Sûreté de fonctionnement de véhicules autonomes à base de méthodes ensemblistes

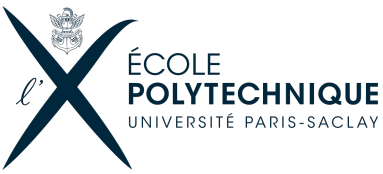
Application aux drones multirotors

Par

FRANCK DJEUMOU

Rapport final de Stage de fin d'études: ISAE-SUPAERO - LIX.

Avril - Septembre 2018



Sous la supervision de :

ERIC GOUBAULT
goubault@lix.polytechnique.fr

SYLVIE PUTOT
putot@lix.polytechnique.fr

JÉRÔME HUGUES
Jerome.HUGUES@isae.fr

Abstract

De manière générale, la sûreté de fonctionnement est une notion générique qui mesure la qualité de service délivrée par un système, de manière à ce que les utilisateurs aient en lui une confiance justifiée. Cette confiance justifiée s'obtient à travers une analyse qualitative et quantitative des différentes propriétés du service délivré par le système. Nous allons plus portés notre attention sur la sûreté de fonctionnement de véhicules autonomes du type quadricoptères. Comment assure t'on de manière offline et online qu'un tel système a t-il les bonnes propriétés ? Par prouver que ce système a les bonnes propriétés, nous sous-entendons pour l'instant faire de la preuve formelle sur les algorithmes de contrôle bas niveau embarqués avec l'auto-pilote d'un tel système. Dans la littérature, Il existe de nombreux outils permettant de synthétiser des "contrôleurs" ou des algorithmes du type haut niveau prouvés par construction de respecter certaines spécifications représentées de manière formelles (logique temporelle, etc...). L'idée ici étant que toutes les fonctionnalités du type haut niveau dépendent fortement des algorithmes de contrôle bas niveau implémentés. Et donc même si de la preuve formelle peut être effectuée sur les fonctionnalités haut niveau, tant qu'il n'y a pas un minimum de garantie sur les algorithmes de contrôle on ne peut rien garantir quant au comportement global du système.

Notre démarche s'inscrit donc dans la logique de prouver d'une part de manière offline et d'autre part en temps réels que l'algorithme de contrôle (contrôle en position, en attitude etc...) en cours d'exécution respecte bien certaines propriétés de performances spécifiées lors de la conception du problème. Ceci requiert notamment une étude approfondie de la dynamique du système en question et des incertitudes sur sa modélisation. Étant donné ce modèle et ses incertitudes, nous spécifions aussi des incertitudes sur le comportement de l'environnement et à partir de là, grâce aux méthodes de sur-approximation et de sous-approximation (Intégration garantie à base de modèle de Taylor), on obtient les trajectoires/traces possibles sur un horizon de temps fini du système. Ces trajectoires nous permettront de conclure si le système se dirige vers un état sûr et le cas échéant, nous appliquerons des corrections pour atteindre cet état sûr. Les résultats obtenus au cours de ce stage comprennent d'une première part l'implémentation d'un simulateur opérationnel pour la plateforme de drone Crazyflie, d'autre part le développement d'un outil de vérification en temps réel suffisamment léger et efficace qui peut s'embarquer sur une plateforme du type quadrotors et qui permettrait de vérifier certaines propriétés au niveau du contrôle et d'appliquer des corrections si possible.

TABLE DES MATIÈRES

List of Figures

Figure **Page**

CHAPTER 1

INTRODUCTION

Dans cette introduction, les objectifs du stage sont détaillés à la suite de la présentation du laboratoire et de l'équipe dans laquelle j'ai évolué.

1.1 Présentation du laboratoire et de l'équipe

1.1.1 Le Laboratoire Informatique de l'Ecole polytechnique (LIX)

Le **LIX** encore connu sous l'appellation du Laboratoire Informatique de l'X est le laboratoire principal de recherche informatique de l'Ecole polytechnique. Le LIX est un laboratoire de recherche mixte sous la tutelle de deux établissements : l'Ecole polytechnique et le Centre National de la Recherche Scientifique (CNRS). Le laboratoire se situe sur le plateau de Saclay, et de part une participation majeure de l'Ecole polytechnique et du CNRS dans le développement d'initiatives scientifiques dans et autour du plateau de Saclay, le LIX intervient donc aussi naturellement dans un grand nombre d'initiatives scientifiques et technologiques ayant lieu sur le plateau de Saclay.

Le LIX entretient des relations très étroites et actives avec **INRIA**. En particulier, plusieurs équipes du LIX sont des EPCs (Equipes-projets commun) Inria : Comète, Parsifal, Crypto (Grace) et Amib. En outre, depuis septembre 2012, le LIX occupe le bâtiment Alan Turing, qui est en co-propriété avec Inria. Entre autre, le LIX fait partie des laboratoires du plateau de Saclay participants à la création du projet ambitieux Université Paris-Saclay dont le but est de rivaliser avec les plus prestigieuses universités au monde.

Au cours des dernières années, le LIX s'est attaché à développer de nombreuses collaborations avec le monde industriel. Outre Thalès, nous pouvons citer parmi les grands partenaires internationaux Microsoft Research, Hitachi Labs, et la NASA. De plus, le LIX abrite la Chaire «

Systèmes Industriels Complexes « financée par Thalès. À tous ces titres, il est devenu un acteur académique essentiel au sein du pôle de compétitivité System@tic

Enfin, la recherche à LIX est organisée autour de la notion d'équipe de recherche. Une équipe combine des chercheurs, des professeurs, des doctorants, des post-docs etc... autour d'un projet scientifique spécifique. La plupart des équipes sont des équipes-projet communes avec l'INRIA. Les trois principales thématiques dans le laboratoire sont :

- Algorithmiques, Combinatoires et modèles
- Systèmes distribués et Sécurité
- Calcul symbolique et Preuves formelles

1.1.2 Les équipes du LIX

Dans la thématique de l'algorithmique, combinatoires et modèles, on peut citer entre autres : **AlCo** qui travaille sur des problèmes d'algorithmique, d'optimisation et de recherche opérationnelle. **AMIB** qui est un groupe de recherche en bio-informatique muni d'un fort intérêt pour les aspects moléculaires de l'organisation cellulaire, ainsi que d'une forte appétence pour les Acides RiboNucléiques (ARN). **COMBI** s'intéresse aux liens entre combinatoire et géométrie et applique des méthodes algorithmiques et d'énumération à des problèmes issus de contextes variés, allant de la physique statistique à la compression de données ou la topologie énumérative. **DASCIM** travaille dans le domaine de la data science avec comme priorité les graphes et le text mining sur des bases de données à grande échelle. **STREAM** travaille principalement sur les problèmes de modélisation géométrique et animation 3D. **CEDAR** étudie la gestion et l'analyse des données pour des données complexes à très grande échelle et possiblement doté d'une sémantique riche.

Dans la thématique des systèmes distribués et Sécurité, on peut citer entre autres : **COMETE** qui focalise ses recherches sur la conception, l'implémentation et les applications de langages formels pour les systèmes distribués, mobiles et sécurisés. **GRACE** sont des experts dans les mathématiques et l'algorithmique de la cryptographie et de la théorie des codes. **NETWORKS** se concentre aussi bien sur les algorithmes et protocoles de routage que sur l'architecture de réseaux, très grands, très dynamiques ou contraints.

Enfin, dans la thématique du calcul symbolique et des preuves formelles, on peut citer entre autres : **MAX** qui se concentre sur l'efficacité et la robustesse des algorithmes de calcul symbolique et des outils pour l'algèbre, le calcul différentiel et la géométrie. **PARSIFAL** qui développe et exploite la théorie des preuves. **TYPICAL** développe et exploite la théorie des types. **COSYNUS**, qui est l'équipe dans laquelle j'effectuais mon stage, travaille sur la sémantique et l'analyse statique des systèmes logiciels, éventuellement distribués, hybrides et cyber-physiques.

1.1.3 Présentation de l'équipe Cosynus

L'équipe **Cosynus** dans laquelle j'effectuais mon stage est composée de 15 personnes dont 6 chercheurs permanents, 6 doctorants et 3 post-doctorants. Mon stage était co-encadré par Eric Goubault et Sylvie Putot, tous les deux membres de l'équipe Cosynus. Les recherches d'Eric Goubault se concentrent sur la vérification des programmes numériques et systèmes, sur les méthodes géométriques pour la concurrence, et sur la topologie algébrique dirigée et plus particulièrement, ses applications à l'étude du comportement des programmes concurrents. Les recherches de Sylvie Putot se basent aussi sur la vérification des programmes numériques et systèmes, en particulier sur l'analyse des systèmes hybrides ou plus généralement des systèmes cyber-physiques. Les sujets d'intérêts du groupe sont variés mais convergent tous sur de la preuve formelle de fonctionnement des programmes/systèmes informatiques(ou encore systèmes hybrides). Pour tout ce qui est expérimentation, le groupe possède une salle en cours de construction qui devra contenir un système de localisation, des véhicules du type drone (crazyflie, Turtlebot) et du matériel permettant de faire tout genre d'expériences avec ce type de véhicules. Cette salle sera aussi un point d'entrée pour des démonstrations à des visiteurs et aussi pour les étudiants de l'X qui suivent des cours faisant intervenir de la preuve formelle.

1.2 Les objectifs du stage

L'objectif premier du stage était de montrer d'un point de vue théorique et pratique q'un algorithme de planification et de contrôle d'un drone fait ce qu'il doit faire sans le mettre en danger. Cela demandait de modéliser la dynamique du drone et de ses incertitudes, de simuler (de manière garantie en particulier) les trajectoires du drone, et si possible d'expérimenter l'algorithme de contrôle et de planification sur de vrais drones. Entre autres, on s'intéresse aussi à développer des algorithmes de vérification suffisamment légers et efficaces pour être embarqués et qui interagiront avec le contrôle et la planification pour garantir des bonnes propriétés sur le système. Ces objectifs se sont raffinés au fil du temps et le stage s'est finalement articulé en trois grandes parties :

1.2.1 L'étude et la synthèse d'un simulateur pour le quadrotor Crazyflie de Bitcraze

Dans cette partie, on s'intéressait plus particulièrement au quadrotor Crazyflie, sa dynamique et les incertitudes concernant quelques de ses paramètres physiques (Matrice d'inertie, masse, etc...). Puis dans l'optique de pouvoir faire de la vérification embarquée et de pouvoir tester les algorithmes de contrôle 'sûr' qui auront été synthétisés, il s'est rapidement posé le problème d'avoir une plateforme de simulation assez réaliste dont **l'utilisation serait absolument similaire à 1 ou 2 commandes près de l'utilisation du drone réel**. L'intérêt ici étant bien évidemment de pouvoir faire des tests avant l'embarquement sur le drone réel pour éviter

toute surprise. Enfin, la salle d'expérimentation avec le système de positionnement indoor a été construite pour pouvoir faire des tests dans le monde réel histoire de valider le comportement de la simulation.

1.2.2 La vérification à base des méthodes de sur/sous approximation

Une fois le simulateur implémenté et testé, on s'est concentré sur la partie vérification en soi. Pour cela, étant donné la dynamique du crazyflie développée dans la première partie, une étude bibliographique a été faite sur la plupart des domaines numériques abstraits et sur les méthodes de sur/sous approximation à base de ces domaines abstraits. Ces deux méthodes seront développées plus tard et permettent notamment l'analyse des états atteignables par le système au cours de son évolution. Ces états atteignables permettent de conclure sur la sûreté de la manœuvre en cours d'exécution.

1.2.3 Implémentation et test de la vérification en temps réel

Une fois les méthodes de vérifications offline présentées, il était question d'implémenter une version efficace et légère des méthodes ci-dessus que l'on pourrait directement embarquée sur la plateforme de Crazyflie et qui interagira directement avec le contrôle pour vérifier la cohérence de la manœuvre à effectuer. La difficulté ici étant de trouver le bon domaine abstrait qui permettrait une vérification rapide, une utilisation bornée en mémoire et dont l'intégration avec les tâches temps réels du Crazyflie n'affecterait pas l'ordonnançabilité du jeu de tâches complet du système. Notons que le rendu final actuel ne permet que la vérification dans l'environnement de simulation. Cependant le simulateur a été construit de manière à ce que le passage dans le monde réel soit identique.

CHAPTER 2

MODÉLISATION ET ÉTUDE DU CRAZYFLIE

Dans ce chapitre, nous nous intéresserons en particulier au Crazyflie, à la modélisation de sa dynamique et à l'estimation des valeurs des paramètres physiques essentiels intervenants dans ses équations de la dynamique. Nous présenterons aussi rapidement ses différents algorithmes de contrôle bas niveau qui seront exploités plus tard par la partie vérification et enfin des résultats de vols obtenus dans la salle d'expérimentation avec son système de localisation indoor.

2.1 Le design de Crazyflie

Le Crazyflie 2.0 est à la base un mini quadricoptère (ils incluent la possibilté de l'étendre pour un plus gros quadricoptère) crée par Bitcraze. L'idée initiale derrière Crazyflie était de créer une plateforme facilement assemblable et dont le but serait de faciliter les expériences dans le milieu de la recherche. Ce qui n'est généralement pas le cas pour les autres drones existants dont la documentation est parfois inexistante ou encore très compliquée à suivre et cela requiert beaucoup de travail supplémentaire de la part des chercheurs pour monter un banc d'essai. De plus, du fait de sa petite taille, si une pièce devient défaillante, il est généralement facile dans la remplacer rapidement. Ce qui est assez pratique car la plupart du temps, ce qui se passe c'est qu'il faut recommander les pièces et attendre un temps non négligeable avant de pouvoir reprendre ses expériences. L'autre avantage de cette plateforme est que du fait de sa petite taille, on peut envisager des manœuvres risquées sans pour autant craindre de casser complètement son drone. Comme on le verra plus tard, ce drone vient avec son système de positionnement en intérieur (LPS) pas très couteux et qui se met en place relativement facilement contrairement aux systèmes beaucoup plus couteux tel que Optitrack ou Vicon. Cependant la précision de ce système est faible comparée à par exemple Optitrack ou Vicon. Dans les lignes qui suivent, nous décriront quelques caractéristiques du Crazyflie 2.0.

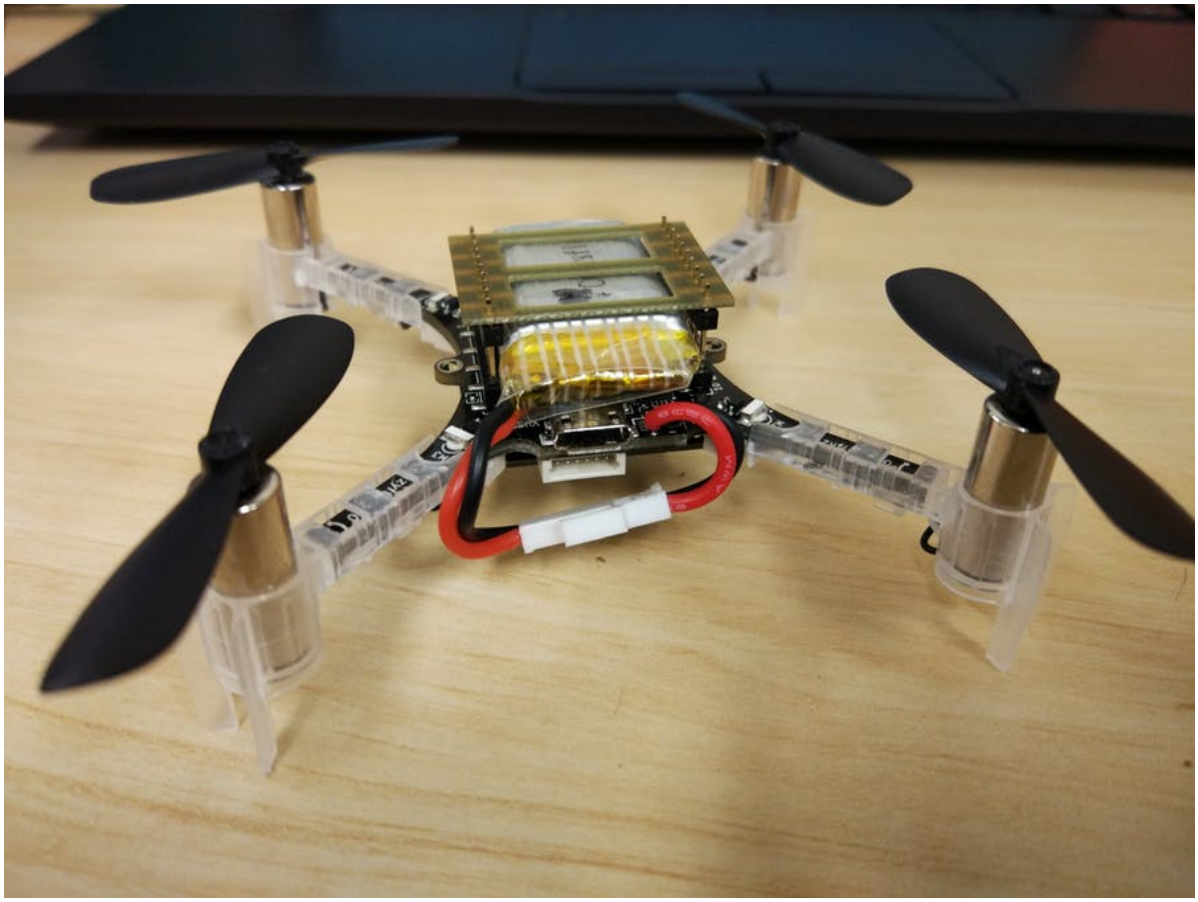


Figure 2.1: Visuel d'un Crazyflie 2.0

2.1.1 Caractéristiques du Crazyflie 2.0

Dans cette partie, nous allons discuter des caractéristiques du Crazyflie 2.0. Par caractéristiques, on sous-entend tous les paramètres nécessaires pour modéliser la dynamique d'un tel système. Ces informations seront primordiaux pour la suite de ce rapport.

2.1.1.1 Masse, Batterie et structure du Crazyflie 2.0

Comme indiqué plus tôt, le Crazyflie 2.0 est un mini quadricoptère pouvant se tenir dans une main. **Sa masse est de 27g** et sa configuration de base est la **configuration X**. Du fait de sa faible masse, la batterie embarquée est moins puissante : **LiPo 240mAh** et ne permet qu'en moyenne **7min de vol**. Concernant les dimensions du Crazyflie 2.0, on a **une distance moteur-moteur = 92mm** et la **Hauteur = 29mm**. Le temps de recharge lorsque la batterie est vide est en moyenne de **40min** et le maximum de payload recommandé au décollage est de **15g**.

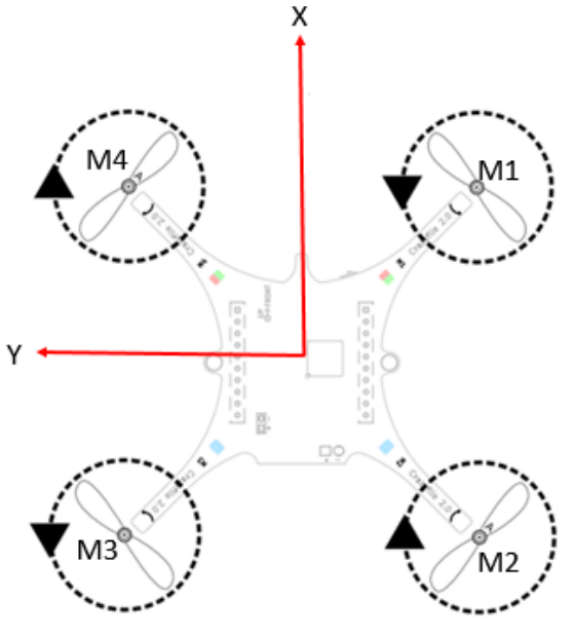


Figure 2.2: Geometrie du crazyflie 2.0

2.1.1.2 Matrice d'inertie

Pour la partie vérification, il est très important de pouvoir estimer cette valeur et d'avoir des incertitudes sur cette valeur. Ces incertitudes seront utilisées dans les calculs de sur et sous approximation. Dans notre situation, beaucoup d'articles font des approximations du type box et utilisent ensuite les dimensions du crazyflie données ci-dessus pour estimer les coefficients de la matrice d'inertie. Cependant, dans l'article [1], on peut retrouver des procédés expérimentaux qui permettent de déterminer ces coefficients avec une incertitude donnée. On a donc extrait de cet article (qui applique ces expériences sur le Crazyflie en particulier) la valeur de la Matrice d'inertie :

$$\boldsymbol{I_{CF}} = \begin{pmatrix} 16.571710 & 0.830806 & 0.718277 \\ 0.830806 & 16.655602 & 1.800197 \\ 0.718277 & 1.800197 & 29.261652 \end{pmatrix} \cdot 10^{-6}\mathrm{kg} \cdot \mathrm{m}^2$$

Figure 2.3: Matrice d'inertie du Crazyflie 2.0

2.1.1.3 Caractéristiques des capteurs inertiels par défaut

Par défaut, le Crazyflie 2.0 possède un capteur du type **MPU-9250** qui est un micro contrôleur possédant à la fois un gyroscope, un accéléromètre et un magnétomètre à trois axes. En plus des

capteurs cités, il possède aussi un capteur de pression à haute précision (**LPS25H**). Nous nous intéressons aux grandeurs suivantes d'un capteur du type capteurs inertiels :

- Le bruit blanc du capteur
- La marche aléatoire du capteur
- La fréquence d'échantillonnage du capteur

Pourquoi ces grandeurs ? Tout simplement parce qu'elles permettent de créer un modèle pour ce type de capteur et ainsi nous permettrons plus tard de simuler le comportement du capteur.

En étudiant la fiche technique de ces capteurs, on retrouve souvent des références à ces grandeurs sous d'autres appellations et souvent pas. Donc étant donné un capteur, le tout est de pouvoir extraire de la fiche technique du capteur ces informations. Le cas échéant, on peut utiliser la méthode de la Variance d'Allan (ou en anglais *Allan Standard Deviation*) pour estimer les grandeurs présentées ci-dessus. L'idée ici est de collecter un grand nombre de données du capteur réel pendant un intervalle de temps assez long et à intervalle de temps régulier. Cette méthode est développée dans l'article [2] et elle sera résumée et testée sur les capteurs du Crazyflie par la suite.

2.1.1.4 Les moteurs et Hélices

Les moteurs utilisés sur le Crazyflie sont des moteurs du type DC qui pèsent chacun 2.7g. D'un point de vue spécification électrique, on a $K_v = 14000 rmp/V$, La tension nominale $U_{nom} = 4.2V$ et enfin le courant nominal $I_{nom} = 1000mA$. Concernant les dimensions du moteur, ce qui nous importe ici c'est son diamètre $D = 7mm$ et sa longueur/hauteur $L = 16mm$ qui seront des grandeurs nécessaires pour le modèle virtuel de Crazyflie. Enfin, les moteurs du Crazyflie sont vendus en pack et sont assez pas couteux pour des hélices de drone. Ceci facilitant la rechange d'une hélice lors d'une mauvaise manoeuvre. Ces hélices sont longues de **45mm**.

2.1.1.5 Les microcontrôleurs au coeur du Crazyflie

L'architecture système du Crazyflie et ses microcontrôleurs sont décrits sur la fig [2.4]. Le Crazyflie 2.0 possède deux cartes "mères" principales. La première est une STM32F405 (Cortex M4, 168MHz, 192Kb SRAM, 1Mb FLASH) qui contient le code source de bitcraze et gère les communications avec les capteurs et actuateurs du Crazyflie. La deuxième est la nRF51822 qui s'occupe de la communication radio et de la gestion de l'alimentation du reste du circuit. La gestion de plusieurs MCU en même temps étant compliquée, la solution adoptée par bitcraze fut la suivante concernant le Crazyflie 2.0 :

- La logique ON/OFF, la gestion de l'energie pour le reste du système (STM32, capteurs etc...), La recharge de la batterie et la mesure du voltage, la communication radio en mode master, le bootloader, sont tous gérés uniquement par la nRF51.

- La STM32 est responsable du reste des tâches. Entre autres, La lecture des capteurs et le contrôle des moteurs, le contrôleur de vol, La télémétrie et les configurations personnelles de l'utilisateur.

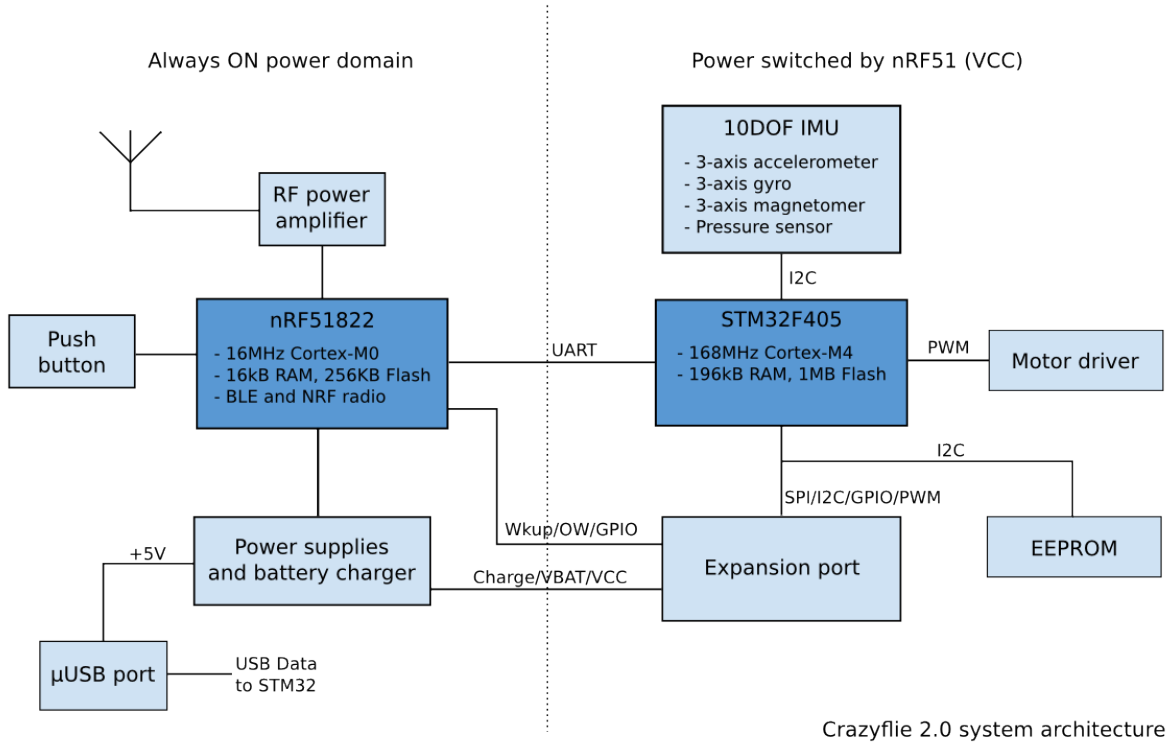


Figure 2.4: Architecture du Crazyflie 2.0

Plus d'informations seront données sur la communication lorsque l'on discutera du contrôle haut niveau du Crazyflie.

2.1.2 Les cartes d'extensions

L'avantage avec le Crazyflie c'est qu'il est possible de le personnaliser autant qu'on le souhaite grâce à ses cartes d'extensions. Par exemple pour pouvoir faire de la localisation indoor du Crazyflie avec le système de localisation LPS conçu par bitcraze, il suffit juste de rajouter au dessus la carte d'extension en question, de recompiler le code et l'utilisation en découle aisément. Il existe aussi une carte d'extension qui permet d'utiliser le crazyflie sur un quadricoptère plus imposant sans grande modifications sur le code source de base ou sur la façon initial d'utiliser l'auto-pilot. Cette fonctionnalité est évidemment super utile mais il est toute fois nécessaire de tuner de nouveau les coefficients PID du contrôleur pour espérer être capable de contrôler décemment le drone. Les différentes extensions pouvant être utilisées sur le Crazyflie sont visible sur le site de Bitcraze [3].

2.1.3 La communication RADIO avec l'extérieur

La figure [2.5] résume le principe de fonctionnement de la communication RADIO du Crazyflie 2.0. Pour communiquer avec l'extérieur, le Crazyflie utilise une communication RADIO dont le master est l'antenne placée sur le Crazyflie (nRF51822). Par conséquent, pour recevoir des messages du Crazyflie, il faut un récepteur compatible avec le protocole particulier RADIO mis en place par l'émetteur Crazyflie. Ceci est possible ici grâce au *Crazyflie PA USB dongle* qui permet cette communication active avec l'auto-pilote. Une fois la connexion établie, il est possible d'interpréter les messages reçus du drone et de le répondre depuis tout type de terminal comme observé sur la figure. Notons aussi que même le flash de la STM32 s'effectue via ce protocole radio.

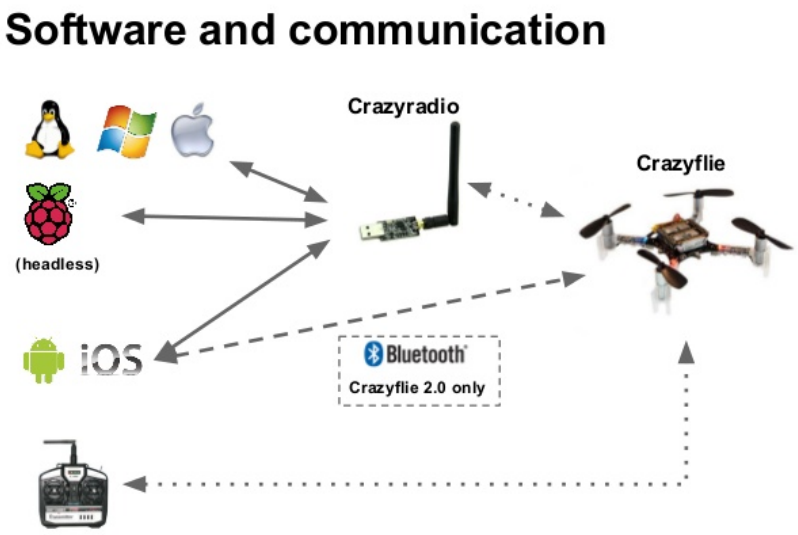


Figure 2.5: Communication avec Crazyflie

La communication se fait de la façon suivante : Le Crazyradio USB envoie un paquet sur un canal et attend un 'ack' (acknowledgement) de la part du Crazyflie. Si le Crazyflie reçoit un paquet sans erreur, il renvoie un paquet du type 'ack'. Si un paquet du type 'ack' est reçu par le Crazyradio USB, ce dernier est autorisé à envoyer le prochain paquet. Et tant qu'aucun paquet 'ack' est reçu, ce dernier enverra automatiquement le même paquet. Ce protocole garantit que tout paquet est éventuellement transmis au Crazyflie à condition que ce dernier soit dans la zone de couverture du Crazyflie radio. Approximativement 80 canaux peuvent être utilisés et chaque paquet est envoyé avec une adresse personnelle. Ceci favorise donc les scenarios du type swarm et par conséquent l'utilisation de plusieurs Crazyflie avec un seul Crazyradio.

D'un point de vue plus pratique, l'enjeu ici sera de comprendre la structure des messages qui sont convertis en paquet et transmis à travers le moyen de la radio. Ces messages constituent

le moyen interne de communication utilisé par le crazyflie et nous verons par la suite comment décoder/encoder ces messages. **Ce système de communication interne a été appelé CRTP (Crazy RealTime Protocol)** par bitcraze et le but plus tard est de convertir ces messages de type CRTP en messages ROS beaucoup plus compréhensibles, facilitant par là les procédures de contrôle haut niveau pour les fanatiques de ROS.

2.2 Le système de positionnement en intérieur

A la suite du développement de Crazyflie, Bitcraze a synthétisé pour le Crazyflie un système de localisation en intérieur pas couteux qu'ils ont appelé *LPS (Loco Positionning System)*. Le LPS est un système de positionnement à base d'onde radio qui utilise principalement la technologie UWB (Ultra WindBand). Le principe est typiquement le même que le GPS, on place quelques ancres (dont on connait les positions) dans une arène et avec un nombre d'ancres suffisants, il est possible de localiser un ancre du même type placé dans le périmètre des premiers (cf 2.6). Des informations détaillées sur la localisation utilisant se système sont fournies dans l'article [4].

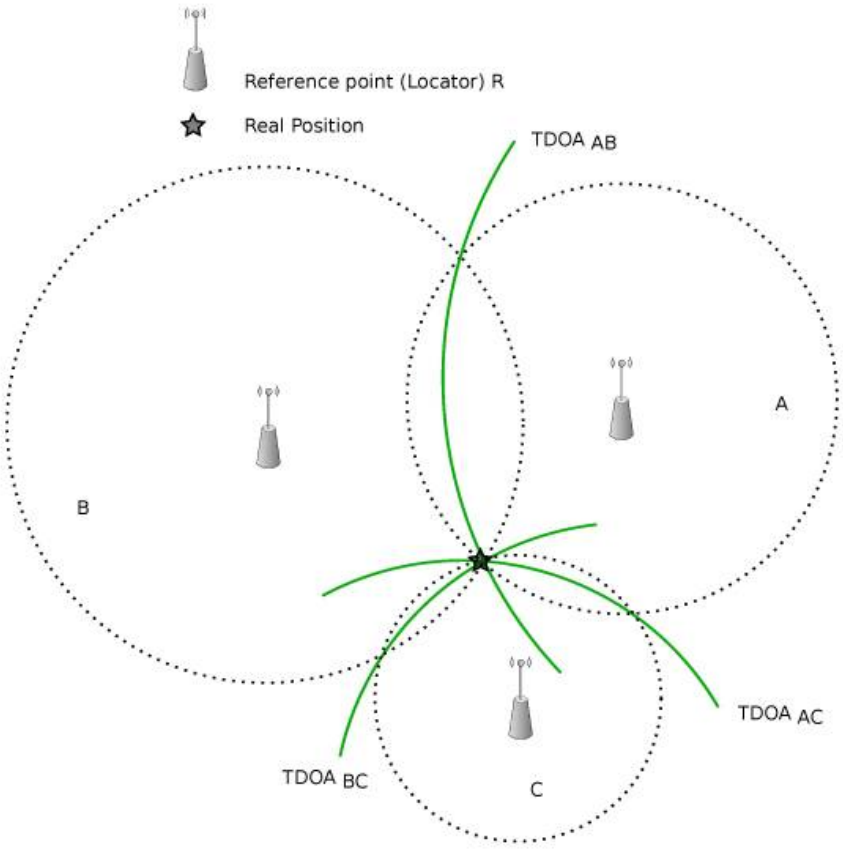


Figure 2.6: Principe de base d'un système de positionnement

Tout le travail effectué ensuite par bitcraze a été de trouver une solution pour combiner les

données obtenues par ces capteurs UWB avec les données de l'accéléromètre, du gyroscope pour faire de l'estimation d'état basée sur le filtre de Kalman pour le drone. Ces résultats ont été développés dans l'article [5].

2.2.1 Installation du système

L'installation du LPS se fait assez rapidement si vous avez les bons outils pour faire des mesures. Le plus d'ancre vous possédez, le mieux c'est pour avoir une bonne estimation de la position et surtout une couverture maximale de l'arène dans laquelle le crazyflie volera. Actuellement bitcraze offre la possibilité d'utiliser deux possibles jeux d'ancre : un ensemble de 6 ancres ou un ensemble de 8 ancres disposés de la façon indiquée sur les figures **??**. Une fois les ancres installés, la procédure de configuration est assez immédiate en suivant les informations sur le site de bitcraze [3]. Il est toute fois nécessaire de spécifier ici que le crazyflie de base ne possède pas la carte d'extension lui permettant d'interagir avec les autres ancres du LPS. Donc il faut s'assurer que cette extension soit fournie avec le crazyflie pour espérer utiliser le LPS.

2.2.2 Performances du système et Comparaison

Dans cette sous-section, on parlera des performances du LPS et on le comparera à d'autres systèmes de localisation en intérieur réputés dans la robotique tel que Optitrack ou encore VICON. Les critères de comparaison seront les suivants :

- **La précision**
- **La disponibilité** : La disponibilité du système de positionnement en terme de temps. Ici on l'exprimera en terme de fréquence maximale de réception de données de position valide .
- **La zone de couverture**
- **L'évolutivité** : A quel point le système assurera t'il le positionnement si l'on augmente des dimensions (la geographie, le nombre d'utilisateurs)
- **Le coût**

Notons que Vicon et Optitrack ne font pas que de la localisation en terme de position. Ces deux systèmes permettent entre autres d'évaluer l'orientation d'un corps. On peut conclure que le LPS est un bon compromis entre coût et précision recherchée. Il a une plus grande couverture que les systèmes du type vision. Un malus concernant le LPS est la fréquence maximale à laquelle on peut extraire des informations de positions. Cette fréquence est un peu plus grande que celle d'un GPS classique mais assez petite pour ne pas favoriser une meilleure estimation de la position dans le filtre de Kalman. Meilleure estimation qui est obtenue grâce aux systèmes tel que Vicon ou Optitrack. Cependant en admettant une bonne configuration du système de positionnement, les résultats obtenus avec LPS sont assez précis pour la stabilisation et le contrôle d'un crazyflie.

2.3 Commande/Contrôle du Crazyflie en mode offboard

Dans cette partie, on se concentrera plus sur le choix que nous avons effectué pour commander le Crazyflie. Rappelons que le mode offboard est un mode de contrôle du drone où les instructions du type haut niveau (contrôle en position, contrôle des angles d'Euler ou encore contrôle des vitesses de rotations suivant les axes x,y et z) sont effectuées à l'extérieur même de l'auto-pilote du drone. Par extérieur je sous-entend un ordinateur à distance ou une unité logique autre que les cartes qui effectuent le contrôle de vol. Le but étant que le mode offboard permette de faire abstraction de toute la logique au niveau des actuateurs et de formuler des *objectifs de mission* sous forme de consigne haut niveau.

2.3.1 Conversion des messages CRTP en message ROS

Dans le protocole de communication radio vu précédemment, il était question que le CRTP était le système de message utilisé au seins de l'auto-pilote pour communiquer entre différents modules. En plus de ça, tout paquet envoyé au travers de la Radio est un message CRTP qui a été converti en paquet pour la transmission. Les messages échangés au travers de la Radio sont classifiés en plusieurs groupes dont les plus importants sont spécifiés ci-dessous :

- *Le Ping* : c'est un message particulier ne contenant aucune payload et qui est envoyé quand la communication est inactive pour récupérer des nouvelles données

- *Message de Console* : Tous les appels à un debug_print (ou une fonction du genre) dans le code source de crazyflie convertit le message à afficher en paquet du type CRTP Console et l'envoie via Radio.

- *Message de Paramètre* : Toutes les grandeurs utiles codées en dur dans le code source de crazyflie peuvent être obtenues ou modifiées en envoyant une requête de ce genre. Typiquement les coefficients PID, l'état initial du filtre de kalmann, l'état des capteurs etc...

- *Message de Log* : C'est le type de message/requête à effectuer si des informations sur les variables d'états du systèmes veulent être obtenues. La requête se crée en spécifiant les variables voulues puis la fréquence à laquelle ses variables doivent être envoyées par le crazyflie.

- *Message de mise à jour de la position du crazyflie* : Lorsque le LPS est utilisé, ce type de message est inutile vu que la position du crazyflie vient directement du LPS. Cependant si un autre système de positionnement est utilisé, un moyen de faire le crazyflie utiliser ces données de positions extérieur est via ce type de message. Typiquement, on l'utilisera dans le simulateur avec comme source de donnée de position le capteur de position dans l'environnement de simulation.

- *Message de Commande haut niveau* : Dans cette appellation, je regroupe tous les messages visant à contrôler le crazyflie tel que des messages de consignes en attitude, en position, en vitesse etc... On détaillera ces fonctionnalités dans la prochaine section de ce rapport.

Étant donné ces informations, le but ici est de convertir tous ces messages en messages/topics ROS pour faciliter encore plus l'interaction avec le crazyflie pour un utilisateur haut niveau. Notons que Bitcraze fournit une interface python pour communiquer avec le crazyflie et effectuer des opérations haut niveau. Cependant, l'utilisateur est forcé à effectuer tout son travail avec Python et par conséquent cela manque le coté évolutif comparé à une utilisation de ROS. Une première implémentation [6] que nous avons beaucoup modifié a été faite dans le but de convertir toutes les actions réalisables via CRTP en actions réalisables via ROS. Pour cela, tout message CRTP du type requête (obtention de paramètres, création de log, requête de décollage etc..) est converti en services ROS. tout message CRTP du type obtention de log (donc à fréquence régulière) est converti en topic ROS du type subscriber (Un autre noeud ne peut que souscrire à ce topic). Enfin tous les messages du type contrôle (envoie de consigne en position, attitude etc...) sont maintenant exécutés en publiant sur des topics ROS spécifiques qui seront convertis en message CRTP approprié et envoyé au crazyflie.

Le graphe [2.8] permet d'observer tous les topics disponibles permettant de communiquer avec le crazyflie. Notons que ces topics sont les mêmes en simulation comme dans le cas du contrôle d'un vrai crazyflie (ce qui facilite d'autant plus la transition).

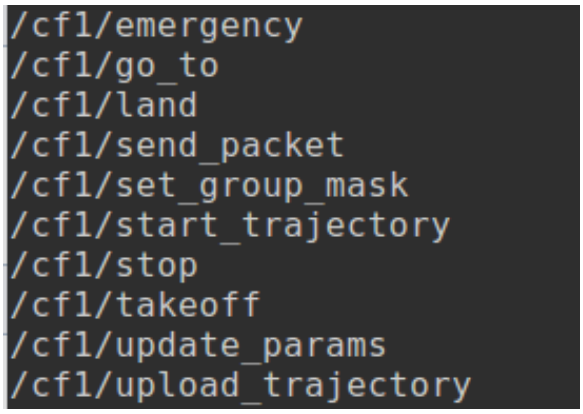


Figure 2.7: Service répliquant le requête du CRTP

La figure [2.7] liste les services fournis par le crazyflie utilisable par un utilisateur en mode offboard. On peut notamment y observer des services tel que le service qui permet de faire décoller ou atterrir le crazyflie. Le service permettant d'envoyer au crazyflie une trajectoire prédéfinie, de modifier des paramètres de vol du crazyflie (du genre coefficients de PID etc...).

Sur le graphe [2.8], il faut faire abstraction du noeud gazebo. Ce graphe a été pris dans le mode simulation. De manière générale, ce noeud représente le crazyflie. Et donc comme on peut

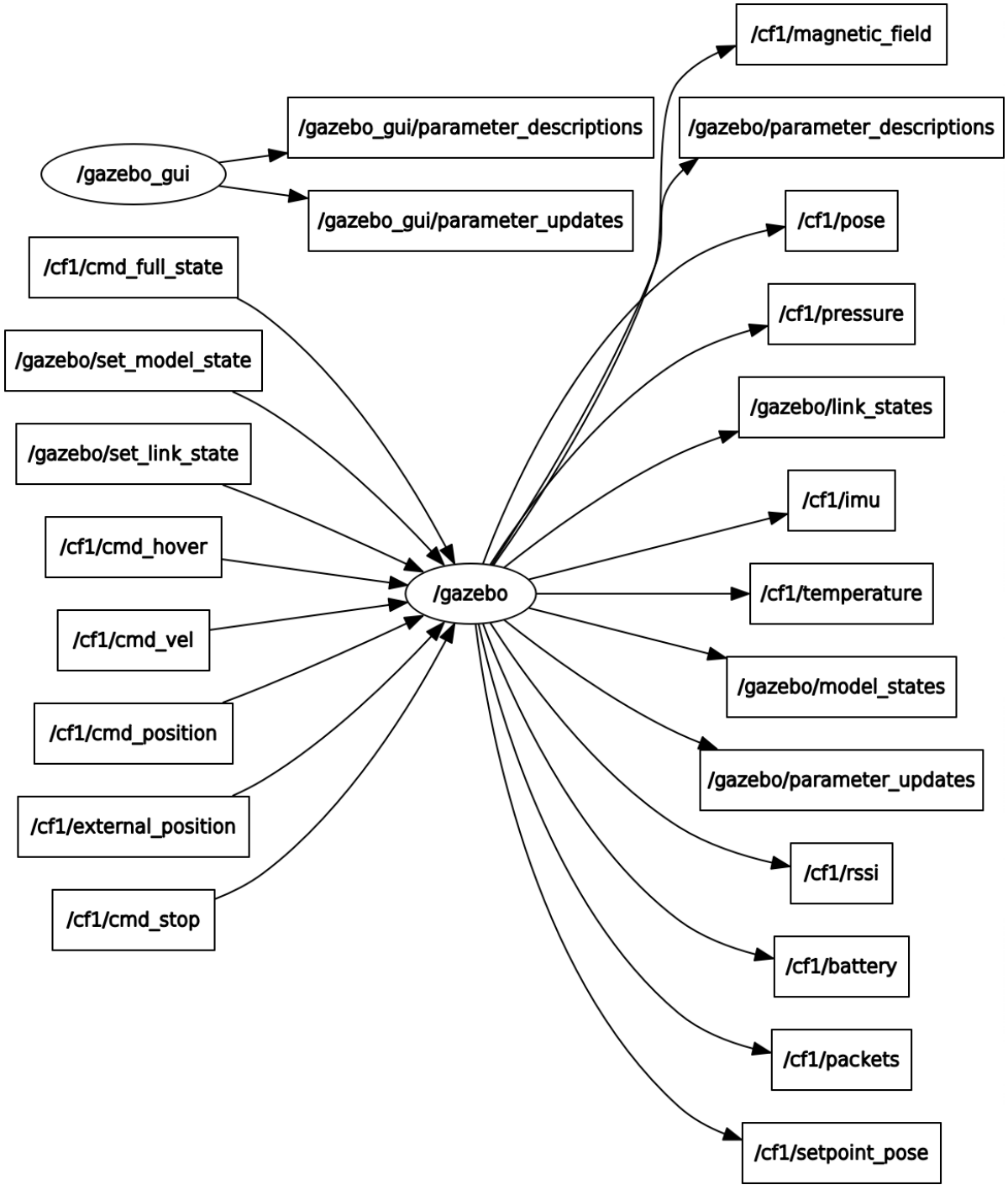


Figure 2.8: Topics ROS répliquants le protocole CRTP

l'observer, il publie sur des topics spécifiques les informations relatives aux données logs qui ont été demandées et reçoit des commandes haut niveau sur d'autres topics.

2.3.2 Expériences sur le crazyflie au LIX

Dans cette sous-section, on va afficher quelques résultats obtenus lors d'une manoeuvre de décollage du crazyflie. Le but étant de montrer ici la stabilité du contrôle du crazyflie avec comme système de positionnement le LPS.

2.3.2.1 Installation du package sim_cf développé tout au long du stage

Tout au long du stage, le package **sim_cf** [7] a été d'une part développé au vue de l'inexistence d'un simulateur pour crazyflie et en plus elle contient un outil de vérification en ligne. D'autre part ce même package peut être utilisé pour contrôler un crazyflie dans la salle d'expérimentation en utilisant les mêmes commandes qu'en simulation. Notons que toutes les informations nécessaires à l'installation et l'utilisation du package sont fournies sur notre github [7]. Entre autres, des exemples de contrôle haut niveau ont été fournies comme point de départ pour utiliser ce package.

2.3.2.2 Quelques résultats

Ici, on va montrer comment le crazyflie réagit à la donnée d'une consigne en position ou encore en attitude. Pour cela on va considérer la manœuvre de décollage qui est facilement exécutable à partir d'un des services mis en ligne par notre package. L'objectif du drone dans ce scenario est d'aller du sol, de décoller jusque 1.5m d'altitude puis d'atterrir après quelques secondes. On trace sur la figure [2.9] la trajectoire suivant l'axe Z du crazyflie dans notre salle d'expérimentation où on a installé un LPS à 6 ancres.

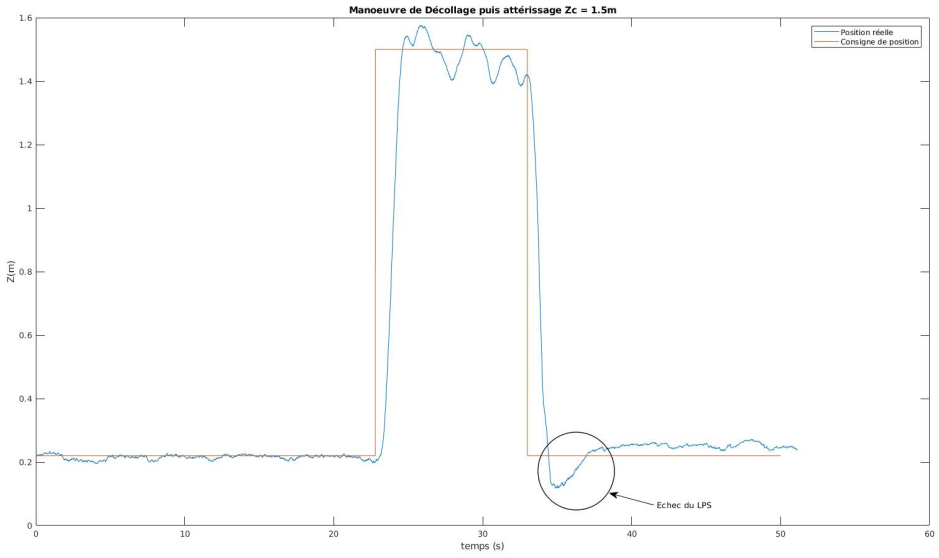


Figure 2.9: Manœuvre de décollage puis d'atterrissage en salle d'expérimentation

On peut observer ci-dessus que le contrôle est satisfaisant et que la réponse du système est rapide. A cause de l'imprécision du LPS, on observe ces oscillations faibles autour de la position désirée. Sur la figure, on observe aussi une partie marquée avec un cercle. Cette partie concerne la manœuvre d'atterrissage, et l'échec de l'estimation de la position vient du fait que dans notre salle d'expérimentation, le système LPS n'arrive pas à bien localiser le crazyflie lorsqu'il est proche du sol. D'où le fait que cette expérience commence à une altitude de 0.21m (qui correspond au crazyfle sur une 'piste' de décollage) au dessus du sol et visible par tous les ancres du LPS.

2.4 Modèle de la dynamique du Crazyflie 2.0

Dans cette dernière partie de ce chapitre, nous allons dériver les équations générales de la dynamique d'un drone et l'appliquer avec les paramètres d'un drone du type crazyflie. Puis nous allons analyser le code source de crazyflie pour savoir quel type de contrôleur est implémenté, comment les sorties des contrôleurs sont transformées en valeur PWM, et Comment convertir ces valeurs PWM en vitesse de rotation qui sera utilisée pour faire une analyse en boucle fermée. Répondre à toutes ces questions sera effectuée dans cette dernière section.

2.4.1 Dynamique d'un quadricoptère

Ici on considère les hypothèses suivantes sur le quadricoptère en question :

- Le drone est un corps solide qui ne peut pas se déformer, par conséquent il est possible d'utiliser les équations de la dynamique d'un corps rigide pour ce drone
- La masse du quadricoptère est constante
- La configuration géométrique du quadricoptère est la configuration X comme obtenue sur la figure [2.2]. Ce détail est important car c'est la configuration prédéfinie du crazyflie.
- On utilise le système de coordonnée ENU (East, North, Up) comme sur la figure [2.10]

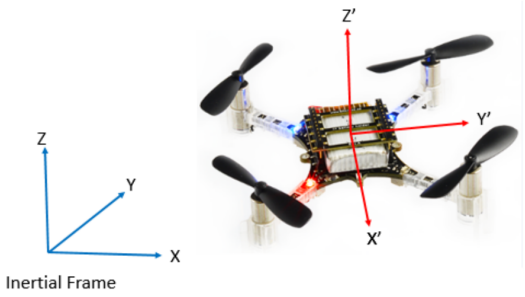


Figure 2.10: Système de coordonnée du crazyflie

Les variables d'états à utiliser dans le modèle du drone seront les suivantes :

- *La position dans le repère inertiel* : $[x \quad y \quad z]^T$
- *La vitesse linéaire du centre de gravité dans le repère lié au quadricoptère par rapport au repère inertiel :* $[u \quad v \quad w]^T$
- *L'orientation du quadricoptère par rapport au repère inertiel* : $[\phi \quad \theta \quad \psi]^T$
- *La vitesse angulaire dans le repère lié au quadricoptère* : $[p \quad q \quad r]^T$ où p est la vitesse de rotation suivant l'axe de roulis, q est la vitesse de rotation suivant l'axe de tangage et r est la vitesse de rotation suivant l'axe de lacet.

L'étude de l'article [8] [9] et de nombreux autres articles dans la littérature sur le modèle dynamique d'un quadricoptère nous ont permis de dériver les équations suivantes :

$$
\begin{cases}
\dot{x} = cos(\theta)cos(\psi)u + (cos(\psi)sin(\theta)sin(\phi) - cos(\phi)sin(\psi))v \\
\qquad +(sin(\phi)sin(\psi) + cos(\phi)cos(\psi)sin(\theta))w \\
\dot{y} = cos(\theta)sin(\psi)u + (cos(\phi)cos(\psi) + sin(\theta)sin(\phi)sin(\psi))v \\
\qquad +(cos(\phi) * sin(\theta)sin(\psi) - cos(\psi)sin(\phi))w \\
\dot{z} = -sin(\theta)u + cos(\theta)sin(\phi)v + cos(\theta)cos(\phi)w \\
\dot{u} = rv - qw + sin(\theta)g \\
\dot{v} = -ru + pw - cos(\theta)sin(\phi)g \\
\dot{w} = qu - pv - cos(\theta)cos(\phi)g + \frac{\boldsymbol{F}}{m} \\
\dot{\phi} = p + cos(\phi)tan(\theta)r + tan(\theta)sin(\phi)q \\
\dot{\theta} = cos(\phi)q - sin(\phi)r \\
\dot{\psi} = \frac{cos(\phi)}{cos(\theta)}r + \frac{sin(\phi)}{cos(\theta)}q \\
\dot{p} = I_p^p p^2 + I_{pq}^p pq + I_{pr}^p pr + I_q^p q^2 + I_{qr}^p qr + I_r^p r^2 + I_{xx}^m \boldsymbol{Mx} + I_{xy}^m \boldsymbol{My} + I_{xz}^m * \boldsymbol{Mz} \\
\dot{q} = I_p^q p^2 + I_{pq}^q pq + I_{pr}^q pr + I_q^q q^2 + I_{qr}^q qr + I_r^q r^2 + I_{xy}^m \boldsymbol{Mx} + I_{yy}^m \boldsymbol{My} + I_{yz}^m * \boldsymbol{Mz} \\
\dot{r} = I_p^r p^2 + I_{pq}^r pq + I_{pr}^r pr + I_q^r q^2 + I_{qr}^r qr + I_r^r r^2 + I_{xz}^m \boldsymbol{Mx} + I_{yz}^m \boldsymbol{My} + I_{zz}^m * \boldsymbol{Mz}
\end{cases}
\tag{2.1}
$$

Où les grandeurs I_*^* sont fonctions de la matrice d'inertie et seront détaillées plus tard. De plus, les entrées du modèle sont les Forces et Moments donnés par les formules suivantes :

$$
\begin{cases}
\boldsymbol{F} & = C_T(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2) \\
\boldsymbol{M_x} & = \frac{L}{\sqrt{2}} C_T(-\omega_1^2 - \omega_2^2 + \omega_3^2 + \omega_4^2) \\
\boldsymbol{M_y} & = \frac{L}{\sqrt{2}} C_T(-\omega_1^2 + \omega_2^2 + \omega_3^2 - \omega_4^2) \\
\boldsymbol{M_z} & = C_D(-\omega_1^2 + \omega_2^2 - \omega_3^2 + \omega_4^2)
\end{cases}
\tag{2.2}
$$

Où les grandeurs $\omega_1, \omega_2, \omega_3, \omega_4$ sont les vitesses de rotations des différents moteurs. Les constantes C_T, C_D sont respectivement les coefficients de poussée et coefficients de moments du

quadricoptère et dépendent de la géométrie, structure et caractéristiques des moteurs et hélices. Enfin L est la longueur du bras du quadricoptère (distance centre de gravité à un des moteurs).

2.4.2 Boucle de contrôle au sein du crazyflie 2.0

Classiquement, Le contrôle d'un quadricoptère est une succession de boucle PID en cascade. Du plus bas au plus haut niveau, on a : Les boucles PID de contrôle pour les vitesses de rotations suivant les axes de roulis, tangage et lacet, ensuite les boucles PID de contrôle pour les angles d'euler (roulis, tangage, lacet), les boucles PID de contrôle pour la vitesse dans le référentiel inertiel, enfin les boucles PID de contrôle pour la position dans le référentiel inertiel. Notons ici que la sortie de la boucle de contrôle au niveau 'i' est utilisée comme consigne dans la boucle de contrôle au niveau 'i-1'. Et pour la boucle au plus bas niveau, sa sortie est directement utilisée et transformée en commande PWM pour les moteurs.

2.4.2.1 Boucle de commande des angles d'euler

Les paramètres pour cette boucle sont nommées de la façon suivante :

- $\boldsymbol{\phi_{sp}}$, $\boldsymbol{\theta_{sp}}$ et $\boldsymbol{\psi_{sp}}$ sont respectivement les consignes d'angles de roulis , lacet et tangage
- $\boldsymbol{\phi}$, $\boldsymbol{\theta}$ et $\boldsymbol{\psi}$ sont respectivement les valeurs actuelles des angles de roulis, lacet et tangages données par les équations de la dynamique
- $\boldsymbol{K_p^r}$, $\boldsymbol{K_d^r}$, $\boldsymbol{K_i^r}$, $\boldsymbol{K_p^p}$, $\boldsymbol{K_d^p}$, $\boldsymbol{K_i^p}$, $\boldsymbol{K_p^y}$,$\boldsymbol{K_d^y}$, $\boldsymbol{K_i^r}$ sont respectivement les coefficients proportionnels, dérivatifs et intégrales du PID pour les angles de roulis, lacet et tangages.
- $\boldsymbol{p_{sp}}$, $\boldsymbol{q_{sp}}$, $\boldsymbol{r_{sp}}$ les consignes de vitesses angulaires qui seront les sorties de cette boucle de contrôle et les entrées de la boucle de contrôle en vitesse de rotation.

Les équations suivantes décrivent la dynamique de la boucle de contrôle en attitude :

$$\begin{cases} p_{sp} = K_p^r * (\phi_{sp} - \phi) + K_d^r * (\dot{\phi_{sp}} - \dot{\phi}) + K_i^r * \int(\phi_{sp} - \phi)dt \\ q_{sp} = K_p^p * (\theta_{sp} - \theta) + K_d^p * (\dot{\theta_{sp}} - \dot{\theta}) + K_i^p * \int(\theta_{sp} - \theta)dt \\ r_{sp} = K_p^y * (\psi_{sp} - \psi) + K_d^y * (\dot{\psi_{sp}} - \dot{\psi}) + K_i^y * \int(\psi_{sp} - \psi)dt \end{cases} \tag{2.3}$$

2.4.2.2 Boucle de commande des vitesses angulaires

Comme stipulé précédemment, les sorties de la boucle de contrôle des angles d'euler seront utilisées comme consigne de cette boucle de contrôle. Les paramètres de cette boucle prolongent les paramètres de la boucle précédente avec les nouvelles grandeurs ci-dessous:

- $\boldsymbol{K_p^{rr}}$, $\boldsymbol{K_d^{rr}}$, $\boldsymbol{K_i^{rr}}$, $\boldsymbol{K_p^{pr}}$, $\boldsymbol{K_d^{pr}}$, $\boldsymbol{K_i^{pr}}$, $\boldsymbol{K_p^{yr}}$,$\boldsymbol{K_d^{yr}}$, $\boldsymbol{K_i^{rr}}$ sont respectivement les coefficients proportionnels, dérivatifs et intégrales de la boucle de PID pour les vitesses angulaires

- $\boldsymbol{cmd_\phi}$, $\boldsymbol{cmd_\theta}$, $\boldsymbol{cmd_\psi}$ sont respectivement les sorties de commande roulis, tangage et lacet qui seront convertis en PWM selon une formule explicitée plus tard

Les équations suivantes décrivent la boucle de contrôle des vitesses angulaires:

$$\begin{cases} cmd_\phi = K_p^{rr} * (p_{sp} - p) + K_d^{rr} * (\dot{p_{sp}} - \dot{p}) + K_i^{rr} * \int (p_{sp} - p) dt \\ cmd_\theta = K_p^{pr} * (q_{sp} - q) + K_d^{pr} * (\dot{q_{sp}} - \dot{q}) + K_i^{pr} * \int (q_{sp} - q) dt \\ cmd_\psi = K_p^{yr} * (r_{sp} - r) + K_d^{yr} * (\dot{r_{sp}} - \dot{r}) + K_i^{yr} * \int (r_{sp} - r) dt \end{cases} \tag{2.4}$$

A la sortie de cette boucle de contrôle, Les valeurs $\boldsymbol{cmd_\phi}$, $\boldsymbol{cmd_\theta}$, $\boldsymbol{cmd_\psi}$ sont contraintes dans une variable de 16 bits (pour PWM). En assumant en plus que le crazyflie utilise sa configuration par défaut, on peut obtenir les valeurs PWM à fournir aux différents moteurs via la relation :

$$\begin{cases} pwm_1 = thrust - 0.5 cmd_\phi - 0.5 cmd_\theta - cmd_\psi \\ pwm_2 = thrust - 0.5 cmd_\phi + 0.5 cmd_\theta + cmd_\psi \\ pwm_3 = thrust + 0.5 cmd_\phi + 0.5 cmd_\theta - cmd_\psi \\ pwm_4 = thrust + 0.5 cmd_\phi - 0.5 cmd_\theta + cmd_\psi \end{cases} \tag{2.5}$$

Où $thrust$ est la poussée désirée. Cette variable peut être obtenu via la sortie *de la boucle en position* (typiquement via la sortie de la position en Z et V_Z) ou peut être réglée manuellement. Sa valeur doit être entre **0** et **65536**. D'après les expériences effectuées dans [1], on a la relation linéaire suivante entre PWM (entier de 16 bits) et vitesse de rotation des moteurs (en rad/s) :

$$\omega_i = C_1 * pwm_i + C_2 \tag{2.6}$$

Où C_1 , C_2 sont aussi fournies dans l'article et sont spécifiques au Crazylfie 2.0. Toutes ces constantes sont résumées dans le tableau [2.11].

Enfin en utilisant les équations [2.2], [2.5], [2.6], on en déduit les équations suivantes (calculs effectués avec l'outil de calcul symbolique de matlab) pour les entrées **Mx , My , Mz , F** permettant le contrôle du Crazyflie en attitude (c'est à dire en consigne en angle de roulis, tangage et lacet) :

$$\begin{cases} Mx = (4C_T C_1^2 d * thrust + 4C_2 C_T d C_1) cmd_\phi - (4C_1^2 C_T d) cmd_\theta cmd_\psi \\ My = (-4C_1^2 C_T d) cmd_\phi * cmd_\psi + (4C_T C_1^2 d * thrust + 4C_2 C_T C_1 d) cmd_\theta \\ Mz = (-2C_1^2 C d) cmd_\phi cmd_\theta + (8C_D C_1^2 * thrust + 8C_2 C_D C_1) cmd_\psi \\ F = C_T C_1^2 cmd_\theta^2 + C_T C_1^2 cmd_\phi^2 + 4C_T C_1^2 cmd_\psi^2 + (4C_T C_1^2) * thrust^2 \\ \qquad +(8C_T C_1 C_2) * thrust + 4C_T C_2^2 \end{cases} \tag{2.7}$$

Où $d = \frac{L}{\sqrt{2}}$

Notons ici que les entrées de ce contrôle sont la poussée ('thrust' dans les équations) et les consignes d'angles de roulis, tangage et lacet. Si on veut utiliser plutôt des consignes en position

x,y,z alors il suffit de prendre la sortie de la boucle de position et l'injecter en tant que consigne de la boucle de vitesse puis la sortie de cette boucle de vitesse sera les consignes d'angles de roulis, tangage et lacet. Les contrôleurs de position et vitesse ne sont pas explicités ici par soucis de similarités avec les deux ci-dessus.

2.5 Résumé du chapitre

Dans ce chapitre, on a vu comment interagir avec le crazyflie via le package ROS développé au cours de ce stage que ce soit en simulation ou dans la salle d'expérimentation. Toutes les informations détaillées nécessaires pour commencer à utiliser le package est open source et trouvable sur le github [7]. De plus, on a résumé les équations de la dynamique et quelques paramètres physiques du crazyflie qui seront utilisés dans les chapitres suivants. Concernant les équations de la dynamique du crazyflie, la formule [2.1] nous résume la dynamique du crazyflie avec comme inconnue les grandeurs M_x, M_y, M_z, F. Ces grandeurs sont plus tard fournies par les boucles de contrôle grâce à la formule [2.7].

Enfin si l'on considère que la matrice d'inertie de la figure [2.3] est en fait diagonale : c'est à dire que l'on suppose que les termes non nuls qui ne sont pas sur la diagonale auront un effet minime dans les équations de la dynamique comparées aux termes diagonaux. La conséquence immédiate est que les coefficients I^*_* des équations de la dynamique en $\dot{p}, \dot{q}, \dot{r}$ sont réduits à zero sauf les coefficients $I^p_{qr}, I^q_{pr}, I^r_{pq}, I^m_{xx}, I^m_{yy}, I^m_{zz}$ dont les valeurs pour un crazyflie sont résumées sur la figure [2.11].

```
#define G_QUAD 9.8f
#define M_QUAD 0.028f

#define Kp_Z   2.0f
#define Kp_VZ  25.0f
#define Ki_VZ  15.0f

#define Kp_rr  250.0f
#define Kp_pr  250.0f
#define Kp_yr  120.0f

#define Ki_rr  500.0f
#define Ki_pr  500.0f
#define Ki_yr  16.7f

#define Ip_qr   -0.760696995059653f
#define Iq_pr   0.761902331719982f
#define Ir_pq   -0.002866960484664f
#define Im_xx   6.034380278196999e4f
#define Im_yy   6.003985926176670e4f
#define Im_zz   3.417442050093412e4f

#define C1      0.04076521;
#define C2      380.8359;
#define Ct      1.285e-8;
#define Cd      7.645e-11;
#define d       0.046/sqrt(2.0);
```

Figure 2.11: Paramètres physiques pour la dynamique du crazyflie

CHAPTER 3

SIMULATEUR GAZEBO POUR CRAZYLFIE

Dans ce chapitre, on présentera de manière succincte le simulateur gazebo pour crazyflie implémenté au cours du stage. Ceci est une nouveauté car jusqu'à ce que l'on s'investisse là dessus, le seul moyen officiel de faire des tests sur du code implémenté pour un crazyflie était de le lancer et de voir si le crazyflie échouait ou pas. L'autre méthode pour faire de la simulation gazebo pour crazyflie repose sur l'utilisation de PX4 comme auto-pilote pour crazyflie au lieu d'utiliser le code source développé par bitcraze. Cette alternative donne la possibilité à l'utilisateur de faire la simulation. Mais cette simulation ne sera pas effectuée avec les paramètres physiques du modèle de crazyflie à moins d'apporter quelques modifications sur le simulateur gazebo de PX4. Le simulateur implémenté ici est actuellement utilisé par plusieurs personnes dans la communauté bitcraze et est en cours de maintenance. Dans la suite de ce chapitre, nous commencerons par une présentation rapide de Gazebo, puis le modèle de crazyflie et des capteurs/actuateurs utilisés, ensuite nous présenterons les modifications faites au niveau du crazyflie pour faire de la SITL et de la HITL et aussi de la communication entre crazyflie et gazebo. Enfin Quelques tracés de trajectoires seront présentés dans l'environnement de simulation pour montrer le comportement en simulation et l'on présentera rapidement quelques scenario swarm en simulation.

3.1 Présentation de Gazebo : Modèle 3D de crazyflie + Modèle de ses capteurs

Gazebo est un simulateur 3D pour véhicule autonome de manière générale. En plus de son interface 3D, il a une API favorable au développement et à la personnalisation de ses fonctionnalités de base. Gazebo offre la possibilité de simuler avec précision et de manière efficace une population de robots dans des environnements en intérieur ou en extérieur complexes.

C'est une plateforme de simulation assez utilisée dans le domaine de la robotique et beaucoup de packages à ce jour ont été crées dans le but de faciliter les démarches d'un utilisateur souhaitant intégrer son propre module de simulation. Un des gros avantage de Gazebo est qu'il peut fonctionner en mode indépendamment de ROS, cependant il inclut aussi la possibilité de fonctionner avec ROS. Ce qui nous arrange étant donné les objectifs initialement fixés de contrôle haut niveau du crazyflie.

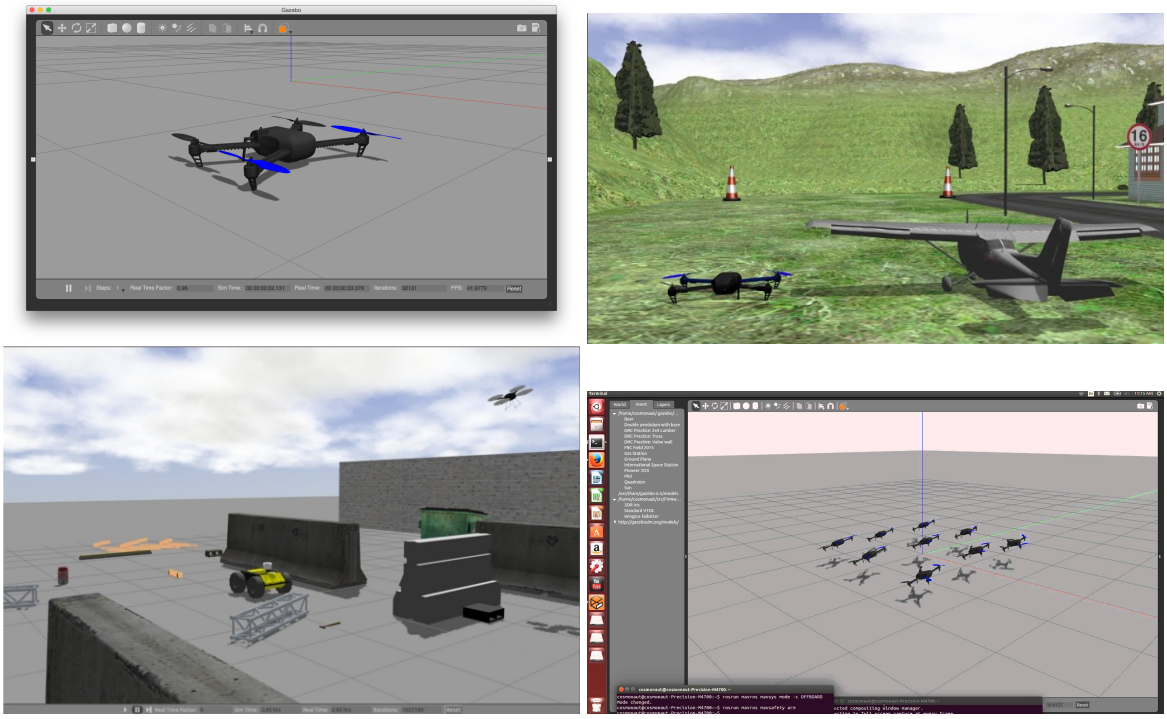


Figure 3.1: Quelques environnements Gazebo

3.1.1 Dimensionnement et Modèle visuel du crazylfie

Le but ici est de trouver un modèle graphique qui représentera de manière conforme le crazyflie dans l'environnement Gazebo. Par manière conforme, on parle d'un modèle dont les dimensions sont les mêmes que le crazyflie réels, dont les coefficients d'inertie et la géométrie correspondent à ceux d'un vrai crazyfie. Concernant juste le coté visuel (c'est à dire l'apparence), on a obtenu de bitcraze un modèle 3D .dae du crazyflie ainsi qu'un modèle de ses hélices. Une fois ce fichier obtenu, le but est de le rajouter dans gazebo en rajoutant des collisions autour des éléments susceptibles de rentrer en contact physique avec l'environnement. Une fois les collisions placées, il faudrait instruire gazebo quant aux valeurs des coefficients d'inertie du modèle et de la masse de l'objet. Toutes ces valeurs sont nécessaires à gazebo pour faire de la simulation des forces et moments qui seront exercés sur le modèle gazebo. Le résultat du modèle visuel peut être observé sur la figure [3.2].

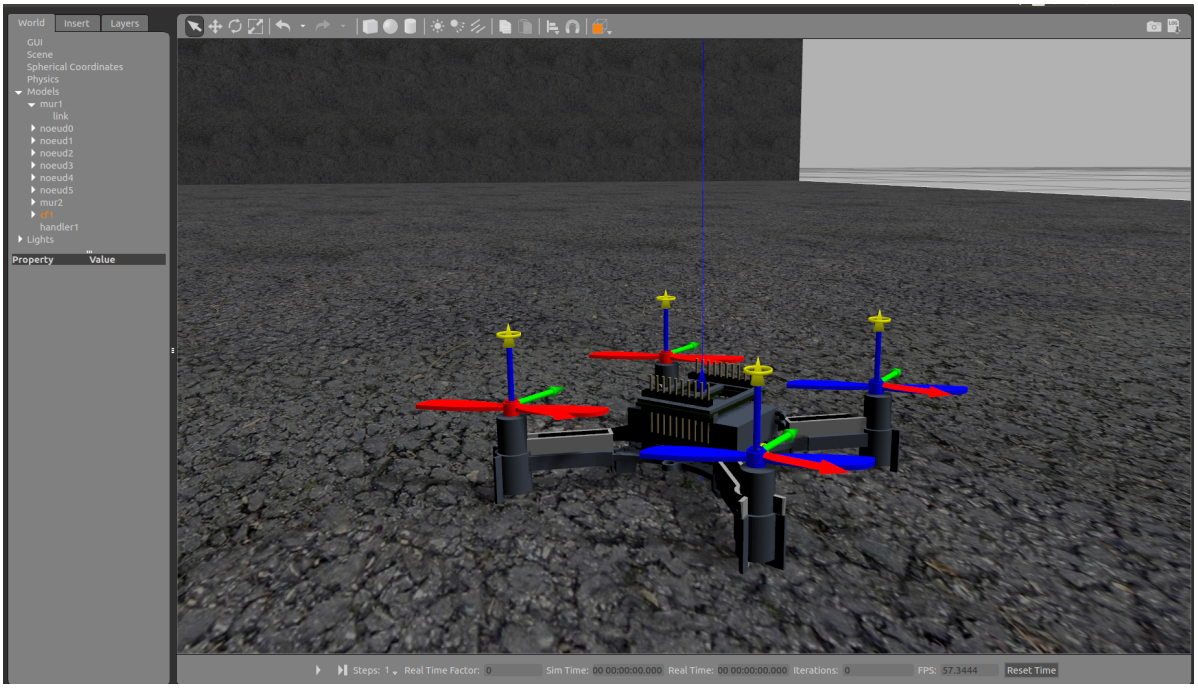


Figure 3.2: Visuel d'un Crazyflie 2.0 dans gazebo

3.1.2 Modèle des capteurs inertiels embarqués dans la simulation

Généralement, le modèle de capteur inertiel (accéléromètre, gyroscope ou magnétomètre) consiste typiquement à un bruit blanc ajouté à un biais de type marche aléatoire. Plus précisément, si par exemple considère la vitesse angulaire sans perdre de généralité, la valeur ω_{mes} qui est la vitesse angulaire mesurée (sur un axe par exemple) peut être écrite [10] :

$$\omega_{mes}(t) = \omega(t) + b(t) + n(t)$$

Où n représente le bruit blanc qui fluctue beaucoup et b représente le biais qui varie lentement. Le même modèle peut être utilisé pour les autres axes mais avec probablement différente valeur caractéristique de b et n. Par ailleurs ce modèle est aussi utilisable pour l'accéléromètre. Notons que $\omega(t)$ correspond à la valeur exacte de la grandeur mesurée qui en cas de simulation est censée être fourni par l'environnement de simulation (du fait de la présence de cet objet dans cet environnement).

Concernant le bruit blanc, on va le modéliser en continue avec une moyenne de zero et un écart type σ_g. Etant donné que la simulation est effectuée en discrétisant le temps, ce processus blanc gaussien va être simulé suivant les règles :

$$n_d[k] = \sigma_{gd} W[k]$$

$$W[k] \equiv N(0,1)$$

$$\sigma_{gd} = \frac{\sigma_g}{\sqrt{\delta t}}$$

Avec δt la fréquence d'échantillonnage des données. Le but plus tard est de déterminer σ_g et d'en déduire $sigma_{gd}$ pour l'utiliser dans la simulation.

D'un autre coté, concernant la biais, il est modélisé ici par un processus Brownien ou marche aléatoire. Plus formellement, ce processus est généré en intégrant le bruit blanc de deviation standard σ_{bg} (pour sigma biais gyro). En continue, cela donne

$$\dot{b}_g(t) = \sigma_{bg} W(t)$$

En discret on a donc l'évolution suivante pour le biais :

$$b_d[k] = b_d[k-1] + \sigma_{bgd} W[k]$$

$$W[k] \equiv N(0,1)$$

$$\sigma_{bgd} = \sigma_{bg} * \sqrt{\delta t}$$

La figure [3.3] résume les différentes grandeurs pour les capteurs inertielles usuels.

Parameter	YAML element	Symbol	Units
Gyroscope "white noise"	`gyroscope_noise_density`	σ_g	$\frac{rad}{s}\frac{1}{\sqrt{Hz}}$
Accelerometer "white noise"	`accelerometer_noise_density`	σ_a	$\frac{m}{s^2}\frac{1}{\sqrt{Hz}}$
Gyroscope "random walk"	`gyroscope_random_walk`	σ_{bg}	$\frac{rad}{s^2}\frac{1}{\sqrt{Hz}}$
Accelerometer "random walk"	`accelerometer_random_walk`	σ_{ba}	$\frac{m}{s^3}\frac{1}{\sqrt{Hz}}$
IMU sampling rate	`update_rate`	$\frac{1}{\Delta t}$	Hz

Figure 3.3: Grandeurs caractéristiques pour le modèle de capteurs

Le modèle du magnétomètre, du baromètre ou du système LPS est beaucoup plus simple car ils correspondent tout simplement à des bruits blancs de déviation standard à déterminer grâce à une collection de données des capteurs du crazyflie. Pour le système LPS on sait au préalable que sa déviation standard est de 10cm.

Pour obtenir les déviations standards des capteurs inertielles, on a utilisé la méthode de la Variance d'Allan présentée dans le chapitre 2. Les données des capteurs ont été collectées pendant 6h à intervalle de temps réguliers et on a obtenu les courbes suivantes en utilisant la méthode de la variance d'Allan :

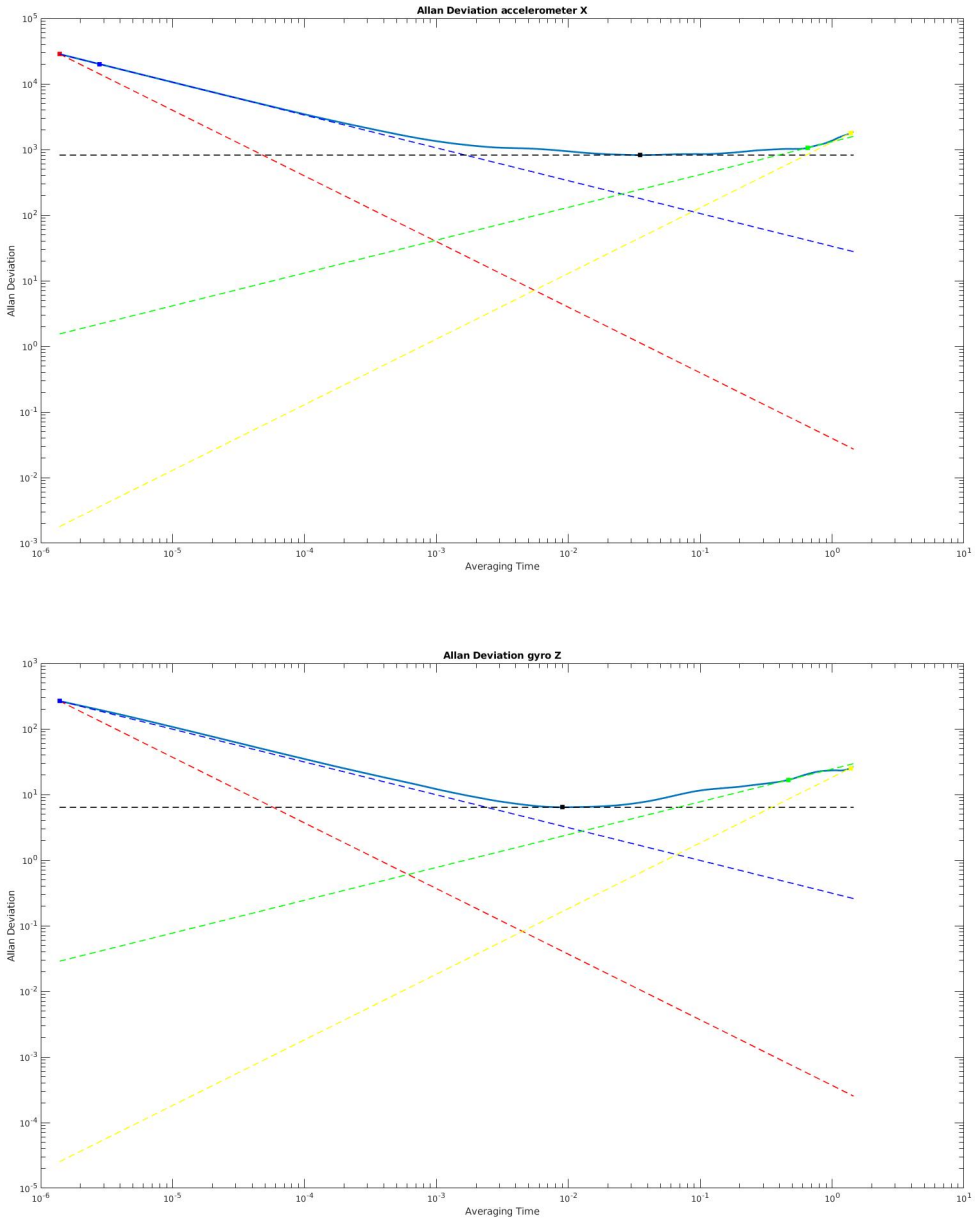


Figure 3.4: Calcul des caractéristiques des bruits des capteurs

Sur la courbe [3.4], les droites d'intérêts dans l'échelle log-log sont les droites de pente 1/2 et -1/2. D'après la théorie de la Variance d'allan, la droite de pente -1/2 permet de repérer la valeur de σ_g. Et la pente de +1/2 permet de remonter à la valeur σ_{bg}. Ces valeurs pour le crazyflie peuvent être retrouvées dans les fichiers de configurations du simulateur.

3.1.3 Modèles des moteurs + hélices utilisés dans le simulateur

Le but ici est de déterminer le modèle qui permettrait de caractériser au mieux l'action combinée de la rotation des moteurs et des hélices sur les forces et moments générés sur le crazyflie. Pour cela, on s'est référé à l'article [11] qui donne des modèles assez réalistes pour moteurs de robots (et donc de drone). Tout le travail ici étant de pouvoir déterminer les bons paramètres pour le robot d'intérêt et sa géométrie. Ces paramètres incluent la constante de moment, le coefficient de trainée et la vitesse maximale de rotation. Le coefficient de trainée pour le crazyflie s'obtient grâce aux expériences menées dans l'article [1]. Cet article permet aussi de retrouver la constante de moment α_M car par définition : $Thrust = \alpha_M \omega^2$. Or dans l'article, on a spécifiquement pour le crazyflie une relation entre la poussée et les valeurs PWM fournies aux moteurs. De plus on a une relation entre les valeurs PWM fournies et la vitesse de rotation donc implicitement on a une relation entre la poussée et la vitesse de rotation. Le coefficient α_M s'en déduit naturellement.

3.1.4 Idée derrière la simulation

Avec toutes ces informations, il est enfin possible de simuler le comportement de crazyflie car dans le modèle gazebo, étant donné une valeur PWM reçue, on connait la relation permettant d'en obtenir une vitesse de rotation. Cette vitesse de rotation devra être envoyée au simulateur qui en fonction des valeurs des paramètres ci-dessus générera les Forces et Moments utiles pour déplacer le drone dans l'environnement graphique et mettre à jour l'état du drone. D'un autre coté, vu que l'on arrive à simuler les capteurs embarqués au sein du crazyflie, l'environnement de simulation pourrait envoyer ces informations à une instance de crazyflie en lui faisant croire que ces informations proviennent d'un capteur et ce dernier fera ses routines usuelles et devra renvoyer les sorties PWM au simulateur et on répète cette procédure en boucle pour avoir une simulation du crazyflie sur gazebo. Le but dans la suite ce sera de faire communiquer l'instance de crazyflie et celle de gazebo.

3.2 Les types de simulations implémentées

Afin d'atteindre nos objectifs, on a réalisé les deux versions de simulation les plus connues : la HITL (Hardware In the Loop) et enfin la SITL (Software In the Loop).

3.2.1 Simulation en mode HITL

Ce mode est le mode qui nous rapproche le plus du comportement réel d'où l'intérêt de l'implémenter. De manière simple, l'auto-pilote est une fois de plus exécuté par le crazyflie même sauf que au lieu d'utiliser les données de ses propres capteurs, il utilise des données de capteur d'une entité extérieur à lui. Puis au moment du contrôle des actuateurs, il ne devra pas

commander ses propres actuateurs mais plutôt envoyer ces valeurs a cet agent externe. C'est le mode de simulation qui nous rapproche le plus du comportement réel car le code source est directement exécuté par la carte elle même et donc toute la logique concernant le coté temps réel est respectée. Ce mode de simulation apporte donc une meilleure garantie mais requiert que l'utilisateur possède au moins un crazyflie avec lui.

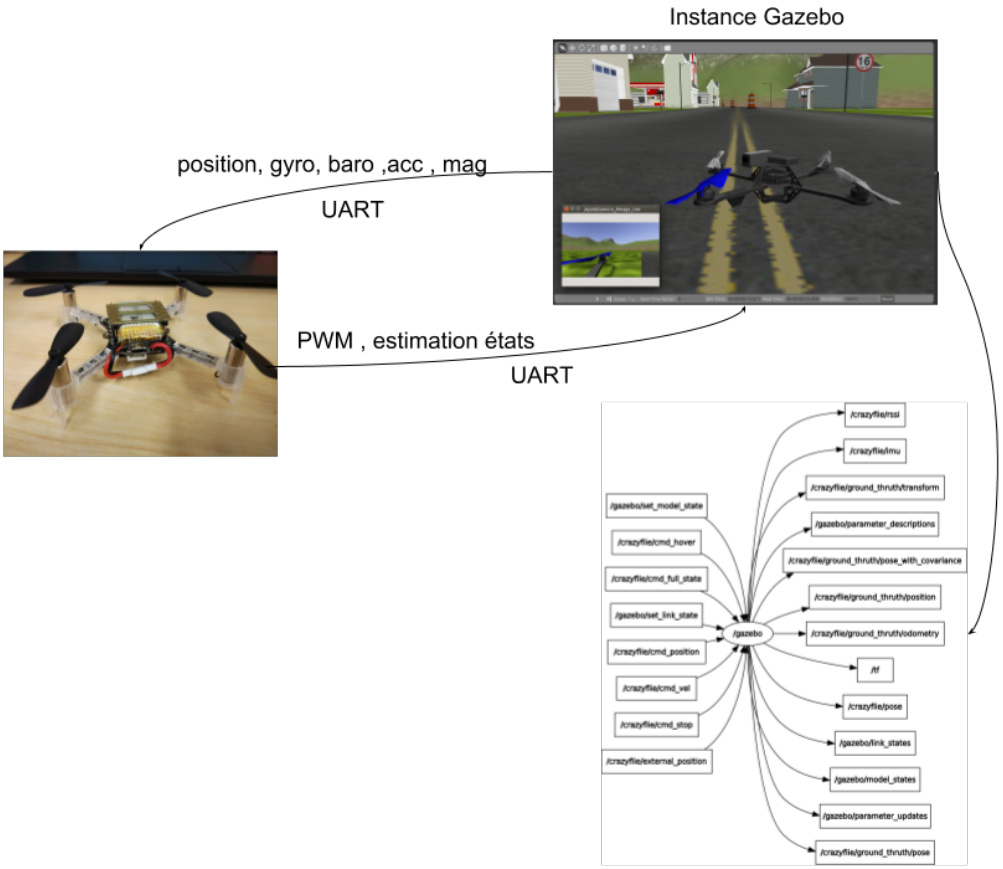


Figure 3.5: Simulation en mode HITL

Concernant le moyen de communication avec cet agent extérieur, dans le cas du crazyflie, il existe deux moyens de communiquer avec l'extérieur : l'UART et la RADIO. Faire de la simulation avec la radio dongle ici s'avère compliqué car les informations des capteurs venant de l'agent extérieur (gazebo) doivent arriver avec un débit assez élevé pour avoir une réactivité correcte du système. Pour cette raison, la version HITL que l'on a implémenté n'utilise que l'UART mais il est facile de le faire utiliser aussi la radio si vous savez exactement ce que vous voulez faire. Les modifications à effectuer pour réaliser une HITL au niveau du crazyflie sont minimes car il suffit juste d'utiliser des macros pour empêcher l'utilisation des données de ses capteurs et l'écriture des sorties PWM directement sur les moteurs. Cependant il est nécessaire de savoir reconnaitre des messages contenant des informations sur l'état des capteurs inertiels lorsqu'un message

CRTP est reçu. Et il est aussi important d'avoir un message CRTP spécial pour reconnaitre et envoyer des informations du type PWM pour contrôler le drone dans l'environnement virtuel. Ceci constitue pratiquement toutes les modifications à faire au sein du crazyflie et ceci en utilisant que des macros pour inclure du code relatif à la HITL quand la HITL est choisie. Au niveau de gazebo, on doit connaitre l'adresse UART pour pouvoir communiquer avec le crazyflie. Ayant cela, l'orientation, la vitesse angulaire, l'accélération et la position vraie du crazyflie dans l'environnement de simulation sont bruités grâce au modèle présenté plus haut puis sont envoyés au crazyflie via UART.

3.2.2 Simulation en mode SITL

La différence flagrante entre les deux modes est que la SITL ne nécessite pas d'avoir la carte d'auto-pilote du crazyflie pour simuler un scénario. Ce mode semble plus pratique la plus part du temps mais il faut pas oublier que le système d'exploitation sur lequel le module sera lancé n'est pas temps réel donc la réponse du système ne sera jamais aussi bonne qu'avec la HITL.

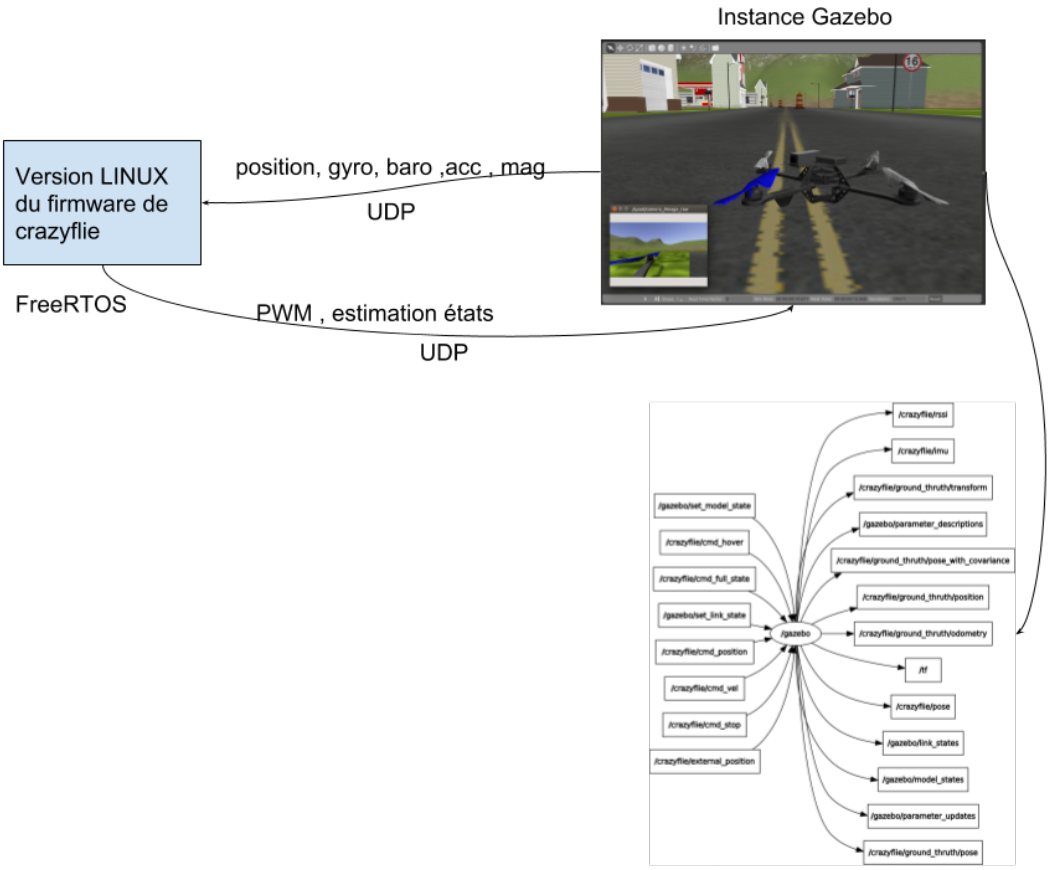


Figure 3.6: Simulation en mode SITL

Pour avoir ce mode de simulation, Le code source du crazyflie est légèrement modifié de

manière à pouvoir le compiler sous un OS du type Linux. Les modifications impliquent :

- l'utilisation des macros pour masquer toute inclusion des capteurs en cas de choix de SITL
- En plus des canaux de transmissions tel que UART et la RADIO actuellement utilisé à l'intérieur du crazyflie, j'ai rajouté un canal du type socket (un protocol UDP) pour autoriser aussi l'échange des messages CRTP via ce canal. Ce nouveau canal est une tâche à part entière comme les deux précédents canaux avec sa priorité et la quantité de mémoire qu'elle est autorisée à utiliser
- La création de nouveaux types de messages CRTP et de nouveaux ports pour permettre au crazyflie de recevoir les informations capteurs et d'envoyer les informations PWM.
- La création d'une version dupliquée des capteurs inertiels qui utilisent ce nouveau canal et décodent ces nouveaux types de message CRTP en 'message capteurs'
- La création d'une version dupliquée du module de commande des moteurs qui utilise ce nouvel canal pour envoyer les commandes PWM
- Utilisation systématique du canal socket en mode simulation. Cependant l'environnement gazebo doit être ouvert au préalable pour initier correctement la communication

3.3 Quelques manœuvres dans l'environnement de simulation

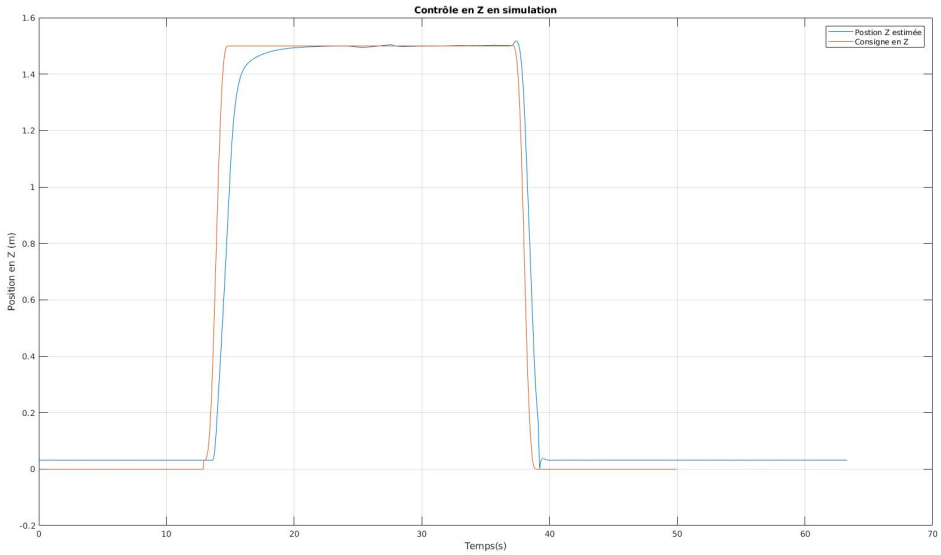


Figure 3.7: Contrôle de l'altitude du crazyflie en Simulation SITL

Dans cette partie, on va essayer de comparer le comportement obtenu dans la simulation avec celui de la salle d'expérimentation du chapitre précédent [2.9]. La manœuvre que doit effectuer le drone ici est : décoller à 1.5m en 2s puis il se déplacera pour la position x=0.8m , y=0 en conservant son altitude et à partir de cette nouvelle position il fera un cercle dont le centre est (0,0,1.5) et le rayon R=0.8m. Tout dabord, la figure [3.5] nous donne les performances en simulation (ici la SITL) en comparant la consigne en altitude et sa valeur estimée par le filtre de Kalman. Une fois de plus il est important de noter que la procédure pour lancer la simulation dans l'environnement gazebo est exactement la même que pour lancer cette expérience sur le vrai drone en salle d'expérimentation. On peut observer une fois de plus la rapidité de la réponse lorsque la consigne de 1.5m est envoyée. D'autre part une des différences avec le cas réel est le modèle du LPS utilisé en simulation. Ce modèle est simpliste vu qu'il prend pas en compte les possibles pertes de connexion entre les ancres et le crazyflie. Ces pertes de connexion entrainent des sauts et rendent le contrôle du drone un peu plus instable. C'est cela qu'on observait lorsque le crazyflie atteignait la position Z=1.5m sur la figure 2.9. L'amélioration qu'on peut apporter est d'augmenter le nombre d'ancres dans la salle d'experimentation pour une meilleure couverture. On peut aussi faire des modifications coté simulation. Le but de cette modification consisterai à augmenter la déviation standard du capteur de LPS pour prendre en compte ces possibles délais de communications ou alors de complètement formuler un nouveau modèle du LPS en simulation.

Après la procédure de décollage, la mission attribuée est d'effectuer un cercle et on peut voir le résultat sur la figure [3.5] suivante.

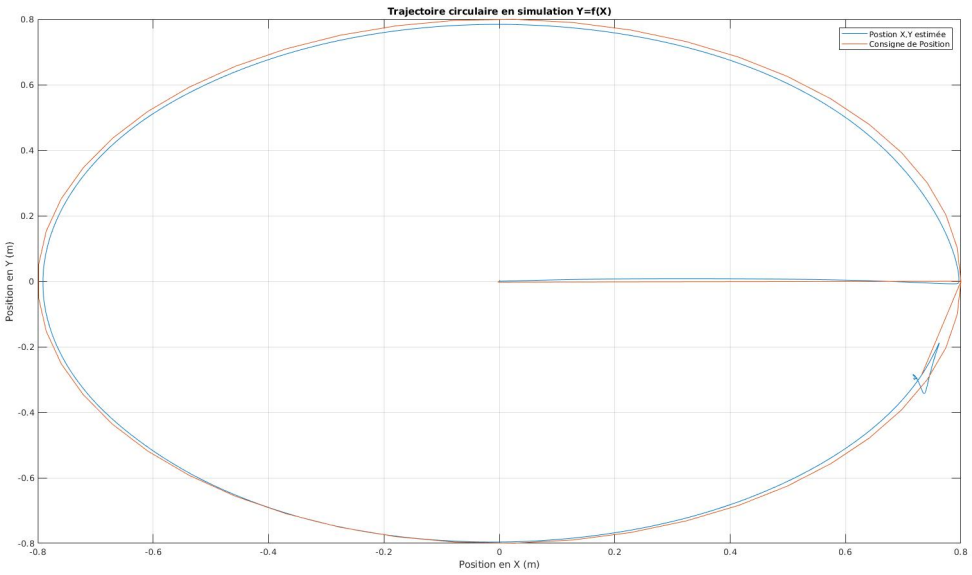


Figure 3.8: Trajectoire circulaire en Simulation SITL

3.4 Ouverture de la simulation aux Swarms

Parmi les fonctionnalités recherchées dans le simulateur, la possibilité de faire de la simulation avec un grand nombre de crazyflie était un challenge et était une des fonctionnalités les plus recherchées parmi les questions posées par ceux qui ont utilisés notre package. Après la première version qui fonctionnait bien mais qui avait du mal à faire tourner plus de deux crazyflie sur une même machine, une restructuration du code et de la logique au niveau du simulateur et module gazebo a été faite dans le but d'avoir de meilleures performances. On est passé d'un maximum de deux crazyflie à 8 crazyflie sur le même ordinateur. Des améliorations restent encore à faire et ce nouveau code n'est pas encore sur la branche principale du package.

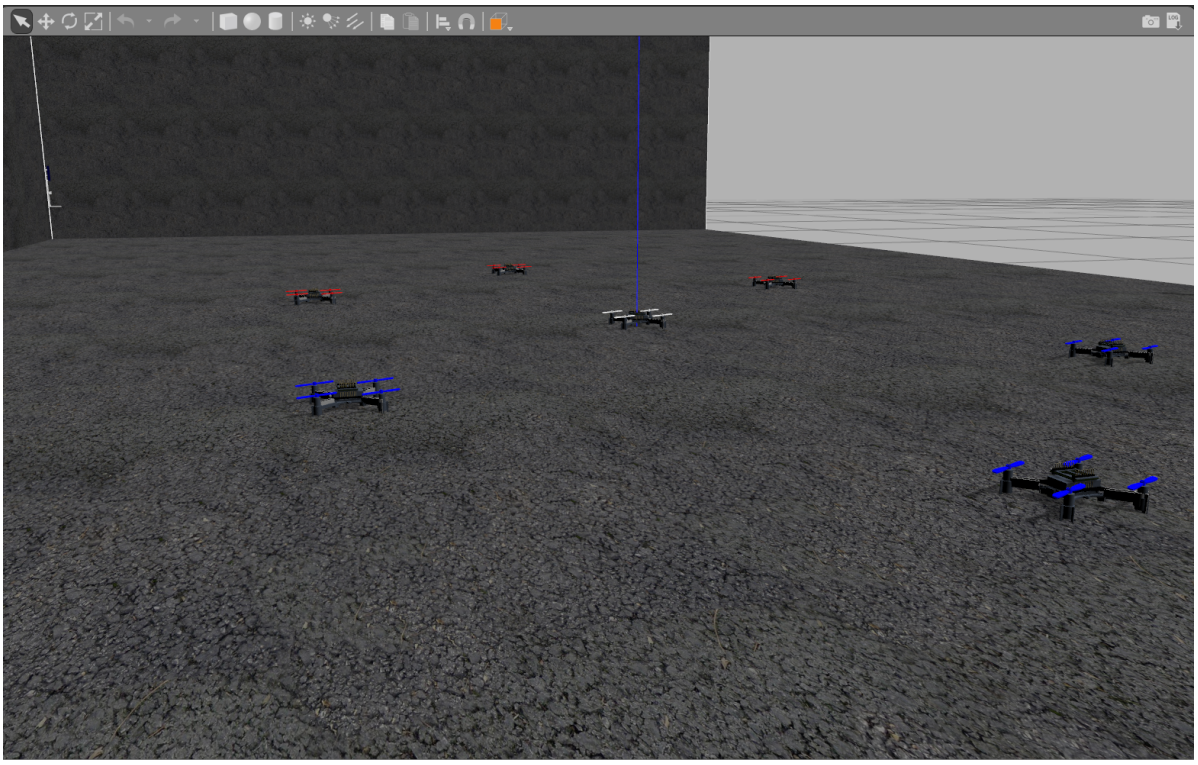


Figure 3.9: 7 crazyflie dans l'environnement virtuel salle Lix

L'idée ayant permise d'augmenter les performances est de séparer les crazyflie en petits groupes et d'utiliser un seul gestionnaire (donc 2 threads : 1 de réception et 1 d'envoi) par petits groupe de crazyflie. Avant, à chaque ajout de crazyflie dans l'environnement virtuel, un thread était crée et tournait en attendant de recevoir des informations (de type log etc..) du crazylfie. En même temps le thread principal de gazebo qui met à jour les capteurs envoient les informations de ces derniers. Au final plus on rajoutait de crazyflie plus on surchargeait le thread principale en créant d'autres threads et plus le thread principal devait effectuer d'actions car le rajout d'un nouveau crazyflie entraine la création de nouveaux capteurs et l'envoi à 400Hz des valeurs des capteurs inertiels via une socket. Cette façon de raisonner n'était clairement pas efficace mais

fonctionnait pour 1 ou 2 crazyflie et peut être plus sur un ordinateur plus puissant.

Avec la nouvelle solution on limite le nombre de thread qui est crée, on ne surcharge plus le thread principal de gazebo vu que les opérations d'envoie des valeurs des capteurs ne sont plus effectuées dans ce thread là mais les valeurs sont placées dans une queue et envoyées plus tard via le thread envoyeur du gestionnaire dans lequel est placé le crazyflie. De plus avec cette nouvelle méthode, on limite le nombre de socket UDP crée. Les crazyflie appartenant à un même gestionnaire doivent avoir un identifiant et doivent envoyer les commandent moteurs au même serveur (le gestionnaire) et le gestionnaire a entre autres pour rôle de retourner les valeurs mesurées par un capteur au bon destinataire.

Par exemple, La figure [3.10] montre une trajectoire en forme de 8 effectuée par trois crazyflie en simulation SITL. Et la figure [3.11] permet de visualiser leur contrôle en altitude.

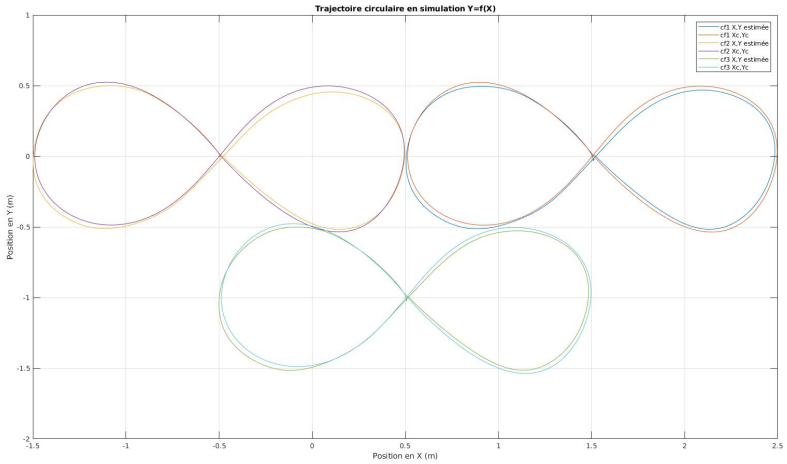


Figure 3.10: 3 crazyflie se synchronisant pour une trajectoire en huit

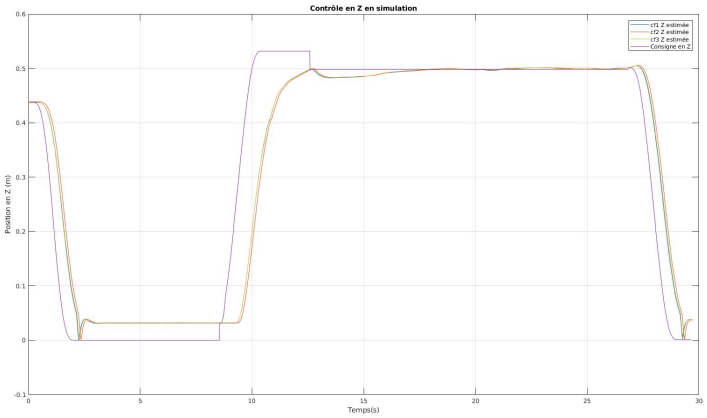


Figure 3.11: Visualisation de l'altitude pour la trajectoire en huit

CHAPTER 4

INTÉGRATION GARANTIE DE LA DYNAMIQUE D'UN DRONE À BASE DE MÉTHODES ENSEMBLISTES

Pour expliquer la notion d'intégration garantie de la dynamique et comment elle est liée à la vérification, considérons une fois de plus le cas d'un drone. Nous possédons un modèle (avec ses incertitudes) des équations qui réagissent la dynamique de notre drone. Si ce modèle était parfait (c'est à dire tous les paramètres des équations de la dynamique etaient parfaitement connus ou encore si aucune approximation n'avait été faite pour écrire cette forme simplifiée de la dynamique du drone) et si la connaissance de l'état initial était certaine, il suffirait d'appliquer l'une des nombreuses méthodes d'intégration d'une ODE pour déterminer l'évolution complète du système. Donc on saurait à l'avance les états atteignables par notre drone à partir de la seule donnée de la condition initiale. Cette connaissance nous permettra par conséquent de valider ou non certaines propriétés du système. Cependant, le monde étant ce qu'il est, rien n'est parfait ! On a pas une confiance absolue aux équations de la dynamique du drone mais on sait qu'ils sont incertains à quelque chose près. De même en cours de vol, on ne saurait presque toujours jamais précisément l'état dans lequel le drone est étant donné l'imprécision des capteurs ou les perturbations de l'environnement. C'est la raison pour laquelle les méthodes ensemblistes sont utiles ici. Le but étant de considérer les variables d'états de notre système non plus comme des valeurs fixes et connues mais comme une enveloppe contenant les valeurs possibles que peuvent avoir ces variables d'états partant d'un état initial lui aussi incertain (sous forme d'enveloppe). On peut facilement imaginer l'utilisation d'un intervalle (avec comme diamètre l'erreur maximale à un moment donné que l'on peut avoir pour la variable en question) comme enveloppe à notre problème. On verra plus tard les inconvénients que possède le domaine des intervalles pour une intégration garantie. Il est clair que ces enveloppes constituent la plus part du temps une sur-approximation de la vraie grandeur. Donc en admettant que des

opérations de bases peuvent être définies sur cette enveloppe (opérations qui conservent le coté sur-approximation), il est possible grâce aux méthodes de Taylor, à partir d'une enveloppe des conditions initiales (incertaines) d'intégrer les équations de la dynamique. Après l'intégration, si l'enveloppe à tout instant n'intersecte pas un état non sûr, on a validé que tous les états accessibles sont sûr. Le cas échéant, on ne peut pas conclure car l'enveloppe intégrée au fil du temps est une sur-approximation. Cette méthode sera appelée grossièrement par la suite sur-approximation et l'enveloppe sera appelée domaine abstrait ou particulièrement dans notre cas : *forme affine*. Inversement, supposez qu'en partant d'une enveloppe sur-approximante il y a moyen d'en avoir une sous-approximante : c'est à dire que l'on peut assurer que le drone passera par ces états. Ces sous-approximations permettraient de juger d'une part la qualité de notre sur-approximation et d'autre part permettrait de valider un état non sûr s'il rentre en intersection avec ce dernier (ce que la sur-approximation était incapable de faire). Pour les ODE (linéaire et non linéaire), l'article [12] est assez complet et détaille la théorie pour obtenir une sur/sous approximation. Si la dynamique n'est pas une ODE mais possède des délais, l'article [13] permet de comprendre comment trouver des sous/sur approximations pour ce genre de modèle.

Dans ce rapport, on résumera les résultats obtenus en implémentant les algorithmes de [12] sur la dynamique du crazyflie. Puis on détaillera un peu plus la version compacte et qui peut être embarquée sur le crazyflie pour calculer la sur-approximation en temps réel. On ne s'intéressera pas à la sous-approximation en embarqué à cause du temps de calcul important nécessaire pour obtenir la sous-approximation. Cependant pour les quelques analyses offline qui seront présentées par la suite, on s'intéressera à la fois à la sous approximation et la sur approximation.

4.1 Sur/Sous approximation des boucles de contrôle en vitesse angulaire

On va considérer pour l'instant uniquement le problème du contrôle des vitesses angulaires du crazyflie. Par vitesse angulaire, on pense notamment aux vitesses de rotation suivant les axes de roulis, de tangage et de lacet. On s'intéresse donc à commander les grandeurs p,q,r où la consigne est spécifiée par $p_{sp},q_{sp},r_{sp},thrust$. L'idée de faire de la vérification sur un algorithme de contrôle bas niveau a été expliqué dans l'introduction. Le but étant d'assurer certaines garanties bas niveau qui sont la plus part du temps supposée comme acquise. Disons par exemple que l'on veuille que notre contrôleur de vitesse angulaire ait un dépassement maximal de 10%. Connaissant la dynamique du système, On peut en offline partant d'une condition initiale avec ses incertitudes générer le tube de trajectoire du drone et à partir de la, statuer si les états atteignables par le drone sont sûr ou pas. Cette garantie sera dépendante des hypothèses faites sur l'incertitude de la condition initiale. Par conséquent une étude offboard permettrai de garantir des propriétés sur la boucle de contrôle étant donné l'état initial incertain. Ceci peut constituer

une preuve de sûreté mais dépend comme on a pu le redire des conditions initiales.

La dynamique du crazyflie tel que spécifiée dans le chapitre 2 nous fournit les équations suivantes:

$$
\begin{cases}
\dot{\phi} = p + cos(\phi)tan(\theta)r + tan(\theta)sin(\phi)q \\
\dot{\theta} = cos(\phi)q - sin(\phi)r \\
\dot{\psi} = \frac{cos(\phi)}{cos(\theta)}r + \frac{sin(\phi)}{cos(\theta)}q \\
\dot{p} = I^p_{qr}qr + I^m_{xx}\boldsymbol{Mx} \\
\dot{q} = I^q_{pr}pr + I^m_{yy}\boldsymbol{My} \\
\dot{r} = I^r_{pq}pq + I^m_{zz}\boldsymbol{Mz} \\
\boldsymbol{Mx} = (4C_TC_1^2d * thrust + 4C_2C_TdC_1)cmd_\phi - (4C_1^2C_Td)cmd_\theta cmd_\psi \\
\boldsymbol{My} = (-4C_1^2C_Td)cmd_\phi * cmd_\psi + (4C_TC_1^2d * thrust + 4C_2C_TC_1d)cmd_\theta \\
\boldsymbol{Mz} = (-2C_1^2Cd)cmd_\phi cmd_\theta + (8C_DC_1^2 * thrust + 8C_2C_DC_1)cmd_\psi \\
cmd_\phi = K^{rr}_p * (p_{sp} - p) + K^{rr}_d * (\dot{p_{sp}} - \dot{p}) + K^{rr}_i * \int (p_{sp} - p)dt \\
cmd_\theta = K^{pr}_p * (q_{sp} - q) + K^{pr}_d * (\dot{q_{sp}} - \dot{q}) + K^{pr}_i * \int (q_{sp} - q)dt \\
cmd_\psi = K^{yr}_p * (r_{sp} - r) + K^{yr}_d * (\dot{r_{sp}} - \dot{r}) + K^{yr}_i * \int (r_{sp} - r)dt \\
p_{sp} = K^r_p * (\phi_{sp} - \phi) + K^r_d * (\dot{\phi_{sp}} - \dot{\phi}) + K^r_i * \int (\phi_{sp} - \phi)dt \\
q_{sp} = K^p_p * (\theta_{sp} - \theta) + K^p_d * (\dot{\theta_{sp}} - \dot{\theta}) + K^p_i * \int (\theta_{sp} - \theta)dt \\
r_{sp} = K^y_p * (\psi_{sp} - \psi) + K^y_d * (\dot{\psi_{sp}} - \dot{\psi}) + K^y_i * \int (\psi_{sp} - \psi)dt
\end{cases}
\tag{4.1}
$$

En observant l'équation ci-dessus, on peut observer que les entrées Mx, My et Mz dépendent de $cmd_\phi, cmd_\psi, cmd_\theta$. Ces derniers dépendent à leur tour de $\dot{p}\dot{q}\dot{r}$ et des derivées des angles d'Euler. Par conséquent vu que $\dot{p}\dot{q}\dot{r}$ dépendent à leur tour de Mx, My et Mz, ces équations différentiels forment une DAE qui est difficile à résoudre.

D'un autre coté, si l'on observe correctement, on voit que le coté DAE de ce système vient de la présence des coefficients dérivateurs dans la loi de commande. Donc pour simplifier le problème, l'idéal serait de faire de la preuve de fonctionnement sur un contrôleur du type PI. S'il est possible de trouver et prouver un tel contrôleur, l'objectif est atteint.

Enfin, dans le cas où on considère uniquement le PI, la dynamique peut être réduite au système d'équations 4.2. Notons que le nombre de variables d'états du système est au minimum égal à 12. Dans le cas où le système n' est pas réduit à un PI mais plutôt un PID, on peut montrer qu'en utilisant un outil de calcul symbolique (Matlab), la nouvelle ODE obtenue (après décomposition structurelle de la DAE) a un comportement assez proche du cas PI. On va donc dans la suite de cet essai ne considérer que des contrôleurs du type PI. Et ce au moins pour le

contrôle des vitesses angulaires.

$$
\begin{cases}
\dot{\phi} = p + cos(\phi)tan(\theta)r + tan(\theta)sin(\phi)q \\
\dot{\theta} = cos(\phi)q - sin(\phi)r \\
\dot{\psi} = \frac{cos(\phi)}{cos(\theta)}r + \frac{sin(\phi)}{cos(\theta)}q \\
\dot{p} = I^p_{qr}qr + I^p_r r^2 + I^m_{xx}\boldsymbol{Mx} \\
\dot{q} = I^q_{pr}pr + I^m_{yy}\boldsymbol{My} \\
\dot{r} = I^r_{pq}pq + I^m_{zz} * \boldsymbol{Mz} \\
int\dot{E}rr_\phi = \phi_{sp} - \phi \\
int\dot{E}rr_\theta = \theta_{sp} - \theta \\
int\dot{E}rr_\psi = \psi_{sp} - \psi \\
int\dot{E}rr_p = p_{sp} - p \\
int\dot{E}rr_q = q_{sp} - q \\
int\dot{E}rr_r = r_{sp} - r
\end{cases}
\tag{4.2}
$$

La figure [**??**] suivante compare l'évolution des grandeurs d'intérêts lorsqu'on considère les K_d nuls et non nuls. Notons que les valeurs des coefficients K qui ont été prises en compte pour la simulation, sont ceux communément utilisés au sein de crazyflie. Donc là on essaye vraiment de faire de la preuve sur le crazyflie vu que la boucle de contrôle est exactement celle du crazyflie.

Notons sur la figure qu'on est en train de réaliser une sur-approximation donc plus le tube est gros, moins on en sait des choses. Prenons l'exemple de la vitesse de rotation suivant l'axe de tangage q. On peut au préalable observer que les deux contrôleurs (PI et PID) mettent quasiment le même temps à se stabiliser. En plus le contrôleur possédant le terme dérivateur a un dépassement plus important (conforme avec la théorie de contrôle). D'un autre coté, on observe que pour la vitesse angulaire suivant l'axe de lacet, le contrôleur PID converge beaucoup plus rapidement que le contrôleur PI. Ceci peut être expliqué par une valeur beaucoup plus importante du coefficient K_d sur cet axe là comparé aux K_d sur les autres axes de rotation.

On peut donc supposer par la suite que nos modèles n'auront pas de coefficients dérivateurs. Ce qui aura pour but de beaucoup réduire la complexité de calcul car l'ODE obtenue à partir de la DAE contient des expressions mathématiques avec beaucoup de termes qui ferait exploser la carte du bitcraze si on embarquait une telle méthode.

4.2 Le contrôle en offline d'une procédure de décollage

Ici le but est de partir d'une position initiale au sol et d'atteindre une altitude z_{sp} en un temps t donné. On a des incertitudes quant à la position au sol dans laquelle on est. On a aussi des incertitudes quant aux mesures du gyroscopes et de l'accéléromètre.

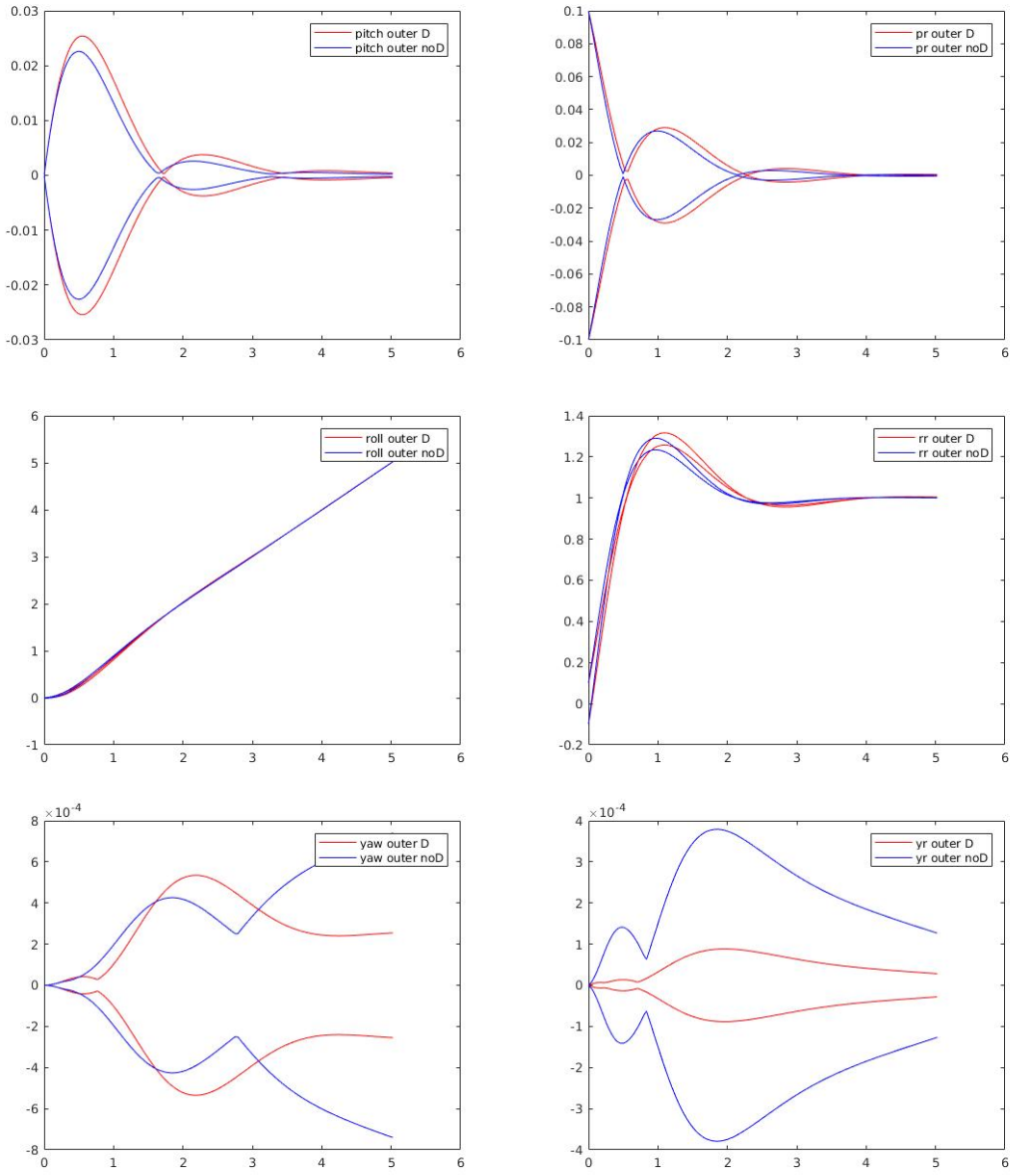


Figure 4.1: Analyse pour : $p_{sp} = 0.0$, $q_{sp} = 1.0$ and $r_{sp} = 0.0$. Du haut en bas, De gauche à droite , les variables sont affichées : tangage , vitesse de rotation suivant l'axe de tangage , roulis , vitesse de rotation suivant l'axe de roulis , lacet , vitesse de rotation suivant l'axe de lacet
.

A partir des équations de la dynamique de la partie 2, on peut extraire les équations utiles à la manœuvre de décollage 4.3.

Notons ici que la différence est qu'une des grandeurs d'intérêts est l'altitude z du crazyflie. Pour capturer cette dernière, on est obligé de capturer en plus les vitesses de rotation dans le

repère lié au crazyflie u, v et$w$. Ceci augmente donc le nombre d'état de notre système.

$$
\begin{cases}
\dot{z} & = -sin(\theta)u + cos(\theta)sin(\phi)v + cos(\theta)cos(\phi)w \\
\dot{u} & = rv - qw + sin(\theta)g \\
\dot{v} & = -ru + pw - cos(\theta)sin(\phi)g \\
\dot{w} & = qu - pv - cos(\theta)cos(\phi)g + \frac{F}{m} \\
\dot{\phi} & = p + cos(\phi)tan(\theta)r + tan(\theta)sin(\phi)q \\
\dot{\theta} & = cos(\phi)q - sin(\phi)r \\
\dot{\psi} & = \frac{cos(\phi)}{cos(\theta)}r + \frac{sin(\phi)}{cos(\theta)}q \\
\dot{p} & = \frac{I_y - I_z}{I_x}qr + \frac{1}{I_x}M_x \\
\dot{q} & = \frac{I_z - I_x}{I_y}pr + \frac{1}{I_y}M_y \\
\dot{r} & = \frac{I_x - I_y}{I_z}pq + \frac{1}{I_z}M_z \\
\dot{x_{i1}} & = 2(z_{sp} - z) - w \\
\dot{x_{i2}} & = p_{sp} - p \\
\dot{x_{i3}} & = q_{sp} - q \\
\dot{x_{i4}} & = r_{sp} - r \\
M_x & = (4C_TC_1^2d * thrust + 4C_2C_TdC_1)cmd_\phi - (4C_1^2C_Td)cmd_\theta cmd_\psi \\
M_y & = (-4C_1^2C_Td)cmd_\phi * cmd_\psi + (4C_TC_1^2d * thrust + 4C_2C_TC_1d)cmd_\theta \\
M_z & = (-2C_1^2Cd)cmd_\phi cmd_\theta + (8C_DC_1^2 * thrust + 8C_2C_DC_1)cmd_\psi \\
F & = C_TC_1^2cmd_\theta^2 + C_TC_1^2cmd_\phi^2 + 4C_TC_1^2cmd_\psi^2 + (4C_TC_1^2) * thrust^2 \\
 & \quad + (8C_TC_1C_2) * thrust + 4C_TC_2^2
\end{cases}
\tag{4.3}
$$

Les paramètres pour cette dynamique sont résumés dans le tableau [**??**]. Il est toute fois possible de rajouter des termes d'erreurs à ces grandeurs en cas de marge d'erreur du procédé expérimental utilisé pour obtenir ces paramètres. Ces paramètres peuvent directement être utilisés en simulation.

I_x	$1.657171e-5$	C_T	$1.285e-8$
I_y	$1.6655602e-5$	C_d	$7.645e-11$
I_z	$2.9261652e-5$	C_1	0.04076521
m	0.028	C_2	380.8359
g	9.81	d	$\frac{0.046}{\sqrt{2.0}}$

Table 4.1: Paramètre du modèle de décollage

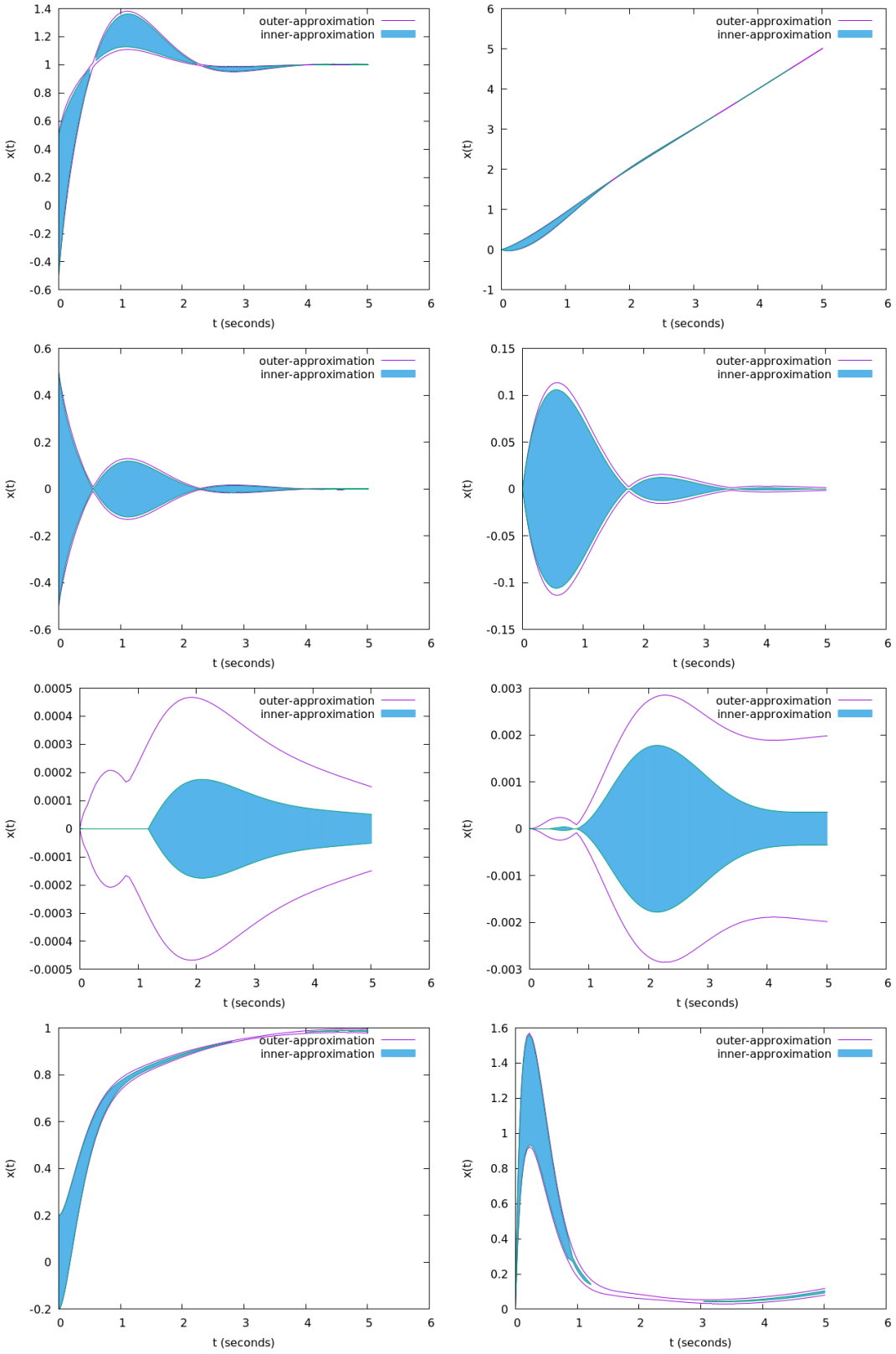


Figure 4.2: Analyse pour : $p_{sp} = 1.0$, $q_{sp} = 0.0$, $r_{sp} = 0.0$ and $z_{sp} = 1.0$. Du haut en bas, De gauche à droite , les variables affichées sont : p , θ , q, ϕ , r , ψ , z , w

Enfin, admettons que le but c'est de rejoindre l'état suivant: $z_{sp} = 1.0m$, $q_{sp} = 0deg/s$, $p_{sp} = 1deg/s$. Supposons de plus que l'état initial n'est pas certain et majorons cette incertitude en écrivant : $z = [-0.2, 0.2]$,$p = [-0.5, 0.5]$, $q = [-0.5, 0.5]$ et toutes les autres valeurs sont initialisées à 0. Notons que cette incertitude prise par exemple pour l'altitude Z est énorme au vu du fait que pour le crazyflie par exemple, le système de positionnement est précis à 10cm près. En utilisant l'outil de vérification sur ce modèle avec un pas variable et un ordre de taylor égal à 5, le temps de calcul pour la sous-approximation uniquement est de 6.3s et le temps de calcul pour la sur-approximation et la sous-approximation est de 996s. Le résultat de cette vérification peut être obtenu sur la figure [**??**]

L'analyse de la figure précédente dépend forcement des propriétés qui sont désirées pour sa boucle de contrôle. Ceci étant dis, précisons quand même que pour les grandeurs d'intérêts qui sont ici p, q, r, z la sous-approximation est incluse et presque confondue avec la sur-approximation. Ceci nous permet de conclure quand à la qualité de notre sur-approximation et cette information est utile pour faire de la vérification de propriété. Cependant comme il a été dis plus tôt, l'obtention d'une telle sous approximation est assez couteuse d'où le fait que l'on s'intéressera plus à la sur-approximation en embarqué.

4.3 Vérification en temps réel

Comme on l'a vu dans ce chapitre, la vérification offline nécessite de savoir la valeur initiale des variables d'état de l'équation de la dynamique. Donc le seul moyen en offline de garantir que notre contrôleur marchera, c'est de prouver qu'il marchera quelque soit l'état initial. Ceci est impossible à effectuer offline au vu du nombre infini de possibilité d'états initiaux. C'est pour cela que la vérification en temps réel peut être un bon concept.

4.3.1 Difficultés pour la vérification en temps réel

Pour commencer, il est naturel de se poser la question du pourquoi ne pas juste embarquer les algorithmes utilisées plus haut pour faire de la vérification en temps réel. La réponse est simple, l'outil de vérification a été implémenté pour les tourner sur un OS non temps réel et qui plus est, cet outil se base sur deux bibliothèques fondamentales:

- **La bibliothèque d'arithmétique affine** : C'est une implémentation en C++ des formes affines et de l'arithmétique affine. Les formes affines sont indispensables pour le calcul de la sur-approximation. Cependant la caractérisation originale de ces formes affines fait que leur 'taille' (nombre de symbole de bruits) ne peut qu'augmenter avec toutes opérations non linéaires (par conséquent le temps de calcul augmente aussi). Cette augmentation de la taille est synonyme d'allocation dynamique dans le jargon programmation par conséquent il est difficile d"intégrer cette version originale de la forme affine dans un microprocesseur.

- **La bibliothèque de différentiation automatique FADBAD++**: Comme son nom l'indique, cette librairie nous permet d'obtenir les coefficients du modèle de taylor en prenant en entrée la dynamique du système. A la base on pensait que cela n'allait pas être le cas d'utiliser en embarqué cette bibliothèque du fait que sa version d'origine utilise de l'allocation dynamique pour sauvegarder les coefficients de taylor jusqu'à un certain ordre du modèle. Il s'avère qu'il y'avait un moyen de faire de l'allocation statique en précisant à la compilation l'ordre maximum de différentiation. Avant de trouver cette possibilité, un bon moment a été passé à essayer de faire de la différentiation automatique à partir du calcul symbolique pour générer les coefficients de taylor. On s'est vite rendu compte que les expressions devenaient ingérables et qu'il y avait des calculs redondants qui sont mal gérés donc pas d'efficacité mémoire (nécessaire en embarqué et surtout pour le crazyflie).

Etant donné ces deux difficultés,il fallait donc développer une version de l'arithmétique affine/ forme affine qui pourrait être compilée et exécutée sur des petits MCU utilisés dans des systèmes embarqués. Donc dans l'idéal pas d'allocation dynamique quitte à sur-approximer un peu plus toutes opérations non linéaire entre forme affine ou à perdre des corrélations.

4.3.2 Forme affine développée pour l'embarqué

Le problème d'allocation dynamique de la forme affine par défaut [14] (ajouts de nouveaux symboles de bruits à chaque opérations non linéaire) est réglé ici par une stratégie de limitation du nombre de bruit maximal que peut avoir chaque variable. Toutes les opérations de bases telle que décrites dans l'article [14] restent basiquement les mêmes sauf que à chaque opération non linéaire deux cas se présentent :

- Soit il n'y a rien à faire car l'opération non linéaire résultera sur une forme affine de taille appropriée
- Soit le résultat est de taille supérieur à la limite imposée et par conséquent, si R est ce surplus par rapport à la limite imposée, le but est de fusionner les R+1 plus récents symboles de bruits obtenues en un nouveau symbole de bruit.

C'est cette stratégie rudimentaire qui a permis de créer une nouvelle forme affine et de définir une arithmétique là dessus conformément à l'arithmétique affine d'origine.

Quelques comparaisons (en terme de vitesse de calcul, de mémoire stack utilisée) ont été effectuées entre mon implémentation du calcul de la sur-approximation (qui utilise cette forme affine) et le calcul de la sur-approximation avec la forme affine classique et la forme affine AF1 présentée dans [15]. Et le résultat est exposé sur la figure [**??**] suivante :

Pour les notations, DAF c'est la forme affine par défaut avec allocation dynamique. MAF1 c'est la forme affine avec symbole de bruit limité plus un terme d'erreur [15] et MAF2 c'est la forme affine présentée dans cette partie.Sur la légende, N represente le Nombre maximal de

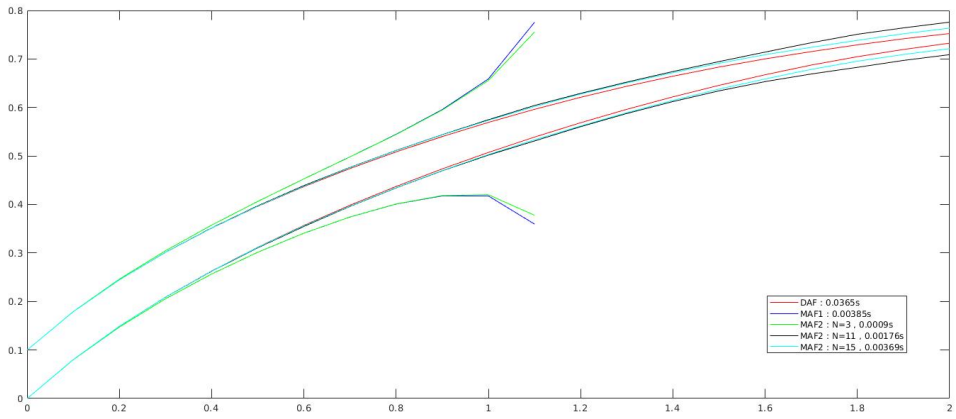


Figure 4.3: Comparaison forme affine sur l'exemple du brusselator

symbole de bruit pour chaque variable. Logiquement, on observe que la forme affine par défaut donne une meilleure approximation que les autres formes affines. On observe aussi que MAF1 n'est pas faite pour être utilisée avec l'intégration garantie car les erreurs s'accumulent assez vite et la forme affine se comporte à ce moment là comme un intervalle. Enfin MAF2, dépendant du nombre de symbole de bruit mis en jeu a une précision raisonnable et reste quand même dix fois plus rapide que la forme affine par défaut. Et cette rapidité est une caractéristique qui sera très importante pour faire de l'intégration garantie en temps réél.

4.3.3 Vérification embarquée en Simulation

Etant donné le fait que fadbad utilise c++ et la surchage d'opérateur, l'outil de vérification en temps réel a été implémenté en c++. Cependant, le code source crazyflie étant en C, pour les tests en SITL, l'instance crazyflie a été compilée avec C++ pour éviter des soucis d'incompatibilité.

Comme preuve de concept de la possibilité de faire de la vérification en temps réels, on a choisi de faire de la vérification pour une procédure de décollage et de voir si pendant son décollage, le drone ne peut pas atteindre des états indésirables (trop de dépassement, temps de réponse lent etc...). Pour intégrer cette fonctionnalité dans le code source existant de crazyflie, les étapes suivantes ont été suivies :

- La création d'une tâche FreeRTOS spécialement pour la partie vérification. Cette tâche communique avec l'estimateur pour obtenir l'état du système

- Cette tâche doit être modélisée de manière à prendre le moins d'espace mémoire possible et de manière à avoir une priorité pour pas atténuer les autres fonctionnalités. Donc une évaluation de l'ordonnançabilité du nouveau jeu de tâche peut être requise

- Definir un message CRTP pour communiquer les résultats de la vérification à un agent extérieur

On a suivi ces étapes et implémenté cette tâche dans le carde de la simulation en mode SITL. La tache de vérification fonctionne de la manière suivante : A chaque initialisation du crazyflie, dès que le filtre de kalman retourne une estimation valide, la tâche de vérification obtient son état de l'estimation et initialise l'ODE avec cet état initial plus une incertitude qui est liée aux incertitudes des capteurs de la variable. En même temps la tâche de vérification rentre en contact avec le stabilisateur pour savoir quelles sont les consignes $p_{sp}, q_{sp}, r_{sp}, z_{sp}$ et commence le processus d'intégration pour obtenir une sur-approximation sur **0.25s**. Si le tube de trajectoire ne rencontre pas d'état non sûr sur ces 0.25s d'horizon alors la tâche s'endors et recommence cette procédure toutes les 100ms. Si par contre le tube de trajectoire intersecte une zone de danger, Vu que la vérification ne peut pas conclure si en vrai elle intersecte l'état non sûr, on essaie de forcer le crazyflie à passer en mode hover.

Pour le décollage à 1.5m du sol, on a tout d'abord changé les coefficients PID du contrôleur en attitude pour mettre à 0 les coefficients dérivateurs (hypothèse de debut de chapitre) puis on sauvegarde dans un fichier chaque intégration éffectuée toutes les 100ms pour obtenir la figure **??**] suivante :

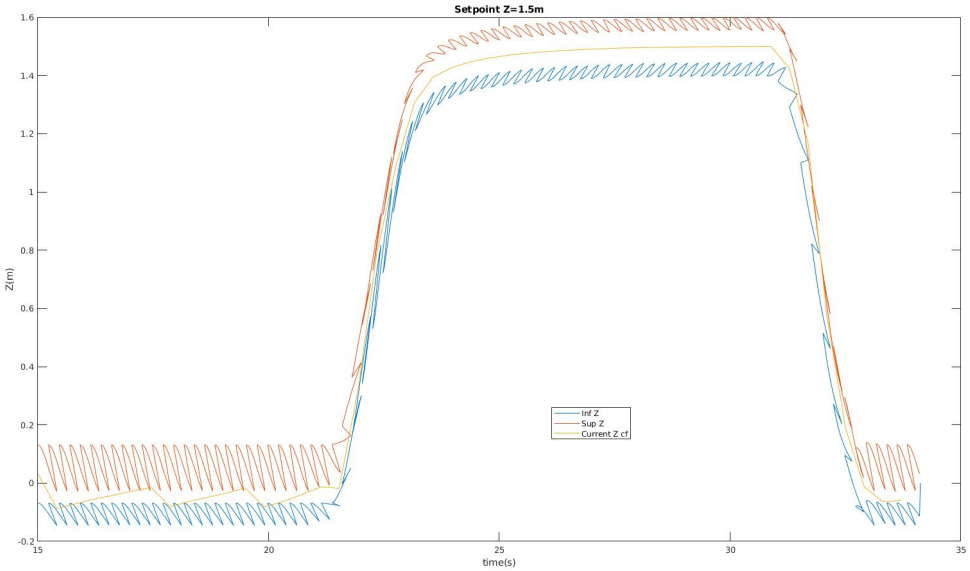


Figure 4.4: Z estimé et Z prédit par intégration garantie – [-0.1, 0.1] est l'erreur autour de Z à chaque réveil de la tâche de vérification

Si on regarde plus attentivement à la courbe, on peut observer la position estimée du crazyflie est toujours comprise à l'intérieur de la sur-approximation et surtout semble suivre le flux de

l'intégration garantie. Si on regarde plus attentivement la partie stationnaire à Z = 1.5m (figure [**??**]), les cercles sont les points de départ ou les conditions initiales de l'intégration, et les carrées sont les points finaux. On observe que le comportement du module de vérification n'est pas aberrant. On peut nettement voir comment la trajectoire définie par la vérification essaie de converger vers Z=1.5m.

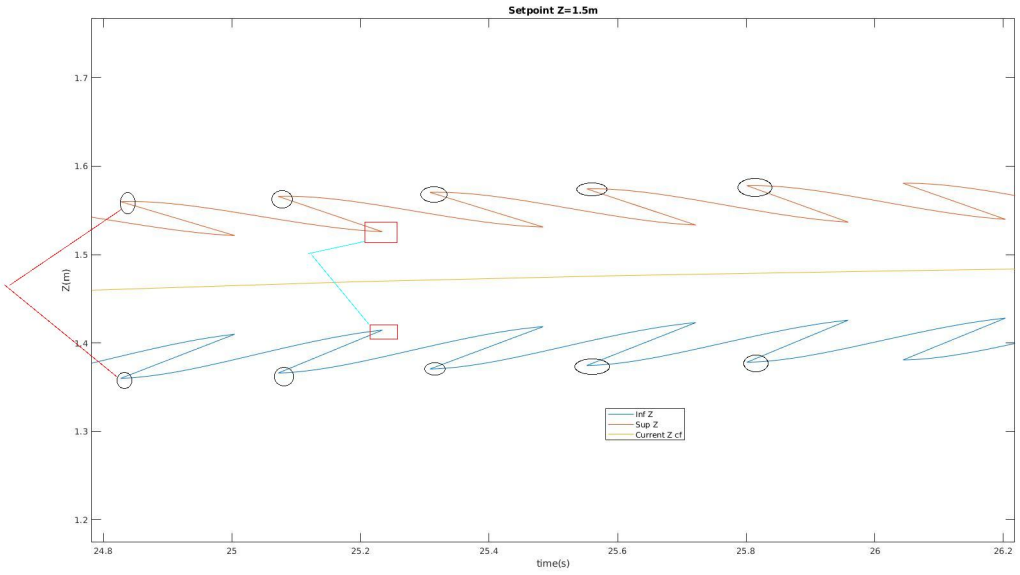


Figure 4.5: Zoom Z prédit par la vérification

Enfin on peut imaginer que si cette opération d'intégration est réalisée en permanence, on peut assurer quelques garanties. Le modèle de décollage est assez simple et ne fait qu'intervenir un système d'équations à 11 variables d'états. Si l'on veut faire de vérification qui incluent l'utilisation des coordonnées de position x et y, l'algorithme prendrait plus de temps et surtout plus de mémoire vu que l'on passerai à 21 variables d'états. Le problème de mémoire est un gros problème concernant le crazyflie. Etant donné le nombre de variable du système d'équations, les variables intermediaires crées par fadbad++ pour la différentiation, on peut imaginer que la vérification prend pas mal de mémoire stack. Le travail actuel repose sur la détermination de la quantité de mémoire stack à prendre pour la tâche de vérification puis de tester cette même simulation en salle d'expérimentation en rajoutant cette fois ci une correction si un état pas sûr est potentiellement atteint.

CHAPTER 5

CONCLUSION

Pour conclure, le stage a été divisé en deux grandes parties : La première partie où je développais le simulateur de vol pour crazyflie dans Gazebo et la deuxième partie où il était question d'implémenter un algorithme de vérification suffisamment léger pour interagir avec le contrôle et dont le but serait de détecter des possibles défaillances. Le première partie du stage a été un succès dans la mesure où il n'y avait pas de simulateur pour crazyflie avant cela malgré le désir de la communauté de bitcraze d'en avoir un comme le font d'autres auto pilot réputé tel que Px4. La version actuelle sur le git est assez rapide pour permettre à faire des simulations avec un grand nombre de crazyflie. La documentation est aussi fournie sur le github [7] mais il se pourrait qu'il y ait de petites erreurs de typo. La seconde partie du stage sur la vérification embarquée a requis plus de temps à cause des différentes tentatives pour avoir une forme affine ou un outil de différentiation approprié. De plus le but n'étant pas de faire que de la simulation sous gazebo, il fallait une garantie que les outils développés et les bibliothèques utilisés sont compatible avec le compiler STM32 du crazyflie.

Tout cela pour dire que pour la partie vérification embarquée, pas mal de choses peuvent être encore réalisées : dimensionner la tâche de vérification avec le reste des tâches de l'auto-pilot, améliorer l'implémentation actuelle de la forme affine développée ici (L'opération de réduction des nombres de bruits à chaque opérations non linéaire est couteux, et une façon intelligente de le faire serait de le faire uniquement lors des affectations à une variables d'état), faire des tests dans la salle d'expérimentation pour la procédure de décollage et étendre cela à une vérification du contrôle en position. c'est bien dommage que le stage de fin d'étude ne dure que 5-6 mois.

BIBLIOGRAPHY

[1] Julian Förster.
System identification of the crazyflie 2.0 nano quadrocopter.
hhttps://doi.org/10.3929/ethz-b-000214143, August 2015.
Bachelor Thesis.

[2] N. El-Sheimy, H. Hou, and X. Niu.
Analysis and modeling of inertial sensors using allan variance.
IEEE Transactions on Instrumentation and Measurement, 57(1):140–149, Jan 2008.
ISSN 0018-9456.
doi: 10.1109/TIM.2007.908635.

[3] Bitcraze website.
https://www.bitcraze.io/.

[4] Abdulrahman Alarifi, AbdulMalik S. Al-Salman, Mansour Alsaleh, Ahmad Alnafessah, Suheer Alhadhrami, Mai A. Al-Ammar, and Hend Suliman Al-Khalifa.
Ultra wideband indoor positioning technologies: Analysis and recent advances †.
In *Sensors*, 2016.

[5] M. W. Mueller, M. Hamer, and R. D'Andrea.
Fusing ultra-wideband range measurements with accelerometers and rate gyroscopes for quadrocopter state estimation.
In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1730–1736, May 2015.
doi: 10.1109/ICRA.2015.7139421.

[6] W. Hönig and N. Ayanian.
Flying multiple uavs using ros.
In Anis Koubaa, editor, *Robot Operating System (ROS):*. Springer, 2017.

[7] Gazebo simulation for crazyflie crtp.
https://github.com/wuwushrek/sim_cf/tree/multi-uav-final.

[8] Carlos Luis and Jérôme Le Ny.

Design of a trajectory tracking controller for a nanoquadcopter.
https://arxiv.org/abs/1608.05786v1, August 2016.
Complete Technical Report.

[9] A. E. C. Da Cunha.
Benchmark: Quadrotor attitude control.
In Goran Frehse and Matthias Althoff, editors, *ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems*, volume 34 of *EPiC Series in Computing*, pages 57–72. EasyChair, 2015.
doi: 10.29007/dc68.
URL https://easychair.org/publications/paper/mwnd.

[10] Ieee standard specification format guide and test procedure for single-axis interferometric fiber optic gyros.
IEEE Std 952-1997, pages 1–84, Feb 1998.
doi: 10.1109/IEEESTD.1998.86153.

[11] Fadri Furrer, Michael Burri, Markus Achtelik, and Roland Siegwart.
Robot Operating System (ROS): The Complete Reference (Volume 1), chapter RotorS—A Modular Gazebo MAV Simulator Framework, pages 595–625.
Springer International Publishing, Cham, 2016.
ISBN 978-3-319-26054-9.
doi: 10.1007/978-3-319-26054-9_23.
URL http://dx.doi.org/10.1007/978-3-319-26054-9_23.

[12] E. Goubault and S. Putot.
Forward Inner-Approximated Reachability of Non-Linear Continuous Systems.
2017.
doi: 10.1145/3049797.3049811.

[13] E. Goubault, S. Putot, and L. Sahlman.
Inner and Outer Approximating Flowpipes for Delay Differential Equations.
actes de Conference on Computer Aided Verification CAV, 2018.

[14] Jorge Stolfi, L H. de FIGUEIREDO, and Estrada Dona.
An introduction to affine arithmetic.
IEEE Std 952-1997, Apr 2003.
doi: 10.5540/tema.2003.04.03.0297.

[15] Frederic Messine and Ahmed Touhami.
A general reliable quadratic form: An extension of affine arithmetic.
Reliable Computing, 12:171–192, 06 2006.

doi: 10.1007/s11155-006-7217-4.