|  |  |  |
| --- | --- | --- |
| Une image contenant texte, arts de la table, vaisselle, assiette  Description générée automatiquement | 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 : Heure :  
  
Composition du Jury :  
  
 - Président (responsable EFREI ) :   
  
 - Responsable du stage (Entreprise) :   
  
 - Invité(e) :

**PUBLICATION DU RAPPORT DE STAGE**  
 Le Responsable du stage :   autorise le stagiaire à publier le rapport de stage sur l’Intranet de l’Ecole. Signature

**Mots clés** Drones, Robots, Capteurs, Asservissement, PID, Microcontrôleur, IHM (*GUI*), Conception Assistée par ordinateur, Imprimantes 3D, Projet éducatif.

Une image contenant texte, arts de la table, vaisselle, assiette

Description générée automatiquement **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

[Remerciements 1](#_Toc129253416)

[Liste des figures 3](#_Toc129253417)

[Résumé technique 4](#_Toc129253418)

[Abstract 4](#_Toc129253419)

[Présentation de l’environnement du stage 5](#_Toc129253420)

[Stage 6](#_Toc129253421)

[1 Introduction 6](#_Toc129253422)

[1.1 Présentation succincte du dispositif 6](#_Toc129253423)

[1.2 Missions et objectifs du stage 7](#_Toc129253424)

[2 Étapes charnières du stage 9](#_Toc129253425)

[2.1 Prise en main de l’outil de gestion de versions 10](#_Toc129253426)

[2.2 Prise en main de STM32 Cube IDE 10](#_Toc129253427)

[2.3 Début du développement de l’IHM 13](#_Toc129253428)

[2.4 Poursuite du développement de l’IHM 15](#_Toc129253429)

[3 Approfondissement du sujet 17](#_Toc129253430)

[3.1 Mise en œuvre des modes « Calibration » et « Séquence de vol » 17](#_Toc129253431)

[3.2 Développement du mode « Séquence de vol » 23](#_Toc129253432)

[Conclusion 25](#_Toc129253433)

[Bibliographie 26](#_Toc129253434)

[Annexes 27](#_Toc129253435)

# Liste des figures

# Résumé technique

…

Mots clés :

# Abstract

… English version

Key words:

# 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 généraliste et expert 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, comportes-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 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 ces mêmes filières et majeure.

# Stage

# Introduction

Ce stage fait suite au développement conceptuel et matériel d’un bras drone à un degré de liberté, à visée éducative, 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.

## Présentation succincte du dispositif

Le dispositif du bras drone est constitué de plusieurs éléments à caractère mécanique et électronique. Le squelette est composé de 2 profilés en aluminium montés en T via des équerres de fixation faisant office de socle. Un troisième profilé est utilisé comme bras, monté en liaison pivot par rapport au socle (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 et une hélice.

Le principe du dispositif est donc d’assurer le mouvement du bras par rapport au socle, afin d’effectuer une rotation sur un seul plan, dans une fourchette de 0 à 90° (0° étant la position de repos du bras à la verticale), grâce à la poussée du drone. Au-delà du fait de simplement déplacer le bras par rapport au socle, 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.

Ainsi, le dispositif comprend également une carte de développement NUCLEO-L476RG de STMicroelectronics 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.

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.

[NOUVELLE PHOTO DU DISPOSITIF A PRENDRE / PHOTOS DETAILLEES ?]

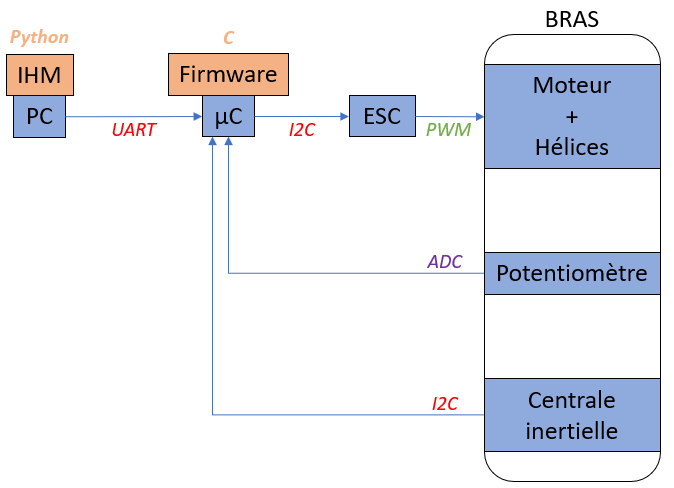


Figure 1: Schéma fonctionnel du dispositif

## 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* » (Fig.1).



Figure 2: Machine d'état initiale

La machine d’état 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.

# Étapes charnières du stage

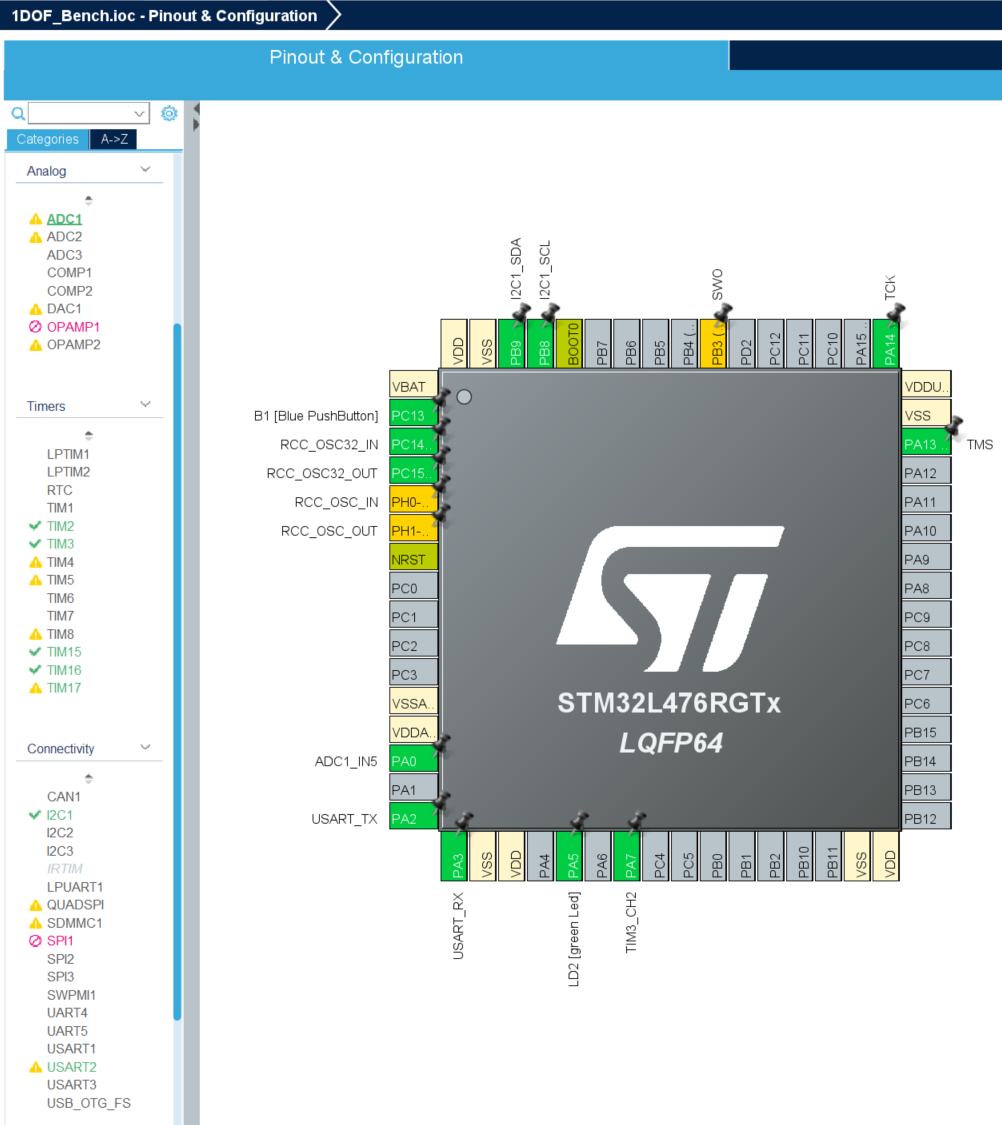
Dans un premier temps, les étapes citées ci-avant ont été suivies afin de remplir les missions principales du stage. Ces dernières m’ont alors permises de développer une première version de l’IHM souhaitée.

## Prise en main de l’outil de gestion de versions

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.

## Prise en main de STM32 Cube IDE



A partir de la machine d’état présentée dans l’introduction (Fig. 2), l’objectif premier a été d’optimiser le fonctionnement de l’automate. 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 et 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, 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.

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, 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 manquer 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* », durant lequel on attend que l’utilisateur presse la touche « Entrée », 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 soit la touche « Espace », 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 serait alors 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 utilité certaine 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 étape 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, qui permettrait à l’utilisateur d’acquérir des couples de données consigne (en % de gaz moteur)/position angulaire mesurée par la centrale inertielle, afin de pouvoir tracer une courbe et de pouvoir étudier les comportements statique et dynamique du dispositif.

## Début du développement de l’IHM

J’ai en premier temps cherché les bibliothèques qui me permettraient de créer une IHM en Python. J’ai préféré débuter une IHM avec *Tkinter*, plutôt que *PyQt* et *PySide*, qui est référencée comme la bibliothèque la plus simple pour commencer à créer des IHM et dont la documentation et exemples étaient suffisamment riches pour pouvoir atteindre mes objectifs dans un court délai.

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 : une unique fenêtre principale, qui s’agrandirait à mesure que l’on cliquerait sur certains boutons. La personne en question présentait le développement d’une IHM en Python pour pouvoir communiquer en série avec une STM32. Ainsi, une certaine partie des explications concernaient également la communication en série UART, ce qui m’est avéré bien utile.

Je me suis donc fortement inspiré de l’aspect de l’IHM de ce tutoriel, notamment de la racine de cette fenêtre principale, similaire au menu de connexion du terminal série que nous utilisons (Fig.2).



Figure 3: 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* ». Une fois qu’ils sont sélectionnés, le bouton « *Connect* » n’est plus grisé (« *disabled* ») 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 (Fig.3).

Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement

Figure 4: 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* (l’interface graphique), 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), 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)* », Fig.2). 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 (Fig.4), nous donne accès à un menu de modes via plusieurs boutons.



Figure 5: 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 à nous renvoyer à une nouvelle classe du même fichier, regroupant les fonctions du mode.

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.

## Poursuite du développement de l’IHM

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

Une image contenant texte

Description générée automatiquement

Figure 6: Extension de fenêtre propre au mode « Manual Mode Term », avant envoi d’une première consigne

Cette section de fenêtre (Fig.5) 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 de revenir au début du mode « *Manual Mode Term* », car cela nous permet d’enclencher directement une autre simulation sans avoir à revenir au menu de modes puis cliquer une fois encore sur le bouton « *Manual Mode Term* ») ;
* 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 la consigne. Les boutons « + » et « - » étaient alors reliés à la valeur textuelle et de par 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 (Fig.6).

Une image contenant texte

Description générée automatiquement

Figure 7: 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 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 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 les valeurs des coefficient PID, en rajoutant de nouveaux *widgets* !



Figure 8: Extension de fenêtre propre au mode « Auto Mode », avant envoi d’une première consigne

La section (Fig.7) se compose donc des mêmes éléments que « *Manual Mode Term* » (Fig.5), 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 (Fig.7) 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 et il est alors encore possible de modifier la valeur consigne et d’appuyer sur le bouton « *GO !* » (Fig.8).



Figure 9: Extension de fenêtre propre au mode « Auto Mode », après envoi de la consigne

# Approfondissement du sujet

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.

## Mise en œuvre des modes « Calibration » et « Séquence de vol »

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 (Fig.9).



Figure 10: 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* (Fig.10), permettant de rechercher puis sélectionner un fichier. Sur la figure 10, on peut observer dans quel répertoire se trouve le fichier texte en question à compléter.

Une image contenant texte

Description générée automatiquement

Figure 11: Boîte de dialogue "File Explorer" permettant de sélectionner un fichier

Une image contenant texte

Description générée automatiquement

Figure 12: 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 11. Il a tout d’abord comporté uniquement un certain nombre de lignes composées de 2 nombres entiers espacés d’une tabulation et a ensuite disposé de commentaires.

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, de par 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 (= 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* » (Fig.9) 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 (Fig.12 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 « *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 potentielle utilisation du dispositif, il nous a donc été demandé 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 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. La première étude étant dite dynamique (Fig.12) et la seconde est dite statique (Fig.13).



Figure 13: Exemple de graphique issu de "Show Graph" du mode "Calibration" - étude dynamique



Figure 14: 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* ».

Ensuite, il m’a été recommandé de m’informer au sujet de la méthode de communication à employer pour pouvoir transmettre les données mesurées car le « *polling* » m’imposait de renvoyer constamment un caractère pour pouvoir accéder aux fonctions de lecture du capteur. 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).

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. 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 Fig.11 dont les temps de mesures sont de 5s et donc 250 points par abscisse sur un graphique). Ce nombre de point 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.

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* » (Fig.14).



Figure 15: Extension de fenêtre propre au mode « Trip Mode »

Le principe est similaire, l’utilisateur est invité à sélectionner un fichier (cette fois-ci le fichier « *test trip* » de la figure 10), 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 (Fig. 15).

Une image contenant texte

Description générée automatiquement

Figure 16: 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 ». L’« *erreur* » proportionnelle correspond à la différence entre la consigne d’angle et la mesure de l’angle déduite des capteurs de la centrale inertielle. L’erreur intégrale correspond à l’accumulation (sommée) des erreurs proportionnelles, tandis que l’erreur dérivée prend en compte la différence entre l’erreur proportionnelle précédente et celle actuelle. Les valeurs successives de ces 3 variables ont été également rajouté dans la transmission de données.

## Développement du mode « Séquence de vol »

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

Il m’est alors paru essentiel de rajouter des *widgets* à l’extension « *TripModeGUI* », afin de pouvoir renseigner de nouveaux coefficients PID rapidement et consécutivement (Fig. 16), sans avoir à intervenir directement dans le « *main.c*» du *firmware* (on revient sur la même problématique de facilitation du renseignement des paramètres).



Figure 17: 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 vite limités en termes d’exploitation … En effet, il n’était alors possible que d’afficher les valeurs de « Position Angulaire X ». Il fallait donc trouver un moyen de pouvoir afficher les autres données/courbes sur un même graphique, comme effectué manuellement sur Excel, via le fichier csv (Fig.17).

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.

Il serait alors possible de créer des graphiques en html, qui s’afficheraient sur mon moteur de recherche (Firefox) en offline. *Plotly* prenant déjà en compte la possibilité d’activer ou désactiver certaines courbes via la légende des courbes. De plus, l’ergonomie et les outils plus développés que ceux de *matplolib*, m’ont fait préférer l’utilisation de *plotly* plutôt que *matplolib*. Par la suite, les graphiques sous *plotly* ont été déployé sur le mode « *Calibration* ».

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, notamment de fournir des données et graphiques aux « clients » de ce projet. Nous avions alors vu ensemble, et plus en détails, ce qui les intéresseraient d’exploiter (des simulations au repos par exemple, etc).

Dans cette optique de continuer à perfectionner le mode asservi, nous avions alors perçu la nécessité de déplacer les commandes d’asservissement de la boucle du « *main* » à la boucle propre au « *Timer interrupt* » évoqué précédemment. Déjà, car jusqu’à présent la lecture du capteur était double (une fois dans le « *main* » et une autre dans celle du *timer*) et par soucis de rigueur, bien que le système fonctionnât, il fut nécessaire de corriger ce problème.

J’ai donc modifié une énième fois, la machine d’état (le « *main* », la boucle principale 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 de modes.

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 90°, 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°.

# Conclusion

**Bilan :**

**Apport personnel :**

# Bibliographie

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

# Annexes

**Annexe 1 : Dépôt GitHub du projet Bras Drone**

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