eFrei PARIS PANTHÉON-ASSAS UNIVERSITÉ	**RAPPORT DE STAGE**	**M1**

ELEVE

Nom : **PANNIER** Prénom : **JULIEN** Filière : **Systèmes Embarqués - SRD**

SUJET

Fabrication de matériel pédagogique :
Asservissement d'un bras à hélice avec un degré de liberté

ENTREPRISE

Nom : **EFREI Paris**
Adresse : **30-32 Avenue de la République 94800 VILLEJUIF**

DATE DU STAGE Date de remise du rapport aux membres du Jury :

du : **07/11/2022** au : **31/03/2023** durée effective en semaines : **20**

SOUTENANCE Date : **28/03/23** Heure : **11h30**

Composition du Jury :

- Président (responsable EFREI) : BEN HEDIA BELGACEM
- Responsable du stage (Entreprise) : GRIOT RÉMI & CONTEVILLE LAURIE
- Invité(e) :

PUBLICATION DU RAPPORT DE STAGE

Le Responsable du stage : R. Griot et L Conteville autorise le stagiaire à publier le rapport de stage sur l'Intranet de l'Ecole.
Signature GRIOT

Mots clés

Drones, Robots, Capteurs, Asservissement, Régulateur Proportionnel Intégral Dérivé, Microcontrôleur, Interface Homme Machine, Conception Assistée par Ordinateur, Imprimantes 3D, Projet éducatif.

2022-2023

SYSTÈMES ROBOTIQUES & DRONES Étudiant : PANNIER JULIEN

Stage technique M1 Référent pédagogique : BEN HEDIA BELGACEM

Fabrication de matériel pédagogique : Asservissement d'un bras à hélice avec un degré de liberté

EFREI PARIS

INNOVATION LAB

VILLEJUIF

Tuteurs : CONTEVILLE LAURIE

GRIOT RÉMI

Remerciements

Tout d’abord, je souhaite remercier l’EFREI Paris pour le financement de ce stage et pour la mise à disposition de ses locaux qui m’ont permis de travailler dans un environnement serein et propice à une expérience constructive et productive.

Je tiens également à exprimer mes remerciements à Madame CONTEVILLE et Monsieur GRIOT, pour leur accompagnement, leur disponibilité, leurs enseignements et pour m’avoir donné la possibilité de participer au projet du bras drone.

J’aimerais aussi remercier Monsieur BEN HEDIA, Monsieur BENCHALLAL et Madame BALTAZE pour m’avoir suivi et épaulé sur la partie administrative de cette expérience professionnelle.

Je remercie également Madame CRAGNOLINI et Monsieur N’DO pour leurs enseignements et pour toute l’aide apportée.

Je souhaite de même remercier pour leur gentillesse, les autres stagiaires et doctorants de mon lieu de travail avec lesquels j’ai pu interagir durant mon stage.

Table des matières

Liste des figures

Résumé technique

Le matériel pédagogique proposant une grande adaptabilité et réparabilité à long terme et à faible coût n'est pas forcément aisé à trouver au sein du marché professionnel. C'est pourquoi l'I'Lab de l'Efrei Paris, en partenariat avec la majeure Systèmes Robotiques & Drones, se sont engagés dans le processus de création de matériel pédagogique en interne et répondant à ces critères. Le projet Bras Drone vient répondre au besoin de développer un dispositif pluridisciplinaire pour des cours de robotique et drones et d'asservissement en position. Ce projet mêle *hardware* et *software* : il est composé de plusieurs éléments mécaniques (ossature, liaison rotative, hélice de drone…) et électroniques (moteur de drone, capteurs sensoriels…). La communication s'effectue en série (UART) entre un PC et une carte microcontrôleur présente sur le bras. Le bras couplé au « drone » subit une rotation sur son axe commun avec le tronc de la structure, grâce à la force de poussée. Optimiser la communication série entre le PC de l'utilisateur et le dispositif a introduit l'idée de développer une interface homme machine en langage Python (Tkinter), à la suite d'une refonte de la machine d'état propre au fonctionnement du dispositif (*firmware* du microcontrôleur en C). Des développements matériels ont également été effectué dans cette optique de consolider le bras drone.

Mots clés : Drones, Robots, Capteurs, Asservissement, Régulateur PID, Microcontrôleur, Interface Homme Machine, Conception Assistée par Ordinateur, Imprimantes 3D, Projet éducatif.

Abstract

Highly adaptable and repairable teaching material is not necessarily easy to find in the market. That is the reason why the I'Lab of Efrei Paris in partnership with the major Robotics Systems & Drones is engaged in the process of educative material's in-house creation which can meet those requirements. The Drone Bench project satisfies the need to develop multidisciplinary devices for classes of robotics & drones and classes of automatic control of position. This project combines hardware and software as it is composed of mechanical (structure, rotary connection, drone's helix…) and electronical (drone's motor, sensors…) parts. The communication is serial (UART protocol) between the PC and the arm's microcontroller. The arm, which is coupled to the "drone part", rotates on a common axis with the stem of the structure thanks to the thrust force. Optimizing serial communication between the user's computer and the device introduced the idea to develop a graphical user interface in Python (Tkinter), just after a complete overhaul of the state machine behind the device's functioning (firmware of the microcontroller in C language). Material developments had also been made on the drone bench, following the same idea of upgrading the device.

Key words: Drones, Robots, Sensors, Automatic Control, PID Controller, Microcontroller, Graphical User Interface, Computer Aided Design, 3D Printers, Educational Project.

Présentation de l’environnement du stage

- ## L’Efrei

L’*École française de radioélectricité, d’électronique et d’informatique* (Efrei) est une école privée d’ingénieurs généralistes et experts du numérique, en perpétuelle évolution depuis sa fondation en 1936, qui dispose de l’habilitation par la Commission des Titres Ingénieur à délivrer le titre d’ingénieur depuis 1957 et qui a rejoint l’université Paris-Panthéon-Assas en tant qu’établissement composante depuis l’année 2022. L’Efrei dispose de plusieurs bâtiments dont deux campus dans les villes de Villejuif et de Bordeaux. Le campus principal à Villejuif comporte en son sein un bâtiment inauguré en 2021, dans lequel s’effectue le stage présenté dans ce document : l’Innovation’Lab.

- ## L’Innovation’Lab

L’Innovation’Lab est un espace de travail dédié aux étudiants, enseignants et associations étudiantes disposant d’outils pour la conception et la réalisation de projets. L’Innovation’Lab est aussi par extension, une entité d’Efrei Paris, avec pour mission principale d’élargir le spectre innovatif de cette dernière en proposant par exemple des projets pédagogiques, tel que le sujet de ce stage, qui à terme permettront à l’école de s’assurer une certaine forme d’auto-suffisance matérielle.

- ## Les acteurs du projet « Bras drone »

Le projet « Bras drone à un degré de liberté », proposé par l’Innovation’Lab (I’Lab) en partenariat avec la majeure Systèmes Robotiques & Drones (SRD), est encadré par Monsieur GRIOT Rémi, en sa qualité de manager de l’I’Lab et de Madame CONTEVILLE Laurie, enseignante-chercheuse et responsable de la majeure SRD.

Les bénéficiaires du projet sont l’Efrei Paris, les enseignants de la filière Systèmes Embarqués et de la majeure SRD, ainsi que les étudiants de cette même filière.

Stage

1 Introduction

L'idée du projet d'un bras drone à un degré de liberté est née du besoin de pouvoir utiliser et fabriquer du matériel pédagogique modifiable et réparable, relativement aisément, mais aussi à faible coût via les moyens mis à disposition par l'Efrei. Or l'I'Lab dispose d'outils (imprimantes 3D, découpeuse laser …) offrant ces possibilités !

L'opportunité est apparue au sein de la majeure SRD, avec l'idée d'une maquette utile à la compréhension du fonctionnement d'un drone, dont les différents composants et capteurs seraient facilement accessibles et qui se proposerait comme pluridisciplinaire.

Ce stage fait suite au développement conceptuel et matériel du bras, qui fut le sujet d'un stage technique réalisé l'an dernier (2021-2022) par mon prédécesseur Monsieur N'DO Yann Kader Axel Obou, également élève de la majeure Systèmes Robotiques & Drones de l'Efrei.

1.1 Présentation succincte du dispositif

Le dispositif du bras drone est constitué de plusieurs éléments à caractère mécanique et électronique. Le squelette est composé de profilés en aluminium :

- Deux de ces profilés sont montés en T via des équerres de fixation, faisant office de socle pour le premier et de tronc pour le second ;
- Un troisième profilé est utilisé comme bras, monté en liaison pivot par rapport au tronc (1 degré de liberté : une rotation possible sur un seul axe, avec un champ de manœuvre limité par le socle), au bout duquel se trouve la « partie drone » soit un moteur couplé à une hélice composée de 3 pâles.

Le principe du dispositif est donc d'assurer le mouvement du bras par rapport au tronc, afin d'effectuer une rotation sur un seul plan, dans une fourchette de 0 à 90° (0° étant la position de repos théorique du bras à la verticale), grâce à la poussée du drone (voir le bras avec le moteur en fonctionnement sur la Figure 1). Au-delà du fait de simplement déplacer le bras par rapport au tronc, le but pédagogique du bras drone est de pouvoir assurer un asservissement de la vitesse de rotation du moteur afin que le bras puisse rester stable autour d'une consigne de position angulaire.

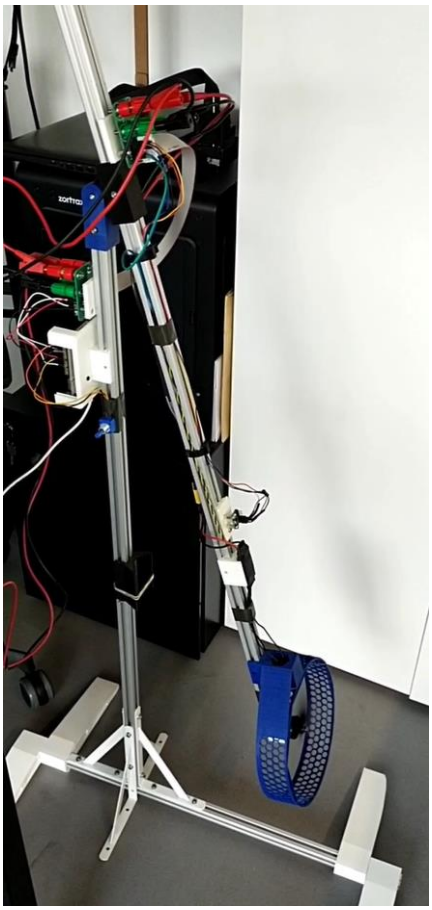


Figure 1: Photographie du dispositif au début du stage

Ainsi, le dispositif comprend également une carte de développement NUCLEO-L476RG de STMicroelectronics (Figure 2) dont le microcontrôleur ARM Cortex M4 STM32L476RGT6 permet la gestion des différents capteurs et éléments électroniques : un moteur *brushless* et un contrôleur de vitesse électronique (ESC *Electronic Speed Controller*) issus du commerce de drones ; un potentiomètre ; une centrale inertielle (MPU-6050 de chez *InvenSense* / protocole de communication I2C) nécessaire pour déterminer la position angulaire du bras et un générateur de laboratoire pour alimenter le système (voir sur la partie gauche de la photographie associée au schéma fonctionnel de la Figure 3).

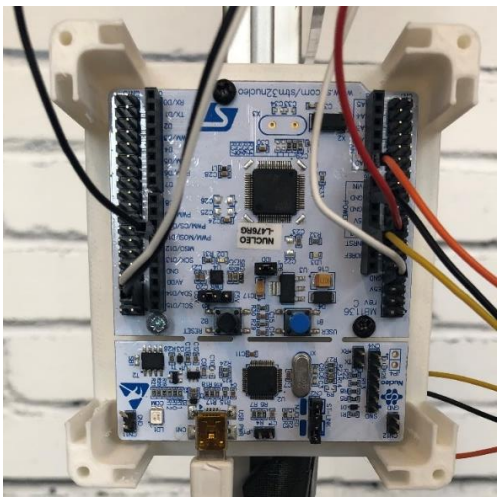
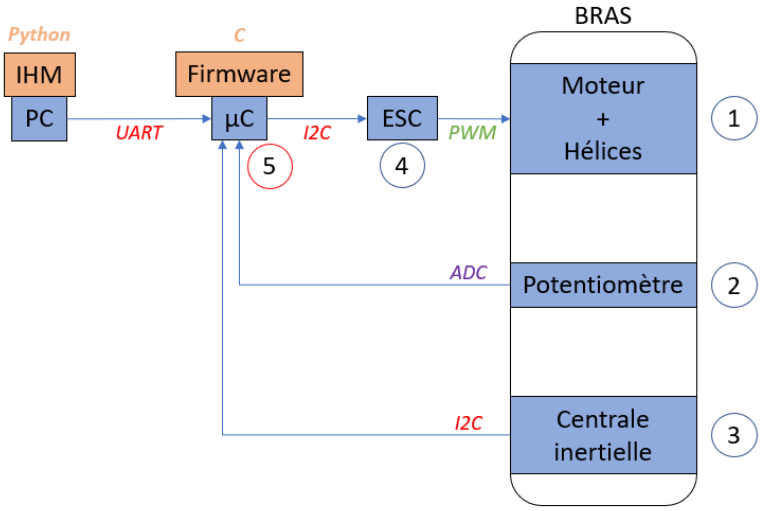


Figure 2: Photographie de la carte STM32

La carte STM32 dispose d'un environnement de développement STM32 Cube IDE permettant d'accéder aux fonctionnalités de cette dernière, à ses registres et de développer en langage C, le *firmware* implémenté dans la carte. La communication avec la carte se fait en série via le protocole UART RS232 avec un Baud Rate de 115200 Bits/s et un « *Word Length* » de 8 Bits. L'utilisation d'un terminal UART permet de rentrer des commandes sur ordinateur.



IHM : Interface Homme Machine
µC : Microcontrôleur
ESC : Electronic Speed Controller

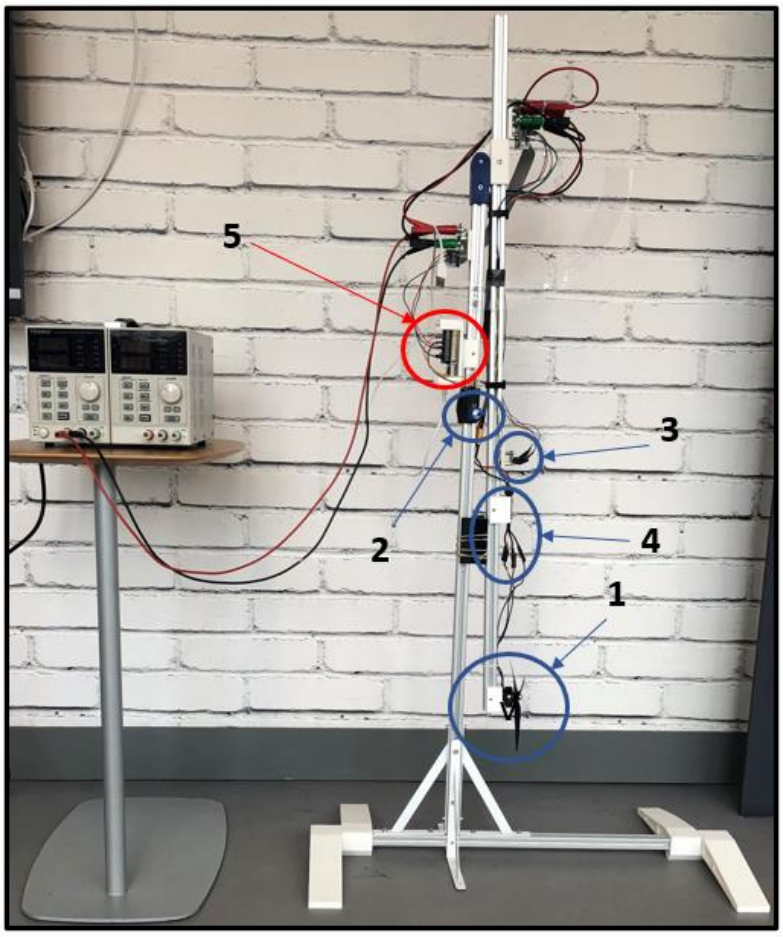


Figure 3: Schéma fonctionnel du dispositif photographié

1.2 Missions et objectifs du stage

En début de stage, le dispositif est dans un état fonctionnel : il est possible de communiquer avec la STM32, via un terminal série, en suivant les conditions de passages d'un état à un autre de la machine d'état développée dans le fichier source « *main.c* » via l'instruction « *switch* » et ses instances « *case* » (voir la machine d'état illustrée par la Figure 4).

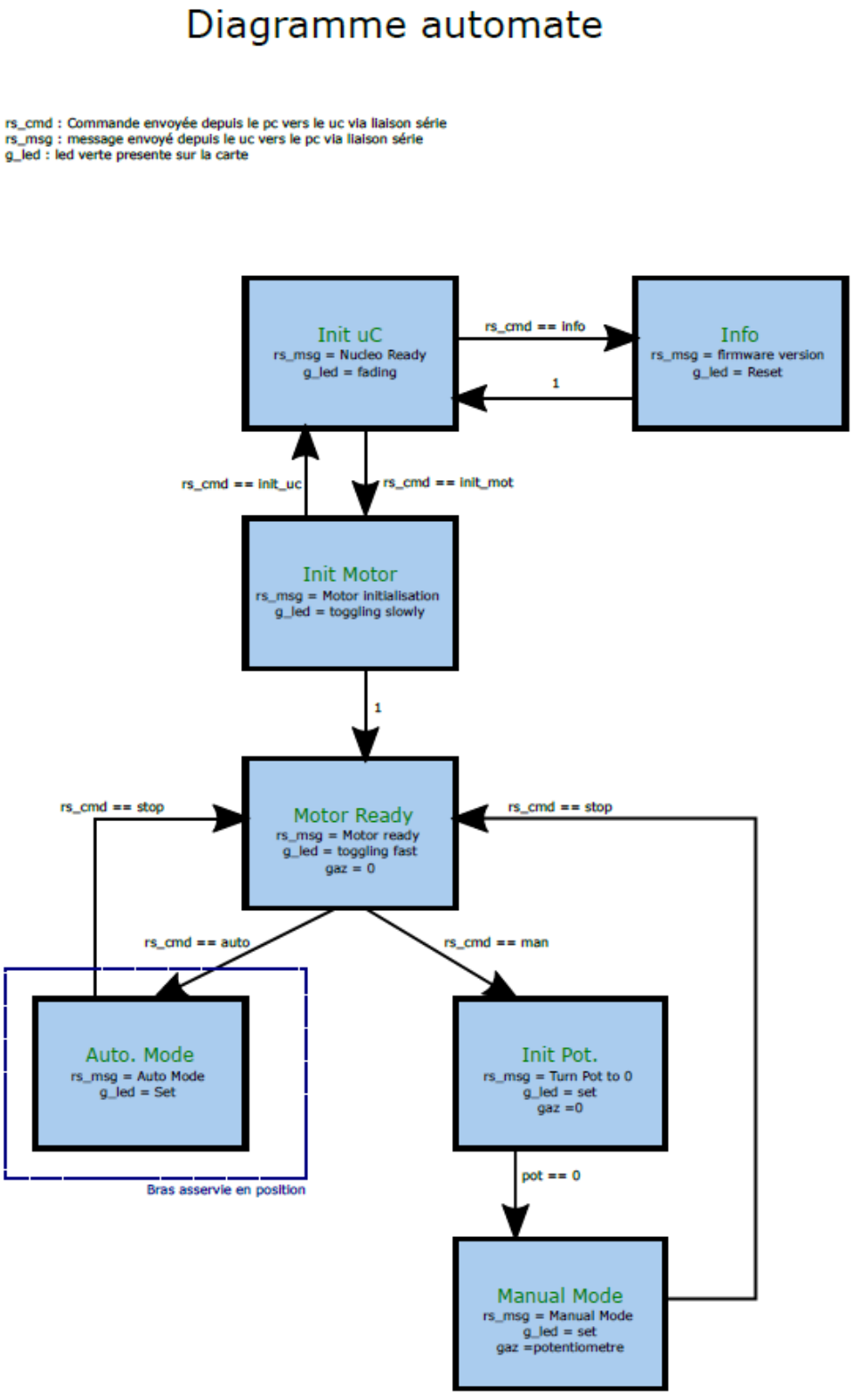


Figure 4: Machine d'état initiale

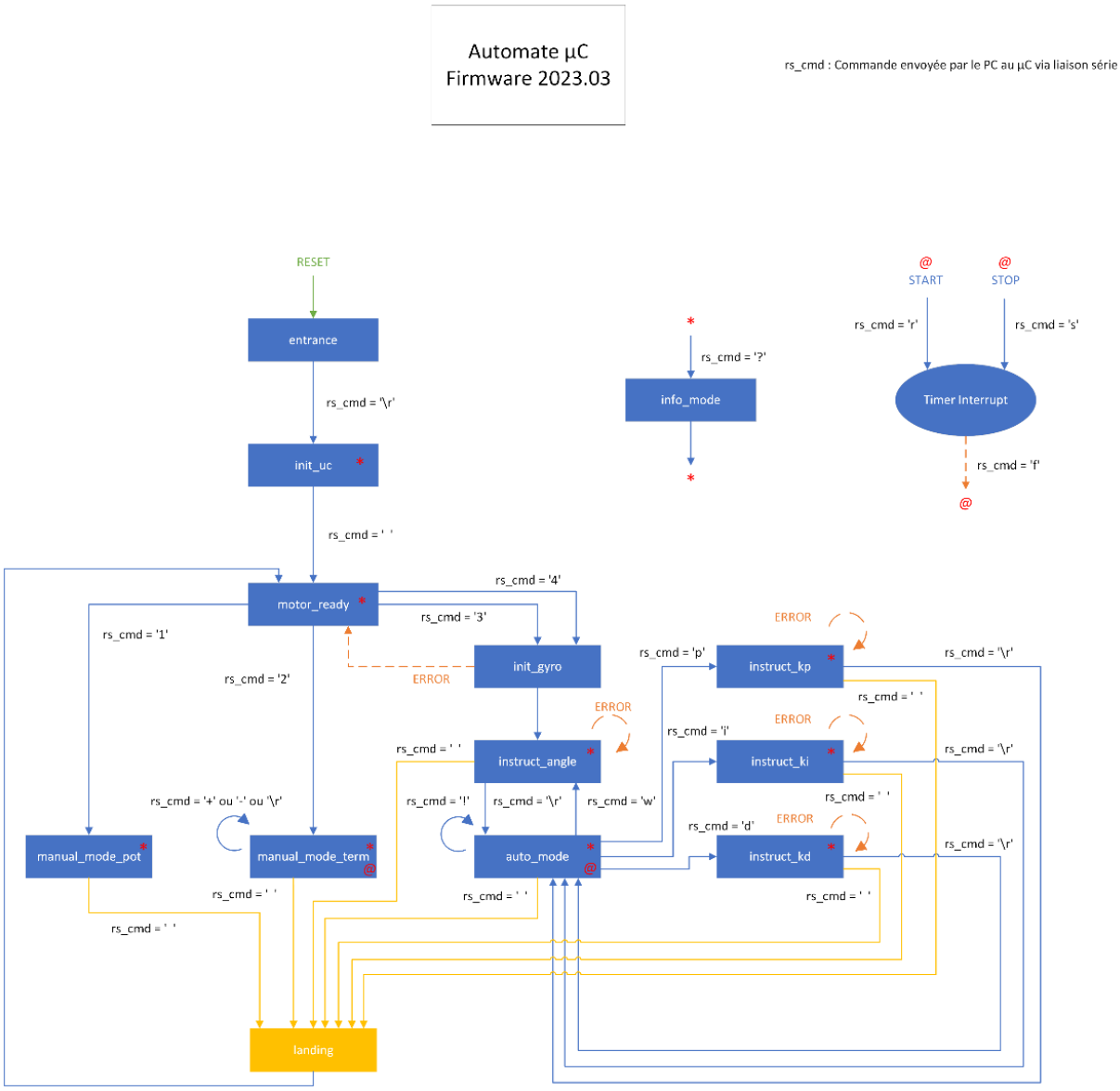


Figure 5: Machine d'état finale

La machine d'état initiale (Figure 4) se déroule de la manière suivante : l'état de départ « *Init_UC* » est un état de transition dans lequel il est possible d'accéder à l'état « *Info* » (qui propose un récapitulatif des différents modes et autres informations relatives à la simulation) ou bien à l'état « *Init Motor* » par sélection du caractère spécifique associé (un chiffre).

L'état « *Init Motor* » vérifie s'il y a bien une acquisition de signal en provenance du moteur, auquel cas la transition à l'état suivant « *Motor Ready* » est possible.

C'est dans « *Motor Ready* » qu'il est alors possible de choisir (par sélection d'un caractère également) entre plusieurs modes d'interaction avec le moteur :

- « *Manual Mode Pot* », pour une interaction manuelle avec le moteur via le potentiomètre ;
- « *Manual Mode Term* », dans lequel l'utilisateur est amené à renseigner une consigne de pourcentage de gaz du moteur ou bien une incrémentation de plus ou moins 1% de cette même variable ;

- « *Auto Mode* », le mode asservi pour une consigne fixe de 45°, prenant en compte des coefficients du régulateur PID (Proportionnel Intégral Dérivé) fixes renseignés dans le fichier source « *main.c* ». L'état « *Init Pot* » est un état transitoire, dans lequel on vérifie si le potentiomètre est bien connecté et initialisé. A savoir qu'il est tout à fait possible de revenir à « *Motor Ready* » si l'on désire changer de mode de fonctionnement.

Les premières missions qui m'ont été attribuées, à partir de cette étape d'avancement du projet, sont :

- L'optimisation de la communication entre la STM32 et le PC ;
- L'amélioration des performances de la boucle d'asservissement ;
- L'amélioration de l'affichage des données.

Pour se faire, il m'a été conseillé de suivre les étapes suivantes :

- Prendre en main l'outil de gestion de version Git et de la plateforme GitHub, afin de pouvoir établir un environnement de travail rigoureux ;
- Prendre en main STM32 Cube IDE, dans le but de pouvoir continuer à développer le *firmware* de la STM32, soit optimiser la machine d'état, donner plus d'ampleur au mode asservi (sélection d'une consigne autre que 45° et pouvoir modifier les coefficients PID plus aisément), établir un mode atterrissage car, à l'heure actuelle, le bras doit être rattrapé manuellement lors de l'arrêt du moteur ou bien encore établir un mode « séquence de vol » (détaillé un peu plus tard dans ce rapport) ;
- Prospecter les librairies d'affichage de graphiques en temps réel et de développement d'IHM (Interface Homme Machine) en langage Python, dans l'optique de développer une IHM et de faciliter la communication STM32/PC.

1.3 Outils et environnements de développement

Afin de pouvoir débuter mon stage dans un cadre de travail rigoureux et efficace, j'ai tout d'abord suivi une courte formation sur l'utilisation de l'outil de gestion de versions Git. Cet outil est à utilisation quotidienne lors de ce stage, car il me permet de mettre à jour les fichiers sur lesquels je travaille, de pouvoir récupérer d'éventuelles nouvelles versions de fichiers sur lesquels mon tuteur serait amené à travailler et également de lui mettre à disposition mon avancement sur telle ou telle partie du projet. L'I'Lab possède son propre dépôt sur GitHub, dans lequel se trouve un sous dépôt dédié au projet du bras drone (Annexe 1).

Une fois l'outil de gestion de versions maîtrisé, j'ai alors pu passer à l'étape suivante soit commencer à utiliser STM32 Cube IDE et regarder en détails les différents fichiers sources, ainsi que la machine d'état.

Le logiciel STM32 Cube IDE est un environnement de développement et d'utilisation de cartes STM32 proposé par STMicroelectronics. Cette plateforme propose à chaque début de projet de retrouver la carte en question en sélectionnant la référence de cette dernière dans un catalogue.

Par la suite, il est alors possible de configurer les différents *PIN* en sortie (Figure 6), de choisir des protocoles de communication, de configurer des *Timers* etc.

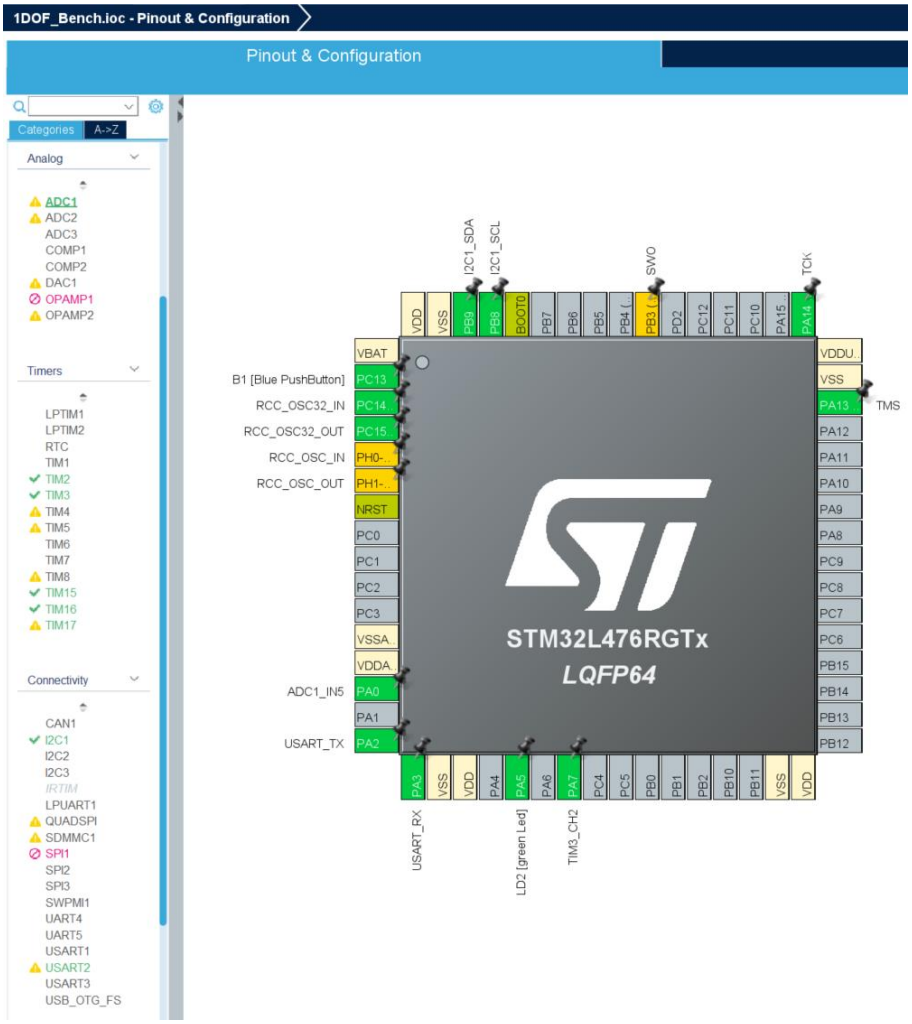


Figure 6: Configuration de la STM32 sur Cube IDE

L'environnement STM32 Cube IDE reprend également les fonctionnalités d'un IDE « traditionnel » soit la possibilité de travailler sur des fichiers en C ou C++ et d'avoir accès à une console et un *debugger*.

2 Firmware & Software

A partir de la machine d'état présentée dans l'introduction (Figure 4), l'objectif premier a été d'optimiser le fonctionnement de l'automate (la partie *firmware*). Par la suite, le développement d'une IHM a été réalisé (partie *software*).

2.1 Refonte de la machine d'état du firmware

Le passage d'un état à un autre se fait par « *polling* » c'est-à-dire qu'à partir d'une certaine ligne du code, au sein d'un état, une boucle « *do while* » est utilisée afin de vérifier constamment si un caractère spécifique est saisi. Si tel est le cas alors il sera possible de sortir de cette condition et d'attribuer à ce caractère, la commande pour atteindre prochainement l'état qui lui est associé.

J'ai, en premier lieu, voulu faire une refonte de la machine d'états, d'une part en y ajoutant plus de messages informatifs pour faciliter l'interaction via le terminal (Figure 7), mais aussi afin de simplifier le développement de cette dernière. Le premier point marquant fut de fusionner l'état « *Init Pot* » avec l'état « *Manual Mode Pot* » afin de diminuer le nombre d'états transitoires et de faciliter la compréhension de l'automate.

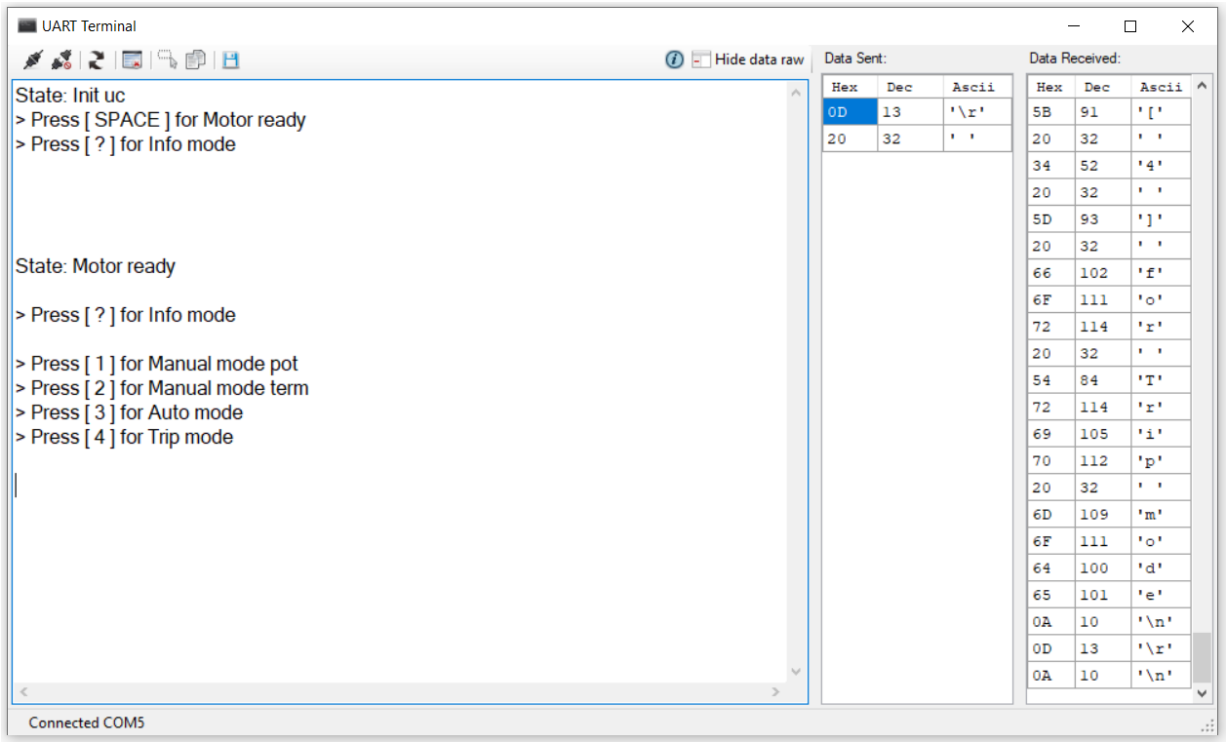


Figure 7: Communication avec le UART terminal

Ensuite, il m'avait été demandé de permettre un accès à l'état informatif « *Info Mode* » depuis n'importe quel autre état et également dans un but de simplification, j'y ai attribué le caractère « ? » qui me semblait plus intuitif qu'un chiffre, que j'ai alors préféré réserver aux modes de fonctionnement : '1' pour « *Manual Mode Pot* » / '2' pour « *Manual Mode Term* » / '3' pour « *Auto Mode* ».

Le mode « *Info* » fonctionne désormais tel qu'une fois avoir demandé l'accès à ces informations, qui sont par la suite affichées sur le terminal (par « *printf* »), on retourne directement à l'état

dans lequel on se trouvait précédemment, ce qui permet entre autres d'accéder aux informations, sans pour autant compromettre l'action en cours (le moteur va pouvoir continuer à fonctionner et le bras va pouvoir rester à une certaine position, par exemple).

L'état « *Motor Ready* » quant à lui est toujours accessible depuis l'un de ces 3 modes, à ceci près qu'il est maintenant possible de sortir d'un mode de fonctionnement à n'importe quel moment où il est demandé d'interagir avec la console.

A cette étape du développement, il m'est paru sensé de créer un état transitoire d'atterrissage « *Landing* », ainsi qu'une fonction associée, car l'arrêt des moteurs devenait assez récurrent lors des tests à la suite du développement de la machine d'état. Cette fonction reprend le principe de décrémentation successive de la valeur consigne assignée au moteur, jusqu'à atteindre la coupure du moteur, ce qui permet au bras de retrouver sa position de départ, tout en douceur.

Afin d'éviter d'autres potentiels « dangers » ou dommages sur le dispositif, j'ai décidé alors d'ajouter des garde-fous accompagnés de messages d'erreurs sur les 2 modes de fonctionnement « *Manual Mode Term* » et « *Auto Mode* » présentant une partie interactive (demande de consigne). Par exemple, le mode 2 « *Manual Mode Term* » requiert une valeur consigne de pourcentage de gaz à attribuer au moteur et par expérimentation, j'ai constaté qu'une valeur supérieure à 10%, appliquée directement au moteur (et non attribuée par incrémentation successive) donnait beaucoup trop d'élan au bras, qui partait alors au-delà de sa position angulaire limite (90°).

Ensuite, il a été convenu que le mode « *Init UC* » ne devrait plus être accessible depuis « *Motor Ready* » car une initialisation de la carte prenait plus de sens par son caractère unique. Un état d'entrée a alors également été suggéré afin de ne pas rater certains messages de l'état « *Init UC* » lors de la lecture des « *printf* », mais aussi dans un but de clarté.

Ainsi, la nouvelle machine d'état se caractérise par l'état « *Entrance* », dans lequel on attend que l'utilisateur presse la touche « Entrée » ('\r' dans la colonne « *Data Sent* » de la Figure 7), afin de pouvoir accéder à l'état « *Init UC* ». Dans cet état, les messages nous proposent d'entrer le caractère ' ?' pour obtenir des informations sur la simulation ou bien d'entrer la touche « Espace » (' ' dans la colonne « *Data Sent* » de la Figure 7), spécifique à l'accès à « *Motor Ready* », mais aussi parallèlement utilisé pour l'atterrissage. Une fois « *Motor Ready* » atteint, il nous est alors demandé de saisir '1', '2' ou '3' (ou encore ' ?') et donc d'exprimer notre choix de mode (« *Motor Ready* » est donc similaire à un menu).

Enfin, le mode 3 « *Auto Mode* » nécessitait d'offrir plus de possibilités d'interaction, car pour rappel, la valeur de position angulaire et les coefficients PID étaient préalablement choisis dans le fichier source et il m'était alors paru plus intéressant de pouvoir saisir ces valeurs via le terminal.

J'ai donc rajouté la possibilité de saisir une valeur consigne de position angulaire via l'accès à un état transitoire « *Instruct Angle* », qui une fois rentrée nous amène à l'état « *Auto Mode* » ou bien nous ramène à ce même état, si l'on a demandé à accéder à « *Instruct Angle* » pour changer de position angulaire (c'est-à-dire qu'il est maintenant possible d'effectuer un changement de position en plein vol, ce qui aura une certaine utilité pour le développement d'un mode séquence de vol).

La saisie des coefficients PID se fait également via des états transitoires, mais par modification. Entre autres, des valeurs par défaut sont déjà présentes dans le code, car l'on suppose qu'il ne sera pas toujours nécessaire de vouloir saisir des coefficients PID (il est également possible de saisir une commande de retour aux valeurs par défaut si besoin).

Une fois toutes ces étapes franchies, il m'a été alors conseillé de débuter le développement d'une IHM en Python, où il serait alors possible de rajouter un mode séquence de vol et un mode calibration. Ce dernier permettrait à l'utilisateur d'acquérir des couples de données [consigne] (en % de gaz moteur) & [position angulaire] calculés via les données récoltées par la centrale inertielle, afin de pouvoir tracer une courbe et de pouvoir étudier les comportements statique et dynamique du dispositif.

2.2 Développement d'une IHM : COM Manager & Menu de modes

J'ai en premier temps cherché les bibliothèques qui me permettraient de créer une IHM en Python. J'ai préféré débuter le développement de l'IHM souhaitée avec *Tkinter* (plutôt que *PyQt* ou *PySide*) qui est référencée comme la bibliothèque la plus simple pour commencer à créer des IHM et dont la documentation était suffisamment riche pour pouvoir atteindre mes objectifs dans le délai qui m'était accordé.

J'ai alors suivi plusieurs tutoriels afin de pouvoir me lancer dans le développement de cette IHM. L'un des tutoriels se présentait de la manière suivante : créer une unique fenêtre principale, qui s'agrandirait à mesure que l'on cliquerait sur certains boutons. Le concept s'apparentait aux attentes de mes tuteurs et je me suis donc orienté sur ce design d'interface. La personne ayant réalisé ce tutoriel présentait le développement d'une IHM en Python pour pouvoir communiquer en série avec une carte STM32. Ainsi, une certaine partie des explications concernait également la communication en série UART, ce qui m'est avéré bien utile pour mon projet. Je me suis donc fortement inspiré de l'aspect de l'IHM de ce tutoriel, notamment de la racine de cette fenêtre principale (Figure 8), similaire au menu de connexion du terminal série que nous utilisons.

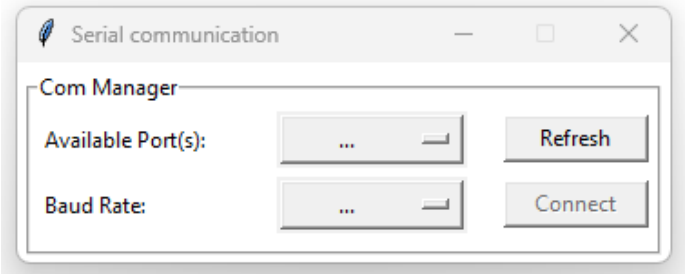


Figure 8: Fenêtre de communication série de l'IHM

L'interface offre donc la possibilité de sélectionner le port de communication détecté, ainsi que le « *Baud rate* » (vitesse de transmission de données en série). Une fois qu'ils sont sélectionnés, le bouton « *Connect* » n'est plus grisé comme dans l'exemple de la Figure 8 (widget à l'état « *disabled* » dans le code) et il est alors possible d'accéder à une première extension de fenêtre si la connexion est bien ouverte et fonctionnelle. Un message de validation s'affiche dans une nouvelle fenêtre « *pop-up* » dans le cas où le port COM est bien accessible et si la connexion est aboutie ; dans le cas contraire, un autre message s'affichera, annonçant que la connexion n'a pas pu se faire (Figure 9).

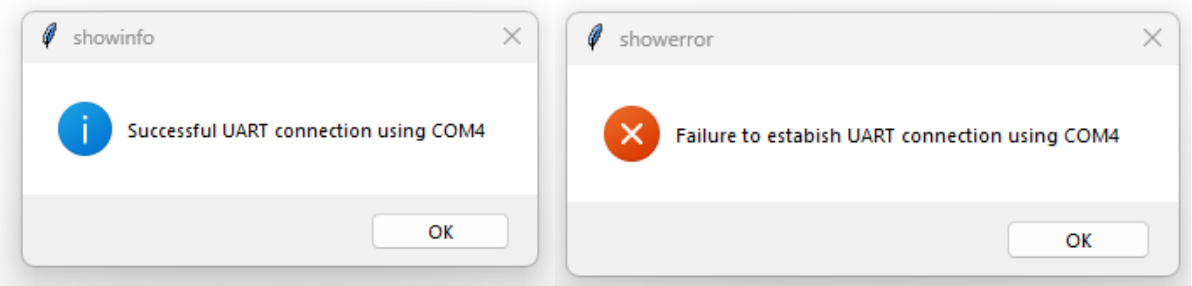


Figure 9: Messages de connexion (bouton Connect)

Une connexion non aboutie peut être due notamment à une communication déjà existante via le terminal série UART par exemple.

Le code Python de cette IHM est développé sur plusieurs fichiers « .py » dont un fichier principal nommé « *Master.py* », lequel est exécuté lorsque l'on veut utiliser l'IHM et qui fait appel à plusieurs autres fichiers « .py » regroupant les différentes classes et fonctions utilisées.

L'un de ces fichiers est appelé « *GUI_Master.py* », GUI pour *Graphical User Interface* (un équivalent courant en anglais d'IHM), et regroupe la majeure partie du code, dont des classes propres à chacune des extensions de fenêtres et leurs fonctions associées qui traitent des *widgets*.

Dans ce fichier, on retrouve donc la classe propre à la fenêtre principale, « *ComGui* » pour la communication série (TAG_IHM_002 dans le répertoire des balises en Annexe 2 ; ces balises sont présentes en commentaires dans les fichiers .py et .c intervenant dans ce projet), et cette même classe contient des fonctions qui sont appelées lors des interactions avec les widgets. Soit par exemple la fonction « *ComOptionMenu* » qui s'occupe de gérer l'obtention et l'affichage de la liste des ports COM dans le widget « *drop_com* » (en face du label « *Available Port(s)* » visible sur la Figure 8).

Cette classe fait également appel au module « *pyserial* » de Python, utilisé pour pouvoir communiquer en série. Les fonctions de « serial » sont essentielles à la réalisation de l'IHM, car elles établissent l'envoi et la réception des données dont nous avons besoin.

Les fonctions « *write* » et « *readline* » de « *pyserial* » représentent à elles deux, la clé de voûte de notre système de communication : entre autres, elles nous permettent d'envoyer les entrées « *char* » que nous entrions manuellement via le terminal série, mais aussi de lire les « *printf* » du *firmware* qui comportent les données que nous cherchons à acquérir.

Un fichier réservé à la communication série « *Serial_COM_ctrl.py* » décline les différentes fonctions nécessaires à l'ouverture et à la fermeture du port COM, mais aussi à l'écriture et à la lecture de données (TAG_IHM_008).

La première extension de fenêtre (Figure 10), nous donne accès à un menu de modes via plusieurs boutons.

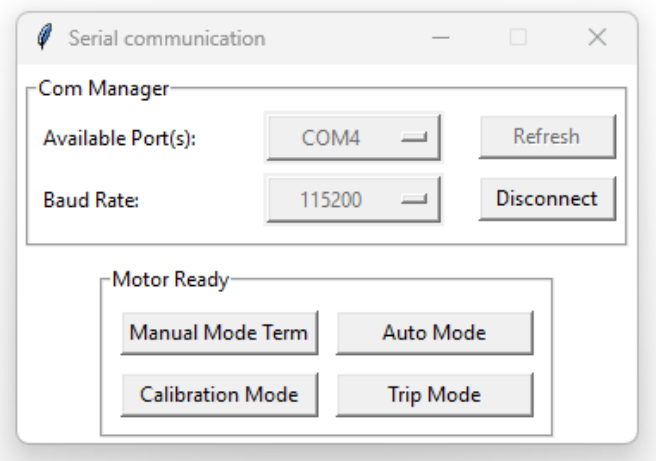


Figure 10: Extension de fenêtre propre au menu de modes

Deux premiers boutons correspondent aux 2 modes interactifs de l'automate : « *Manual Mode Term* » et « *Auto Mode* ». Chacun de ces boutons sont définis dans la classe « *MotorReadyGUI* » (TAG_IHM_003) et renvoient à des fonctions bien précises, qui elles-mêmes sont destinées à étendre encore une fois la fenêtre, à envoyer les bons caractères propres aux états correspondants et donc à nous renvoyer vers une nouvelle classe du même fichier (classes qui regroupent les fonctions du mode en question).

Deux autres nouveaux modes proposent à partir des deux modes originaux, d'automatiser une séquence de vol « *Trip Mode* » et une simulation de calibration « *Calibration Mode* ».

On notera aussi que le bouton « *Connect* » est maintenant désigné comme « *Disconnect* » et permet alors de fermer la communication série à tout moment.

2.3 Développement d'une IHM : Modes non asservi et asservi

La seconde extension de fenêtre se produit donc lorsque l'utilisateur interagit avec l'un des 4 boutons du frame « *Motor Ready* » (Figure 10) et l'on voit alors apparaître à droite une nouvelle section propre à chacun de ces modes.

En premier temps, j'ai réalisé le contenu de l'extension du mode « *Manual Mode Term* » afin de pouvoir mettre en place, un peu plus tard, le mode calibration.

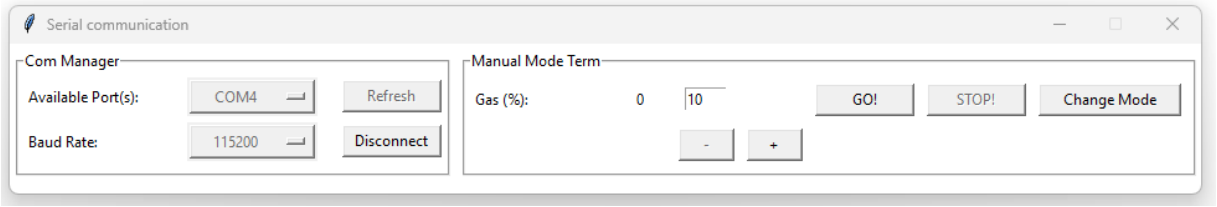


Figure 11: Extension de fenêtre propre au mode « Manual Mode Term », avant envoi d'une première consigne

Cette section de fenêtre (Figure 11) est définie dans la classe « *ModeTermGUI* » du fichier « *GUI Master* » (TAG_IHM_004) et se caractérise via les widgets suivants :

- Un premier label comportant l'intitulé de la consigne, qui pour ce mode correspond à la valeur de pourcentage des gaz du moteur ;

- Un second label utilisé en tant qu'écran factice, qui se trouve être relié à la prise en compte de l'entrée, qui pour cette dernière peut être renseignée textuellement via une « *entry box* » ou bien via les boutons « + » et « - » ;
- Un bouton « *GO !* » qui permet de vérifier si la valeur consigne textuelle est correcte (si elle répond aux critères d'être un entier positif et inférieure à une valeur maximale que l'on aurait fixé au préalable pour des raisons de sécurité par exemple) et ensuite d'envoyer cette valeur, soit démarrer les moteurs (pour une 1ère valeur) ou bien sélectionner une nouvelle consigne en plein vol ;
- Un bouton « *STOP !* » qui s'active une fois que la simulation a débuté et qui permet d'amorcer l'atterrissage et ensuite de revenir au début du mode « *Manual Mode Term* » (utile car cela nous permet d'enclencher directement une autre simulation sans avoir à revenir au menu de modes, à cliquer une fois encore sur le bouton « *Manual Mode Term* » et ainsi de suite) ;
- Un bouton « *Change Mode* » qui répond donc à un besoin de changer de mode sans avoir à redémarrer l'IHM.

Ainsi, on peut débuter une simulation soit en renseignant une valeur correcte dans l'« *entry box* » puis en cliquant sur le bouton « *GO !* » ou bien en cliquant sur le bouton « + » ce qui fera alors varier la valeur du label écran de « 0 » à « 1 ».

Ce label a une importance particulière, car il m'a permis de résoudre certains soucis liés à la prise en compte de la consigne, lorsqu'il n'y avait alors que l'« *entry box* » pour garder en mémoire et afficher cette même consigne. Les boutons « + » et « - » étaient alors reliés à la valeur textuelle et du fait de son caractère modifiable, il y avait alors des erreurs de cohérence entre valeur affichée et valeur prise en compte par l'automate. J'ai donc résolu ce petit problème grâce à cette idée d'écran factice qui vient s'actualiser selon les 2 options de renseignement de la consigne (Figure 12).

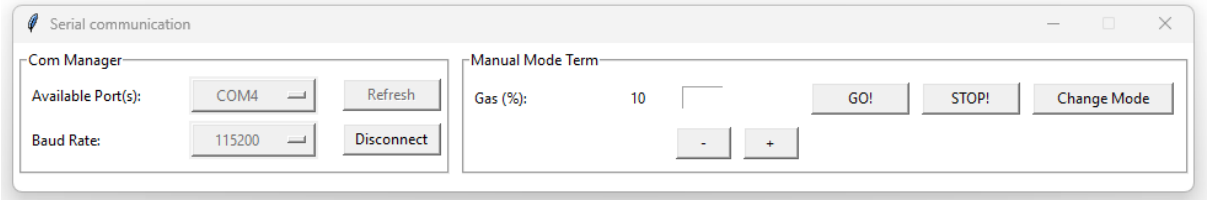


Figure 12: Extension de fenêtre propre au mode « Manual Mode Term », après envoi de la consigne

Lorsque la valeur consigne prise en compte n'est plus à « 0 », le bouton « - » et le bouton « *STOP !* » s'activent (et sont dégrisés). Il est alors possible de renseigner une nouvelle valeur ou bien d'arrêter la simulation via un atterrissage qui peut s'effectuer conventionnellement via le bouton « *STOP !* » ou bien via le bouton « *Change Mode* » ou encore si l'on ferme la fenêtre principale (par mesure de sécurité).

Par la suite, j'ai défini une nouvelle classe « *AutoModeGUI* » (TAG_IHM_005) qui reprend le même principe que celle attitrée au mode « *Manual Mode Term* ». Son apparence est assez similaire, car il faut pouvoir renseigner une consigne textuellement, mais il faut aussi prendre en compte, cette fois-ci, des valeurs des coefficient PID, en rajoutant de nouveaux *widgets* !

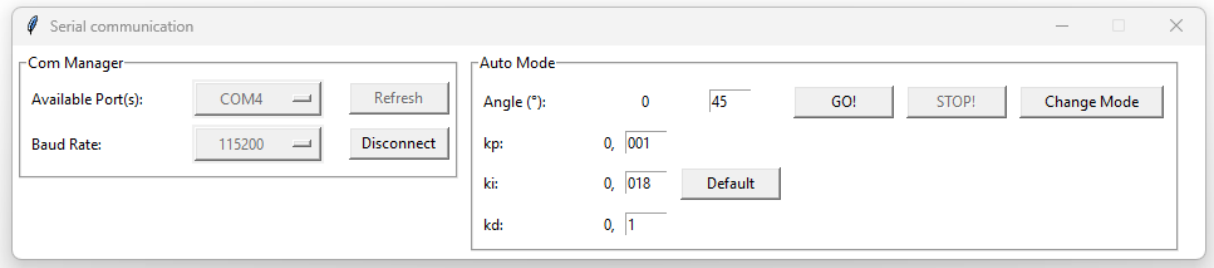


Figure 13: Extension de fenêtre propre au mode « Auto Mode », avant envoi d'une première consigne

La section (Figure 13) se compose donc des mêmes éléments que « *Manual Mode Term* » (Figure 11), à ceci près qu'il n'y a plus de boutons « + » et « - » mais 3 nouvelles lignes de labels et « *entry box* », ainsi qu'un bouton « *Default* » permettant de revenir aux valeurs des coefficients PID par défaut.

Les *entry boxes* des coefficients sont accessibles lorsque le moteur est éteint, c'est-à-dire lors de l'attente d'une première consigne (Figure 13) ou bien après avoir actionné le bouton « *STOP !* ».

Lorsque la simulation est lancée, les *entry boxes* des coefficients, ainsi que le bouton « *Default* » sont grisés. Il est alors encore possible de modifier la valeur consigne et d'appuyer sur le bouton « *GO !* » si l'on veut changer de position en plein vol (Figure 14).

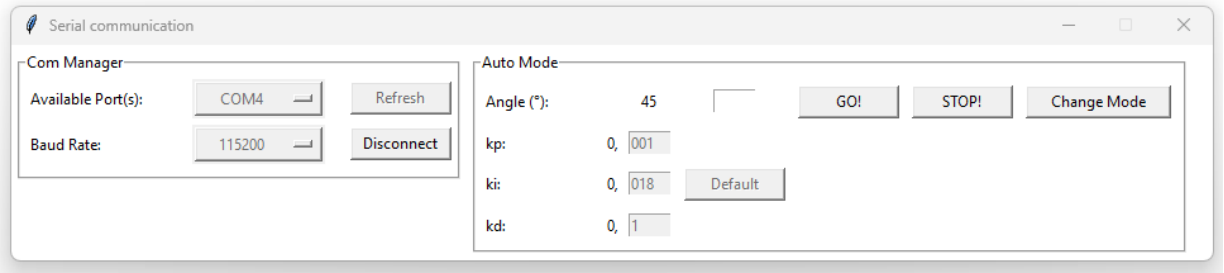


Figure 14: Extension de fenêtre propre au mode « Auto Mode », après envoi de la consigne

A cette étape du stage, les objectifs de départ ont été atteints et vient alors le moment d'approfondir les missions du stage : continuer le développement de l'IHM en mettant en place les modes calibration et séquence de vol ; améliorer ces mêmes modes afin d'avoir un plus grand champ de manœuvre et aussi dans le but d'améliorer les conditions d'asservissement du dispositif.

2.4 Mode non asservi avec prise de mesures : « *Calibration* »

La classe « *CalibrationGUI* » (TAG_IHM_006) du fichier « *GUI Master* » reprend encore la même structure d'extension de fenêtre, mais cette fois-ci elle est un peu plus épurée (Figure 15).

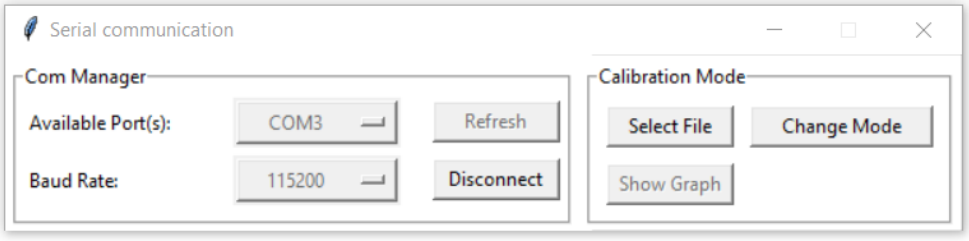


Figure 15: Extension de fenêtre propre au mode « Calibration »

En effet, elle dispose de moins de *widgets*, en grande partie car l'interaction et le renseignement des consignes n'est plus manuelle mais automatisée via un script préétabli dans un fichier .txt que l'on va alors pouvoir compléter en suivant quelques règles.

Un premier bouton « *Select File* » fut nécessaire pour répondre au besoin spécifique de cette méthode d'interaction. Lorsque l'utilisateur clique sur ce dernier, une seconde fenêtre laisse apparaître une boîte de dialogue grâce au module « *filedialog* » de la librairie *Tkinter* (Figure 16), permettant de rechercher puis sélectionner un fichier. Sur la Figure 16, on peut observer dans quel répertoire se trouve le fichier texte en question qui est à compléter.

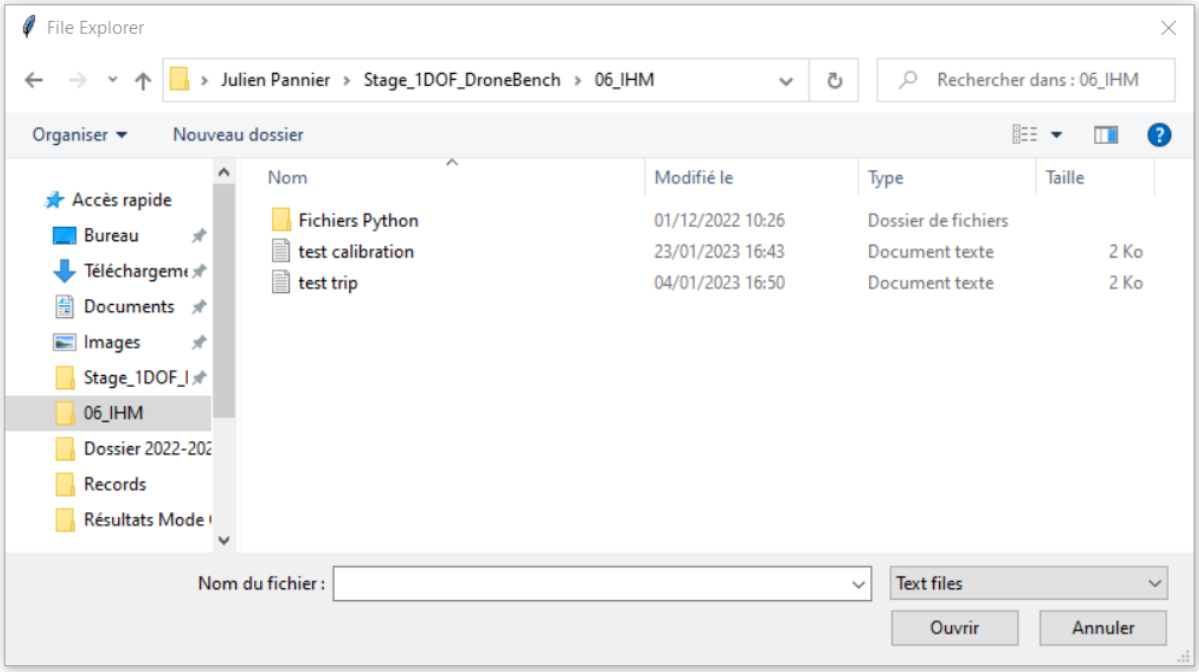


Figure 16: Boîte de dialogue "File Explorer" permettant de sélectionner un fichier

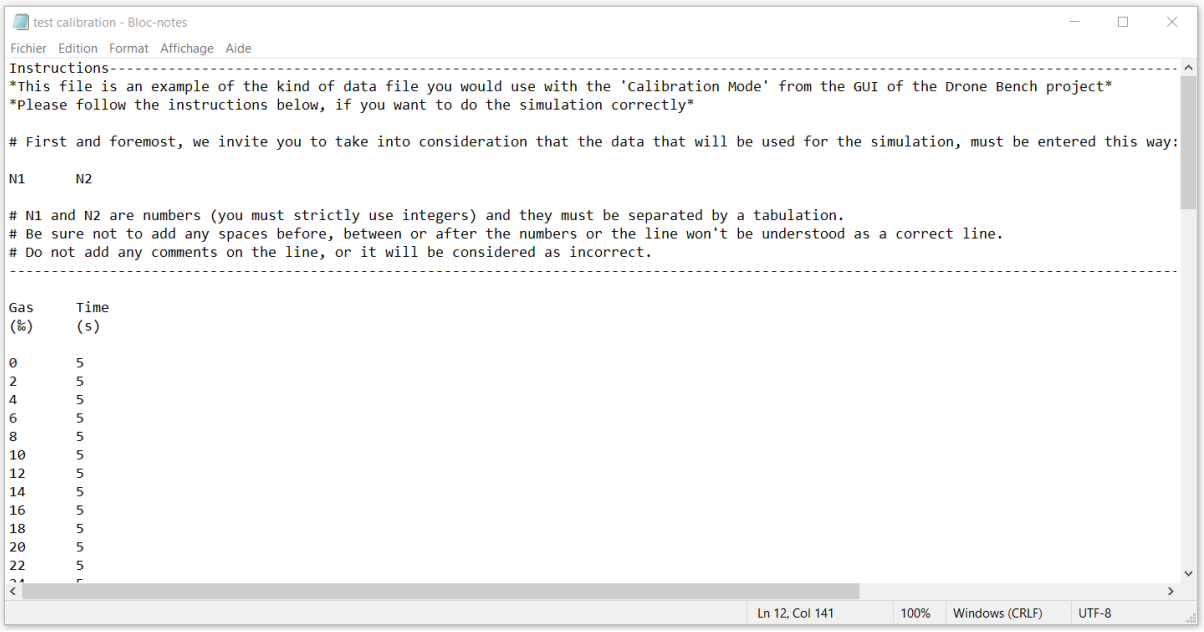


Figure 17: Fichier "test calibration" rempli

Le fichier dit « *test calibration* » a évolué au cours du stage de manière à atteindre le résultat illustré sur la Figure 17. Il a tout d'abord comporté uniquement un certain nombre de lignes composées de 2 nombres entiers espacés l'un de l'autre d'une tabulation et a ensuite disposé de commentaires instructifs.

Ces 2 nombres représentent, pour le premier, une valeur consigne de pourcentage de gaz, qui par la suite a évolué en pour mille (‰) afin d'obtenir une plus grande précision car la valeur de 17% de gaz correspondait déjà à notre valeur d'angle maximale (avant basculement du bras au-delà des 90°). Passer en ‰ nous permet donc d'avoir des courbes beaucoup plus détaillées. Pour le second nombre, il s'agit du temps (en secondes), pendant lequel on veut faire l'acquisition de données.

Il m'a donc fallu trouver un moyen pour différencier ce qui s'apparenterait aux valeurs consignes de la simulation et ce qui porterait plus sur l'aspect décoratif du fichier, soit les commentaires et instructions.

Poursuivre la lecture du fichier, malgré la présence d'erreurs ou d'oublis au niveau des consignes, était inéluctable. J'ai donc utilisé la fonction « *compile* » du module « *re* » (« *regular expression operations* ») qui m'a alors permis de solutionner ces différents problèmes, via la création d'un motif (« *pattern* ») composé de 2 nombres entiers séparés d'une tabulation. Néanmoins, ce motif, du fait de son caractère strict, impose une certaine vigilance lors de l'entrée des données : un espace de trop engendrerait la non prise en compte d'une ligne de consignes par exemple (d'où la nécessité d'ajouter des instructions). A noter que les lignes vides ne viennent pas perturber la simulation.

Une fois le fichier sélectionné, la fenêtre de dialogue se ferme et s'il s'agit bel et bien du bon fichier non vide (c'est à dire comportant des lignes de consignes valides), la simulation pourra alors débuter. Lorsque cette dernière se termine, une nouvelle boîte de dialogue apparaît et propose cette fois-ci de sélectionner un emplacement de sauvegarde des résultats sous format .csv. Un titre de fichier horodaté est également proposé par défaut.

Après avoir enregistré ou non les données issues de la simulation, le bouton « *Show Graph* » (Figure 15) est actif et donne la possibilité à l'utilisateur d'ouvrir une nouvelle fenêtre mettant à disposition un graphique issu des résultats avec une barre d'outils permettant de jouer sur l'affichage, mais aussi de sauvegarder l'image sous plusieurs formats (Figure 18 par exemple). Il est possible d'afficher ce graphique, autant de fois que l'utilisateur le désire.

Le bouton est désactivé une fois que l'utilisateur a décidé de relancer une simulation (via « *Select File* ») ou bien tout simplement en quittant le mode d'une manière ou d'une autre.

Le principe et but du mode « *Calibration* » est d'obtenir les mesures issues du capteur MPU6050 sous un mode non asservi, soit via « *Manual Mode Term* », afin de pouvoir en tirer des études sur l'évolution mécanique du bras en fonction du moteur. Ces études peuvent notamment faire office de sujets de travaux pratiques propres au dispositif.

Dans le cadre d'une future utilisation du dispositif, il nous a donc été demandé (par nos « clients » / professeurs des matières concernées par l'utilisation du dispositif) de pouvoir récupérer les données brutes issues du gyromètre et de l'accéléromètre de la centrale inertielle

(sur 3 axes), ainsi que les positions angulaires calculées sur 2 axes (X sur laquelle la rotation du bras s'effectue et Y).

A noter que les graphiques issus de « *Show graph* » portent uniquement sur la position angulaire selon X :

- Soit en fonction du temps pour un échelon donné (on donnera alors une seule ligne de consigne d'entrée, ou bien 2 lignes avec un court temps d'initialisation sur 0% si l'on désire éviter des premiers points qui peuvent parfois être complètement faussés « *glitch* ») ;

- Soit en fonction de la consigne en pour mille de gaz moteur.

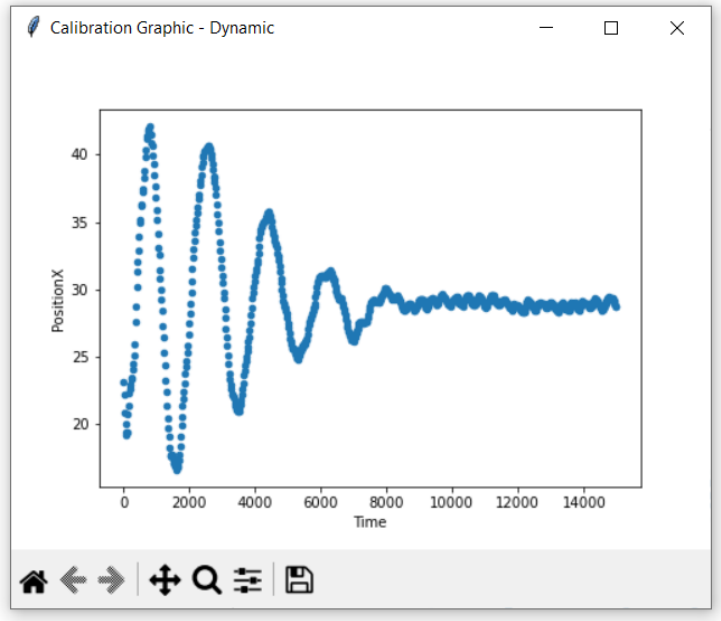


Figure 18: Exemple de graphique issu de "Show Graph" du mode "Calibration" - étude dynamique

La première étude est dite dynamique (Figure 18) et la seconde est dite statique (Figure 19).

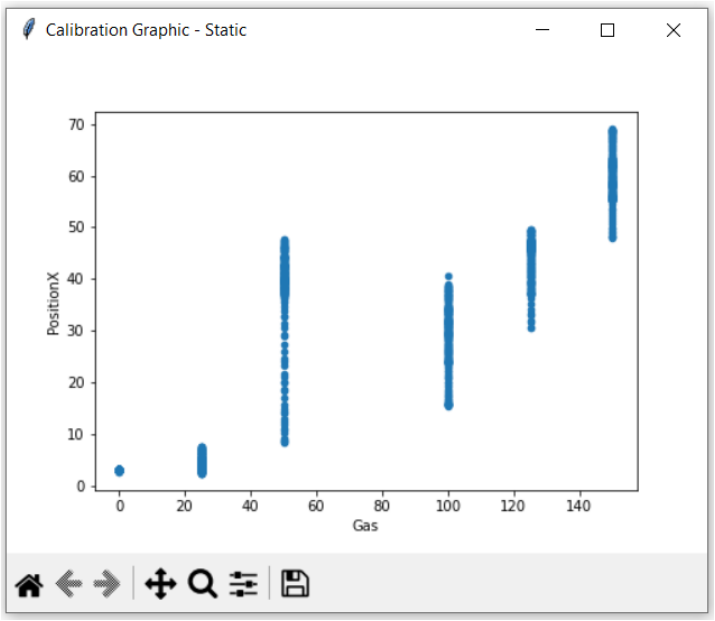


Figure 19: Exemple de graphique issu de "Show Graph" du mode "Calibration" - étude statique

Pour revenir en détail sur l'obtention des mesures, il m'a été recommandé en premier temps, d'affiner le pas de la consigne de gaz moteur. Comme évoqué précédemment, le pas était de 1% et sachant que 17% de gaz suffisait à atteindre un seuil « critique » pour la position du bras, nous ne pouvions alors pas obtenir des courbes très détaillées. C'est ainsi que l'on m'a demandé de réduire le pas à 1‰ soit 0,1%. Cela s'est donc fait en modifiant la fonction « *mapping_adc_value_percent* » (TAG_UC_007) définie dans le fichier « *yann.c* ».

2.5 Utilisation des « *Timers Interrupts* »

Il m'a été recommandé de m'informer au sujet de la méthode de communication à employer pour pouvoir transmettre les données mesurées, car d'une part le « *polling* » m'imposait de renvoyer constamment un caractère pour pouvoir accéder aux fonctions de lecture du capteur et d'autre part il n'y a pas de déterminisme temporel que l'on va pouvoir retrouver dans une méthode par interruption (par exemple).

Je me suis alors orienté vers un mode par interruption, via les « *Timers interrupts* » de la STM32. Il faut d'abord sélectionner une période de tic (« *counter period* ») qui nous permet de connaître le nombre de fois que sera exécuté une routine définie dans la fonction « *Callback* » : « *HAL_TIM_PeriodElapsedCallback* » (TAG_UC_004). On est donc, dès à présent, sûr de l'instant temporel associé à la lecture des données.

En connaissant le nombre de tics par seconde selon la valeur de « *counter period* » choisie, on a alors le nombre de mesures effectuées et par conséquent le nombre de points par lignes de consignes (fichier .txt) sur nos graphiques. Il s'agit du plus gros avantage de cette méthode de communication, car nous pouvons être sûr, cette fois-ci, que nous avons une lecture périodiquement stable.

J'ai opté pour une « *counter period* » de 200 qui est équivalente à 20ms. On a donc 50 points par seconde de prise de mesures (soit 250 points par ligne de consigne sur la Figure 17 dont les temps de mesures sont de 5s et donc 250 points par abscisse sur un graphique). Ce nombre de points très élevé est amplement suffisant pour notre étude.

Il m'a ensuite été demandé de m'informer sur la communication via le DMA, mais qui après quelques recherches et tests, m'a paru ne pas être nécessaire pour cette mission.

2.6 Mode asservi avec prise de mesures : « *Séquence de vol* »

Le mode « *Séquence de vol* » est défini selon la classe « *TripModeGUI* » (TAG_IHM_007) du fichier « *GUI Master* » et reprend cette fois-ci le même principe que l'extension de fenêtre du mode « *Calibration* », bien qu'il n'y ait pas de bouton « *Show Graph* » (Figure 20).

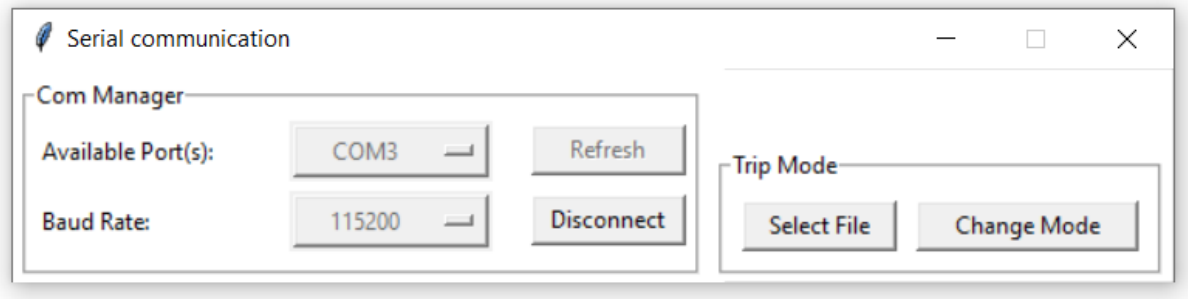


Figure 20: Extension de fenêtre propre au mode « Trip Mode »

Le principe est assez similaire, l'utilisateur est invité à sélectionner un fichier (cette fois-ci le fichier « *test trip* » de la Figure 16), puis la simulation s'effectue selon le mode « *Auto Mode* ». Le fichier texte « *test trip* » reprend exactement la même disposition que « *test calibration* », à ceci près que la première colonne de nombres entiers n'est plus en consigne de gaz moteur, mais en consigne de position angulaire.

Le principe et but du mode « *Test Trip* » est aussi d'obtenir les mesures issues du capteur MPU6050, cette fois-ci sous un mode asservi, afin de pouvoir en tirer des études sur l'asservissement du bras en fonction du moteur. Les données récupérées comportent en plus les termes d'erreurs pris en compte dans le calcul de l'asservissement (Figure 21 et Figure 22).

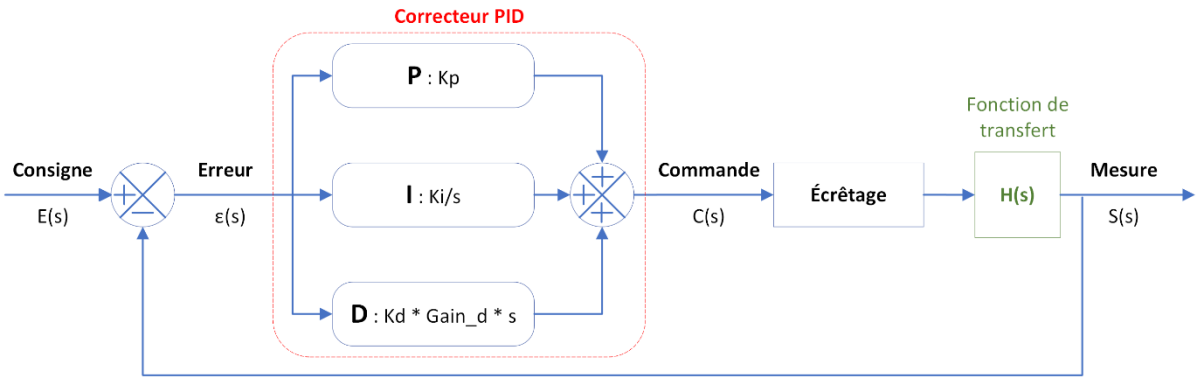


Figure 21: Schéma bloc de l'asservissement

```
// Asservissement

// Calcul des erreurs
_erreur = erreur;
erreur = consigne - position_angulaireX;
integre_erreur += erreur;
derive_erreur = erreur - _erreur;

// Calcul de la commande
commande = kp * (erreur) + ki * (integre_erreur)
+ gain_d*kd * (derive_erreur);

// Conditions d'écrêtage
if (commande > valeur_max_moteur) {
commande = valeur_max_moteur;
}
if (commande < valeur_min_moteur) {
commande = valeur_min_moteur;
}

// Transmission de la commande
load_pwm(htim3, commande);
```

Figure 22: Routine du calcul de la commande dans la boucle d'asservissement

Les variables « *erreur* », « *integre_erreur* », ainsi que « *derive_erreur* » sont associées, respectivement, aux coefficients « proportionnel intégral dérivé » (« gain_d » est une constante qui vient s'appliquer en facteur de « kd » et qui est apparu à la suite des tests PID).

Les valeurs successives de « erreur », « integre_erreur » et « derive_erreur » ont été également rajouté dans la transmission de données, sous les noms Erreur P/I/D (respectivement).

Une fois que le calcul de la commande est effectué, il y a 2 conditions d'écrêtage qui s'appliquent sur la valeur « commande », si la valeur est supérieure à une valeur maximale relative au moteur, alors la variable « commande » prendra la valeur maximale. Si « commande » est inférieure à la valeur minimale relative au moteur, alors « commande » prendra la valeur minimale (Figure 22).

La transmission s'effectue toutes les 20 ms (période de tic du « *Timers Interrupt* ») via la fonction « *load_pwm* » définie dans le fichier « yann.c ».

Après avoir effectué des premiers tests concluants sur les modes « *Calibration* » et « *Trip Mode* », j'ai poursuivi le développement de ce dernier, car nous avions eu alors l'envie de trouver d'autres configurations de coefficients PID, dans un but de meilleures performances.

A noter que cette étude de l'asservissement et des coefficients PID est purement empirique. Nous n'avons pas de modèle théorique et donc de fonction de transfert (Figure 21). Cette étude théorique pourra faire l'objet d'un sujet de TP par exemple.

Il m'est alors paru essentiel de rajouter des *widgets* à l'extension « *TripModeGUI* », afin de pouvoir renseigner des nouveaux coefficients PID rapidement et consécutivement (Figure 23), sans avoir à intervenir directement dans le « *main.c* » du *firmware* (on revient sur la même problématique de facilitation du renseignement des paramètres, propre à ce qui a motivé le développement d'une IHM pour ce projet).

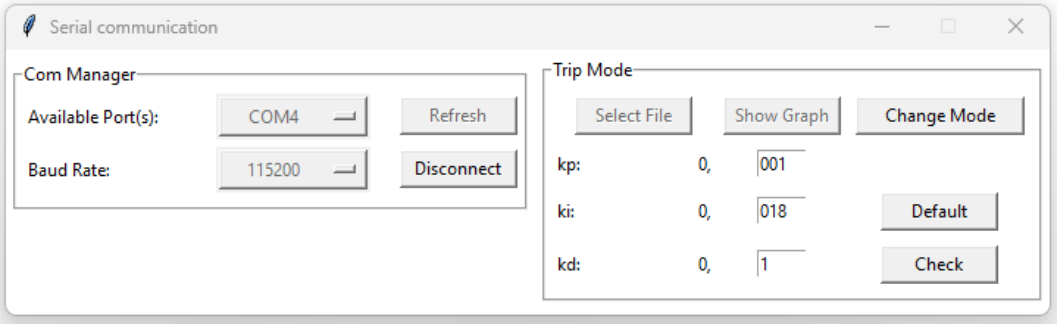


Figure 23: Extension de fenêtre propre au mode « Trip Mode », version avec widgets PID

Ensuite, après avoir effectué des premiers essais, je me suis vite rendu compte que les graphiques résultants du bouton « *Show Graph* » étaient très limités en termes d'exploitation… En effet, il n'était alors possible que d'afficher les valeurs de « Position Angulaire X » et il me fallait donc trouver un moyen de pouvoir afficher toutes les autres données demandées, avec la possibilité de les représenter sur un même graphique (avoir plusieurs courbes sur ce graphique et pouvoir sélectionner celles qui nous intéressent). En somme, comme effectué manuellement sur Excel, via le fichier csv (Figure 24 et Figure 25). Ce nouvel ajout présenterait alors un gain de temps formidable pour les tests empiriques successifs.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Consigne Angle (°)	Time (ms)	Position Angulaire X (°)	Position Angulaire Y (°)	Ax Raw (m/s²)	Ay Raw (m/s²)	Az Raw (m/s²)	Gx Raw (rad/s)	Gy Raw (rad/s)	Gz Raw (rad/s)	Erreur P (°)	Erreur I (°)	Erreur D (°)
2	60	0	2,25	-123,01	0,328	-9,9431	-0,2421	0,0448	0,0011	0,0105	57,75	57,75	57,75
3	60	20	2,52	-128,24	0,328	-9,9838	-0,3945	0,0441	0,0017	0,0093	57,48	172,75	-0,04
4	60	40	2,51	-128,32	0,3543	-9,9527	-0,3183	0,0442	-0,0023	0,0096	57,49	230,25	0,01
5	60	60	2,51	-128,97	0,3519	-10,0149	-0,351	0,0436	-0,002	0,01	57,49	287,74	0
6	60	80	2,5	-129,74	0,3376	-9,9144	-0,3591	0,0433	0,0019	0,0085	57,5	345,24	0,01
7	60	100	2,48	-130,34	0,3352	-10,0054	-0,3591	0,0402	-0,0033	0,0105	57,52	402,76	0,02
8	60	120	2,47	-131,16	0,3232	-9,9982	-0,3809	0,046	-0,0028	0,0097	57,53	460,29	0,01
9	60	140	2,45	-130,58	0,3639	-9,9335	-0,2966	0,045	0,0023	0,01	57,55	517,84	0,02
10	60	160	2,38	-129,75	0,3256	-9,9335	-0,234	0,0444	0,0001	0,0067	57,62	575,46	0,06
11	60	180	2,34	-130,83	0,2634	-9,9647	-0,3537	0,0449	-0,0021	0,0103	57,66	633,12	0,04
12	60	200	2,38	-131,19	0,3974	-9,9814	-0,4326	0,0434	-0,0004	0,0079	57,62	690,73	-0,04
13	60	220	2,37	-131,35	0,3424	-9,9886	-0,351	0,0441	-0,0025	0,0079	57,63	748,37	0,01
14	60	240	2,37	-131,53	0,3639	-9,9527	-0,3782	0,0449	-0,0001	0,0099	57,63	805,99	-0,01
15	60	260	2,38	-131,78	0,3615	-10,0269	-0,3891	0,0436	-0,0003	0,0095	57,62	863,61	0
16	60	280	2,38	-131,7	0,3687	-9,8689	-0,3482	0,0466	0,0011	0,0085	57,62	921,23	0
17	60	300	2,37	-131,77	0,3543	-9,9168	-0,3564	0,0424	0,0011	0,0104	57,63	978,86	0,01
18	60	320	2,34	-131,5	0,3304	-9,9431	-0,2857	0,045	-0,0021	0,0099	57,66	1036,52	0,03
19	60	340	2,33	-131,04	0,3807	-9,9671	-0,2993	0,043	-0,0001	0,0112	57,67	1094,18	0,01
20	60	360	2,35	-130,74	0,4046	-9,9718	-0,3401	0,0458	-0,0013	0,0105	57,65	1151,84	-0,01
21	60	380	2,33	-130,36	0,3567	-9,9766	-0,2857	0,0429	-0,0007	0,0105	57,67	1209,51	0,02
22	60	400	2,29	-129,57	0,34	-9,9814	-0,2204	0,0404	0,0005	0,0088	57,71	1267,22	0,04
23	60	420	2,3	-130,14	0,3352	-9,991	-0,4081	0,0461	0,0004	0,0103	57,7	1324,92	-0,01
24	60	440	2,3	-130,01	0,3639	-9,9264	-0,3238	0,043	-0,0017	0,012	57,7	1382,62	0
25	60	460	2,29	-129,82	0,3448	-9,9479	-0,2993	0,0446	0,0012	0,0075	57,71	1440,33	0,01
26	60	480	2,3	-129,95	0,3687	-10,0293	-0,3673	0,046	-0,0041	0,01	57,7	1498,03	-0,01
27	60	500	2,31	-129,86	0,3783	-9,9383	-0,3428	0,0436	-0,004	0,0085	57,69	1555,72	-0,01
28	60	520	2,34	-129,75	0,4118	-9,9766	-0,37	0,0401	0,0011	0,0092	57,66	1613,38	-0,03
29	60	540	2,32	-130,16	0,2945	-9,9958	-0,3346	0,0441	0,0021	0,0088	57,68	1671,06	0,02
30	60	560	2,31	-130,25	0,316	-9,9407	-0,3129	0,0461	-0,0017	0,0108	57,69	1728,75	0,01
31	60	580	2,3	-129,6	0,3759	-9,9742	-0,2612	0,0437	0,0008	0,0115	57,7	1786,45	0,01
32	60	600	2,31	-129,5	0,3615	-9,9934	-0,3238	0,0446	0,0016	0,0091	57,69	1844,14	0
33	60	620	2,35	-130,25	0,3496	-9,9718	-0,4625	0,0454	-0,0032	0,008	57,65	1901,79	-0,04
34	60	640	2,39	-130,55	0,395	-9,9695	-0,4326	0,0429	-0,002	0,0091	57,61	1959,41	-0,04
35	60	660	2,39	-131,42	0,2825	-9,9814	-0,4054	0,0434	0,002	0,0087	57,61	2017,02	0

2023-02-16-13h36-Trip_mode_Resu

Figure 24: Fichier csv résultant de "Trip Mode" ouvert sur Excel

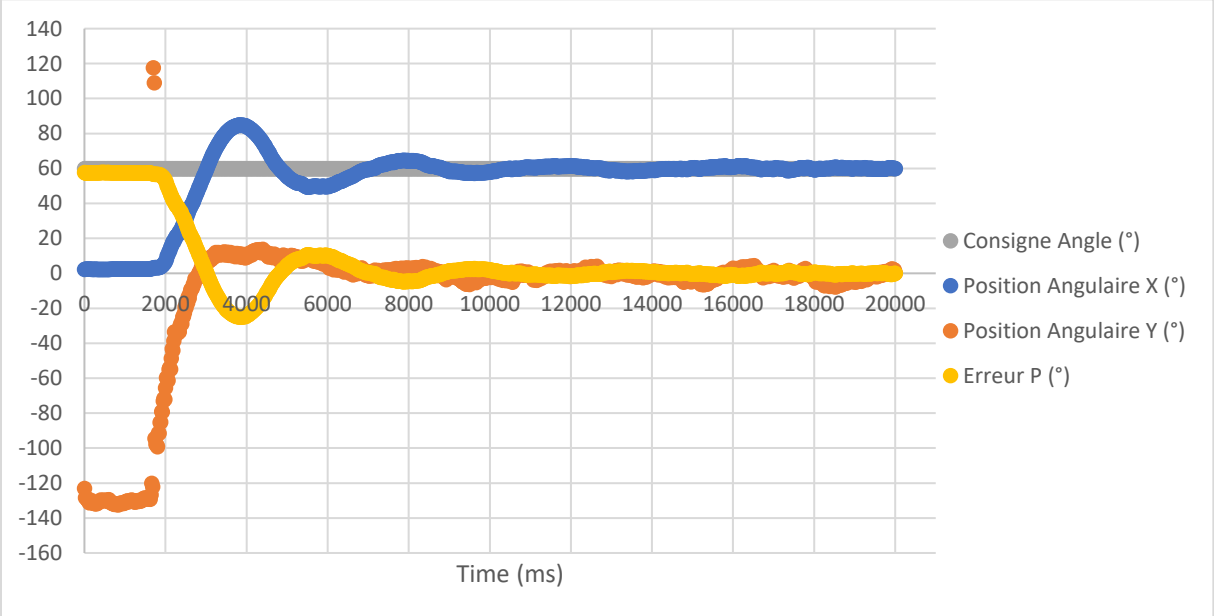


Figure 25: Graphique Excel issu d'un fichier csv de Trip Mode

J'ai donc, en premier temps, cherché à adapter le graphique sous *matplotlib*, en ajoutant des « *checkboxes* » permettant de faire apparaître ou disparaitre une ou plusieurs courbes. J'ai rencontré à ce moment-là quelques difficultés relatives aux « *IntVar()* » de Python et aux multiples fenêtres de l'IHM, qui ont par la suite été résolues. Parallèlement à la résolution de ce contre-temps, j'ai fait une tentative de création de graphique via le module « *plotly* » qui est adapté à l'analyse de données. L'idée était telle qu'il serait alors possible de créer des graphiques en html, qui s'afficheraient sur mon moteur de recherche (en l'occurrence Firefox) en offline.

Plotly prenant déjà en compte la possibilité d'activer ou de désactiver certaines courbes via la légende des courbes, ce module répondait donc tout à fait à mes attentes. De plus, l'ergonomie et les outils plus développés que ceux de *matplolib*, m'ont fait préférer l'utilisation de *plotly*

plutôt que ce dernier (Figure 26 et Figure 27). Par la suite, les graphiques sous *plotly* ont été déployé également sur le mode « *Calibration* ».

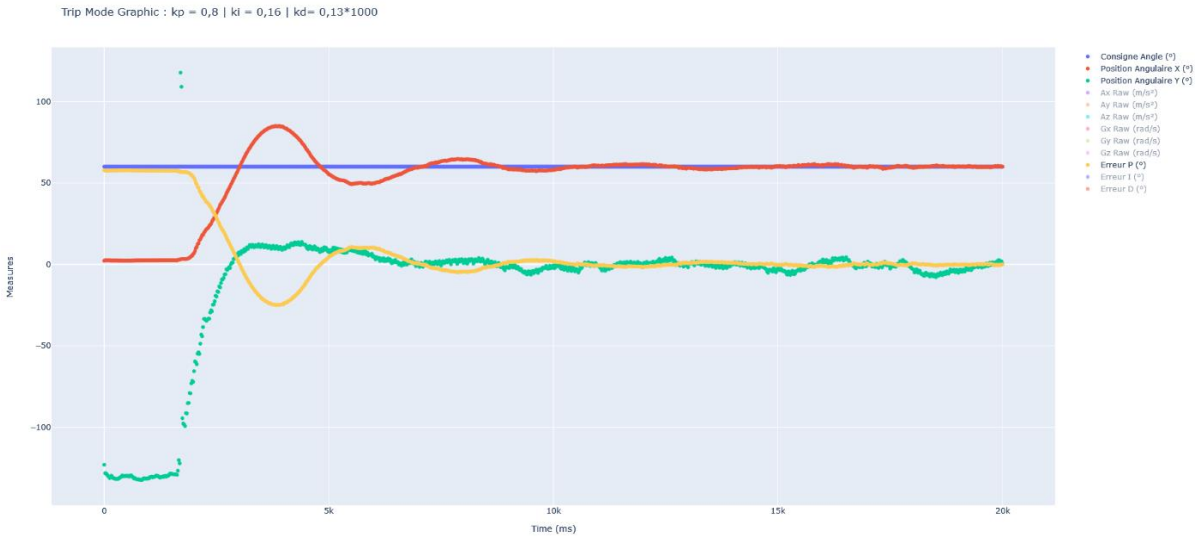


Figure 26: Graphique Plotly issu d'un fichier csv de Trip Mode (vue d'ensemble)

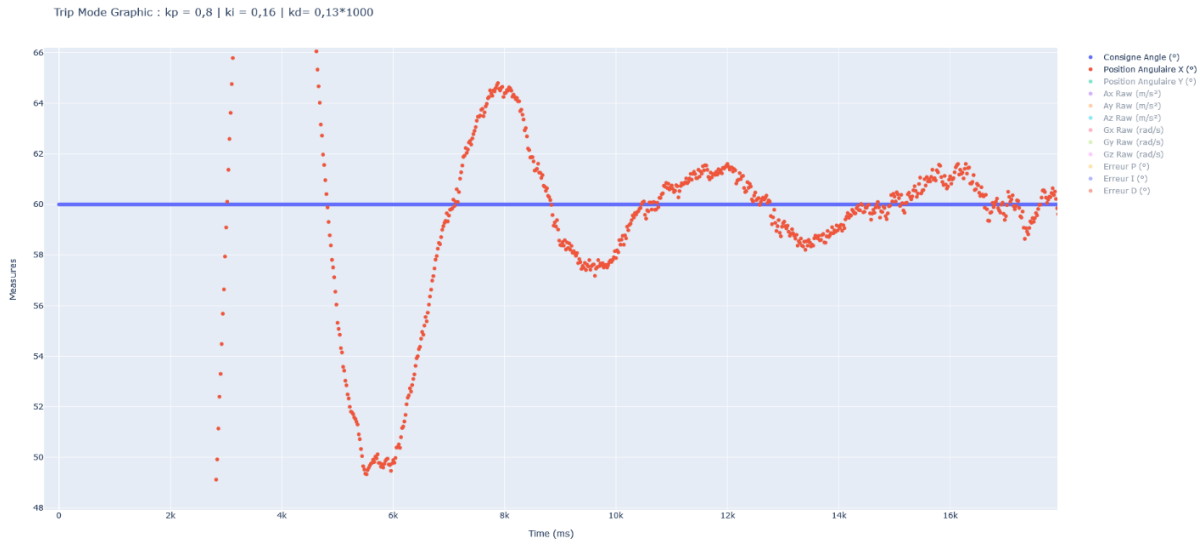


Figure 27: Graphique Plotly issu d'un fichier csv de Trip Mode (vue zoomée sur Position Angulaire X)

Une fois armé de ces nouveaux outils, il a tout de suite été plus simple d'effectuer des tests de coefficients PID et de répondre à certaines attentes plus efficacement, notamment de fournir des données et graphiques aux « clients » de ce projet.

Nous avions alors vu ensemble, lors d'une réunion, ce qui les intéresserait d'exploiter : des simulations en mode non asservi (via le mode « *Calibration* ») au repos par exemple, ou bien une simulation en déplaçant le bras manuellement, sans gaz, pendant quelques dizaines de secondes (Figure 28, Figure 29 et Figure 30). Ces premières données leurs serviraient alors à enclencher une certaine démarche d'inclusion du dispositif dans leurs cours.

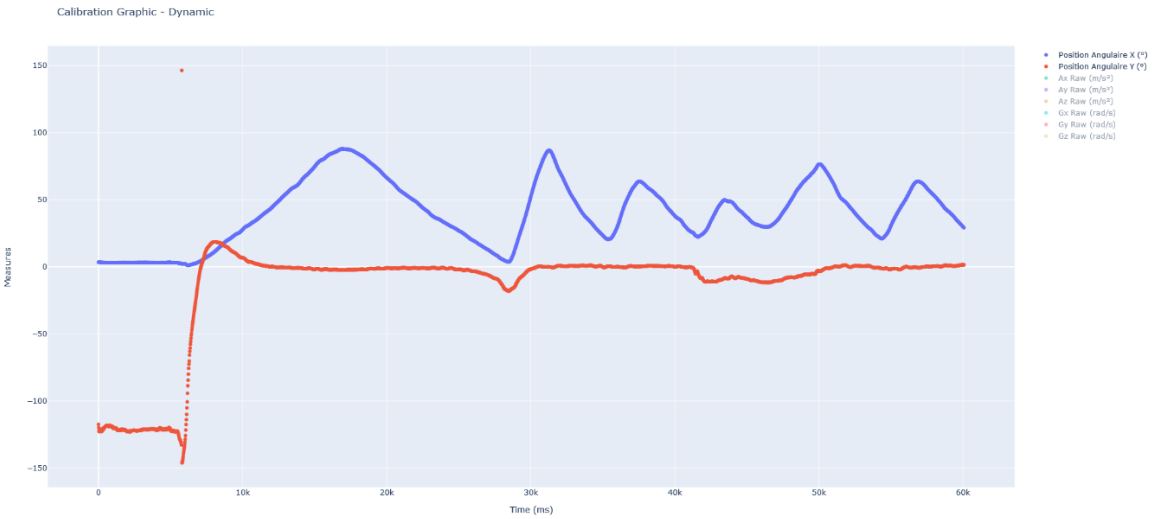


Figure 28: Simulation du bras manipulé manuellement sans gaz (Positions Angulaires X et Y)

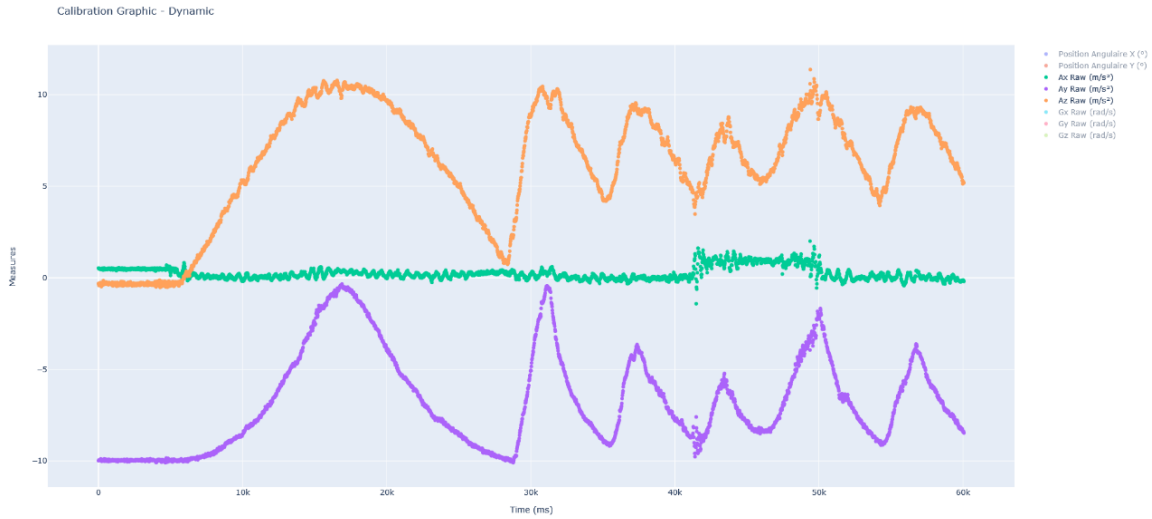


Figure 29: Simulation du bras manipulé manuellement sans gaz (Ax,y,z)

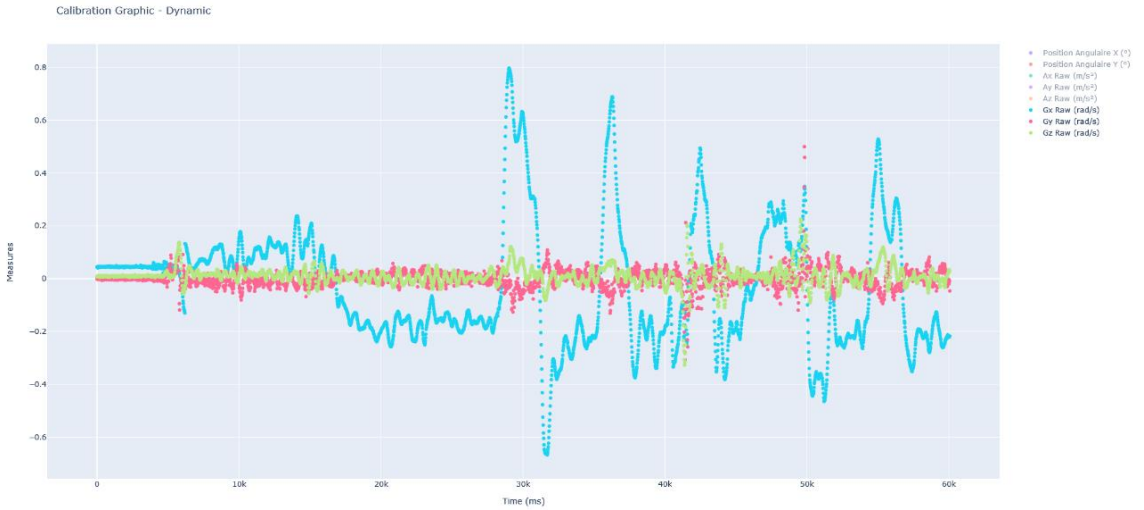


Figure 30: Simulation du bras manipulé manuellement sans gaz (Gx,y,z)

Dans cette optique de continuer à perfectionner le mode asservi, nous avions alors perçu la nécessité de déplacer les commandes d'asservissement, initialement présentes dans la boucle de l'état « *Auto Mode* » (TAG_UC_013), vers la boucle propre au « *Timer interrupt* » évoqué précédemment (TAG_UC_004).

Déjà, car jusqu'à présent la lecture du capteur était double (une première fois dans la boucle d'« *Auto Mode* » et une seconde fois dans celle du « *Timer Interrupt* ») et par soucis de rigueur, bien que le système fonctionne correctement, il fut nécessaire de corriger ce problème.

J'ai donc eu à modifier, une énième fois, la machine d'état du « *main.c* » en concordance avec les fichiers Python de l'IHM, et j'ai réussi à appliquer ce changement sous une condition de passage lors du choix du mode sélectionné dans le menu des modes. Entre autres, lorsque l'on fait le choix d'intervenir en « *Auto Mode* » classique, il est nécessaire de ne pas recevoir (du côté de l'IHM) les données mesurées par *printf*. Alors qu'en choisissant « *Trip Mode* » il est indispensable de recevoir les *printf*.

A noter qu'il a été également développé, au sein de la boucle du « *Timer Interrupt* » une condition telle qu'à partir d'un certain nombre de données de position angulaire mesurées supérieures ou égales à 85°, une procédure d'atterrissage forcé serait alors exécutée dans un but d'éviter cette situation classique où le bras bascule et se bloque au-delà de 90°.

Ainsi s'achève le développement de l'IHM durant ce stage : la Figure 5 présente la nouvelle machine d'état qui vient supplanter la précédente (Figure 4). Bien que certains aspects tels que « *Manual Mode Pot* » ou le mode « *Info* » ne soient plus d'actualité et ne sont pas accessibles via l'IHM, le système élémentaire de cette machine d'état est de proposer un mode, faire une simulation, atterrir sereinement et recommencer le processus.

3 Hardware

L'amélioration du point de vue matérielle du projet est intervenue assez tardivement, car il ne s'agissait pas d'une priorité vis-à-vis des objectifs de mon sujet de stage. Cependant, aux vues de mon avancement sur les objectifs principaux, il a alors été envisagé que je puisse débuter, en parallèle de mes dernières missions explicitées précédemment, une formation sur le logiciel de CAO (Conception assistée par ordinateur) SolidWorks.

3.1 Conception et fabrication d'un rapporteur

Cette formation s'est déroulée sous plusieurs semaines (en parallèle de mes autres missions) durant lesquelles j'ai pu suivre des cours avec mon tuteur, mais aussi avec l'I'Lab Manager, ou bien encore en autonomie avec des tutoriels.

Si l'idée derrière cette formation fut que je puisse à la fois varier mes activités, mais aussi d'intervenir sur l'ajout de composants, voire l'amélioration de composants déjà existants, elle m'est parue tout à fait opportune dans le cadre d'un stage au sein d'un environnement comme celui de l'I'Lab dont la CAO fait partie intégrante du processus de fabrication d'objets via les imprimantes 3D et découpeuse laser.

De plus, l'intervention de cette partie CAO pour l'*hardware* prend tout son sens dans ce stage, non seulement car je viens « ajouter ma patte » à l'objet (cela me permet de m'investir sur le travail de mon prédécesseur, pour qui l'*hardware* fut le sujet principal de son stage), mais aussi car tout au long du stage, des idées d'améliorations matérielles se sont profilées au cours du développement de la partie *software*.

A la suite de cette formation, mon premier travail de conception fut de réaliser un outil de mesure d'angle : un rapporteur peu obstruant d'une forme de quart de cercle (prenant en compte les angles de 0 à 90°) avec lequel il serait possible de vérifier approximativement si la valeur consigne est respectée et atteinte et s'il y a concordance avec les données de la centrale inertielle.

J'ai donc suivi une idée de modélisation provenant de mon tuteur, sur la forme que prendrait l'objet, ainsi que la prise en compte de rainures permettant son insertion et sa fixation au profilé d'aluminium faisant office de tronc. Ensuite, je me suis mis à le modéliser sur SolidWorks en mettant à profit ma formation.

Une fois le modèle finalisé dans le fichier formaté .SLDPRT (*SolidWorks Part Document*), il m'a été demandé de convertir ce fichier (Figure 31) au format DXF (*Drawing eXchange Format*) afin de pouvoir amorcer l'étape de fabrication via la découpeuse laser (Figure 32).

Nous avons alors pu découper le rapporteur dans du plexiglas et la découpeuse laser a également pris en compte de légères découpes (peu profondes) pour l'ajout des tracés et écriture des annotations (Figure 33).

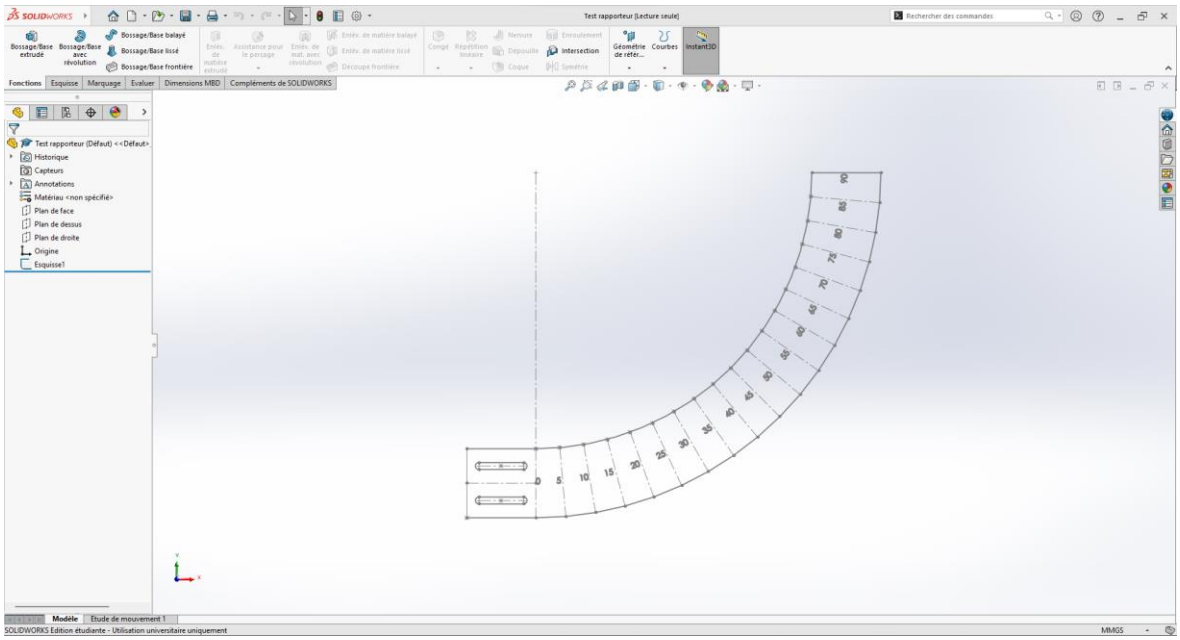


Figure 31: Rapporteur modélisé sur SolidWorks

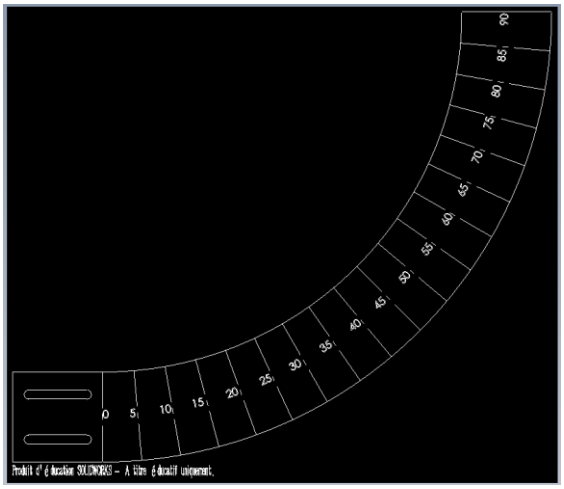


Figure 32: Rapporteur au format DXF

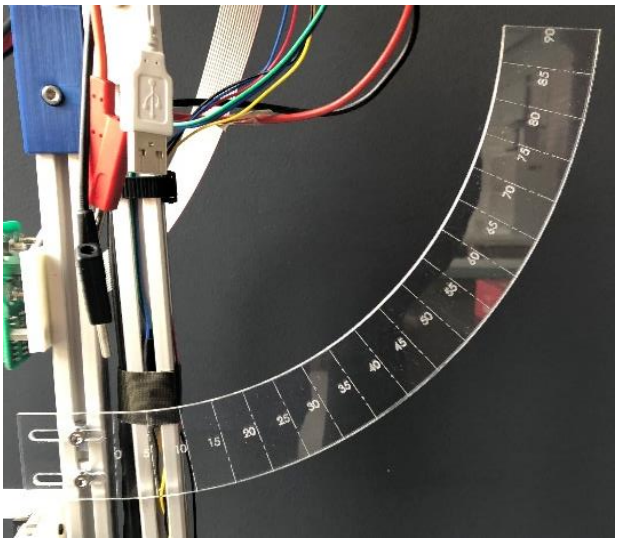


Figure 33: Rapporteur en plexiglas fixé au bras

3.2 Conception d’un nouveau carter

Ma seconde et ultime réalisation sur SolidWorks a été de concevoir un nouveau carter de protection, à la suite de l’enlèvement de l’ancienne version lors des tests effectués sur les coefficients PID, nous avions alors préféré le démonter car ce dernier augmentait conséquemment le bruit provoqué par le drone en fonctionnement, en plus d’alourdir le bras.

J’ai alors reçu la mission de concevoir une nouvelle version du carter, basée sur la version actuelle, mais en réduisant de moitié son épaisseur, en augmentant son diamètre de quelques centimètres et en prenant en compte cette fois-ci d’un espace suffisant au niveau de la partie qui vient s’insérer dans la fente du profilé en aluminium (ceci permettant le passage des fils reliant l’ESC au moteur). Lors du démontage du carter, il nous est alors paru évident qu’il fallait éviter d’avoir à débrancher puis rebrancher ces connexions à chaque changement de matériel.

J’ai donc repris le fichier .SLDPRT que j’ai alors pu modifier en éditant les différentes esquisses et fonctions réalisées. Le nouveau modèle est présenté sur la Figure 34.

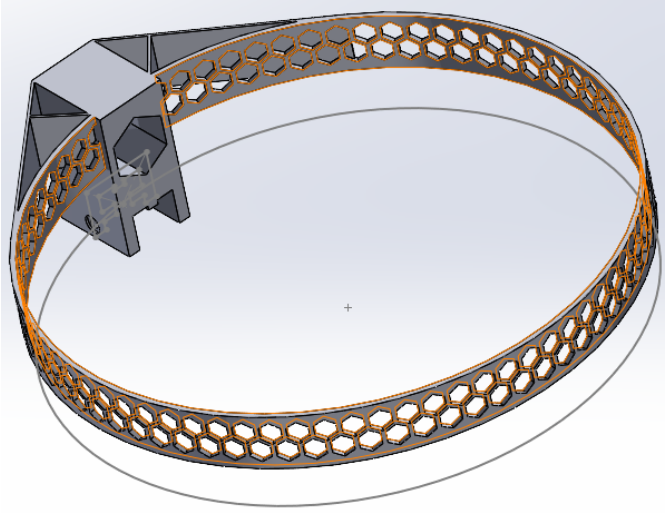


Figure 34: Modélisation du nouveau carter

Conclusion

- ## Fin du stage

Pour cette fin de stage, j'ai dû orienter mes dernières semaines sur la finalisation de la rédaction de ce rapport, dans l'optique de réaliser ma présentation sur la toute dernière semaine de stage.

Ainsi, à l'heure où j'écris ces lignes de conclusion, j'entrevois encore de préparer mes diapositives pour la présentation prévue, mais également de finaliser la conception et l'impression du nouveau carter de protection.

Je vais également passer un certain temps sur une dernière relecture du code effectué durant ce stage, afin de pouvoir m'assurer de la clarté de ce dernier, mais aussi afin de m'assurer de la stabilité du projet (je souhaite ajouter quelques garde-fous et corriger les quelques bugs qui subsisteraient).

Dans cet état d'esprit, je prévois également de remettre au propre le *README* du GitHub, ainsi que la disposition des dossiers présents sur ce dernier. J'aimerais laisser le GitHub dans un état un peu plus épuré, afin de pouvoir s'y retrouver un peu plus facilement pour l'utilisateur commun, comme pour les « clients » de ce projet ou pour l'hypothétique prochain étudiant qui interviendrait sur le bras drone.

- ## Idées de poursuite du développement du projet

Afin de rebondir sur le point évoqué dans le paragraphe précédent, j'aimerais étayer l'idée d'une potentielle poursuite du développement du projet bras drone qui me semble encore avoir beaucoup à offrir sur cette thématique.

Au cours de mon stage, j'ai eu, à de nombreuses occasions, l'opportunité de présenter mon projet et j'ai eu une multitude de retours positifs qui ont été, pour certains, agrémentés de propositions plus ou moins techniques, qui me font donc penser que d'une part ce projet a un intérêt concret et a encore de nombreux « jours heureux » devant lui. Le fait est que ce projet semble attiser naturellement la curiosité des étudiants, mais aussi des professeurs ou autres intervenants.

Par exemple, j'ai récemment été convié à présenter ce projet au sein de l'I'Lab, à un public assez fourni et hétéroclite, et j'ai pu échanger notamment avec la personne en charge d'avoir une vue globale sur les projets de l'Efrei Paris. Nous avions discuté des potentielles améliorations et ajouts à apporter au bras, ainsi que des possibles manières d'accentuer sa transversalité et sa démontabilité. Cet entretien se voit comme une concrétisation et un moment de globalisation de ces nombreuses propositions d'apports.

Ainsi, l'idée de diminuer la taille du dispositif afin de réduire d'une part les coûts de production, mais aussi d'autre part de permettre son utilisation en cours avec plusieurs exemplaires (soit un bras pour un binôme par exemple, et non plus pour un groupe plus fourni d'élèves), a été proposé. Sans oublier d'évoquer le fait qu'un plus petit bras serait moins encombrant, plus facilement transportable (et rangeable) et verrait sa « dangerosité » amoindrie dans l'éventualité d'un accident.

Je souhaite m'attarder sur ce point non négligeable, qui m'est paru être d'une importance capitale dès le début de mon stage. En effet, le bras tel qu'il est présenté aujourd'hui dispose de peu d'éléments de protection, autant pour l'utilisateur que pour sa propre « carcasse ». Bien que j'aie pu implémenter des garde-fous au niveau du code afin de diminuer les possibilités d'accidents dû à l'interaction et via la lecture de mesures préoccupantes, cela n'empêche pas que d'autres accidents purement mécanique ou électronique puissent arriver.

Or il me paraîtrait intéressant de venir consolider le bras en le stabilisant à son socle, mais aussi au niveau de la liaison pivot. Nous avions pensé qu'il aurait peut-être été envisageable de modifier complètement notre partie mécanique utilisée actuellement pour cette liaison pivot, afin de limiter ce souci de basculement au-delà de 90° ainsi que le jeu présent entre les 2 profilés d'aluminium créant des tremblements peu souhaitables.

L'ajout de poids au niveau du socle permettrait également d'éviter que le bras ne s'envole dans le cas où le moteur s'activerait brusquement et entraînerait une trop forte poussée en arrière (nous avons déjà vécu cette malencontreuse expérience…).

Si la fonction de détection d'un dépassement de la valeur de 85° a déjà une utilité certaine, le concept d'un bouton d'arrêt d'urgence a été introduit de nombreuses fois durant ce stage. L'idée d'une butée mécanique empêchant le bras d'aller au-delà d'une certaine position angulaire fut aussi le sujet d'une requête de l'un des professeurs intervenant sur ce projet.

Cette liste d'idées n'est donc pas limitative et souligne donc tout le potentiel qu'a ce dispositif et par extension le potentiel qu'a un « *fablab* » sur le sujet de la fabrication de projets éducatifs en interne.

- ## Apport du stage sur les *Soft*/*Hard Skills*

Au cours de ce stage, j'ai pu acquérir et développer certaines compétences tant au niveau technique (*Hard Skills*) qu'au niveau social (*Soft Skills*).

Tout d'abord, concernant les compétences techniques développées, elles interviennent principalement sur le développement du code : reprendre le travail d'un autre n'est jamais une tâche aisée et demande une certaine patience quant à la relecture complète et rigoureuse du code.

Le fait d'avoir eu à travailler sur une carte STM32, via l'environnement STM32 Cube IDE et en langage C m'a permis de mettre à profit des connaissances acquises en cours tout en voyant ces compétences se développer au rythme de ma progression sur le sujet de mon stage.

Pouvoir rapidement prendre en main un nouveau sujet est une compétence qui me semble être salvatrice dans la mesure où cela touche de nombreux sujets outre le domaine technique dans lequel je compte me professionnaliser.

Mes capacités d'adaptation s'en sont donc vues améliorées et je pense pouvoir affirmer que cela aura un impact certain sur la suite de ma carrière.

Cela tient aussi pour mes compétences en Python, qui étaient néanmoins un peu plus développées que celles en langage C. Cependant, avoir eu à développer l'IHM de zéro fut un défi assez conséquent et le fait d'avoir pu le surmonter me fait croire que j'ai acquis une certaine dextérité sur ce langage en particulier. Je n'avais pas eu à développer une IHM auparavant et je pense être apte, à partir d'aujourd'hui, à intervenir sur d'autres projets sous *Tkinter*.

Afin de conclure sur les *Hard Skills*, je tiens également à préciser que ce stage m'a permis d'apprendre à utiliser Git et GitHub, ainsi que SolidWorks. Provenant d'une formation autre qu'informatique, je n'ai pas eu, jusqu'à présent, l'occasion d'utiliser GitHub malgré sa forte popularité et je suis donc amplement reconnaissant et satisfait d'avoir pu être formé sur cet outil.

Par le passé, j'ai eu à travailler sur un logiciel de CAO nommé *Creo Parametric* et ce fut une expérience assez particulière, qui n'a ni abouti à une maîtrise du logiciel en lui-même, ni abouti au désir de développer des compétences en CAO. Néanmoins, cette formation sur SolidWorks a eu l'effet inverse, je ne dirais pas que j'ai une parfaite maîtrise du logiciel, mais je peux tout de même affirmer que faire de la CAO est devenu une activité que je trouve agréable et satisfaisante.

Concernant les *Soft Skills*, travailler globalement en autonomie, bien qu'il y ait eu un suivi fréquent de mon travail, a indubitablement eu un impact sur ma capacité à prendre des décisions, à m'adapter à des situations problématiques sans avoir nécessairement recours à un soutien quelconque. Être apte à chercher par soi-même la solution à un problème est une preuve d'indépendance et de maturité qui me semble être absolument essentielle d'avoir en tant que futur ingénieur.

Le fait d'avoir eu à présenter, maintes fois, mon projet fut également très formateur sur ma capacité à synthétiser mes pensées et à m'adapter au type de public visé, car présenter un projet scientifique à un public ne possédant pas le « bagage technique » facilitant la compréhension du sujet, n'a vraiment rien à voir avec un public qui le possède. Cette expérience m'a aussi permis de développer mon aisance à l'oral, ma capacité à outrepasser des éléments perturbateurs (en termes de concentration), ainsi que d'affronter un certain stress qui peut intervenir lors des présentations par exemple.

La multidisciplinarité du projet bras drone m'a permis de comprendre que d'une part j'apprécie travailler dans le domaine de la robotique et des drones et que d'autre part j'aime à avoir des missions mêlant *Software* et *Hardware* car j'y trouve un plus grand plaisir de varier entre le monde « concret » du matériel et le monde « abstrait » de la programmation.

- ## Place du stage dans mon projet d'avenir

En tant que projet orienté robotique et drone, ce stage a été une expérience enrichissante, grandissante et aura sans aucun doute une place importante dans mon curriculum vitae car elle s'inscrit parfaitement dans mon projet professionnel qu'est de devenir ingénieur en robotique. Ayant eu des formations diverses, tout au long de mes études, je pense que cette expérience professionnelle, en particulier, viendra affirmer mes choix de spécialisation et sera perçu comme un gage de mes compétences dans les domaines de l'informatique et de la robotique.

Je compte donc bien continuer sur cette voie et j'espère que j'aurai de nombreuses opportunités de travailler sur des projets aussi passionnants par la suite !

Bibliographie

D. Norris, *Programming with STM32. Get Started with the Nucleo Board and C/C++*, McGraw-Hill Education, 2018.

Liste de tutoriels vidéo *Getting started with STM32 and Nucleo*, Digi-Key Electronics, Shawn Hymel, 2019.

https://www.youtube.com/playlist?list=PLEBQazB0HUyRYuzfi4clXsKUSgorErmBv

Liste de tutoriels vidéo *Full tutorial Python Live streaming Data with Graphic UI from Arduino-STM32 using Tkinter, PySerial, Threading, Matplotlib, Numpy*, WeeW – Stack, 2021.

https://www.youtube.com/playlist?list=PLtVUYRe-Z-meHdTlzqCHGPjZvnL2VZVn8

Site web officiel de STMicroelectronics : https://community.st.com/s/

Rapport de stage de Monsieur N'DO Yann Kader Axel Obou, 2022.

https://github.com/InnovationLab-EFREIParis/Stage_1DOF_DroneBench/blob/main/01_doc/Rapports/Dossier%202021-2022%20Yann/rapport%20technique%20Stage%201DOF.pdf

Annexes

Annexe 1 : Dépôt GitHub du projet Bras Drone

https://github.com/InnovationLab-EFREIParis/Stage_1DOF_DroneBench

Annexe 2 : Répertoire des balises TAG

```
Répertoire des TAG

TAG_UC :

TAG_UC_001      État 'entrance' de l'automate
TAG_UC_002      État 'init_uc' de l'automate
TAG_UC_003      État 'info_mode' de l'automate
TAG_UC_004      Fonction Callback du Timer Interrupt
TAG_UC_005      État 'motor_ready' de l'automate
TAG_UC_006      État 'manual_mode_pot' de l'automate
TAG_UC_007      État 'manual_mode_term' de l'automate
TAG_UC_008      État 'init_gyro' de l'automate
TAG_UC_009      État 'instruct_angle' de l'automate
TAG_UC_010      État 'instruct_kp' de l'automate
TAG_UC_011      État 'instruct_ki' de l'automate
TAG_UC_012      État 'instruct_kd' de l'automate
TAG_UC_013      État 'auto_mode' de l'automate
TAG_UC_014      État 'landing' de l'automate


TAG_IHM :

TAG_IHM_001     Classe RootGUI, fenêtre principale accueillant l'interface de communication série
TAG_IHM_002     Classe ComGUI, interface de communication série
TAG_IHM_003     Classe MotorReadyGUI, extension de fenêtre accueillant le menu de modes/état 'Motor Ready'
TAG_IHM_004     Classe ModeTermGUI, extension de fenêtre accueillant l'état 'manual_mode_term'
TAG_IHM_005     Classe AutoModeGUI, extension de fenêtre accueillant l'état 'auto_mode'
TAG_IHM_006     Classe CalibrationGUI, extension de fenêtre accueillant le mode calibration
TAG_IHM_007     Classe TripModeGUI, extension de fenêtre accueillant le mode séquence de vol
TAG_IHM_008     Classe SerialCtrl, protocole de communication UART-MCU
TAG_IHM_009     Classe DataMaster, contient les fonctions relatives aux données d'entrée et au protocole de décodage de messages
```