctionne correctement et que notre astuce pour passer rapidement d'une orientation positive à une orientation négative fonctionne bien pour passer du 2ème au 3ème quadrant du cercle trigonométrique.

6.6.4 Régulation en X et Y

Cette dernière sous-section de la régulation est la partie la plus importante. C'est en effet la régulation en X et en Y qui va permettre au drone d'accomplir la mission. Il n'est pas possible d'atteindre la majorité des objectifs lorsque le contrôle dans le plan n'est pas assuré. Pour commencer, nous définirons de manière arbitraire la position du drone dans le plan comme étant les coordonnées de la projection de la caméra ventrale sur le sol.

Le principe de fonctionnement de la régulation en X et Y est plus complexe que les autres. Tout d'abord, il faut se rappeler que la régulation en position se produit dans un repère absolu alors que le drone est commandé dans un repère relatif qui bouge avec lui et dont l'axe X est toujours dirigé dans la même direction que sa caméra frontale et l'axe Y vers la gauche lorsque l'on regarde dans la direction des X positifs. Il faut donc inclure une étape supplémentaire qui tient compte de ce changement de repère.

Outre cette transformation de repère, le contrôle repose sur un concept assez intuitif. L'intelligence artificielle envoie les coordonnées d'un point et le drone s'y rend. Ainsi, le suivi d'une cible ou encore l'exploration de l'environnement correspondent à une succession de points de référence envoyés au régulateur vers lesquels le drone se redirige à chaque nouvelle valeur. Une technique plus élaborée anticipant les mouvements de la cible et régulant alors le drone en vitesse et non plus en position a été envisagée mais n'a pas pu être implémentée et testée par manque de temps. Cependant, nous pouvons voir que les résultats sont très satisfaisants avec la méthode que nous avons utilisée dans la sous-section c).

Afin d'expliquer notre contrôle dans le plan XY, nous expliquerons en premier lieu la fonction de régulation pour l'axe X du drone uniquement. Cet axe sera appelé Xd étant donné qu'il correspond à l'axe X dans le repère relatif du drone (voir sous-section a)). La régulation selon l'axe Yd fonctionne de façon similaire et son fonctionnement ne sera donc pas développé. Puis, nous évoquerons les problèmes rencontrés lors de l'implémentation de cette régulation (b)). Pour finir, nous discuterons des résultats obtenus lors de nos essais (c)).

a) Fonction de régulation dans le plan

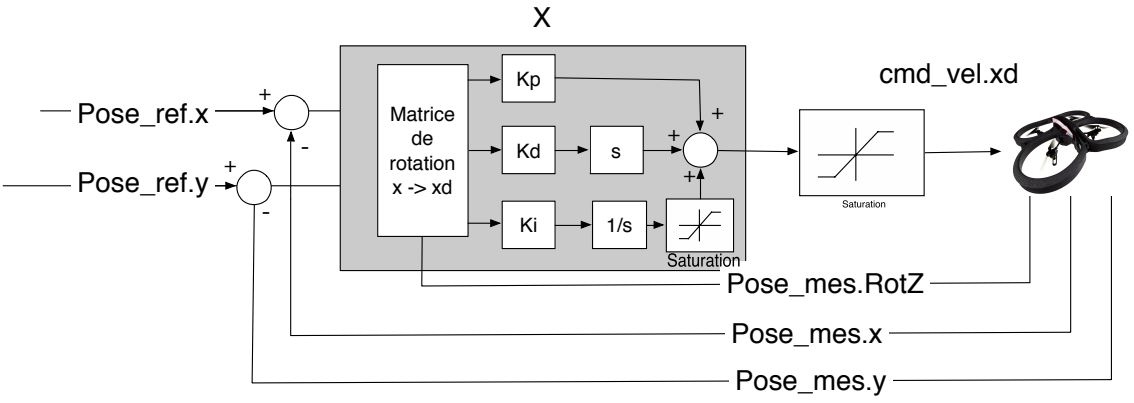


FIGURE 6.16 – Schéma bloc de la régulation en X

La FIGURE 6.16 nous permet de remarquer que le schéma de la régulation en Xd est plus complexe que pour les autres régulations vues précédemment. Premièrement, on remarque que, contrairement aux autres boucles de correction, la régulation en Xd a besoin de connaitre l'état de 3 variables : la position du drone en x, en y et en $RotZ$. Comme annoncé *supra*, ceci s'explique par le fait que les commandes du drones sont relatives et que la régulation est absolue.

Deuxièmement, on peut constater la présence d'une matrice de rotation au début de la boucle de régulation. Celle-ci va permettre de projeter les différences de position selon les axes X et Y sur l'axe Xd du drone. Le terme que l'on régule est dès lors obtenu de la manière suivante :

$$p_term_xd = (delta_x) * cos(Pose_mes.RotZ) - (delta_y) * sin(Pose_mes.RotZ)$$

Cette équation correspond à l'addition des deux premiers graphiques de la FIGURE 6.17 dont le résultat est disponible sur le 3ème graphique de la même figure.

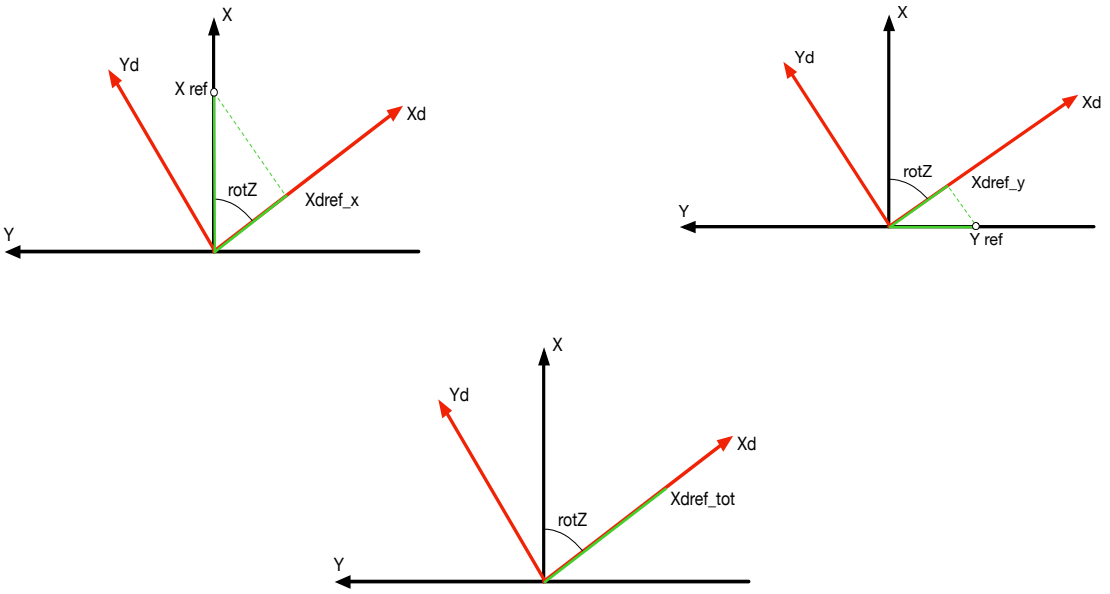


FIGURE 6.17 – Projection orthogonale des différences de position selon les axes absolus sur l'axe relatif x du drone.

Troisièmement, une fois cette valeur obtenue pour réguler le drone selon son axe X, i.e. le faire avancer ou reculer plus ou moins vite, elle doit être régulée afin que le drone atteigne la position en abscisse demandée. Dans cette optique, nous utilisons un régulateur PID. Celui-ci est plus complexe qu'un régulateur PD car il comporte un terme supplémentaire. Ce dernier, le terme intégral, a pour objectif d'éliminer les erreurs statiques, c'est-à-dire un petit *offset* qui ne disparait pas avec le temps. L'ajout de cet intégrateur demande une petite adaptation du code. En effet, contrairement aux autres termes de la régulation (le terme proportionnel et dérivé), le terme intégral a une mémoire. C'est-à-dire que sa valeur est modifiée en ajoutant ou en enlevant du poids à la valeur qu'il avait lors de l'étape précédente de la boucle. Afin d'éviter d'obtenir des termes intégraux trop grands qui créeraient une instabilité du drone, nous avons mis en place une saturation du terme intégral. Ceci permet de maintenir le terme intégral entre deux bornes. De cette façon, ce dernier garde toujours une valeur raisonnable et le système reste stable.

Enfin, la dernière étape visible sur le schéma bloc correspond à un limiteur de commande. Celui-ci permet d'adapter la commande envoyée au drone afin que celui-ci se déplace plus ou moins doucement jusqu'à sa position de référence.

b) Problèmes rencontrés

Dans le cas de la régulation en X et en Y, le régulateur interne ne nous a pas posé de gros ennuis. Le seul problème rencontré provenait des mouvements brusques réalisés par notre drone. En effet, lorsque celui-ci devait atteindre une position éloignée, le régulateur envoyait une commande importante amenant à faire avancer le drone d'une manière

brusque. Un premier à-coup propulsait alors le drone dans la direction souhaitée. Malheureusement, cette impulsion introduisait des erreurs dans la cartographie implémentée par l'autre groupe d'étudiants mémorants. En effet, pour avancer plus vite, le drone se penchait trop fort et la caméra ne filmait donc plus de manière assez perpendiculaire au sol pendant un court instant. Les points remarquables étaient alors perdus et la mise à jour de la carte était faussée. Afin d'éviter ce problème, nous avons réduit la valeur du limiteur de commande de notre boucle de régulation. Après quelques tests et essais-erreurs, nous avons trouvé une bonne limitation qui permettait à l'implémentation de la carte de se faire correctement. En conséquence, notre drone se déplace désormais plus lentement.

c) Résultats

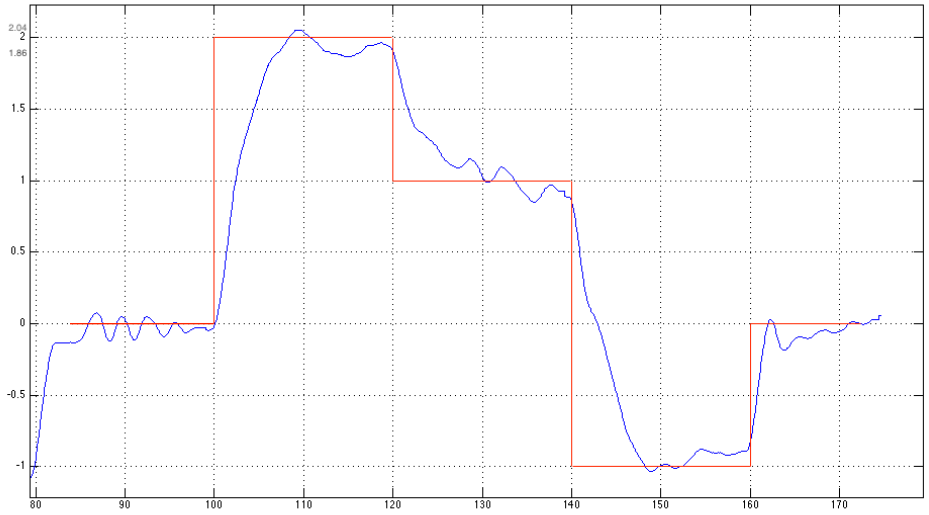


FIGURE 6.18 – Réponse du drone face à une consigne en position dans une direction du plan.

On peut voir à la FIGURE 6.18, que la régulation en X est moins précise que pour les cas précédents. En effet, ce graphique montre que le drone met environ une petite dizaine de secondes afin d'atteindre sa position de référence. Une fois ce point atteint, on constate que le drone oscille pendant une autre dizaine de secondes autour de son objectif avant de se stabiliser. Durant ces oscillations, une erreur de maximum 7% pour une différence de 2 mètres entre deux consignes consécutives a été observée. Nous estimons que cette erreur

est acceptable et qu'elle pourrait être réduite si les courants d'air dans le Hall GC étaient moins forts. Étant donné que la régulation suivant l'axe Y est identique à celle de l'axe X, les résultats ne seront pas illustrés dans ce rapport.

Nous pourrions réduire ces oscillations en augmentant le gain de la partie différentielle de notre régulateur. Cependant, ceci diminuerait encore plus la dynamique du drone. Celui-ci deviendrait alors fort lent pour se rendre d'une position à une autre et ces ralentissements ne permettraient pas à la batterie de tenir assez longtemps que pour explorer l'entièreté de la carte.

Chapitre 7

Validation

Ce chapitre a pour objectif de montrer le bon fonctionnement de la mission. En effet, après avoir détaillé et validé chaque élément de notre code de manière individuelle dans le chapitre précédent, il est maintenant temps de lancer l'entièreté du programme afin de valider notre application. Pour ce faire, nous nous sommes rendus dans la *DroneZone*, une partie du hall GC aménagée afin de pouvoir faire voler des drones. La *DroneZone* a l'avantage d'offrir un grand espace de vol vide de tout obstacle et délimité par des filets qui amortissent le choc des drones, une infrastructure dont nous ne disposions pas dans notre laboratoire situé dans le bâtiment Euler.

Ce chapitre débute avec la description l'environnement dans lequel nous validons notre mission (7.1). Nous présentons ensuite les résultats obtenus lors de notre expérience (7.2). Nous expliquons ensuite quels éléments peuvent nuire au bon fonctionnement de la mission (7.3).

7.1 Environnement de vol

Afin de pouvoir y réaliser la mission, l'espace de vol doit être préparé. Pour ce faire, nous disposons de grandes plaques en carton sur lesquelles nous avons collé des affiches. Celles-ci, disposées au sol, permettent de créer du contraste par terre. En effet, nos drones utilisent principalement leur caméra ventrale afin de connaître leur position. Le sol monotone en béton de la *DroneZone* ne permettait donc pas à nos drones de se repérer de manière satisfaisante et ceux-ci dérivaient très facilement. Les affiches que nous utilisons sont principalement composées d'images simples fortement pixelisées faisant allusion à divers jeux vidéo. Ces images présentent beaucoup de contraste, c'est pourquoi elles ont été utilisées par les étudiants mémorants de l'année précédente. Cependant, notre drone fonctionne avec tous types d'images du moment qu'elles ne sont pas unicolores. Nous avons donc rajouté de nouvelles feuilles cartonnées avec des affiches provenant d'un magazine afin d'augmenter notre zone de vol. Les différents styles d'images utilisés sont présentés à la FIGURE 7.1.

FIGURE 7.1 – Affiches utilisées pour ajouter du contraste au sol

Une fois ces images disposées au sol, celles-ci recouvrent une surface d'environ $9m^2$. Ces dimensions correspondent à la taille de la pièce virtuelle que nos drones explorent (voir FIGURE 6.9). Une photo de cet environnement est disponible à la FIGURE 7.2. On peut y observer les affiches au sol formant un carré et disposées de manière à ne pas laisser d'espaces vides trop grands qui seraient propices à l'introduction d'une erreur dans l'estimation de la position du drone.

FIGURE 7.2 – Photo de la zone de vol dans la DroneZone

Afin d'avoir une cible mobile, nous utilisions, dans un premier temps, une image que nous déplacions à l'aide d'une ficelle. Ceci nous permettait de changer facilement de cible en réimprimant simplement une nouvelle image. Cependant, cette cible "mobile" était peu convaincante et nécessitait quelqu'un pour la déplacer.

Nous avons donc eu l'idée d'acheter une voiture télécommandée. Cette dernière permet d'aller en avant, en arrière, à gauche et à droite. Ceci nous permet donc de pouvoir la positionner où nous voulons tout en continuant à observer le déroulement de la mission

sur nos écrans. Afin que le drone puisse facilement reconnaitre la voiture comme étant la cible lorsqu'il passe dessus, nous avons collé des images sur son capot et sur son toit. En effet, la couleur orange métallisé de notre petite voiture produisait de nombreux reflets qui variaient selon l'éclairage et l'angle de vue du drone. Il était alors très difficile pour celui-ci d'être en mesure de repérer la voiture comme étant la cible pré-enregistrée. Une photo de la voiture après la modification est disponible à la FIGURE 7.3. On peut remarquer que la voiture est recouverte par la répétition d'un même logo. Ce dernier, bicolore, offre de nombreux points de repère facilement détectables par le drone.

FIGURE 7.3 – Photo de la cible mobile

7.2 Résultat de la mission

Cette section a pour objectif de décrire et d'illustrer l'entièreté d'une mission. Afin de rendre nos explications plus claires, nous avons mis une carène verte au drone principal et une carène classique (cfr. FIGURE 2.2) au drone secondaire.

On peut noter sur la FIGURE 7.4 que les drones sont positionnés d'une façon bien précise avant le début de la mission. En effet, il est important que les deux drones décollent à partir du milieu d'un coté de la pièce virtuelle comme cela est arbitrairement renseigné dans l'oracle communiquant avec nos algorithmes d'exploration. De plus, afin que les drones puissent communiquer leur position l'un avec l'autre, nous positionnons le drone secondaire à 50 centimètres à droite du drone principal. De cette manière, nous pouvons introduire un *offset* dans la lecture de la position d'un drone par son coéquipier.

FIGURE 7.4 – Photo des drones à leurs emplacements de départ

Lors du lancement de la mission, le premier drone décolle et commence à explorer la pièce alors que le deuxième drone reste en *standby* à la base, attendant que le premier drone l'appelle. Les photos de la FIGURE 7.5 montrent le drone principal en train d'explorer la zone.

FIGURE 7.5 – Photos du premier drone, explorant l'environnement

Ces photos montrent que le drone effectue bien sa stratégie d'exploration jusqu'à ce qu'il trouve la cible. Une fois cette dernière trouvée, il arrête sa stratégie d'exploration et commence à suivre la voiture en se positionnant au-dessus d'elle. Nous pouvons voir sur la première image de la FIGURE 7.6 que le premier drone est au-dessus de la cible qu'il a détectée. La deuxième image de la même figure illustre l'état de la grille d'exploration au même moment.

FIGURE 7.6 – Photo du drone positionné au dessus de la cible (à gauche) et schéma de la grille d'exploration à l'arrêt de la stratégie d'exploration (à droite).

A partir du moment où la cible est détectée, le drone régule sa position au-dessus de cette dernière. Il la suit alors en restant au dessus d'elle jusqu'à ce que son niveau de batterie atteigne un certain seuil. Une fois au-dessous de ce dernier, le drone principal envoie un signal au drone secondaire lui demandant de venir le remplacer. Afin que cet échange se passe bien, le premier drone monte à une altitude de 2 mètres dans le but d'éviter une collision avec le drone remplaçant, volant à l'altitude de croisière qui est de 1 mètre. Une fois que le drone secondaire est assez proche du drone principal, ce dernier

rentre à la base. Ces différentes étapes sont illustrées par les photos se trouvant à la FIGURE 7.7.

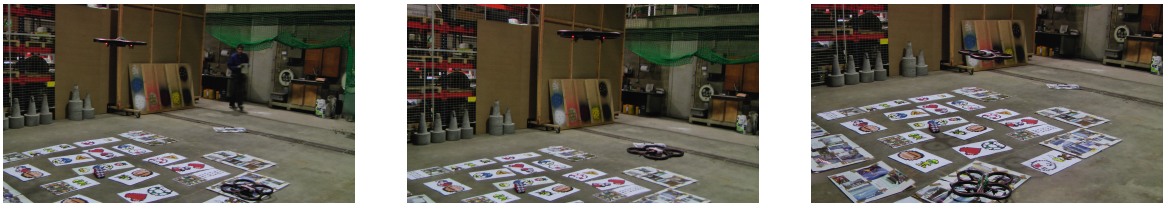


FIGURE 7.7 – Photos de l'opération de transition entre le drone principal et le drone secondaire

Une fois le premier drone rentré à la base et le drone secondaire au dessus de la cible, celui-ci se met directement en stratégie de suivi de cible. Les photos de la FIGURE 7.8 montrent que le drone suit effectivement bien la cible sur toute la longueur de la pièce.

FIGURE 7.8 – Photos du drone secondaire qui se déplace en restant au dessus de la cible mobile.

La FIGURE 7.9 reprend le graphique des différents parcours que les drones et la voiture ont empruntés. On remarque rapidement que l'axe des Y est inversé, i.e. les positifs sont à gauche et les négatifs sont à droite. Cela est dû au fait que les axes sont orientés de la même façon que le repère du drone.

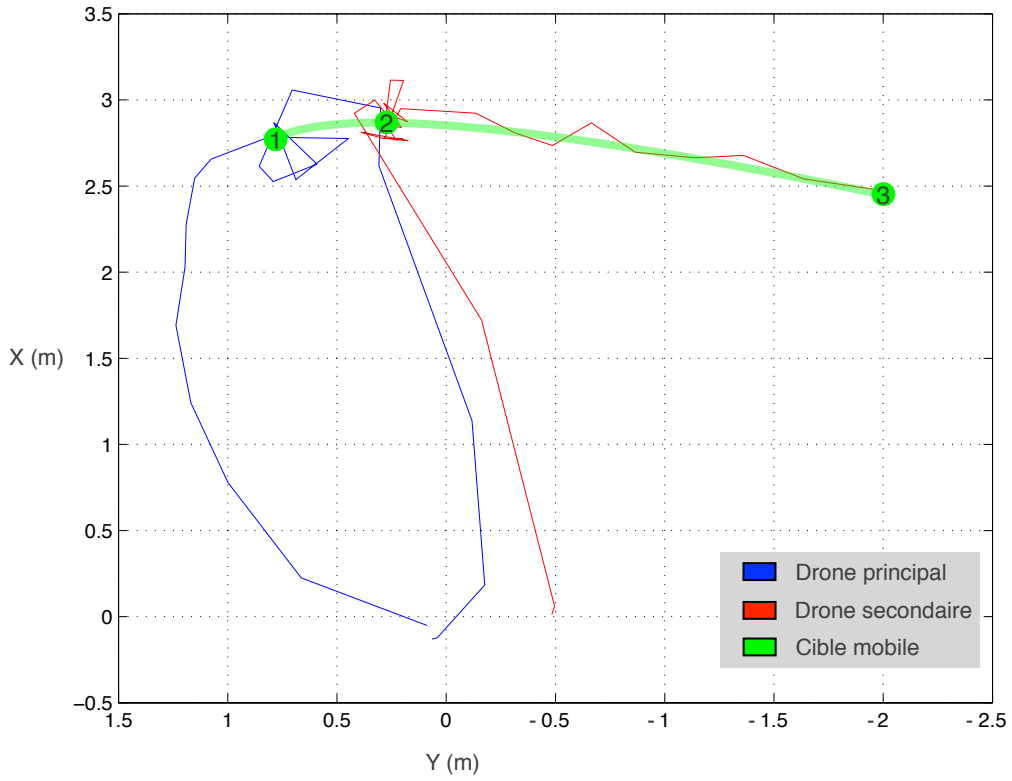


FIGURE 7.9 – Graphe des différentes trajectoires parcourues par les drones et la cible mobile pendant la mission

On peut voir sur ce graphe que le premier drone décolle et explore l'environnement en commençant par le coté gauche. Ceci corrobore bien le schéma de la grille d'exploration de la FIGURE 7.6. Ensuite, on observe que le drone avait fini d'explorer le coté gauche de son terrain virtuel et qu'il commençait à explorer les autres parties lorsqu'il a détecté la cible mobile. Ce lieu de rencontre est schématisé par le point vert numéro 1 sur le graphique. On remarque alors que le drone se stabilise autour de ce point et arrête son exploration. Le drone suit ensuite la voiture qui se dirige vers la droite.

Après environ 50 centimètres de suivi de cible, la batterie du premier drone passe en dessous du seuil critique. Celui-ci appelle alors le deuxième drone et rentre à la base. On voit alors que le drone secondaire se dirige directement vers le point vert numéro 2 d'où le premier drone l'a appelé. Pendant que le deuxième drone se rend sur place, on peut constater que le premier drone est bien retourné à la base. Le drone ayant pour consigne d'atterrir dans les 15 centimètres autour de son point de départ.

Une fois le deuxième drone en vol stationnaire au dessus de la cible, nous avons recommencé à faire avancer la cible mobile vers la droite. On peut observer que le drone secondaire suit très bien la cible et que les deux trajectoires sont ainsi presque confondues.

7.3 Facteurs d'échec de la mission

Lors de nos différents essais, il est arrivé à plusieurs reprises que des imprévus fassent échouer notre mission. Nous avons donc décidé d'améliorer notre code et avons ensuite testé un certain nombre de cas différents. Nous pouvons donc considérer que notre code est assez robuste et répond de manière adéquate aux différents cas de figure auxquels il pourrait devoir faire face. Cependant, parfois l'échec est imprévisible et indépendant de notre implémentation. Cette section a pour objectif de lister les différentes causes d'échec auxquelles nous avons été confrontés durant la réalisation de la mission.

La première erreur, qui est également la plus fréquente, a déjà été introduite dans la section 6.6.3. Il s'agit du drone qui pivote de manière inconsciente. Cette rotation du drone par rapport à son alignement de départ fausse sa position car le drone se déplace comme s'il était toujours bien aligné avec le repère absolu. Même si ce problème n'empêche pas toujours le drone de trouver la cible, il devient impossible pour le drone principal d'envoyer sa position réelle au drone secondaire lors du remplacement. De plus, il devient incapable de rentrer à la base. Malgré un nombre conséquent de tests réalisés, nous n'avons pas trouvé la cause de cette rotation du drone. Que le drone exécute son premier vol de la journée ou son dixième, que la batterie soit pleine ou presque vide, cette erreur survient de manière aléatoire. Nous avons cependant remarqué que certains drones sont beaucoup plus enclins que d'autres à présenter ce mode de défaillance mais aucun des drones que nous avons utilisés n'y a échappé.

La deuxième erreur, qui n'a des conséquences que lorsqu'elle prend une amplitude importante, est la dérive du drone. Étant donné que nous utilisons des drones *low-costs*, il n'est pas étonnant d'observer que les capteurs inertiels ne reflètent pas avec grande précision les mouvements réels du drone. Tant que ces erreurs restent petites, les drones n'ont pas de problèmes à effectuer la mission. En effet, si une erreur minime est introduite dans la position du drone principal, le drone secondaire ira tout de même au bon endroit et tombera probablement sur la cible même si elle n'était pas à l'endroit exact indiqué. Si cette erreur prend de l'ampleur, comme c'est souvent le cas lorsque le premier drone suit la cible mobile, alors le deuxième drone ne parvient jamais à retrouver la cible et la mission est un échec.

Une troisième erreur pouvant entrainer l'échec de la mission est due à la non détection de la cible par un des drones. En effet, il arrive souvent qu'un drone passe au dessus de la cible sans pour autant la détecter. Ce phénomène peut s'expliquer de deux manières différentes. La première est que, pour le moment, le drone ne rafraichit l'image de sa caméra ventrale qu'une fois toutes les deux secondes. Lorsque le drone est en phase d'exploration, il est dès lors possible que celui-ci passe au dessus de la voiture sans pour autant rafraichir son image. La cible n'est donc pas détectée et le drone continue son exploration. Ce problème peut également s'expliquer par le fait que, même s'il la voit, le drone ne reconnait pas toujours la cible. Cela peut être dû à une trop grande variation de l'altitude entrainant

le fait que la cible pré-enregistrée ne ressemble pas à ce que le drone voit réellement ou alors au fait que le drone crée une ombre sur la cible dont l'apparence est alors altérée. Une solution possible à cette troisième erreur serait d'augmenter le nombre d'images analysées par seconde et d'améliorer la qualité de la caméra ventrale du drone (*cfr.* section 8.1).

Chapitre 8

Amélioration du matériel

8.1 Ajout d'une caméra

Au cours du premier quadrimestre, nous nous sommes rendus compte que la caméra ventrale du drone avait une résolution insuffisante pour pouvoir obtenir des informations suffisamment intéressantes lors du traitement d'images. Afin d'obtenir de meilleurs résultats, nous avons décidé de modifier le drone et de lui fixer une nouvelle caméra ventrale de meilleure qualité. Après quelques recherches effectuées en collaboration avec l'autre groupe d'étudiants mémorants, nous en sommes arrivés à devoir faire un choix entre quatre options différentes qui nous semblaient toutes envisageables. Ces options étaient :

- Ajouter une caméra IP-adressable par wifi.
- Ajouter une caméra compatible avec un nouveau module capable de communiquer avec l'ordinateur.
- Utiliser une caméra miniature pour laquelle il existe déjà un driver sous forme de noeud ROS.
- Rajouter une nouvelle carte mère Parrot et y brancher une caméra frontale pointant vers le bas.

La dernière option a été choisie. Celle-ci était plus économique et plus sûre. En effet, nous savions avec certitude que cette caméra fonctionnait avec ROS et qu'il était facile de communiquer avec elle. Dans le pire des cas, cet achat aurait toujours pu servir comme pièce de rechange au cas où une de ces pièces casserait sur un des drones que nous utilisons.
Rajouter une caméra frontale reliée à une carte mère revient à créer un drone virtuel qui n'a ni moteurs ni capteurs et que l'on viendrait coller sur un drone normal. Ainsi, nous pouvons accéder aux données de la caméra virtuelle comme nous le ferions sur un drone normal et ensuite donner nos instructions au drone réel.

Un tableau comparatif des différentes options citées *supra* est disponible à l'Annexe A.

Afin de permettre le bon fonctionnement de ce drone virtuel, il a fallu dédoubler la

connexion de la batterie afin que les deux cartes mères puissent être alimentées. Même si une seule batterie alimente maintenant 2 drones, nous n'avons pas remarqué d'usure plus rapide de la batterie. En effet, le drone virtuel ne possédant que la carte mère reprenant la carte réseau et la caméra frontale, celle-ci est très peu énergivore. Cependant, le drone éprouve plus de mal à décoller à cause de la masse rajoutée. Effectivement, lorsqu'il est équipé de sa carène de vol intérieur, il arrive très fréquemment que le drone ne parvienne pas à s'élever du sol lorsqu'on lui rajoute le poids du drone virtuel. La plupart des tests réalisés par l'autre équipe d'étudiants ont donc été faits sans l'utilisation des moteurs en déplaçant le drone manuellement par rapport au sol en le fixant à une altitude donnée.

8.2 Un routeur pour l'utilisation simultanée de plusieurs drones

Les AR.Drones 2.0 ne sont pas faits pour être utilisés simultanément à partir d'un seul appareil. Afin d'établir une connexion avec un ordinateur ou un smartphone, les drones émettent un réseau *wifi* auquel un appareil peut se connecter. Le drone prend alors le rôle d'hôte du réseau.

Dans le cadre de notre mission, cela n'est pas acceptable. Comme expliqué dans la section 6.3, notre implémentation de la mission compte sur un réseau de drones où tous sont interconnectés. Pour arriver à cela, nous avons utilisé la solution de Nicolas Rowier décrite dans [36]. Comme cette solution n'est pas le résultat de notre travail, nous n'allons décrire que brièvement son fonctionnement.

L'utilisateur doit utiliser un routeur branché à son ordinateur permettant de créer un réseau auquel tous les drones se connecteront. Pour ce faire, un script permet de se connecter à chacun des drones et de leur attribuer une adresse IP qui permet de rendre les drones clients du réseau et non plus hôtes. Ainsi, tous les drones seront connectés à un même réseau, créé par le routeur, et pourront communiquer entre eux par l'intermédiaire de celui-ci.

Afin d'effectuer convenablement cette procédure, il faut au préalable suivre une démarche consistant à récupérer des informations sur les drones (leur adresse MAC) et à les enregistrer dans le routeur. Ensuite, il faut leur attribuer une adresse IP définitive choisie par l'utilisateur selon certaines contraintes. Cette démarche doit être effectuée une seule fois par drone sur le routeur et n'entraine aucune modification du *software* ou du *hardware* des drones. Le lecteur attentif pourrait penser que cette façon de connecter les drones va à l'encontre de la section 6.3, précisant que les drones sont directement connectés pour communiquer entre eux et qu'il n'y a pas de *master*. Ce n'est en fait pas le cas. En effet, le routeur ne sert que de relais de communication entre les drones et n'agit pas comme un "calculateur central". Il sert simplement à relayer l'information. L'implémentation de

notre application est totalement indépendante au fait que toutes les informations transitent par un routeur et fonctionnerait également dans un cadre où tous les drones seraient connectés les uns aux autres directement sans effectuer de changement.

Chapitre 9

Conclusion

Les objectifs de ce mémoire étaient d'implémenter des AR.Drones 2.0 de Parrot avec ROS afin qu'ils puissent explorer un environnement *indoor* et suivre une cible mobile sur base de l'estimation de leur position et que plusieurs drones puissent communiquer des informations entre eux. Pour valider le fonctionnement de tous ces objectifs, une mission dont nous avons choisi les spécificités a été réalisée avec succès.

Nous tenons à souligner le fait que le passage à ROS nous a contraint à recommencer une grosse partie du travail déjà effectué les années précédentes, mais que ce passage était nécessaire et qu'il s'est révélé être très utile pour la conception de notre programme. ROS est en effet une plate-forme offrant de nombreuses possibilités et répondant aux besoins de notre application d'une manière très satisfaisante.

Étant donné que notre mémoire s'inscrit dans un travail continu, repris par différents étudiants d'année en année, un autre de nos objectifs est d'assurer la transition à l'année suivante. Dans cette optique, un mode d'emploi de lancement du code est déjà fourni dans notre dossier *git*, que nous transmettrons à nos successeurs. Nous allons également mettre notre *package* en ligne et sa documentation sur le site de ROS, de sorte à ce que chacun puisse l'installer et l'utiliser sur ses AR.Drones 2.0.

Nous sommes heureux d'avoir pu réaliser notre mémoire avec ces drones et d'avoir développé cette application dont nous sommes très satisfait des résultats. Pour conclure, nous allons aborder trois dernières sections. La première reprend les difficultés liées à ce mémoire, qui resteront sans doutes les mêmes pour nos successeurs. La deuxième présente les pistes que ceux-ci pourraient aborder pour améliorer notre application. Enfin, dans la troisième, nous donnons notre avis et nos conseils quand à l'objectif final de la continuité des mémoires sur les drones à l'EPL, qui est la réalisation d'une application de cartographie à l'aide de plusieurs drones communiquant ensemble où ceux-ci sont autonomes.

9.1 Difficultés liées à ce mémoire

La principale difficulté rencontrée durant la réalisation de ce mémoire est liée à l'utilisation des AR.Drones 2.0. En effet, ces drones ne sont pas d'une grande qualité. Leur caméra ventrale par exemple n'a qu'une résolution de $320 * 240$ pixels. Cela implique que les drones doivent voler relativement près du sol pour pouvoir détecter la cible. Comme l'angle de vue de la caméra n'est que de 64°, le champ de vision du drone est par conséquent très réduit. Les accéléromètres quant à eux ne sont pas d'une excellente qualité non plus. En effet, lorsque le drone est à l'arrêt au sol et que ses caméras sont masquées, le drone croit généralement se déplacer à une vitesse pouvant approcher 1 mètre par minute dans une direction aléatoire. Quant au sonar, celui-ci effectue une erreur de mesure de 5 à 10% entre l'altitude réelle et l'altitude mesurée (voir [12] pour de plus amples informations). Trouver des solutions pour contourner ces problèmes prend toujours du temps, qui pourrait être passé à travailler sur d'autres points de l'application. On peut cependant souligner le fait que l'utilisation de ces drones présente également des avantages. En effet, les pièces sont facilement remplaçables et bon marché (il nous est arrivé plusieurs fois de devoir remplacés des pièces qui étaient cassées) et puisqu'ils sont sur le marché depuis longtemps, il existe beaucoup de documentation en vue de leur utilisation. De plus, un *driver* ROS est déjà existant.

Une autre difficulté notable liée à l'utilisation de ces drones est l'inaccessibilité du code interne de Parrot. Celui-ci gère les opérations de base, comme le décollage ou l'atterrissage, effectue toutes sortes de régulations sur la position et le mouvement des hélices du drone et adapte les mesures de l'altitude au profil détecté. Cela engendre des comportements imprévisibles et nous oblige à créer un régulateur qui englobe un système de contrôle embarqué dont les actions nous sont inconnues. Ces comportements imprévisibles entrainent notamment des décollages ratés, durant lesquels le drone peut se décaler d'un mètre par rapport à sa position d'origine. Ils peuvent aussi entrainer le fait que le drone ignore les commandes qui lui sont données pendant un certain temps ce qui peut amener le drone à monter au plafond.

9.2 Améliorations possibles

Même si notre application se déroule bien dans la majorité des cas, nous avons pensé à quelques pistes d'amélioration que les étudiants des prochaines années pourraient rajouter à notre implémentation. Certaines d'entre elles sont présentées dans cette section.

9.2.1 Implémentation d'un calcul de trajectoire

Dans notre méthode d'exploration, lorsque le drone se déplace d'un point à un autre, il le fait toujours en ligne droite. Ceci implique que la forme d'une pièce explorée par le drone doit toujours être convexe et qu'il ne peut y avoir d'obstacle dans celle-ci pour que le drone puisse l'explorer, sous peine d'entrer en collision avec un mur ou un de ces

obstacles. Une méthode pour résoudre ce problème est l'implémentation d'un algorithme, par exemple A*, permettant au drone de trouver le plus court chemin praticable entre deux points.

9.2.2 Remplacement de l'oracle par une fonction de détection des murs

Pour l'instant, la forme de l'espace exploré par le drone est définie par un oracle interne au programme. Une très bonne amélioration pour notre application serait d'implémenter une fonction de détection des obstacles. Cependant, l'implémentation de celle-ci se révélerait compliquée avec les AR.Drone 2.0. En effet, deux solutions sont possibles.

La première est d'utiliser la caméra frontale pour détecter les obstacles. Cela implique de toujours orienter le drone dans la direction vers laquelle il se déplace et d'utiliser des fonctions de reconnaissance d'images potentiellement compliquées et peu fiables pour la localisation de l'obstacle. Cette solution implique également de modifier le *hardware* du drone, celui-ci étant dans l'incapacité d'envoyer ses deux flux vidéos simultanément.

La seconde est d'utiliser la caméra ventrale afin de détecter le bas des murs (les plinthes peuvent être reconnaissables) et les obstacles. Cependant, comme la qualité de cette caméra est très faible et que le drone doit voler bas, il ne voit pas loin et il serait obligé de voler très lentement pour ne pas entrer en collision avec un mur avant qu'il ne le détecte. Avec notre oracle, le drone ne sortait jamais de la zone autorisée. Ceci s'explique par la capacité du drone à détecter immédiatement un mur au moment où celui-ci rentre dans son champ de vision, ce qui ne serait pas le cas avec une fonction de reconnaissance d'images.

Afin d'aider les générations futures à réaliser cette piste d'amélioration, notre implémentation de l'oracle a été faite de manière à facilement pouvoir être remplacée par un autre module de détection d'obstacles sans nuire au reste du code.

9.2.3 Implémentation d'un algorithme d'exploration multi-drones

Une autre amélioration possible serait de modifier l'algorithme d'exploration afin de profiter du fait que plusieurs drones peuvent chercher la cible en même temps. Ceci impliquerait d'établir une encore plus grande collaboration entre les drones que celle réalisée dans ce mémoire, mais cela est aisément réalisable grâce à ROS. La plus grande difficulté dans ce cas-ci est que lorsque plusieurs AR.Drones 2.0 volent simultanément, ils se perturbent très fort l'un l'autre à cause du vent créé par leurs hélices. Dans notre mission, les deux drones volent simultanément seulement pendant un laps de temps très court ce qui ne provoque pas trop de problèmes, ce serait différent dans le cas de l'amélioration proposée ici.

9.3 Notre avis sur la réalisation d'une application de cartographie multi-drones autonomes

Tout au long de cette année, nous avons été surpris de voir que peu de chercheurs semblent se pencher sur l'implémentation d'une réelle application multi-drones autonomes avec création de carte sur base de l'intégration du flux vidéo et des capteurs inertiels. Comme dit dans la section 3 de ce rapport, la plupart des drones ne sont pas autonomes et dépendent de capteurs extérieurs. Nous avons donc décidé de clôturer notre mémoire en nous posant la question de savoir si une mission à plusieurs drones totalement autonomes pouvait être réalisée avec un taux d'échec quasiment nul ?

De notre point de vue, ce genre d'application est réalisable. D'ailleurs, la mission que nous avons implémentée, accompagnée du programme de cartographie de l'autre groupe d'étudiants mémorants, se rapproche fortement de cet objectif. En effet, nous avons établi une stratégie et une communication multi-drones ainsi qu'un système de traitement d'images et une analyse de la pose basée sur le flux vidéo.

En théorie, des capteurs inertiels et un flux visuel sont suffisants pour que des drones puissent se repérer. Cependant, pour avoir une application qui aurait un taux élevé de réussite, il semblerait qu'un très bon matériel soit nécessaire. En effet, toute odométrie intègre au fur et à mesure une erreur et doit donc être recalibrée. Plus la qualité des capteurs diminue, plus la fréquence de recalibration doit être élevée. Or, dans ce type d'application, il est impossible de recalibrer un drone pendant de sa mission. De plus, afin de pouvoir cartographier son environnement, le drone a besoin d'une bonne caméra et d'un traitement d'images qui est assez performant que pour savoir si ce que le drone observe a déjà été enregistré ou représente un nouvel échantillon.

Dans le cadre de notre mémoire, les capteurs inertiels et le flux vidéo ne sont pas d'une qualité suffisante que pour permettre aux drones de connaître précisément leur position dans le temps. Il arrive même parfois que les drones dévient lors du décollage. Un changement drastique mais essentiel pour que l'EPL puisse réaliser de bonnes applications dans ce domaine serait l'investissement dans de nouveaux drones. Cela impliquerait un gros investissement financier mais également en programmation car il faudrait recréer toute une architecture pour ce nouveau type de drones. De plus, il n'est pas certain qu'un *package* semblable à *ardrone_autonomy* existe déjà pour un drone capable d'effectuer ce genre de tâches.

En conclusion, il est très facile de programmer un drone pour lui faire réaliser une série d'objectifs tant complexes que variés. Aussi, nous avons vu que certains *frameworks* existent afin de faciliter la communication entre les drones. Cependant, même avec un code parfait, le facteur limitant reste surtout la partie *hardware* des drones. Nous sommes cependant convaincus que, d'ici quelques années, les drones auront fortement évolué et qu'il sera possible de s'en procurer de très performants à un prix très démocratique. Ceux-

ci seront alors capables d'être utilisés à des fins que nous sommes loin d'imaginer pour le moment.

Annexes

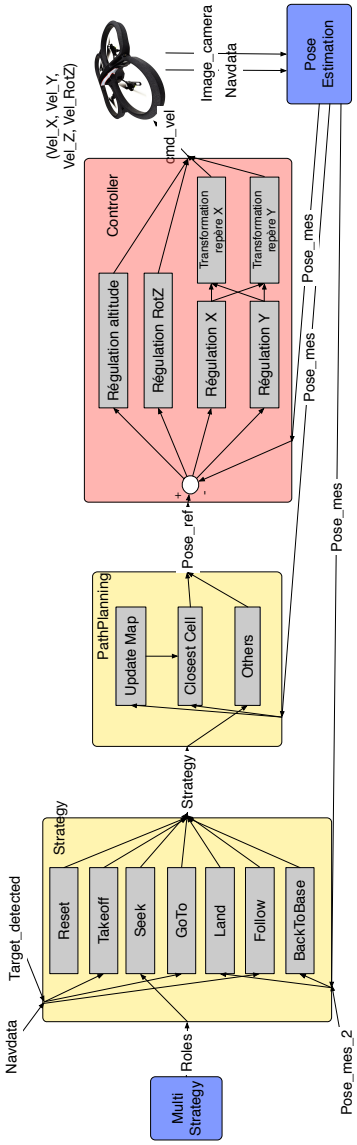
Annexe A

Tableau comparatif des caméras

	Option 1	Option 2	Option 3	Option 4
	Camera Axis	Camera « type GoPro »	Via carte PixHawk	2ème carte AR.Drone
Résolution	HDTV 720p/1MP quality	[640x480 : 4K]	752*480	720p
Modèle	AXIS M1004-W	Spycam, Arlo, Relee RL102, HMNC500, SJCAM, Gopro	Aptina MT9V032	Caméra Frontale AR.Drone 2.0
Matériel Requis	Code ROS disponible sur GitHub et un système d'attache au drone.	Système d'attache au drone.	PixHawk et « Odroïde C1 »	-Carte mère pour AR.Drone 2.0 -Structure inférieure AR.Drone 2.0
Complexité	- Poids et taille assez conséquents (110g + batterie + système d'attache). - Code de Tum à adapter par rapport à la nouvelle caméra	-Hacker la caméra pour la rendre client et non hôte du réseau. -Savoir récupérer les données envoyées sous ROS. -Bande passante inconnue. -Poids peut être conséquent en fonction de la caméra.	-Très peu de documentation. -Fonctionnement assez flou.	
Prix	+- 225 €	De 70$ à 300€	+- 250 €	148,99 €

Annexe B

Structure du code



Bibliographie

[1] The Drone Company a Liège. Video for porsche. http://www.thedronecompany.be/fr/, 2016. [web ; consulté le 5-16-2016].

[2] Markus Achtelik, Stephan Weiss, and Simon Lynen. Ethzasl ptam. http://wiki.ros.org/ethzasl_ptam, 2015. [web ; consulté le 05 - 22 - 2016].

[3] Mubarak Shah Afshin Dehghan, ... Keynote : Automatic detection and tracking of pedestrians in videos with various crowd densities. 2011.

[4] AirDog. The world's first auto-follow drone for action sports. https://www.airdog.com/meet-airdog/, 2016. [web ; consulté le 6-16-2016].

[5] Henry Bradlow. The lily camera developped using ros. http://www.ros.org/news/2015/05/the-lily-camera-developed-using-ros.html, 2015. [web ; consulté le 05 - 22 - 2016].

[6] Jean-Charles CATTEAU. Drones et agriculture biologique : un mariage d'avenir. https://ecotrophologie.com/2014/11/22/drones-et-agriculture-biologique-un-mariage-davenir/, 2014. [web ; consulté le 05 - 20 - 2016].

[7] Jesus S. Cepeda, Luiz Chaimowicz, Rogelio Soto, José L. Gordillo, Edén A. Alanis-Reyes, and Luis C. Carrillo-Arce. A behaviour-based strategy for single and multi-robot autonomous exploration. 2012.

[8] Carnets d'avenir. Perspectives du marché des drones civils. http://carnets-davenir.com/perspectives-du-marche-des-drones-civils/, 2016. [web ; consulté le 05 - 20 - 2016].

[9] de Radigues Brieuc et Van Hijfte Florent. Simultaneous localisation and mapping (slam) and investigation of multi-drone capacities. *EPL TFE*, Juin 2015.

[10] Stephane Piskorski Nicolas Brulez Pierre Eline Frederic D'Haeyer. *AR.Drone Developer Guide*. Parrot, 2012.

[11] Thomas Edlinger and Owald von Puttkamer. Exploration of an indoor environment by an autonomous mobile robot. September 1994.

[12] Nicolas Beghin et Thibault Martin. Localisation et cartographie simultanees (slam) par un quadricoptere ar.drone autonome. *EPL TFE*, Juin 2013.

[13] Guard from above BV. A low tech solution for a high tech problem. http://guardfromabove.com/, 2016. [web ; consulté le 05 - 20 - 2016].

[14] Jean Herman. Tfe 2013-2014. *EPL TFE*, Juin 2014.

[15] Kashmir Hill. Joggobot, the companion drone that makes you run faster, longer, harder. http://www.forbes.com/sites/kashmirhill/2012/06/07/joggobot-the-companion-drone-that-makes-you-run-faster-longer-harder/#65f57230480e, 2012. [web ; consulté le 05 - 21 - 2016].

[16] Emma Hutchings. Domino's tests flying pizza drone [video]. http://www.psfk.com/2013/06/dominos-pizza-delivery-drone.html, 2013. [web ; consulté le 05 - 20 - 2016].

[17] irlock. Ir-lock. www.irlock.com, 2016. [web ; consulté le 05 - 21 - 2016].

[18] Daniel Cremers Jakob Engel, Jürgen Sturm. Autonomous navigation of a camera-based quadrocopter. 2011.

[19] Daniel Cremers Jakob Engel, Jürgen Sturm. Accurate figure flying with a quadrocopter using onboard visual and inertial sensing. 2012.

[20] Daniel Cremers Jakob Engel, Jürgen Sturm. Camera-based navigation of a low-cost quadrocopter. 2012.

[21] Daniel Cremers Jakob Engel, Jürgen Sturm. Scale-aware navigation of a low-cost quadrocopter with a monocular camera. 2014.

[22] Daniel Cremers Jürgen Sturm and Chrstian Kerl. Autonomous navigation for flying robots. https://www.edx.org/course/autonomous-navigation-flying-robots-tumx-autonavx-0, 2015.

[23] kbogert. cmdvel is acceleration in x and y, velocity in z's 116. https://github.com/AutonomyLab/ardrone_autonomy/issues/116, 2014. [web ; consulté le 05 - 26 - 2016].

[24] Georg Klein and David Murray. Parallel tracking and mapping for small ar workspaces. 2007.

[25] Aurélien Deligne Le HuffPost. VidÉo. oubliez les feux d'artifice, place aux spectacles de drones. http://www.huffingtonpost.fr/2016/05/02/spectacle-drones-japon-mon-fuji-insolite_n_9819440.html, 2016. [web ; consulté le 05 - 20 - 2016].

[26] Bird MURI. Learning monocular reactive uav control in cluttered natural environments. https://www.youtube.com/watch?v=hNsP6-K3Hn4, 2012. [web ; consulté le 05 - 22 - 16].

[27] OpenCV. Opencv. www.OpenCV.org, 2016. [web ; consulté le 05 - 19 - 2016].

[28] Mani Monajjemi & others. ardrone autonomy. http://ardrone-autonomy.readthedocs.io/en/latest/, 2014. [web ; consulté le 05 - 19 - 2016].

[29] Parrot. Les chiffres clés du drone en une infographie. http://blog.parrot.com/2015/08/28/chiffres-cles-drone-infographie/, August 2015. [web ; consulté le 05 - 29 - 2016].

[30] pcl. pcl official website. http://www.pointclouds.org/, 2016. [web ; consulté le 05 - 23 - 2016].

[31] Lily Robotics. Site officiel de la caméra lily. https://www.lily.camera/, 2015. [web ; consulté le 05 - 22 - 2016].

[32] ROS. Ros community metrics report. Juillet 2015.

[33] ROS. Is ros for me ? www.ROS.org/is-ros-for-me/, 2016. [web ; consulté le 05 - 18 - 2016].

[34] ROS. Ros. www.ROS.org, 2016. [web ; consulté le 05 - 18 - 2016].

[35] Stéphane Ross, Narek Melik-Barkhudarov Kumar, Shaurya Shankar, Andreas Wendel, Debadeepta Dey, J. Andrew Bagnell, and Martial Hebert. Learning monocular reactive uav control in cluttered natural environments. 2012.

[36] Nicolas Rovier. Interconnexion de multiples drones "ar parrot". December 2014.

[37] RTE. Les hélicoptères du réseau à haute tension, 2016.

[38] Nathan Michael Shaojie Shen and Vijay Kumar. Autonomous aerial navigation in confined indoor environment. https://www.youtube.com/watch?v=IMSozUpFFkU, 2010. [web ; consulté le 13 - 06 - 2016].

[39] Nathan Michael Shaojie Shen and Vijay Kumar. Autonomous multi-floor indoor navigation with a computationally constrained mav. May 2011.

[40] Klaus Kursawe Susanne Albers and Sven Schuierer. Exploring unknown environments with obstacles. 1999.

[41] Wikipedia. Algorithme a*. https://fr.wikipedia.org/wiki/Algorithme_A*, 2016. [web ; consulté le 13 - 06 - 2016].

[42] Wikipedia. Algorithme de dijkstra. https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra, 2016. [web ; consulté le 13 - 06 - 2016].

[43] Wikipedia. Histogramme de gradient orienté. https://fr.wikipedia.org/wiki/Histogramme_de_gradient_orient%C3%A9, 2016. [web ; consulté le 05 - 21 - 2016].

[44] William. Hexo+, le drone autonome capable de filmer tous vos exploits. http://hitek.fr/actualite/hexo-plus-drone-autonome-filmer_3042, 2014. [web ; consulté le 05 - 21 - 2016].

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**