CentraleSupélec

Projet double de deuxième année

Asservissement "rigide" de drone

Auteurs :
M. Hugo Levy–Falk
M^lle Joanne Steiner

Encadrants :
M Jean-Louis Gutzwiller
M. Hervé Frezza-Buet

Version du 7 juin 2019

Résumé

Dans ce rapport, nous détaillons la mise en place d'un asservissement en position d'un drone BeBop en utilisant le flux optique qu'il fournit. Pour cela nous utilisons diverses corrections (PI, PD, PID) et le framework RobotOS.

Table des matières

Chapitre 1

Introduction

Le campus de Metz de CentraleSupélec dispose de drones quadricoptères Bebop 2 (Parrot). Ces derniers sont capables de réaliser des mouvements brusques ce qui rend leur pilotage complexe. Il faut donc les manipuler avec précautions, ce qui revient à sous-exploiter leurs capacités.

Jusqu'à présent, le drone était asservi à l'aide d'une cible présente dans son champ visuel. Le drone suivait la cible (bleue) et se positionnait en face de cette dernière. Toutefois, le drone se déplaçait lentement et se montrait prudent. De plus, une fois face à la cible, le drone n'était pas stable. Il oscillait verticalement face à la cible. Ce résultat avait été obtenu suite à un projet d'élèves.

L'objectif de ce projet est donc de rendre l'asservissement du drone plus "rigide" afin de mieux exploiter les capacités de ce dernier.

Chapitre 2

Modélisation du problème

2.1 Modélisation mécanique

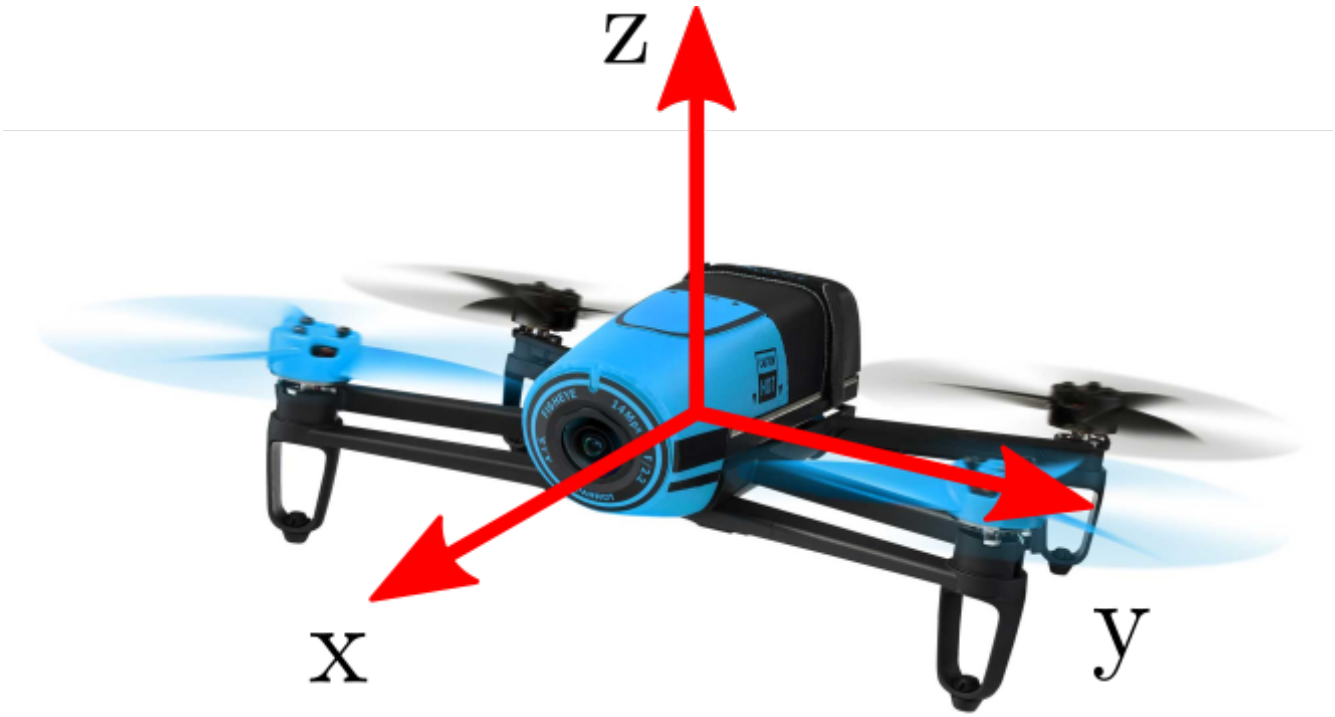

FIGURE 2.1 – Axes du drone parrot [3]

Le contrôle du drone selon les différents axes (figure 2.1) n'est pas identique. Ainsi on contrôle le drone en vitesse pour la translation et la rotation sur l'axe z, alors que le contrôle sur les axes x et y s'effectue par l'inclinaison de la machine.

2.1.1 Axe z

Les drones BeBop nous permettent de spécifier directement une vitesse en rotation et en translation sur l'axe z. L'asservissement interne du drone se chargeant de faire respecter la consigne. On a donc une une réponse fréquencielle de la forme de celle de la figure 2.2. On peut alors assimiler la fonction de transfert sur cet axe à un simple intégrateur dans notre domaine fréquentiel de travail. Pour la correction, on pourra utiliser un simple PID.

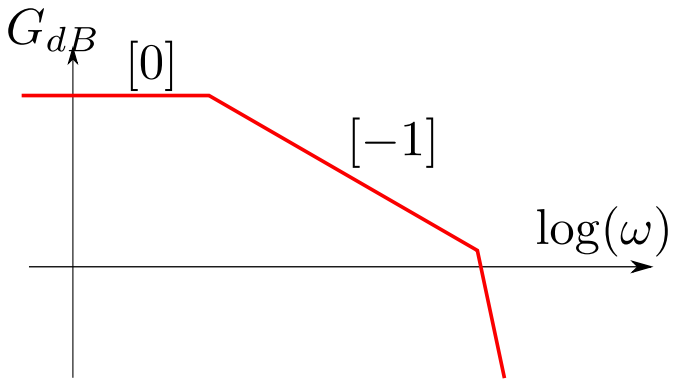

FIGURE 2.2 – Diagramme de Bode simplifié de la réponse sur l'axe z

2.1.2 Axes x et y

Les axes x et y ayant des comportements similaires, nous pouvons les assimiler à un axe u pour les calculs. On suppose que les moteurs du drone fournissent une force de portance constante $\vec{F_m}$ et que le drone doit faire face à une force de frottements fluides $\vec{F_f} = -\alpha\dot{u}$.

En appliquant le principe fondamental de la dynamique dans le cadre des forces appliquées dans le schéma de la figure 2.3, on a :

$$m\ddot{u} = \sin(\theta)F_m - \alpha\dot{u} \tag{2.1}$$

Dans le domaine de Laplace, et en se plaçant dans le domaine des faibles inclinaisons, on a donc :

$$mp^2u = \theta F_m - \alpha p u \tag{2.2}$$

$$u = \frac{F_m}{\alpha p(\frac{m}{\alpha}p + 1)}\theta \tag{2.3}$$

Ce qui donne un diagramme de Bode similaire à celui de la figure 2.4.

Malheureusement, nous ne connaissons pas le paramètre α. Cependant, d'après M. Frezza, à pleine vitesse, le drone met plusieurs seconde à atteindre sa vitesse limite (correspondant à la fréquence à laquelle la fonction de transfert peut être assimilée à un simple intégrateur). On peut donc supposer que pour les fréquences de travail du cadre de ce projet, les axes x et y se comportent comme des doubles intégrateurs. C'est pourquoi le correcteur qui semble adapté pour ces deux axes est un correcteur Proportionnel Dérivé.

2.2 Modélisation primaire de l'asservissement

On peut dans un premier temps modéliser l'asservissement de la manière de la figure 2.5.

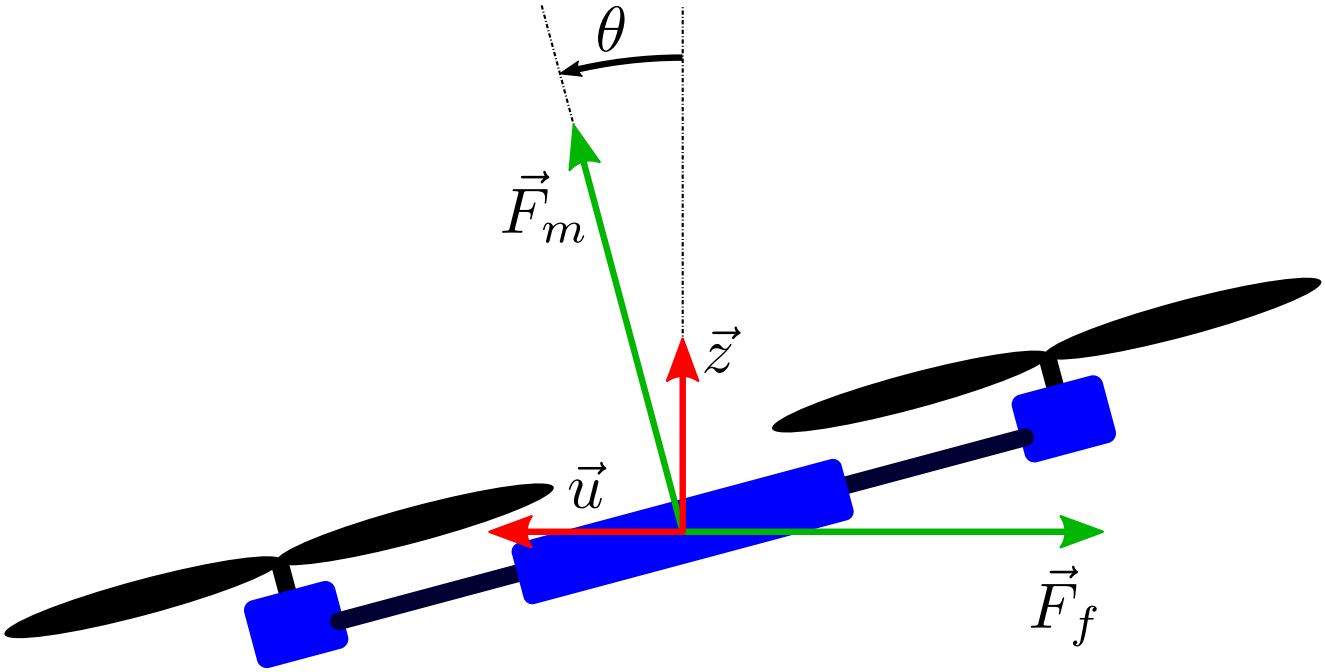

FIGURE 2.3 – Modélisation des forces appliquées au drone dans le cas d'une translation dans le sens positif de $\vec{u}$

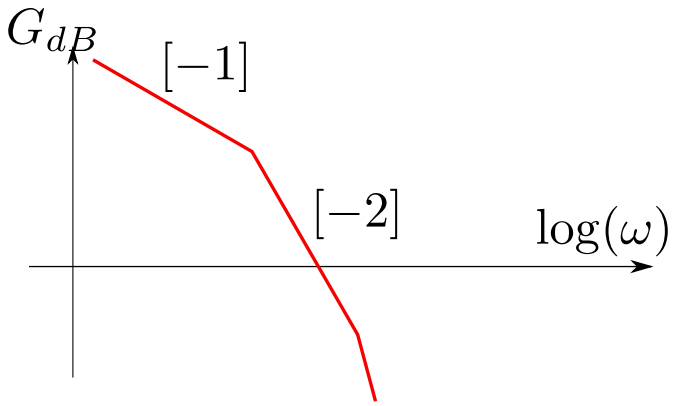

FIGURE 2.4 – Diagramme de Bode simplifié de la réponse sur l'axe u

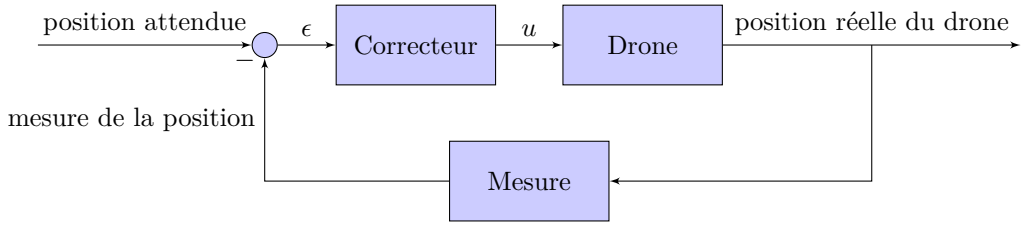

FIGURE 2.5 – Modélisation simple de l'asservissement

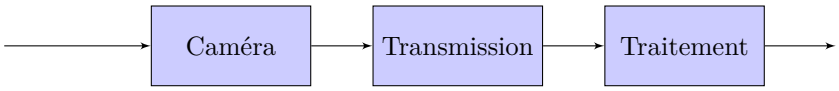

FIGURE 2.6 – Décomposition de la mesure

Le drone diffuse un flux optique, qui permet de calculer sa position vis-à-vis d'une cible. La mesure peut se décomposer en trois éléments, comme le montre la figure 2.6.

Comme le traitement ne se fait pas directement sur le drone, il existe des retards aléatoires dûs au temps de transit sur le réseau, ainsi qu'au temps de calcul sur le poste de contrôle. Ces retards limitent la fréquence des mesures à environ 20Hz[7].

Chapitre 3

Réalisation du projet

3.1 Prise en main de ROS et du drone présent à la smartroom

Pour la prise en main de ROS, nous avons suivi les étapes du tutoriel de M. Frezza-Buet. Cela nous a permis de comprendre comment fonctionne ROS et de pouvoir l'utiliser de manière basique et simple.

Pour installer et utiliser ROS, ainsi que notre projet, on pourra se référer à l'annexe B.

3.2 Réduction du bruit

3.2.1 Problème et solution liés au bruit causé par le calcul de la position du drone par rapport à l'image

Nous avons constaté qu'une forte incertitude existait lors du calcul de la position du drone par rapport au panneau. En effet, celle-ci, même si le drone était immobile, bougeait beaucoup et faisait des "sauts". Il s'agissait d'un bruit de type de Poisson, difficile, voire impossible, à traiter en automatique pour obtenir un asservissement satisfaisant. En effet, le drone, avec ce bruit, y aurait été asservi, rendant impossible l'obtention d'une position fixe. Notre drone aurait fait de grands écarts par rapport à la position attendue.

Afin de régler ce problème, l'idée proposée a été de modifier l'algorithme en place. Ce dernier calcule la position des trois carrés bleus dans l'image renvoyée par le drone par les position H, R et L. Cette détermination se faisait par gngt. Nous avons donc développé un nouveau script Python qui utilise la labellisation [2] afin de déterminer la position des panneaux.

Dans cette partie, nous ne nous intéresserons qu'au fichier `find_targets.py`. Il permet, à partir d'un seuil RGB défini, de trouver les parties "bleues" de l'image[1]. Une fois ces positions trouvées, on ne garde que les trois plus grandes parties et on en calcule le centre de masse. Les résultats obtenus sont plutôt satisfaisants.

3.2.2 Détails des fonctions exposées par `find_targets.py`

Le module contient deux fonctions.

`find_targets`

Cette fonction prend en argument l'image dont on souhaite extraire la cible, des valeurs de seuil bleu, rouge et vert minimales et maximales, un booléen `return_slices` qui indique s'il faut, ou non, retourner les slices localisant les limites de la cible et un booléen `return_binary` qui indique s'il faut, ou non, retourne l'image binaire. Ce dernier sera utile pour régler des seuils lorsque, pas exemple, la couleur des cibles est modifiée.

1. On pourrait cependant chercher d'autres couleurs en changeant les valeurs des seuils.

La fonction retourne les coordonnées des centres de masse des trois carrés de la cible sous la forme d'un tuple `(H,L,R)` avec `H` le point le plus haut, `L` le point le plus à gauche et `R` le point le plus à droite ainsi que les slices correspondant aux zones bleues le cas échéant [2].

La première étape de l'algorithme consiste à détecter tous les points répondant aux conditions de seuil définis dans les arguments de la fonction. Lorsqu'un pixel de l'image remplit les conditions, un 1 est placé sur ce pixel, sinon on y place un 0. On construit ainsi une matrice de 0 et de 1 correspondant aux points satisfaisant les critères de seuil.

On définit ensuite une structure de labellisation. Pour cela, on utilise la matrice suivante :

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Cette matrice permet de demander à l'algorithme de labellisation de considérer que deux pixels A et B sont dans la même surface si et seulement si le pixel B se trouve à gauche, à droite, au-dessus ou en-dessous du pixel B. L'algorithme numérote ensuite ces surfaces de manière unique. Nous récupérons ainsi l'image labellisée ainsi que le nombre de zones détectées.

Dans le cas où moins de trois zones sont trouvées, on considère que la détection s'est mal déroulée et une erreur est levée.

On récupère ensuite les zones détectées sous la forme d'une liste de 3-uplets `(a[0],a[1],i)` où `a[0]` et `a[1]` sont les slices des zones (elles délimitent le parallélépipède le plus petit contenant la zone) et `i` est le numéro de la zone.

On trie ensuite les zones par aire et on ne garde que les trois zones ayant les aires les plus grandes. On suppose ici qu'aucune autre surface bleue importante, qui pourrait être confondue avec les cibles, ne sera présente sur notre image.

Nous déterminons ensuite les centres de masse des zones trouvées grâce à `center_of_mass`.

Enfin, on trie les centre de masse selon l'axe vertical et on définit `H` comme le point le plus haut. On trie les centres de masse restants selon l'axe horizontal et on définit `L` par le point le plus à gauche et `R` comme le point le plus à droite. On retrouve ainsi les trois centres de masse de la cible. On retourne ces derniers, avec leurs slices le cas échéant.

`normalize_coordinates`

Cette fonction prend en argument un point, la largeur et la hauteur d'une image et retourne des coordonnées `(x,y)` normalisés. Cette fonction nous permet d'adapter le format des coordonnées renvoyés au code qui existait déjà. Il s'agit donc d'un changement de repère : on place l'origine au centre de l'image, on inverse l'axe y [3] et on normalise les positions de manière à ce que la plus grande dimension (hauteur ou largeur) de l'image vaille 1.

En posant w la largeur, h la hauteur et $j = max(w, h)$, cela revient à calculer les nouvelles coordonnées (x', y') par la transformation affine de l'équation 3.1.

$$(x', y') = (x, y) \cdot \begin{pmatrix} \frac{1}{j} & 0 \\ 0 & -\frac{1}{j} \end{pmatrix} + \left(-\frac{w}{2j}, \frac{h}{2j} \right) \tag{3.1}$$

3.2.3 Conclusion et résultats

Nous obtenons désormais une position bien plus stable du drone. Évidemment, le bruit n'a pas été totalement supprimé, mais il ne provoque plus de "sauts" dans la position et reste, en moyenne, autour de la position du drone. Le bruit restant est probablement causé par la qualité de l'image renvoyée par le drone et par le seuil que nous avons défini. Des résultats de l'algorithme sont donnés figure 3.1. Les positions des cibles seront ensuite publiées grâce au nœud `target_publisher.py` dans un format compréhensible des nœuds de triangularisation.

Ces résultats ont été obtenu à l'aide du script `test_find_targets.py` donné en annexe E.

2. Cela permet de dessiner les rectangles, ce qui est utile lorsqu'on souhaite procéder à des vérifications
3. Dans une image classique, l'axe y est orienté "vers le bas"

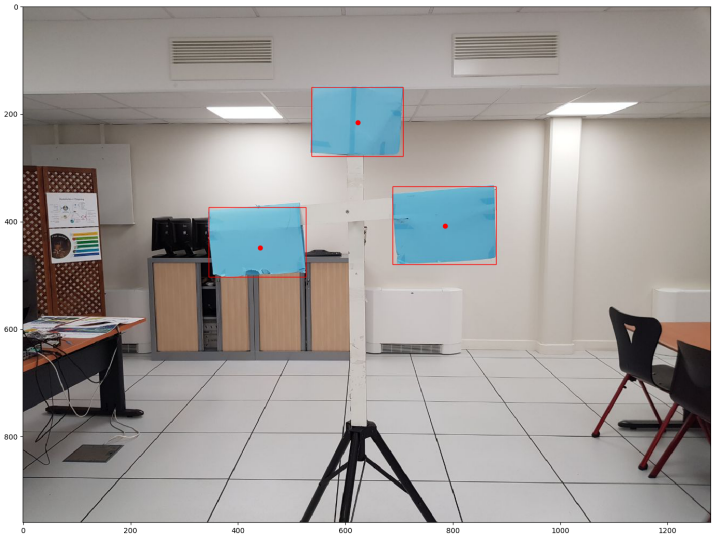

FIGURE 3.1 – Résultat de l'algorithme

3.3 Premier asservissement sommaire

Le premier asservissement qui a été réalisé n'utilisait qu'une seule boucle et asservissait la position du drone. On se référera à la modélisation mécanique réalisée précédemment.

Pour ce premier asservissement sommaire, nous avons estimé la courbe de la réponse fréquentielle de notre drone afin d'en déduire le type de correcteur à utiliser.

Nous savons que selon l'axe z, le drone répond seulement en vitesse. On peut approximer sa réponse fréquentielle par une pente nulle en basse fréquence, puis une pente `[-1]` en haute fréquence. Afin d'asservir le drone, nous aurons donc besoin d'un correcteur de type proportionnel-intégral

$$C(p) = K_p + \frac{K_i}{p} \tag{3.2}$$

Concernant les axes x et y, nous savons que le drone ne peut être correctement asservi en vitesse. En effet, le réglage de la vitesse et de l'accélération du drone se fait selon un angle d'inclinaison : le drone accélère en permanence s'il conserve le même angle ce qui ne permet pas le maintien d'une vitesse constante. La réponse fréquentielle selon ses axes est donc représentée par une pente de `[-1]` dans les basses fréquence et une pente de `[-2]` dans les hautes fréquences. Afin de corriger ces axes, nous allons donc utiliser une correction de type Proportionnelle-Dérivée :

$$C(p) = K_p + K_d p \tag{3.3}$$

3.3.1 Script `triangle_control.py`

Ce script reprend, en partie, le script provenant du projet initial qu'il est possible de retrouver sur le Gitlab de M. FREZZA-BUET [4]. Nous l'avons adapté à notre projet et en avons repris les éléments principaux. Il utilise le fichier de configuration (Voir Annexe F) `triangle_control.cfg` qui a également été repris, en partie, du projet initial. La classe `TriangleControl` implémente 6 méthodes :

`on_reconf` Cette méthode permet, à partir du fichier de configuration, de définir les valeurs à donner aux différents paramètres des 4 PID, ainsi que les valeurs limites de l'accélération et de la vitesse du drone, l'angle de la caméra, la largeur et la longueur de la cible et la distance à laquelle on souhaite que le drone se trouve par rapport à la cible ;

`clear_controls` : remet les erreurs à 0 ;

`saturate_twist` : limite la vitesse selon tous les axes ;

`on_comp` : méthode qui n'a pas été modifié et qui permet de définir la position des trois rectangles bleus de la cible ;

`triangle` : prend en paramètre les trois points de la cible et calcule la consigne de sortie à l'aide de la bibliothèque SimplePID de Python.

3.3.2 Conclusion sur l'utilisation d'une simple boucle

La régulation simple donne des résultats satisfaisants. Cependant quelques problèmes subsistent. Tout d'abord celui du dépassement. En effet la régulation étant linéaire, le dépassement est proportionnel à la consigne. Or si le drone se trouve à grande distance de la cible, le dépassement est tel que cette dernière finit par se trouver hors champ de la caméra, provoquant un comportement chaotique du drone. Une solution pour cela serait de saturer la vitesse du drone afin de limiter l'inertie, et donc le dépassement. D'autre part le système reste "assez lent", dans le sens où il ne donne pas vraiment le sentiment d'un asservissement rigide à la cible. Enfin, il serait intéressant de rendre notre code plus modulaire afin de pouvoir facilement changer la régulation dans un fichier `.launch` de ROS. Afin de régler les deux premiers problèmes, nous choisissons d'implémenter une double régulation en vitesse pour laquelle nous développons une bibliothèque pour effectuer de l'automatique simple sous ROS

3.4 Développement de la bibliothèque d'automatique sous ROS

Nous avons développer une bibliothèque permettant de créer d'implémenter des PID sous ROS et de les régler dynamiquement par un protocole expérimental simple. Afin d'implémenter les correcteurs, nous avons créé des nœuds ROS que nous avons ensuite connectés dans des fichiers `.launch`.

L'idée est de créer plusieurs classes de nœuds dans un script Python dont les paramètres seront obtenus à l'aide de fichiers `.cfg`.

3.4.1 Mesure de la vitesse

Afin d'implémenter la partie dérivée de nos PID, il faut pouvoir calculer la vitesse du drone.

La mesure de vitesse doit se faire en dérivant la mesure de position. Cependant le bruit présent sur cette mesure empêche d'utiliser une dérivation naïve : en effet, on aurait alors de brusques sauts sur la valeur mesurée dus aux hautes fréquences dans le spectre du bruit. Afin d'obtenir une valeur lissée de la vitesse, on se propose d'utiliser un filtre de Savitzky–Golay.

Principe

Un filtre de Savitzky–Golay [8] est un filtre à réponse impulsionnelle finie, qui correspond à une approximation locale du signal par un polynôme de degré faible. Ainsi, un filtre moyenneur est un filtre de Savitzky–Golay simple. Afin d'obtenir la réponse indicielle $[b_0, \dots b_n]$ du filtre, on utilise la méthode des moindres carrés. En posant $((x_i, y_i))_{i\in[\![1\,;n]\!]}$ les n points de la fenêtre, et en écrivant le polynôme d'approximation $a_0 + a_1X + \cdots + a_kX^k$, Cela revient donc à déterminer le jeu de coefficients $(a_0, \dots, a_k)$ qui minimise la grandeur de l'équation 3.4.

$$\left\| \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} - \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^k \\ 1 & x_2 & x_2^2 & \cdots & x_2^k \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^k \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_k \end{pmatrix} \right\|^2 \tag{3.4}$$

En supposant de plus que la fenêtre de points $((x_i, y_i))_{i \in [\![1\,;k]\!]}$ que l'on souhaite lisser est centrée en 0 (on peut toujours se ramener à ce cas par changement de variable), telle que les x_i soient espacés régulièrement d'un pas h, et k impair, on peut s'affranchir des valeurs prises par (x_i) en calculant les $(a'_0, \ldots, a'_k)$ qui minimisent

$$\begin{aligned} L((y_k),(a_k)) &= \left\| \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}}_{=Y} - \underbrace{\begin{pmatrix} 1 & -\left\lfloor \frac{n}{2} \right\rfloor & \left(-\left\lfloor \frac{n}{2} \right\rfloor\right)^2 & \cdots & \left(-\left\lfloor \frac{n}{2} \right\rfloor\right)^k \\ 1 & \left(-\left\lfloor \frac{n-1}{2} \right\rfloor\right) & \left(-\left\lfloor \frac{n-1}{2} \right\rfloor\right)^2 & \cdots & \left(-\left\lfloor \frac{n-1}{2} \right\rfloor\right)^k \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & -1 & (-1)^2 & \cdots & (-1)^k \\ 1 & 0 & 0 & \cdots & 0 \\ 1 & (1) & (1)^2 & \cdots & (1)^k \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) & \left(\left\lfloor \frac{n-1}{2} \right\rfloor\right)^2 & \cdots & \left(\left\lfloor \frac{n-1}{2} \right\rfloor\right)^k \\ 1 & \left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \left(\left\lfloor \frac{n}{2} \right\rfloor\right)^2 & \cdots & \left(\left\lfloor \frac{n}{2} \right\rfloor\right)^k \end{pmatrix}}_{=M} \cdot \underbrace{\begin{pmatrix} a'_0 \\ a'_1 \\ \vdots \\ a'_k \end{pmatrix}}_{=A} \right\|^2 \qquad (3.5) \\ &= (Y - M \cdot A)^\intercal \cdot (Y - M \cdot A) \qquad (3.6) \end{aligned}$$

On a donc :

$$\frac{\partial L}{\partial (A)}(A) = -2 \cdot M^\intercal \cdot (Y - M \cdot A) \tag{3.7}$$

Et finalement, en cherchant le point où cette dérivée s'annule :

$$A = \underbrace{(M^\intercal \cdot M)^{-1} \cdot M^\intercal}_{=B} \cdot Y \tag{3.8}$$

On peut ainsi obtenir facilement les valeurs "lissées" en chaque point i : en effet, la valeur de la fonction en un point est donné par le coefficient a_0, celle de sa dérivée par $\frac{a_1}{h}$ etc. Plus formellement, pour approximer la dérivée d'ordre d, on peut convoluer le signal par le filtre RIF $[b_1, \ldots b_n]$, avec

$$\forall i \in [\![0\,;n]\!], b_i = B_{d,n-i} \tag{3.9}$$

Dans le cas de notre projet, on ne s'intéresse qu'a la vitesse (donc $d = 1$). Il faut maintenant déterminer la taille du filtre (n) et le degré k du polynôme d'approximation. En effet on souhaite minimiser n car la mesure de vitesse induit un retard de $\frac{n}{2} \times h$.

Protocole expérimental

Afin de déterminer le filtre que nous utiliserons, nous choisissons d'en tester plusieurs sur le même relevé de position. Pour relever la position, on se contente de tenir le drone allumé face à la cible puis de marcher dans sa direction. On se référera à l'annexe C pour la méthode employée pour exporter les données depuis ROS vers un fichier `output.txt`. On peut ensuite traiter les données avec le script Julia `test_filter.jl` (annexe D).

Choix du filtre

Les résultats obtenus sont donnés figures 3.2, 3.3 et 3.4. On remarque que la vitesse est mieux lissée lorsqu'un filtre quadratique est utilisé. En effet, plus le degré du polynôme est élevé, plus il sera proche du signal mesuré et donc lissera moins bien.

Sans surprise, le fait d'augmenter la taille du filtre permet de mieux lisser la vitesse. Toutefois, pour éviter de causer un retard trop important dû à l'attente des points pour calculer la dérivée, il sera nécessaire de trouver un compromis entre signal lissé et retard inhérent au filtre, ceci afin d'augmenter la stabilité du système.

Après étude des courbes obtenues, nous avons choisis de commencer les expérimentations d'asservissement avec un filtre quadratique de 19 points, ce qui semble assurer un bien meilleur lissage. Cependant il faut prendre en compte le retard induit par la taille de fenêtre.

Lors de l'implémentation du filtre, il faut également s'assurer que la fréquence d'échantillonage du signal d'entrée correspond à celle pour lequel le filtre est calculé [4].

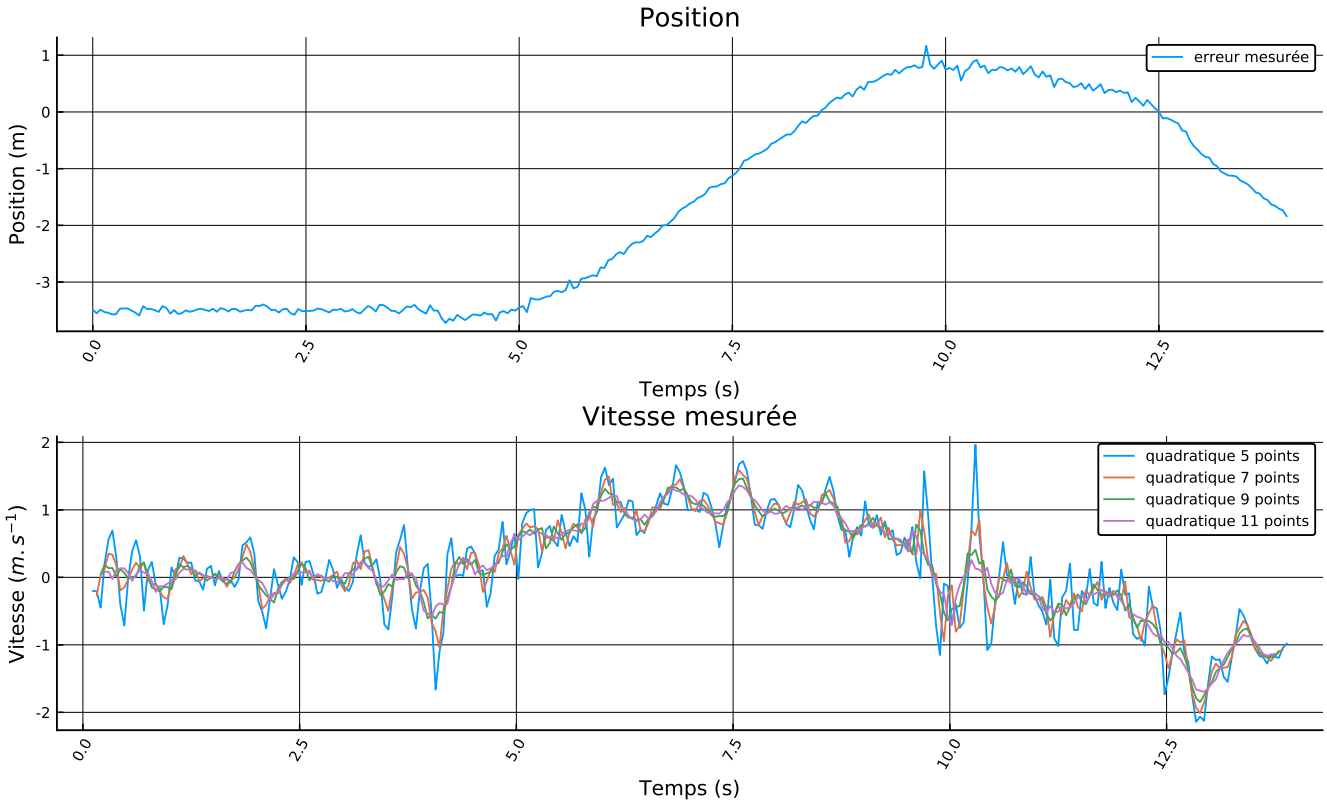

FIGURE 3.2 – Performance des filtres quadratiques

4. Nos expérimentations ont montré la très mauvaise stabilité du filtre dans les cas où la fréquence d'échantillonage n'est pas constante.

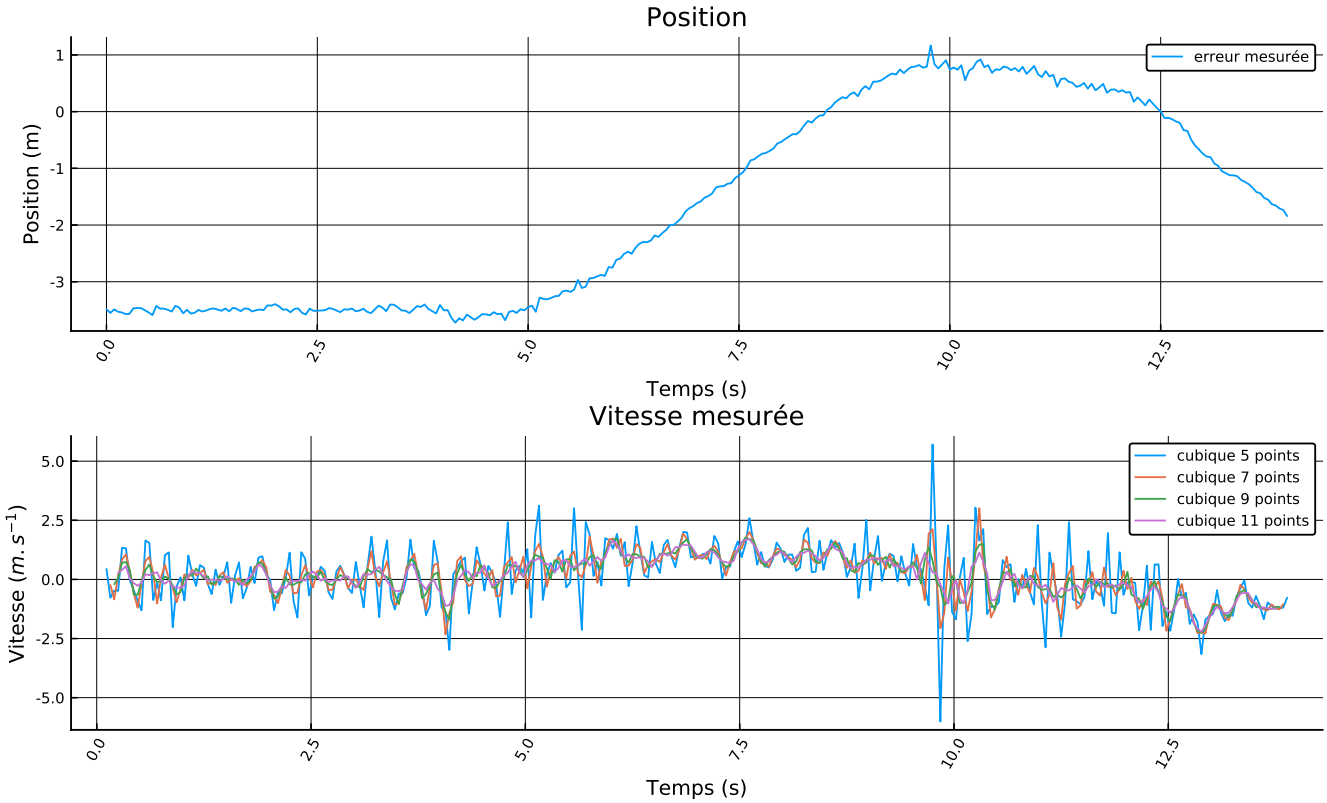

FIGURE 3.3 – Performance des filtres cubiques

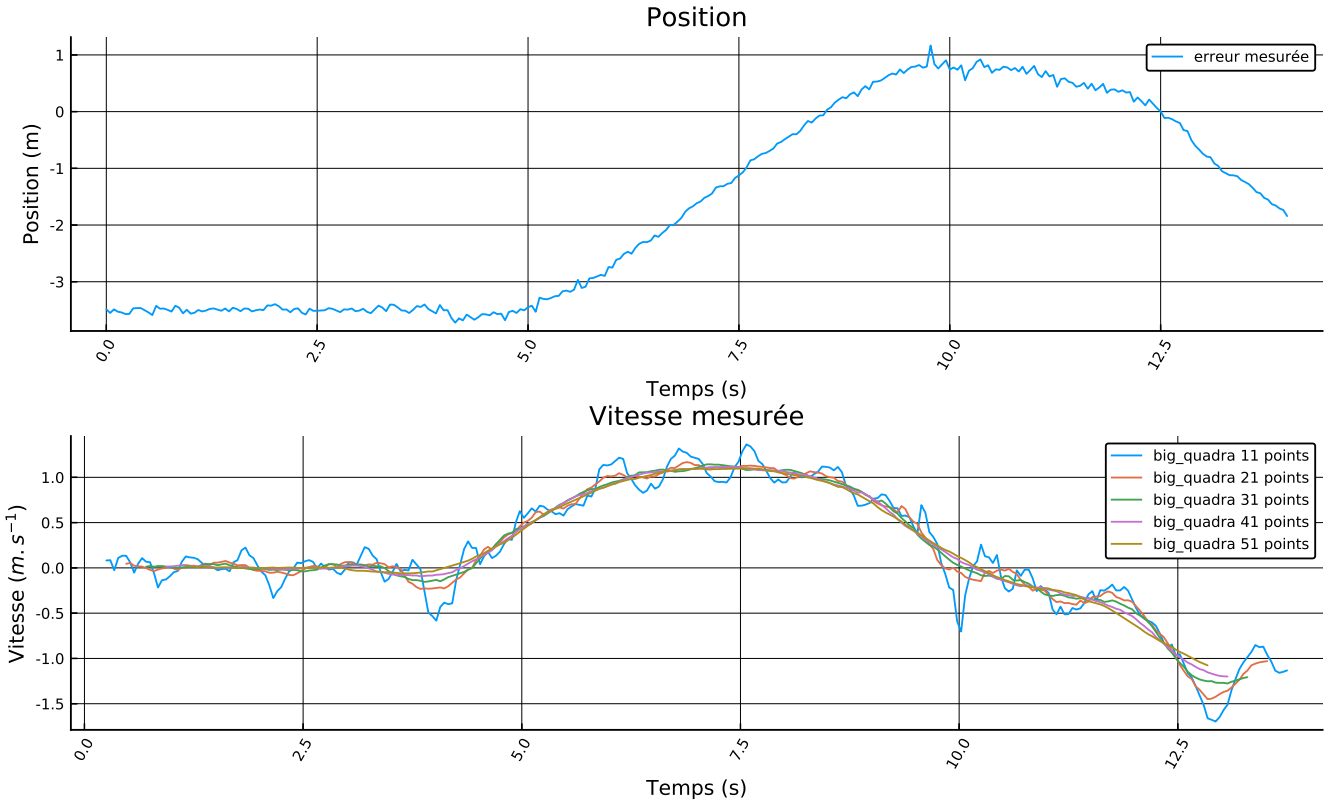

FIGURE 3.4 – Performance des filtres quadratiques pour des nombres de points élevés

3.4.2 Création de fichiers de configuration

Les fichiers de configuration vont nous permettre de régler, à l'aide d'une interface graphique, les paramètres de nos correcteurs dans la double boucle. Pour une explication plus détaillée du fonctionnement des fichiers de configuration, on se réfèrera à l'annexe F.

Nous avons créé les fichiers de configurations suivants :

`DerivativeNode.cfg` : permet de choisir le gain de la partie dérivée du correcteur et le filtre de Savitzky-Golay adapté (ordre et taille du polynôme à utiliser) ;

`IntegralNode.cfg` : permet de choisir le gain de la partie intégrale du correcteur et les valeurs minimale et maximale de l'intégrale ;

`ProportionalNode.cfg` : permet de choisir le gain de la partie proportionnelle du correcteur ;

`InputNode.cfg` : contient la valeur d'une entrée ;

`SaturateNode.cfg` : permet de choisir les valeurs de saturation (max et min) ;

`RateNode.cfg` : permet de forcer une fréquence d'échantillonage constante.

3.4.3 Création des classes de nœuds dans un script Python

L'idée du script est de créer une classe de nœud `ControlNode` qui se spécialise en d'autres nœuds utiles pour notre PID. Chacun des noeuds héritant de `ControlNode` prendra alors une ou plusieures entrées afin de générer une sortie. On peut également leur appliquer un reset afin de mettre la sortie à 0.

Dans la classe `ControlNode`, on définit un Publisher `output_topic` qui publie sur le topic "output" ainsi qu'un Subscriber qui s'inscrit au topic "reset".

Le diagramme d'héritage de notre script est donné figure 3.5, page 17.

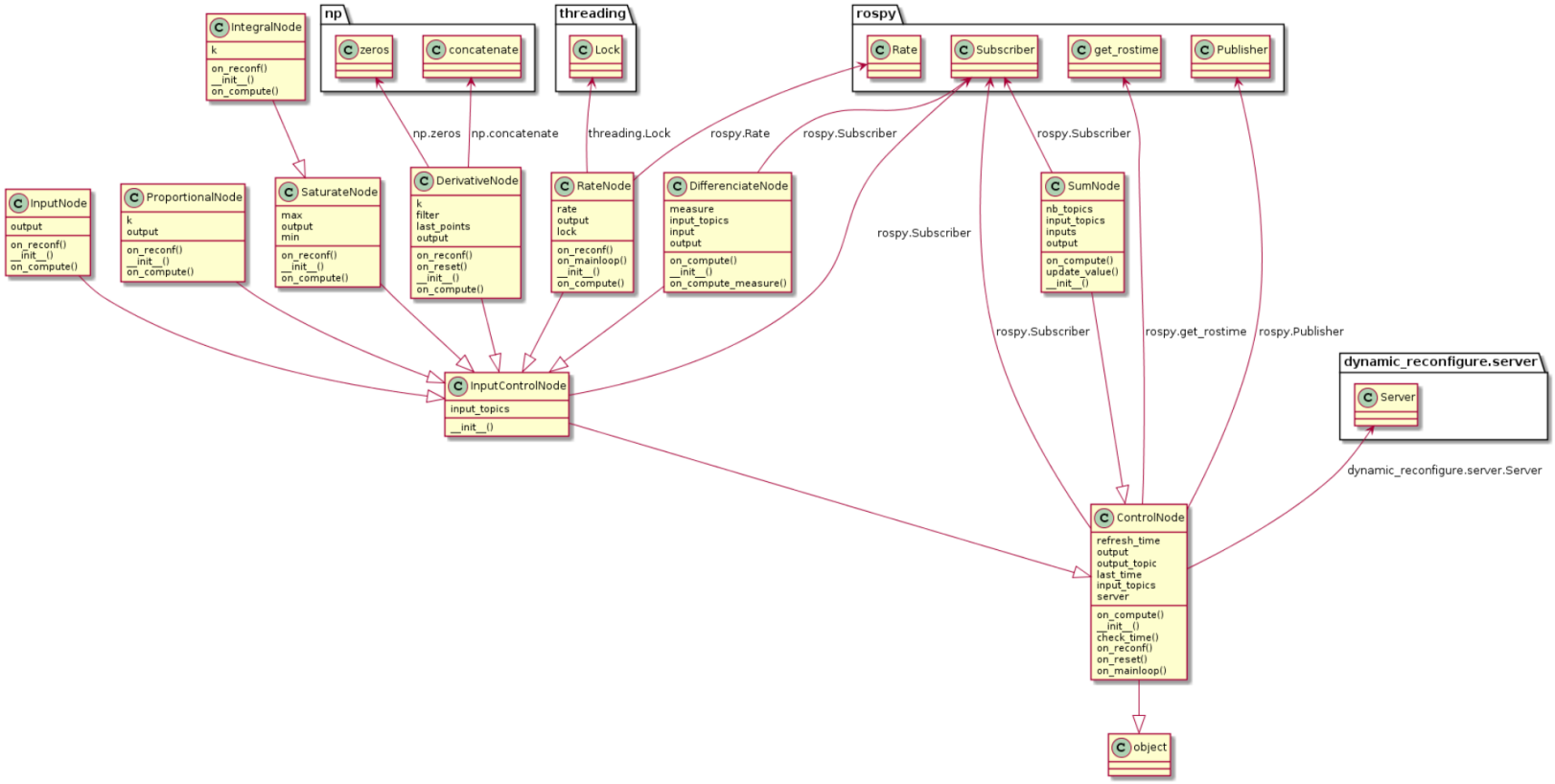

FIGURE 3.5 – Diagramme UML

Les méthodes de la classe mère sont les suivantes :

`on_reset(self, value)` : la valeur de l'output est remise à 0. Certaines classes filles comme `DerivativeNode` surchargent cette méthode pour remettre d'autres valeurs à 0 ;

`on_reconf(self, config, level)` : permet de récupérer la configuration, réalise également un appel à `on_reset` ;

`on_compute(self, value)` : publie sur le topic "output" et met à jour la date du dernier calcul pour la limitation de fréquence ;

`check_time(self, delta_time=0.0)` : permet de vérifier qu'on ne réalise pas de calculs trop souvent et donc de manière irrégulière [5]. On vérifie que le temps passé depuis le dernier calcul est supérieur au `refresh_time` que l'on s'est imposé.

On définit une classe fille `InputControlNode` qui hérite de `ControlNode`. Ceci a été l'occasion d'expérimenter un effet de bord intéressant de l'héritage en environnement multi-threadé. En effet, `ControlNode` lance le Subscriber sur `on_reset` dès l'appel à la méthode `__init__` de la classe mère, mais c'est la méthode `on_reset` de la classe fille qui est appelée, méthode qui n'est possiblement pas encore correctement initialisée, menant à des erreurs d'exécutions. Cette classe fille nous permet de rajouter un Subscriber qui récupère un input d'un topic "input".

Toutes les classes filles de `InputControlNode` s'inscriront donc à un topic pour récupérer une entrée et publieront sur un topic "output". Les classes héritant de `InputControlNode` sont :

`ProportionalNode` : partie proportionnelle du correcteur, permet de définir la valeur du gain et de publier l'output sur un topic avec `on_compute` après vérification du temps ;

`SaturateNode` : permet d'imposer une valeur maximale et minimale à l'output. La classe `IntegralNode` en hérite. Cette classe permet de définir la valeur du gain de l'intégrale et de publier l'output sur un topic (output qui sera donc saturé). Ceci permet d'éviter des instabilités dûes à l'action intégrale. L'intégration est réalisée avec la méthode des rectangles [6] ;

`DerivativeNode` : partie dérivée du correcteur, permet de définir la valeur du gain, la façon de calculer la valeur de la dériver et de publier l'output de la correction dérivée sur un output. La valeur de la dérivée est calculée à l'aide d'un filtre de Savitzky-Golay dont on récupère le paramètre dans la configuration ;

`DifferenciateNode` : permet de faire la différence entre une valeur d'entrée et une valeur mesurée ;

`InputNode` : permet de définir un nœud publiant une sortie ;

`RateNode` : permet de forcer une fréquence d'échantillonage fixe sur la sortie.

Enfin, `SumNode` hérite de `ControlNode`. Il va permettre de faire la somme des différentes parties du correcteur PID. Nous ne le faisons pas hériter de `InputControlNode` car il ne prendra pas toujours une seule entrée. La classe prend donc en paramètre un nombre de topics auxquels elle s'inscrit et qui seront les entrées de la somme.

Ces classes permettront la création de nos nœuds ROS et à terme devraient permettre l'implémentation de la double boucle d'asservissement.

Le script que nous avons écrit prend en paramètre le type de nœud que l'utilisateur souhaite créer (`sum`, `differenciate`, `input`, `saturate`, `derivative`, `integral`, `rate` ou `proportional`) afin de pouvoir aisément lancer beaucoup de nœuds en les organisant dans un fichier `.launch`

3.4.4 Fichier `control.launch`

Puisque dans la suite nous aurons régulièrement à créer des régulateurs PID, nous décidons d'ajouter à notre bibliothèque un fichier `launch` permettant d'en inclure un facilemetn. Ce fichier utilise 8 paramètres : `input`, `output`, `measure`, `reset`, `param_P`, `param_I`, `param_D` et `param_input`. Les quatre premiers paramètres permettent de régler les topics sur lesquels le block doit publier ou s'inscrire, tandis que les quatre derniers donnent les chemin relatif vers les fichiers de configuration des différents nœuds par rapport au répertoire `params` de notre projet.

5. En effet, comme il est dit dans la section portant sur le filtre dérivateur, ce dernier nécessite un pas d'échantillonage constant.
6. Il serait intéressant d'étudier l'effet de méthodes d'intégration plus performantes, notamment sur la stabilité.

Le bloc est composé des noeuds suivants : une entrée, une différence qui entre l'entrée et la mesure, les parties proportionnelle, intégrale et dérivée du PID qui font leurs calculs et les publie sur des topics de sorties et une somme sur ces sorties et publie le résultat sur le topic `output`.

On crée les nœuds de la manière suivante :

```
<node name="nome name" pkg="package" type="control_compute.py" args="name of the class">
    <remap from="name of the argument in control_compute.py" to="new name" />
    ...
</node>
```

Le remap permet de relier les noeuds entre eux. Par exemple, la sortie de la différence est remapé sur un epsilon et les entrées du PID sont remapées sur epsilon. La sortie de la somme devient donc l'entrée du PID.

Durant les phases de test, nous avons rencontré des problèmes lors du remap, notamment lors de l'utilisation du noeud de somme. Nous avons donc décidé de simplifier notre diagramme d'héritage en ne faisant pas hériter `SumNode` de `InputControlNode` ce qui nous permettait de définir plusieurs inputs et de régler le problème de remap. Nous ne savons toutefois pas pourquoi le problème existait au préalable.

Le fichier `control.lauch` permet ainsi de générer le graphe de nœuds de la figure 3.6.

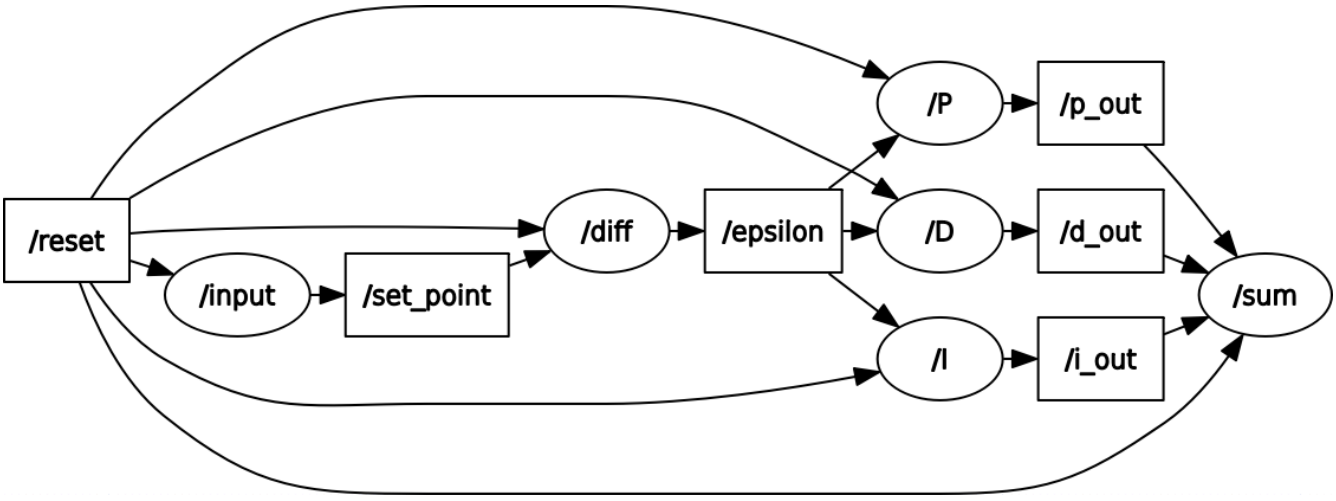

FIGURE 3.6 – Nœuds ROS du block control

3.5 Nœuds divers réalisés pour mettre en place l'asservissement

Afin d'interfacer notre bibliothèque avec les drones BeBop, divers nœuds ont étés réalisés ou adaptés du projet existant de M. FREZZA.

3.5.1 Script `safe_drone_teleop.py`

Ce script a été presque entièrement repris du projet initial, à l'exception de l'ajout d'une publication sur le topic `reset_pid` lors du passage en mode automatique afin de déclencher la remise à zéro des régulateurs, par exemple pour désaturer les actions intégrales. Il s'agit donc de l'interface de contrôle principale des drones.

3.5.2 Script `triangle.py`

Ce script a pour mission de calculer la position du drone à partir de la position normalisée des cibles dans l'image. Il publie ensuite les positions au format `Float64` de ROS, que les nœuds de notre bibliothèque comprennent.

3.5.3 Script `twist_controls.py`

Ce script est destiné à lire la sortie des différents contrôleurs afin de générer un objet `Twist` qui sera envoyé à `safe_drone_teleop.py` pour être interprété comme la consigne à donner au drone.

3.6 Asservissement simple boucle avec la bibliothèque

3.6.1 Méthode de Ziegler-Nichols

Afin de déterminer les paramètres des correcteurs PID (axes z et angle z), nous avons utilisé la méthode de Ziegler-Nichols [9]. Pour réguler les boucles selon les quatre axes, nous allons générer des oscillations (qui sont faciles à voir) en augmentant progressivement le gain proportionnel. Une fois des oscillations d'amplitude et de fréquence constantes obtenues, nous allons pouvoir calculer le régulateur à utiliser en fonction de K_u le gain proportionnel obtenu et T_u la fréquence des oscillations.

Type de contrôle	Paramètres de réglage
P.I.D	$K_p = 0.6K_u \ K_i = \frac{T_u}{2} \ K_d = \frac{T_u}{8}$
P.I.D peu de dépassement	$K_p = 0.33K_u \ K_i = \frac{T_u}{2} \ K_d = \frac{T_u}{3}$
P.I.D aucun dépassement	$K_p = 0.2K_u \ K_i = \frac{T_u}{2} \ K_d = \frac{T_u}{3}$

3.6.2 Méthode empirique

Pour les deux autres axes (x et y), nous avons utilisé une méthode empirique de réglage. Tous les autres gains étant par ailleurs nuls, on augmente progressivement le gain D [7]. Lorsque l'on atteint la limite du comportement oscillant du système, on augmente alors le gain D [8].

3.6.3 Protocole de réglage

Afin de faciliter le réglage du drone, il convient de régler les différents axes dans le "bon" ordre. Pour appliquer la méthode de Ziegler-Nichols, on pourra filmer les oscillations du drone (par exemple avec un téléphone portable) afin de mesurer la période des oscillations *a posteriori* [9].

Nous conseillons donc de procéder au réglage du drone dans cet ordre :

1. Axe z ;
2. Lacet (angle selon z) ;
3. Axe y ;
4. Axe x.

Cependant on notera que l'asservissement des différents axes n'est pas totalement indépendant. En particulier, nous avons pu nous rendre compte que le réglage de l'axe x a tendance à instabiliser le lacet. De plus, il est difficile de régler correctement l'axe y si l'axe x est instable, et *vice-versa*. On procédera donc à un réglage par itérations successives. Enfin on notera que du fait de la mesure de l'erreur, il est nécessaire de donner des gains négatifs pour les axes x et de lacet.

3.6.4 Fichier `launch`

Nous avons créé un fichier `simple_loop.launch` qui permet de lancer la simple boucle tout en chargeant les paramètres de drone que nous avons déterminé. Ce fichier fait usage du fichier `control.launch` de notre bibliothèque.

7. Pour rappel, cela revient à augmenter le gain P du PI équivalent du fait du comportement double intégrateur du système.
8. Pour les mêmes raisons que précédemment, cela revient à augmenter le I du PI équivalent.
9. Il existe de nombreux logiciels de relevé de position à partir d'une vidéo, par exemple avimeca.

3.7 Asservissement double boucle avec la bibliothèque

Afin d'améliorer les performances, on souhaite intégrer une boucle de régulation interne sur la vitesse. En outre, cette boucle devrait nous permettre d'ajouter une saturation en vitesse limitant le dépassement lorsque le drone arrive depuis une grande distance vers la cible. On modélise la nouvelle régulation avec la figure 3.7 (page 21).

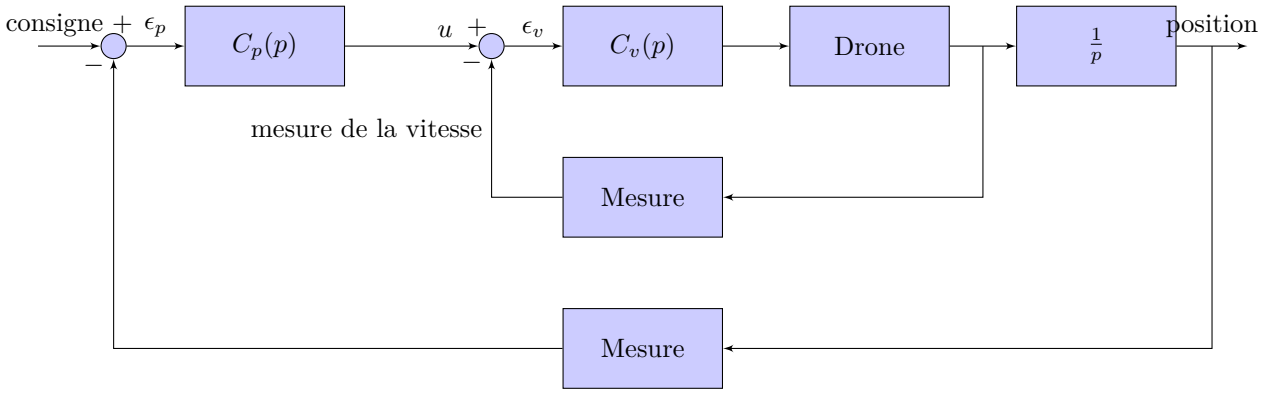

FIGURE 3.7 – Modélisation simple de l'asservissement

3.7.1 Réglage de la double boucle

Méthode

Afin de régler la double boucle, on utilisera la méthode de ZIEGLER-NICHOLS. Cette méthode permet de déterminer les paramètres du PID à implémenter de deux façons possibles.

— On peut générer des oscillations d'amplitudes et de fréquence constante sur la mesure à régler en augmentant doucement le gain proportionnel. Avec les valeurs du gain K_u et de la période des oscillations T_u, on calcule les paramètres du PID de la façon suivante : $K_i = 2/T_u$, $K_p = 0.6K_u$, $K_d = T_u/8$;

— On enregistre la réponse du système non régulé à un échelon et on en déduit la valeur des paramètres par analyse de cette réponse comme donné sur la 3.8.

Réglage de la boucle interne

Nous allons commencer par régler la boucle interne, soit la boucle asservissant la vitesse. Pour cela, nous allons utiliser la réponse du système non régulé à un échelon afin de trouver les paramètres du PID interne.

Pour cela, non envoie un échelon en inclinaison à notre drone (il s'agit donc d'une accélération) car nous ne pouvons pas directement lui donner une commande en vitesse, et on enregistre la réponse indicielle en boucle ouverte. Pour la calculer, nous avons utilisé un filtre de SAVITSKY-GOLAY quadratique de 19 points (avoir beaucoup de retard dans les calculs ne pose, ici, aucun problème, car nous ne cherchons pas à avoir une réponse rapide, mais une réponse la plus lissée possible pour déterminer au plus juste les paramètres découlant de la méthode de ZIEGLER-NICHOLS.

Afin de déterminer la réponse indicielle de la vitesse en boucle ouverte, nous avions besoin d'un grand espace. Nous avons donc déterminé cette réponse dans le gymnase en modifiant la couleur des cibles (jaunes) pour que le drone ne soit pas perturbé par le sol qui est bleu.

Cependant nos essais n'ont pas été très concluants pour plusieurs raisons. Tout d'abord parce que le drone a tendance à perdre la cible lorsqu'il est à grande distance, et à cause des changements de luminosité. Cependant un réglage à proximité de la cible n'est pas envisageable car le drone met quelques secondes à atteindre sa vitesse maximale : il dépasse donc la cible. La régulation en double boucle n'est donc pas fonctionnelle à l'heure de la cloture de ce rapport.

PID Type	K_p	$T_i=K_p/K_i$	$T_d=K_d/K_p$
P	$\frac{T}{L}$	∞	0
PI	$0.9\frac{T}{L}$	$\frac{L}{0.3}$	0
PID	$1.2\frac{T}{L}$	$2L$	$0.5L$

Table 2: Ziegler-Nichols Recipe – First Method

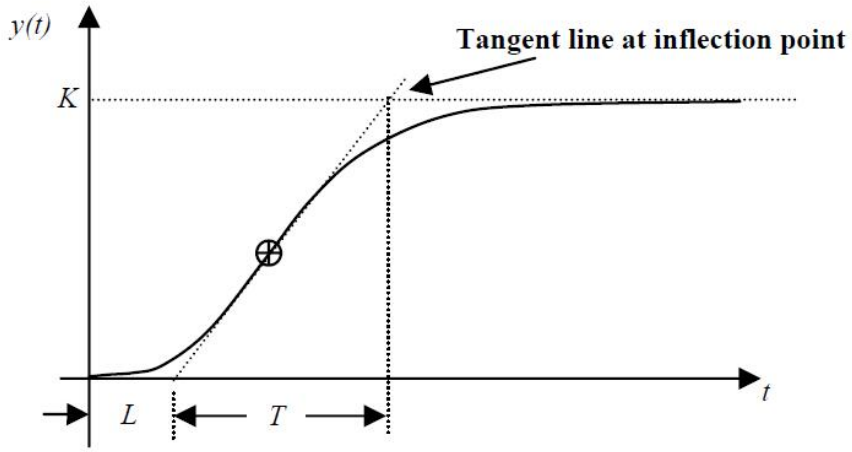

Figure 4: Response Curve for Ziegler-Nichols First Method

FIGURE 3.8 – Méthode de Ziegler-Nichols pour déterminer les paramètres d'un PID [6]

3.7.2 Création du fichier launch et connection des noeuds

Pour réaliser la double boucle, on crée un fichier `launch` dédié.
On crée des noeuds afin de récupérer la position du drone

```
  <node name="targets" pkg="detect_targets" type="target_publisher.py">
</node>

<node name="triangle" pkg="detect_targets" type="triangle_control.py" output="screen">
  <remap from="component_centers" to="targets"/>
</node>
```

On organise grâce à des groupes :

```
<group ns="name">
  ...
</group>
```

À l'intérieur de ces groupes, on récupère les noeuds crées avec `control.launch` en incluant le fichier :

```
<include file="$(find detect_targets)/launch/control.launch" ns="name of the loop">
```

Et on donne les bonnes valeurs aux arguments grâce à

```
<arg name="name of the argument in control.launch" value="value of the argument" />
```

Pour les boucles en x et en y, on crée deux boucles, tandis que pour les boucles en z, on en crée qu'une seule (car on ne régulera pas cet axe avec une double boucle, ce dernier n'étant pas soumis aux problèmes de lenteur et de dépassement étant donné les faibles déplacements sur cet axe).

Il "suffit" ensuite de bien relier les entrées et les sorties des noeuds. Nous obtenu le résultat de nos noeuds 3.9.

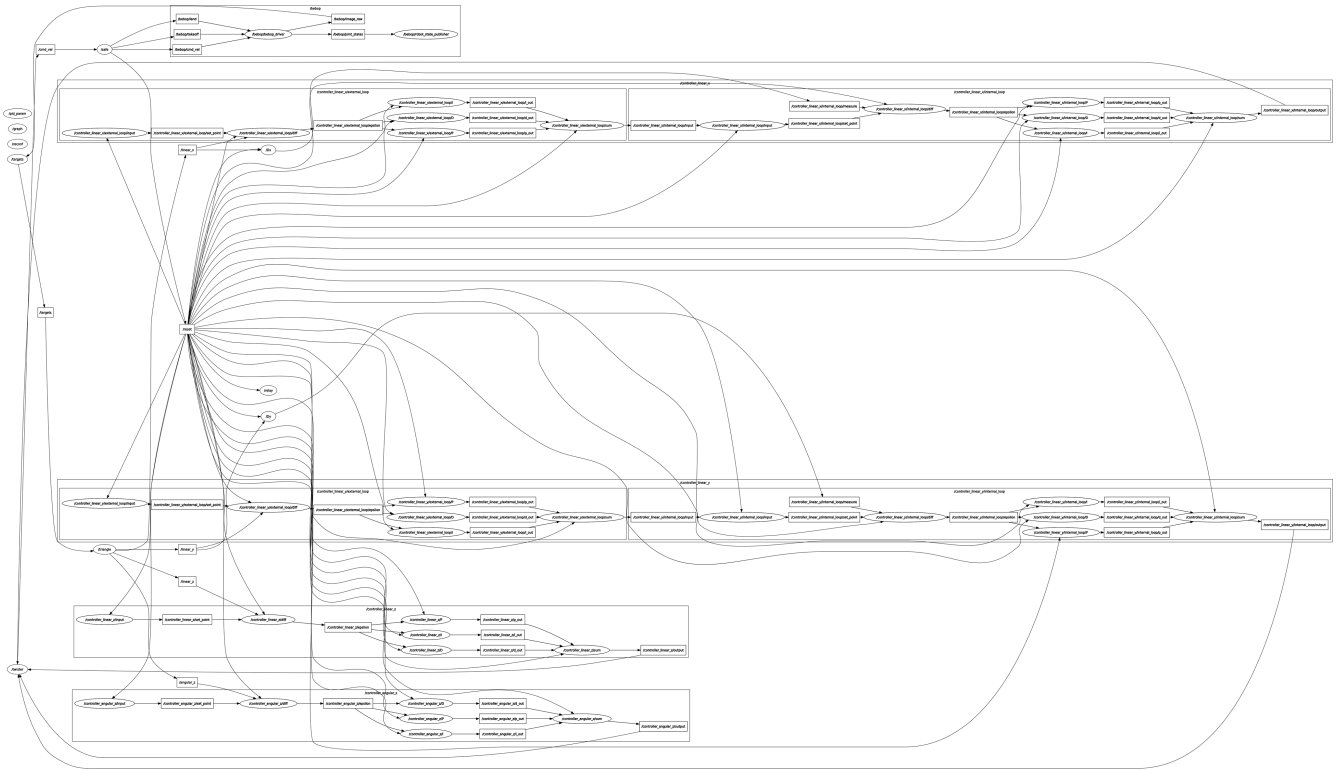

FIGURE 3.9 – Nœuds ROS pour la double boucle

Chapitre 4

Utilisation du livrable

4.1 Installation du module

Pour l'installation du module, se référer à l'annexe B.

4.2 Utilisation

4.2.1 Lancement

Le lancement du projet est décrit dans l'annexe B. Pour rappel :

1. Connectez-vous au réseau WiFi du drone ;
2. Ouvrez un terminal et lancez :

```
roscore
```

3. Dans un nouveau terminal :

```
source ~/drone-rigide/devel/setup.bash
roslaunch detect_targets simple_loop.launch
```

Cinq fenêtres s'ouvrent : un terminal, deux fenêtres de visualisation, un graphe des nœuds et une fenêtre de configuration.

4.2.2 Contrôle manuel

Dans la fenêtre de terminal (figure 4.1) qui s'est ouverte, il est possible de contrôler manuellement le drone. Au bout de quelques secondes après le décollage, si aucun ordre n'est donné le contrôle automatique prend la main.

4.2.3 Fenêtre de visualisation

Ces fenêtres (figure 4.2) permet de donner un aperçu de la sortie de la caméra, afin notamment de vérifier que le drone est bien en mesure de visualiser la cible.

La fenêtre en haut à droite permet de régler les seuils de couleur. Il y a également deux options "Binary" qui permet d'afficher l'image binaire (en haut à gauche) et "Targets" qui permet d'afficher la position des cibles trouvée. Il faut bien penser à décocher ces deux options lorsque le drone est en vol, car leur génération provoque beaucoup de calculs et induit donc un retard, source d'instabilités.

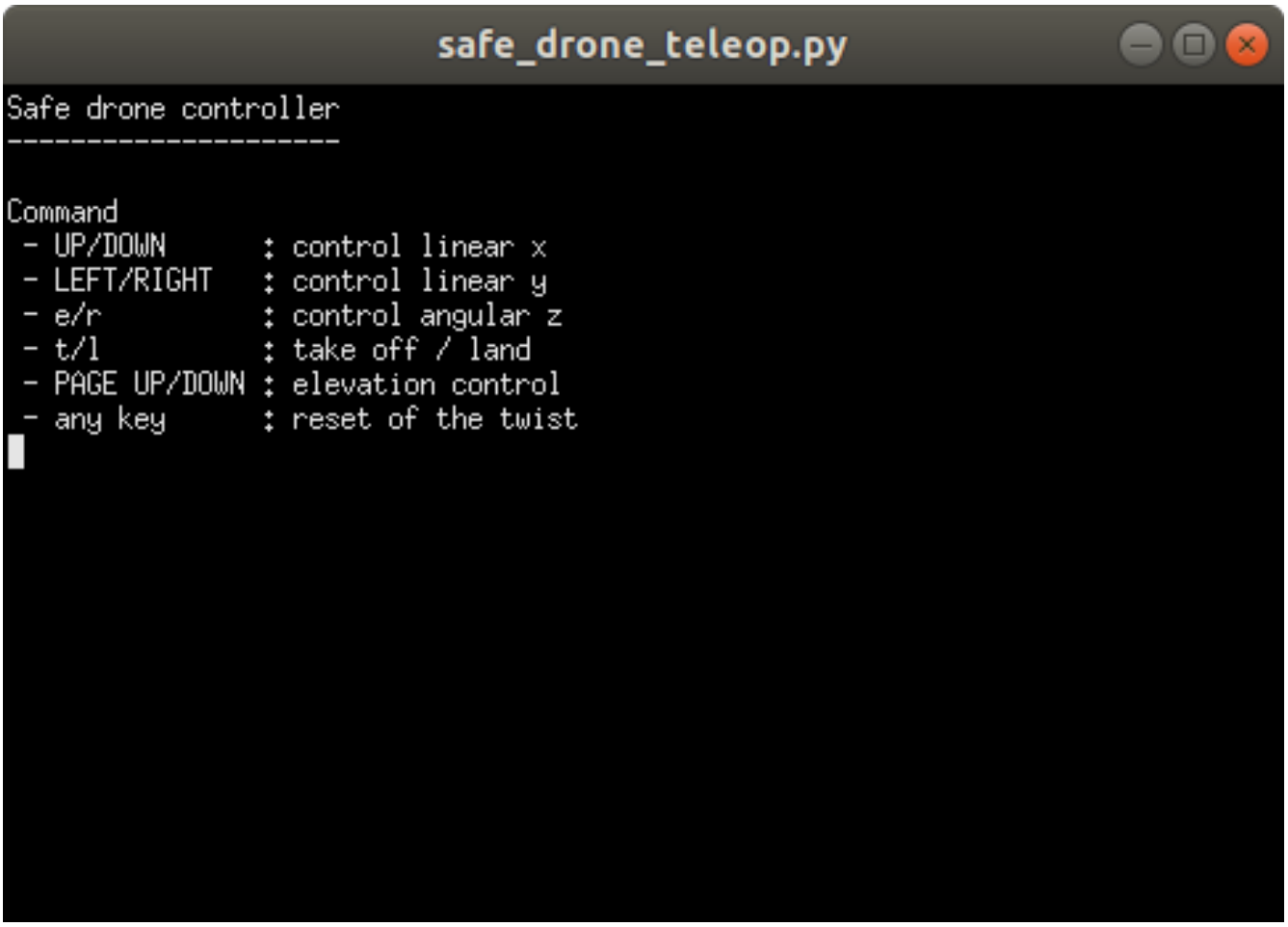

FIGURE 4.1 – Terminal de contrôle manuel

4.2.4 Fenêtre de paramétrage

La dernière fenêtre qui s'ouvre (figure 4.3) est la fenêtre de paramétrage. Elle permet de paramétrer tous les nœuds qui utilisent `dynamic_reconfigure`. Ainsi, pour donner une consigne de distance sur l'axe x, il suffit de changer le paramètre `value` de `controller_linear_x/external_loop/input`.

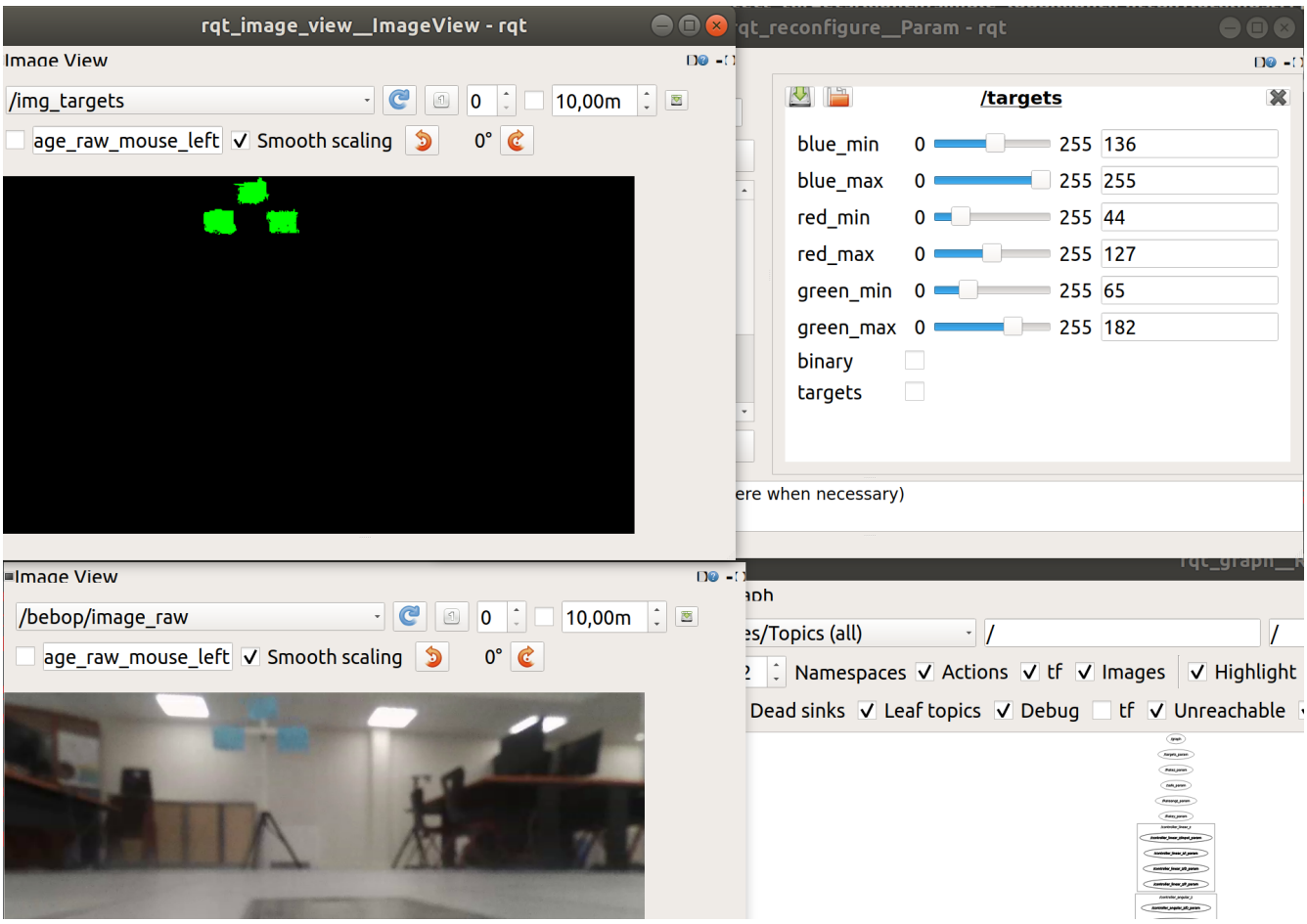

FIGURE 4.2 – Visualisation de la sortie de la caméra

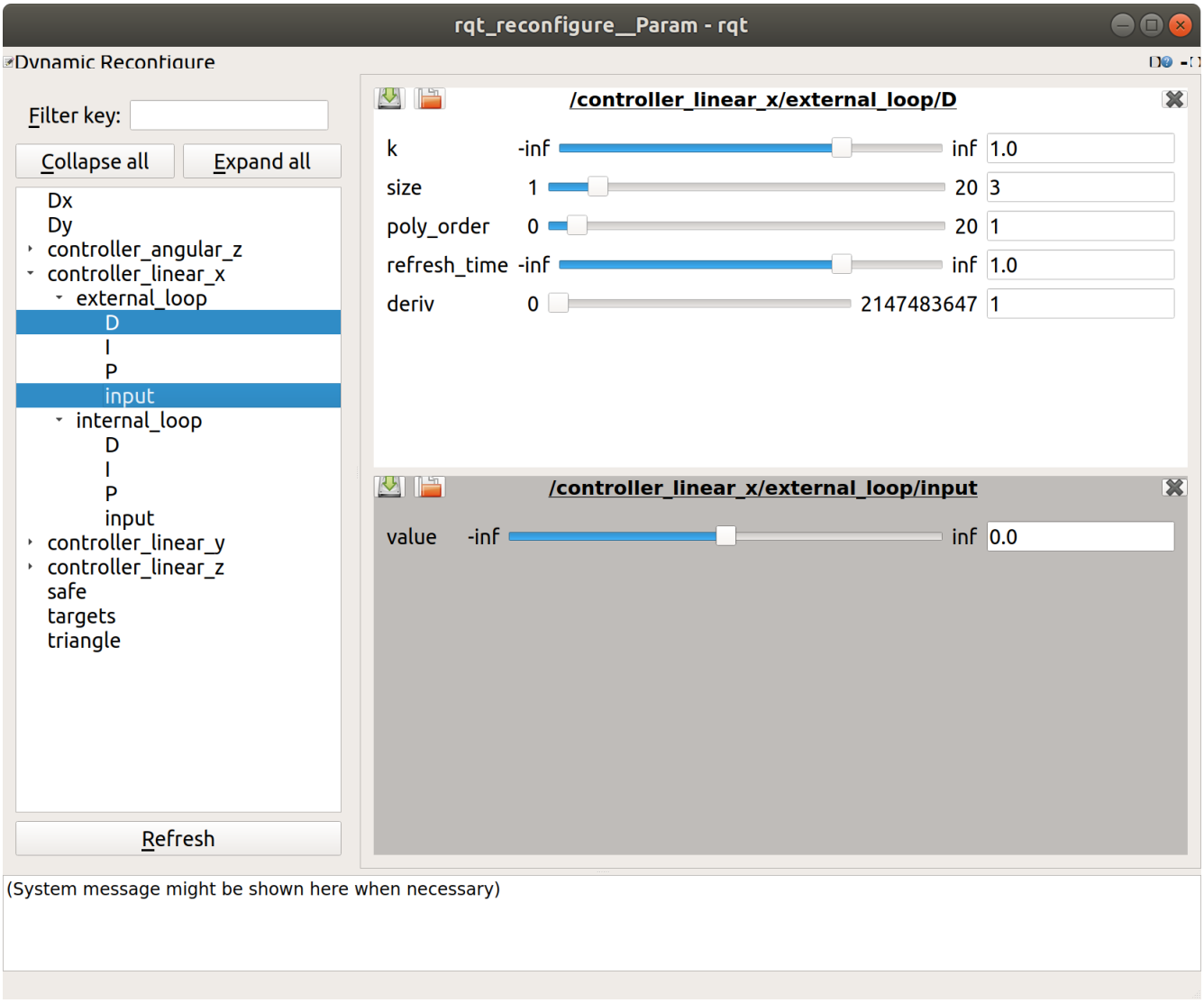

FIGURE 4.3 – Fenêtre de paramétrage

Chapitre 5

Conclusion

L'objectif du projet est atteint. L'asservissement en position du drone est désormais bien plus stable. En outre, le script `control_compute.py` peut aisément être réutilisé dans d'autres projets avec ROS pour implémenter facilement des modules de contrôle.

Ce projet a également été l'occasion pour nous de découvrir et de nous familiariser avec ROS, en plus de mettre en pratique de nombreux cours de Supélec (Génie logiciel, Signaux et Systèmes, Automatique et Commande d'entraînements à vitesse variable notamment).

Nous souhaitons en particulier remercier messieurs Frezza-Buet et Gutzwiller pour leur accompagnement et leurs conseils tout au long du projet.

L'intégralité du travail réalisé pour ce projet peut être retrouvé à cette adresse : `https://gitlab.rezometz.org/klafyvel/drone-rigide`. Les codes ont été testés sous Ubuntu 18.04 avec la version de ROS "melodic" sous une architecture 64 bits. Si à l'avenir le dépôt se trouvait indisponible, les auteurs peuvent être contactés :

Joanne Steiner `joannesteiner@hotmail.fr`

Hugo Levy–Falk `me@klafyvel.me`

Annexe A

Problèmes connus

Lors de notre projet, nous avons rencontré quelques difficultés liés à des problèmes rencontrés avec le matériel :
— les batteries sont peu performantes, elles ont une autonomie très faible (de l'ordre de la dizaine de minutes maximum lorsque le drone est en vol) ce qui implique d'avoir à redémarrer le drone de façon répétée ;
— sur les quatre drones présents dans la Smart Room, trois voient flou ;
— il arrive que l'image de la caméra se fige, il faut alors redémarrer le drone.

Annexe B

Installation de ROS et du projet

Tout d'abord, il faut installer ROS sur votre machine [1]. Afin d'installer ROS et les modules nécessaires à l'utilisation du projet, nous allons nous référer au tutoriel de M. Frezza-Buet disponible sur son site.

Suivez le lien suivant http ://wiki.ros.org/ROS/Installation afin d'installer ROS sur votre machine. Lors de notre projet, nous avons utilisé la version suivante : ROS Melodic Morenia qui était alors la plus récente disponible.

Une fois ROS installé, installez les "catkins tools" qui permettront de créer des espaces de travail ROS

```
sudo apt install python-catkin-tools
```

Créez un workspace sur lequel vous enregistrerez le projet :

```
cd ~
mkdir -p drone-rigide/src
cd drone-rigide/src
```

Procédez à l'installation de bebop_autonomy :

```
sudo apt-get install build-essential python-rosdep
```

Récupérez également le dépôt teleop-tools, nécessaire à l'utilisation de bebop_autonomy :

```
git clone https://github.com/ros-teleop/teleop_tools
```

Procédez à l'installation de Parrot-ARSDK et indiquez enfin le chemin de Parrot-ARSDK pour bebop_autonomy :

```
sudo apt install ros-melodic-parrot-arsdk
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/ros/melodic/lib/parrot_arsdk
git clone https://github.com/AutonomyLab/bebop_autonomy.git bebop_autonomy
```

Pensez à copier le répertoire `bebop_driver` qui se trouve dans `bebop_autonomy` dans le répertoire `src`.

```
cp -r bebop_autonomy/bebop_driver ./
```

Attention à la version de ROS qui n'est pas forcément melodic.

Installez vq2

```
https://github.com/HerveFrezza-Buet/vq2
```

Ramenez les modules du projet sur votre workspace :

```
git clone https://gitlab.rezometz.org/klafyvel/drone-rigide
```

Récupérez le dépôt Gitlab de M. Frezza-Buet suivant :

```
git clone https://github.com/HerveFrezza-Buet/demo-teleop
git clone https://github.com/HerveFrezza-Buet/vqimg
```

Ce module permet de contrôler le drone à l'aide du clavier et de reprendre la main sur ce dernier à tout instant.

Il faut maintenant procéder à la compilation. Tout d'abord, indiquez sous quelle version de ROS vous travaillez (à adapter à votre version) :

```
source /opt/ros/melodic/setup.bash
```

Puis compilez votre workspace :

```
cd ~/drone-rigide
catkin build
```

Désormais, à chaque fois que vous ouvrirez un nouveau terminal pour travailler sous ROS, il faudra indiquer au terminal sous quel version de ROS vous voulez travailler et dans quel workspace. Pour cela, écrivez la commande suivante :

```
source ~/drone-rigide/devel/setup.bash
```

Ouvrez un terminal et lancez :

```
roscore
```

Dans un nouveau terminal, placez-vous dans drone-rigide/src :

```
cd drone-rigide/src
```

Et lancez le fichier `simple_loop.launch` par l'instruction suivante :

```
roslaunch detect_targets/launch/simple_loop.launch
```

Annexe C

Export du relevé de position

C.1 Relevé

ROS permet, grâce à la commande `rosbag` d'exporter les données publiées sur un topic ROS pour les jouer plus tard. Pour cela on peut utiliser la commande :

```
rosbag record <topic> -O record.bag
```

où `<topic>` est le topic à enregistrer.
Après avoir lancé `roscore`, on enregistre la sortie dans un format que le script de l'annexe D accepte.

```
rostopic echo <topic> | python3 parse_topic.py
```

Le script `parse_topic.py` accepte plusieurs arguments.

```
```

On peut alors, dans un autre terminal, jouer l'enregistrement avec la commande suivante.

```
rosbag play record.bag
```

Enfin, on peut arrêter le script `parse_topic.py` en pressant `Ctrl + C`.

C.2 Script `parse_topic.py`

```python
#! /usr/bin/python3

import yaml
import sys
import time

import click


@click.command()
@click.option('--output', default="output.csv", help='Output file')
@click.option('--time/--no-time', 'use_time', default=False, help='Add the number of seconds')
@click.option('--field', default="data", help='YAML field to store')
```

```
14 def main(output, use_time, field):
15     with open(output, 'w') as f:
16         s = ""
17         last_line = ""
18         try:
19             for line in sys.stdin:
20                 if line == "---\n":
21                     v = yaml.load(s)[field]
22                     print(v)
23                     if use_time:
24                         f.write(str(time.time()) + ',' + str(v) + "\n")
25                     else:
26                         f.write(str(v)+"\n")
27                     s = ""
28                 else:
29                     s += line + '\n'
30         except KeyboardInterrupt:
31             print("Output saved to " + output)
32 
33 if __name__ == '__main__':
34     main()
```

Annexe D

Script `test_filter.jl`

Ce script teste différents filtres de Savitsky-Golay à l'aide du langage Julia[5]. Outre le fait qu'il illustre la souplesse et la simplicité d'utilisation du langage, il permet de donner un exemple de calcul des coefficients du filtre.

```
using Plots
using DSP
using CSV

pyplot()

# Parameters
filename = joinpath(@__DIR__, "data", "walk.csv") # input file
h = 1/22 # sample time
orders = [
    (title="quadratique", order=2, sizes=5:2:11),
    (title="cubique", order=3, sizes=5:2:11),
    (title="big_quadra", order=2, sizes=11:10:51)
]


function savgol(size::Int64, poly_order::Int64, deriv::Int64=0, delta::Float64=1.0,
 ↪  conv::Bool=false)
    half_size, rem = divrem(size, 2)
    if rem == 0
        throw(ArgumentError("size must be odd."))
    end
    M = [-half_size:half_size;] .^ [0:poly_order;]';
    y = zeros(poly_order+1)';
    y[deriv+1] = factorial(deriv) / delta^deriv;
    scal = y*inv(M'*M)*M'
    if conv
        scal = scal[end:-1:1]
    end
    scal
end

error = (CSV.read(filename, header=false) |> Matrix{Float64})[:,2]
```

```
33
34
35  for order in orders
36      plot(
37          0:h:(length(error)-1)*h,
38          error,
39          label="erreur mesurée",
40          layout=(2,1),
41          subplot=1,
42          title="Position",
43          xrotation=60,
44          xlabel="Temps (s)",
45          ylabel="Position (m)",
46          reuse=false,
47          size=(1000, 600)
48      )
49      plot!(
50          subplot=2,
51          title="Vitesse mesurée",
52          xrotation=60,
53          xlabel="Temps (s)",
54          ylabel="Vitesse (\$m.s^{-1}\$)"
55      )
56      for size in order.sizes
57          filter = savgol(size, order.order, 1, h, true)
58          speed = conv(filter, error)[size:end-size]
59          plot!(
60              size*h/2:h:size*h/2+(length(speed)-1)*h,
61              speed,
62              label=string(order.title, " ", size, " points"),
63              subplot=2
64          )
65      end
66      savefig(joinpath(@__DIR__, "results", string("mesure_vitesse_", order.title, ".eps")))
67  end
68  show()
```

Annexe E

Génération des images afin de vérifier le script pour trouver la cible, script `test_find_targets.py`

```
import matplotlib.pyplot as pl
from matplotlib.patches import Rectangle

from find_targets import find_targets, normalize_coordinates

ax = pl.subplot(211)
img = pl.imread('image.jpeg')


H, L, R, objects = find_targets(img, return_slices=True)

for o in objects:
    x, y = o
    r = Rectangle((y.start, x.start), y.stop-y.start, x.stop-x.start,
    ↪  linewidth=1,edgecolor='r',facecolor='none')
    ax.add_patch(r)
ax.imshow(img)
ax.plot([H[0], L[0], R[0]], [H[1], L[1], R[1]], 'o', color='red')

ax = pl.subplot(212)

w = len(img[0])
h = len(img)

H = normalize_coordinates(H, w, h)
L = normalize_coordinates(L, w, h)
R = normalize_coordinates(R, w, h)
ax.plot([H[0], L[0], R[0]], [H[1], L[1], R[1]], 'o', color='red')
ax.grid()
ax.set_title("Positions normalisées")
```

```
30
31
32 pl.show()
```

Annexe F

Utilisation et génération des fichiers de configuration

À plusieurs reprises lors de notre projet, nous avons eu à utiliser des fichiers de configuration.

Ce type de fichier nous a permis de définir des paramètres et de générer une fenêtre dans laquelle nous pouvons les régler en direct. Nos fichiers de configuration débutent par :

```
#!/usr/bin/env python
PACKAGE = "detect_targets"

from dynamic_reconfigure.parameter_generator_catkin import *

gen = ParameterGenerator()
```

Nous allons donc utiliser dynamic_reconfigure afin de créer un générateur de paramètres. Dans ce fichier, l'ajout d'un paramètre dans le générateur se déroulera toujours de la même façon :

```
    gen.add("name",type,0,"description",default value, min, max)
```

Cela nous a permis de générer de nouveaux paramètres et de les régler à la main tout au long du projet.

Bibliographie

[1] Documentation ros. http://wiki.ros.org/.

[2] Documentation scipy. https://docs.scipy.org/doc/.

[3] Fotogallery | Parrot Bebop Drone. https://www.thedigeon.com/it/robot/fotogallery-parrot-bebop-drone.html.

[4] Gitlab du projet initial.

[5] Langage julia. https://julialang.org.

[6] Pi controller design (ziegler-nichols) and unit step response graphing for 4th order system. https://electronics.stackexchange.com/questions/396028/pi-controller-design-ziegler-nichols-and-unit-step-response-graphing-for-4th-o.

[7] Reading from bebop. https://bebop-autonomy.readthedocs.io/en/latest/reading.html.

[8] Savitzky-golay filter. https://en.wikipedia.org/wiki/Savitzky\%E2\%80\%93Golay_filter.

[9] Ziegler-nichols tuning rules for pid.