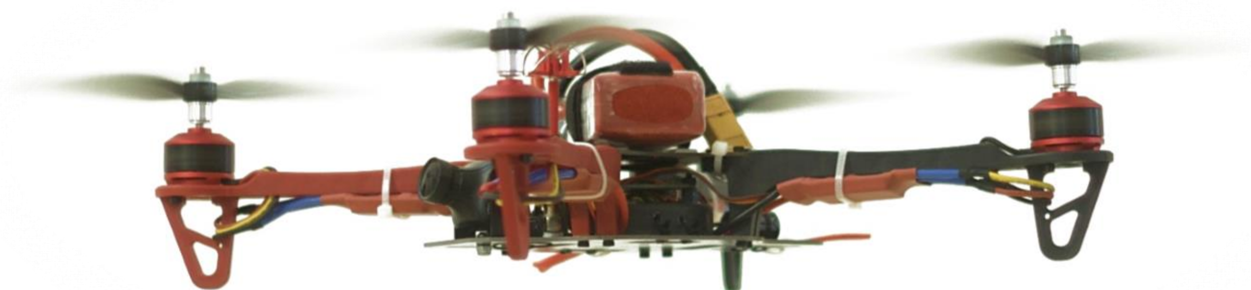
Travail de maturité

Construction et programmation d'un quadricoptère



Lycée cantonal de Porrentruy

Discipline : Informatique

Maître responsable : Didier Müller

Expert : Alain Stucki

Auteur : Gael Fleury

2019 - 2021

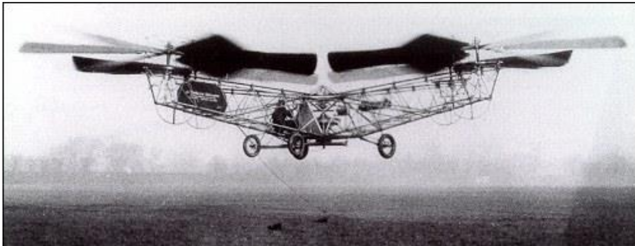
1. Table des matières

1. Table des matières ___ 1
2. Introduction ___ 2
2.1. Définition et histoire ___ 2
2.2. Principe de base et fonctionnement ___ 2
2.3. Recherches ___ 5
3. Construction ___ 6
3.1. Composants ___ 6
3.2. Construction du châssis et branchements principaux ___ 7
3.3. Mise en place de la carte MicroPython et câblage du « hub » ___ 9
4. Programmation ___ 10
4.1. Introduction ___ 10
4.2. Sortie PWM ___ 11
4.3. Lecture du signal PWM ___ 11
4.4. Lecture du signal de l'IMU ___ 11
4.5. Filtre complémentaire ___ 11
4.6. Systèmes de régulation ___ 12
4.7. Programme final ___ 13
5. Tests ___ 14
5.1. Premiers tests virtuels ___ 14
5.2. Tests réels ___ 15
5.2.1. Première tentative ___ 15
5.2.2. Banc d'essais ___ 16
5.2.3. Premiers décollages ___ 17
6. Conclusion ___ 18
6.1. Résultat ___ 18
6.2. Problèmes rencontrés ___ 18
6.3. Prix ___ 19
6.4. Temps ___ 20
7. Déclaration d'authenticité ___ 21
8. Autorisation ___ 21
9. Remerciements ___ 21
10. Bibliographie ___ 22
11. Annexes ___ 23
11.1. Fonctionnement PWM ___ 23
11.2. Images et vidéos ___ 24
11.3. Régulateurs ___ 24
11.4. Programmes ___ 24
11.4.1. main.py ___ 24
11.4.2. mpu6050.py ___ 30

2. Introduction

2.1. Définition et histoire

Un quadricoptère (ou quadrirotor) est un type particulier d'hélicoptère composé de quatre rotors horizontaux. Le tout premier ayant réussi à décoller date de 1907. Il a été développé par la société française *Breguet Aviation,* d'où son nom, le Bréguet n°1, d'un poids de 580 kg. Il tire son inspiration d'un roman de Jules Verne, *Robur le Conquérant* publié en 1886.



Le premier quadricoptère à avoir décollé de manière contrôlée est le quadrirotor de Bothezat (image ci-contre) en 1923. Malgré plusieurs vols réalisés avec succès, l'armée américaine va mettre fin au contrat de Bothezat. L'intérêt pour les quadrirotors s'est ensuite peu à peu dissipé, avant de se populariser il y a une quinzaine d'années à la suite de nombreuses avancées technologiques.

De nos jours, les quadricoptères (souvent appelés drones) se sont popularisés et sont utilisés dans de multiples domaines : le militaire, la recherche, le sauvetage, l'inspection et pour le loisir également.

2.2. Principe de base et fonctionnement

Pour pouvoir se déplacer dans l'espace, le quadricoptère doit pouvoir réaliser quatre mouvements (caractérisés par les quatre axes des joysticks de la télécommande). Tous ces mouvements sont possibles en modifiant indépendamment les vitesses des quatre moteurs. Ces quatre mouvements sont :

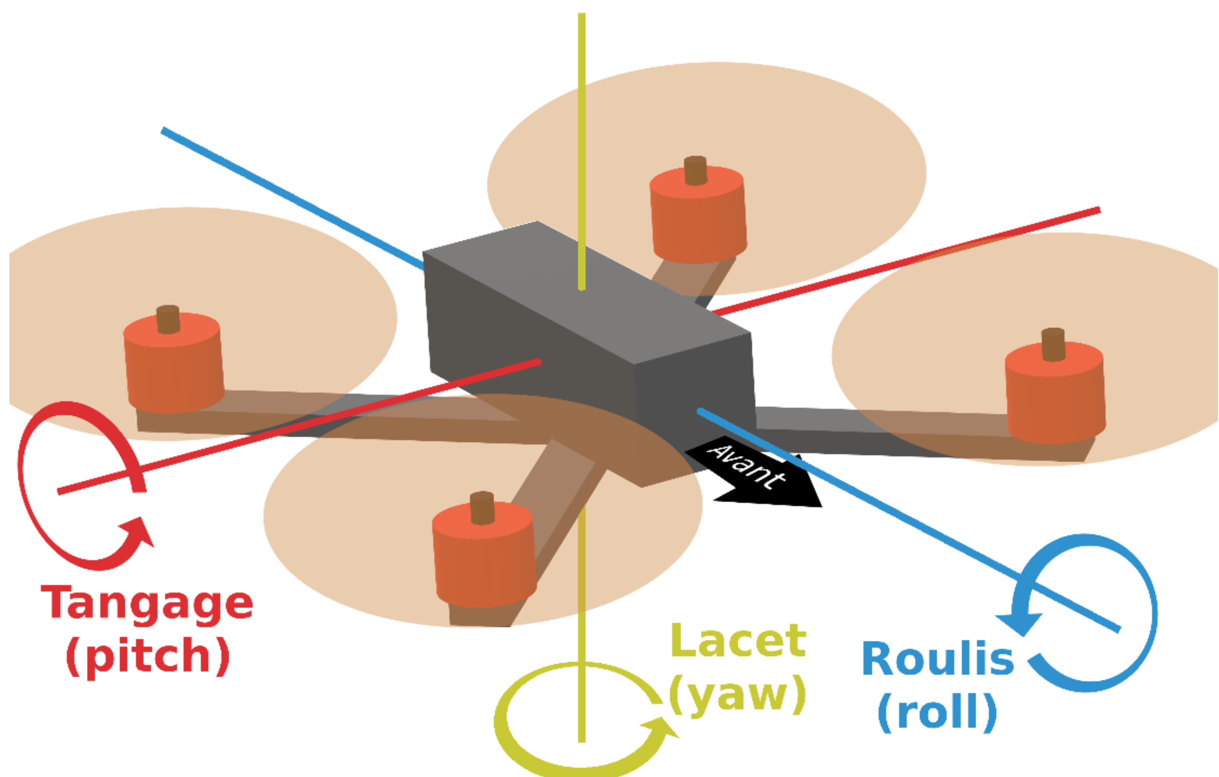
1. Les déplacements verticaux ou autrement dit, le contrôle de l'altitude (monter et descendre) (déplacement sur l'axe Z)
2. Les déplacements frontaux (avancer et reculer) (déplacement sur l'axe X)
3. Les déplacements latéraux (se déplacer sur le côté à la manière d'crabe) (déplacement sur l'axe Y)
4. Les rotations sur lui-même (tourner sur lui-même sur le même plan) (rotation autour de l'axe Z)

Nous allons prendre l'exemple d'un quadricoptère en X car c'est la forme que j'ai utilisée et également la plus courante. Il existe d'autres formes de quadricoptère (comme celui en croix) mais elles sont moins répandues.

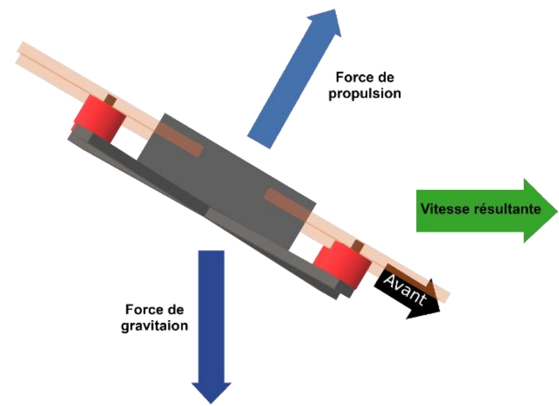
Premièrement, pour pouvoir monter et descendre, il suffit de modifier la vitesse des quatre moteurs simultanément pour créer plus ou moins de portance.

Ensuite, pour pouvoir avancer par exemple, on augmente la vitesse des deux moteurs arrière afin de faire pivoter le quadricoptère en avant. Une fois l'inclinaison désirée atteinte, on stabilise à nouveau la vitesse des moteurs, et l'appareil va avancer, tel un hélicoptère. Les déplacements latéraux se font selon le même principe que les déplacements frontaux, il faut augmenter la vitesse des moteurs du côté opposé auquel nous souhaitons déplacer le quadricoptère. Ces deux types de rotation portent les noms suivants : le tangage (pitch en anglais) pour les rotations frontales et le roulis (roll en anglais) pour les rotations latérales.

Pour pouvoir tourner sur lui-même, le quadricoptère utilise un principe un peu moins connu : l'inertie de rotation. C'est en raison de ce principe que les hélicoptères ont un rotor de queue (la petite hélice verticale à l'arrière). Par le troisième principe de la dynamique de Newton, (lorsqu'on applique une force sur un objet dans une direction, on subit une force égale dans le sens opposé), le rotor principal tournant dans un sens, l'hélicoptère va subir une accélération opposée et va se mettre à tourner sur lui-même dans le sens opposé à cette rotation. Le rotor de queue est là pour contrer ce mouvement et permettre à l'hélicoptère de rester stable. Sur un quadricoptère, cet équilibre est déjà accompli, car deux hélices (opposées) tournent dans un sens et les deux autres dans l'autre sens ; les forces de réaction des quatre moteurs s'annulent donc. En faisant accélérer deux moteurs opposés et en freinant les deux autres moteurs, le quadrirotor va tourner sur lui-même. Les hélices accélérant dans un sens font accélérer le quadricoptère dans l'autre sens. Cette rotation porte le nom de lacet (en anglais yaw).



Ces quatre mouvements combinés permettent aux quadricoptères de se mouvoir dans l'espace. On pourrait également imaginer un drone comme étant un vecteur dont on peut modifier l'intensité (la puissance des moteurs) ainsi que la direction en trois dimensions (grâce aux trois axes de rotation).



Ces quatre mouvements sont caractérisés par les quatre canaux de contrôle de la télécommande (deux joysticks se mouvant sur deux axes chacun). Pour une télécommande en mode 2 (le type le plus courant en modélisme), les quatre canaux sont numérotés de la manière suivante :

- L'axe horizontal du joystick de droite, soit le contrôle du roulis (roll) est associé au canal numéro 1
- L'axe vertical du joystick de droite, soit le contrôle du tangage (pitch) est associé au canal numéro 2.
- L'axe vertical du joystick de gauche, soit le contrôle de l'altitude (de la puissance) est associé au canal numéro 3. Cet axe a la particularité de rester à sa dernière position (il n'a pas de ressorts qui le ramène au centre). Cela permet au pilote de lever le doigt sans que la puissance ne soit changée.
- L'axe horizontal du joystick de gauche, soit le contrôle du lacet (yaw) est associé au canal numéro 4

2.3. Recherches

Avant même le début des inscriptions des travaux de maturité, j'avais envie de faire quelque chose qui sortait de mon imagination et donc qui me tenait vraiment à cœur. Je me suis mis à réfléchir bien avant les inscriptions. L'idée d'un drone m'est venu assez naturellement, puisque c'était pour moi déjà une passion car j'en avait déjà construit un sans le programmer moi-même. Je suivais en ce temps le cours d'option complémentaire en informatique, ce qui m'a donné les bases du langage *Python*[1]. J'avais également quelques sommaires connaissances en *javascript*[2] et *HTML*[3]. La problématique était que tous ces langages ne conviennent pas à la programmation d'un drone. Ils ne peuvent en effet pas être codés sur un ordinateur assez petit et léger pour pouvoir être embarqué à l'intérieur d'un objet volant. La seule option restante était donc *Arduino*[4] ; un langage spécialement créé pour des *microcontrôleurs*[5], de petits puces simples et très légères qui convenaient assez bien au projet que j'avais en tête. Seulement, Arduino est basé sur le langage C que je ne connaissais pas du tout et qui est assez complexe. J'allais devoir apprendre ce langage totalement nouveau pour moi afin de réaliser mon travail de maturité.

Mais tout a changé, quand, quelques mois avant le choix du travail de maturité, j'ai découvert *MicroPython*, un langage encore très jeune et peu connu, également créé pour des microcontrôleurs, mais cette fois sur une base de Python. J'ai immédiatement pensé qu'il pourrait m'éviter de devoir apprendre l'Arduino. Je me suis alors commandé un microcontrôleur compatible pour voir comment ça fonctionnait. Malgré une forte parenté avec Python, le langage demande passablement d'adaptation dû à la possibilité d'interagir avec les interfaces d'entrée-sorties. De cette découverte, j'ai réalisé plusieurs tests et programmes, pour vérifier que mon projet soit techniquement réalisable, je ne voulais pas être bloqué en pleine réalisation par un obstacle infranchissable. Après plusieurs tests, les quelques points pouvant être source de problèmes se sont éclaircis, le projet avait des chances d'être réalisable et je me suis donc lancé. J'ai également regardé sur internet si quelqu'un avait déjà réussi à développer un drone en MicroPython, afin de vérifier la faisabilité, mais je n'ai rien trouvé. Je serais donc une sorte de pionnier, si mon projet venait à se réaliser.

[1] Langage de programmation créé par Guido van Rossum en 1991 très populaire ces dernières années.
[2] Langage de programmation principalement utilisé dans la conception d'applications web.
[3] *HyperText Markup Language* ou abrégé *HTML* est un langage utilisé pour la représentation de pages web.
[4] Marque de cartes électroniques libres de droits, désigne par extension le langage utilisé pour les programmer qui a une très proche du langage C++.
[5] Petit ordinateur basic rassemblant uniquement les composant essentiels.

3. Construction

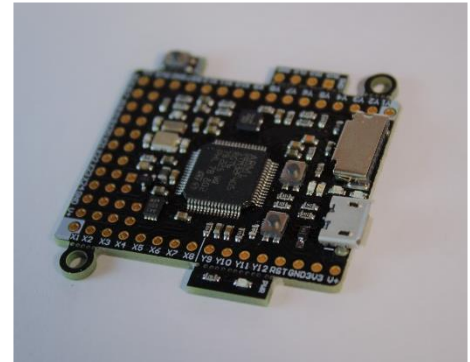
3.1. Composants

Plusieurs composants sont indispensables au fonctionnement d'un quadrirotor :

- Un châssis supportant tous les composants.

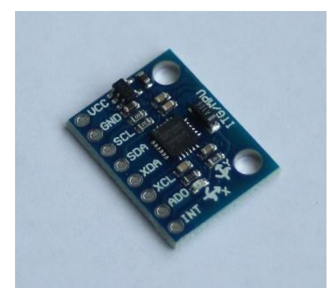
- Quatre moteurs accompagnés de quatre hélices, deux dans le sens horaire et deux dans le sens anti-horaire (même sens dans le moteur opposé) afin de propulser notre quadricoptère dans les airs.

- Un contrôleur de vol, aussi appelé FC (de l'anglais Flight Controler). C'est le « cerveau » du quadricoptère, il va lire toutes les entrées pour pouvoir calculer les vitesses des moteurs puis envoyer le signal correspondant. J'ai choisi d'utiliser une Pyboard. C'est une carte qui a été spécialement conçue par les créateurs de MicroPython pour faire fonctionner leur langage.



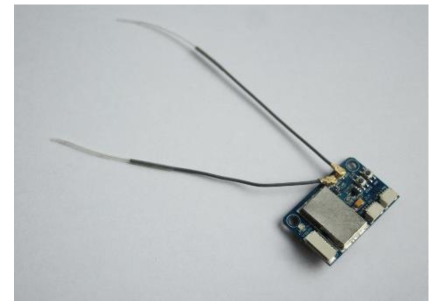
- Quatre régulateurs de vitesse pour les moteurs, appelés ESC (Electronic Speed Contrôler ou contrôleur de vitesse électronique en français), qui transforment le signal PWM (Pulse Width Modulation) venant du contrôleur de vol en signal pour les moteurs. Ceux que j'ai choisis possèdent un régulateur 5 volts, indispensable pour pouvoir alimenter la Pyboard et le récepteur.

- Un gyroscope et un accéléromètre qui, regroupés forment l'IMU (Inertial Mesurment Unit). C'est grâce à cette pièce que l'on peut calculer l'angle de notre quadricoptère. Ici, je vais utiliser un mpu6050.



- Une batterie qui va alimenter tous les composants électroniques. Je vais utiliser d'anciennes batteries lithium-ion polymère (abrégé LiPo) trois cellules fournissant un voltage d'environ 11,1 volts. Elles ont une capacité de 1500 mAh (milliampères heure) à 1800 mAh. Le connecteur est de type XT60.

- Et enfin, comme notre quadrirotor n'est pas autonome, un récepteur qui va recevoir les informations de la radiocommande. J'ai choisi le modèle Flysky X6B car il est compatible avec ma télécommande (Flysky fs-i6x) et il comporte plusieurs sorties différentes du signal reçu.

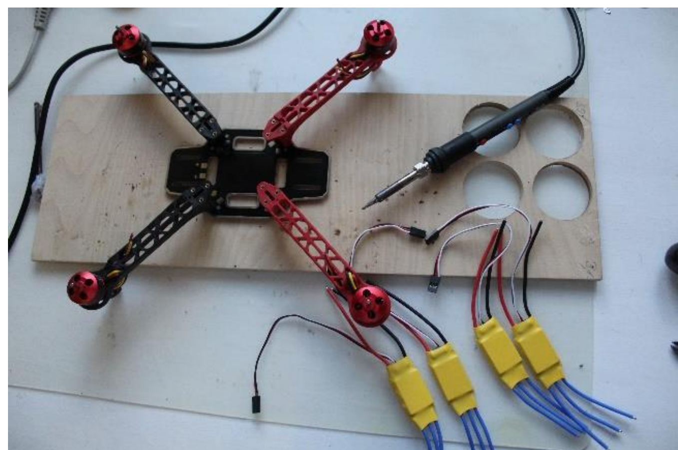


Si certains termes sont en anglais, c'est parce qu'ils ne sont quasiment jamais utilisés en français, ils seront donc utilisés dans leur terme le plus courant dans le reste du rapport. Toutes les pièces que j'ai utilisées ont été commandées sur le site *banggood.com* qui est un site chinois spécialisé dans la vente de produits électroniques (en particulier pour le modélisme).

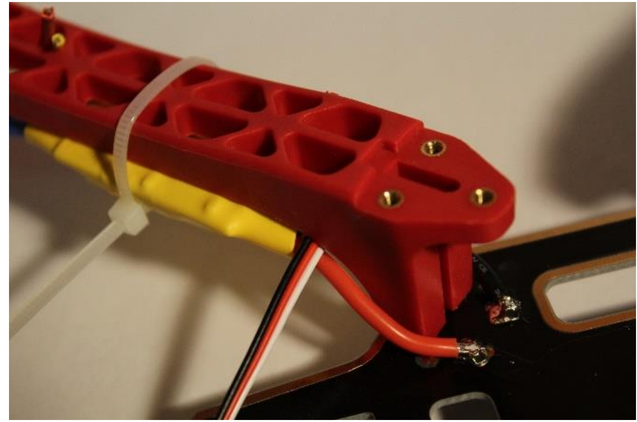
3.2. Construction du châssis et branchements principaux

Après la réception de toutes les pièces nécessaires, je me suis attelé à la construction de mon engin. La construction n'est pas une étape très difficile et fastidieuse, pour autant que l'on ait les outils nécessaires et l'habitude.

La première étape est le châssis. C'est l'étape la plus simple, il suffit de visser quelques vis et le tour est joué. Viennent ensuite les moteurs qui se fixent également à l'aide de vis. Puis les ESC qui sont maintenus aux bras du châssis par des serre-câble[6].

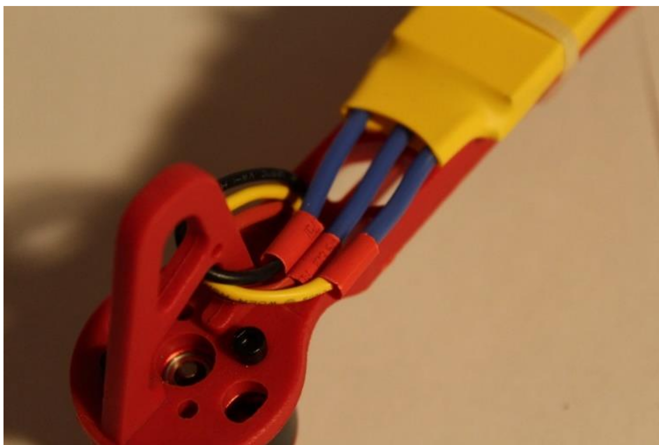


[6] Aussi appelé collier de serrage ou riselan, petite pièce en plastique utilisé pour maintenir des câbles ensemble.



Pour la deuxième étape, il faudra souder toutes les connexions entre elles. Comme mon châssis contient des emplacements de soudure, cette étape est facilitée. On fait fondre un peu d'étain sur les petits emplacements carrés prévus à cet effet. On coupe ensuite les fils positifs et négatifs des ESC à la bonne longueur. Puis à l'aide d'une pince, on vient enlever un demi-centimètre environ de gaine protectrice pour pouvoir faire fondre un peu d'étain sur la partie mise à nue. Une fois cette étape réalisée, on pose le bout du fil sur le bon carré et l'on appuie avec le fer à souder sur le fil, ce qui fait fondre l'étain sur le fil et sur la carte (où il a été mis au préalable) et la soudure a lieux. Enfin, on peut un peu tirer sur les câbles pour s'assurer de la solidité de la connexion. Il faudra répéter les étapes ci-dessus pour chaque ESC et pour le connecteur de la batterie.

Vient ensuite la soudure des trois fils du moteur aux trois fils de sortie de l'ESC. Cette étape se déroule de la même manière que l'étape précédente, à la seule différence que la soudure se fait de fil à fil. Pour protéger la soudure, j'ai l'habitude d'ajouter une gaine thermoréractable[7] (en rouge sur la photo ci-dessous), mais un petit bout de ruban adhésif peut très bien faire l'affaire. A ne pas oublier pour cette étape : le sens de rotation du moteur dépend de l'ordre des trois fils, si la rotation se fait dans le mauvais sens, il suffit d'échanger deux connexions entre elles. Pour cause, ce sont des moteurs à trois phases (symbolisés par les trois câbles, 1 2 3) cycliques ; en échangeant deux des câbles entre eux, la suite se lit à l'envers (1 3 2), le cycle est donc inversé et le moteur tourne dans l'autre sens.



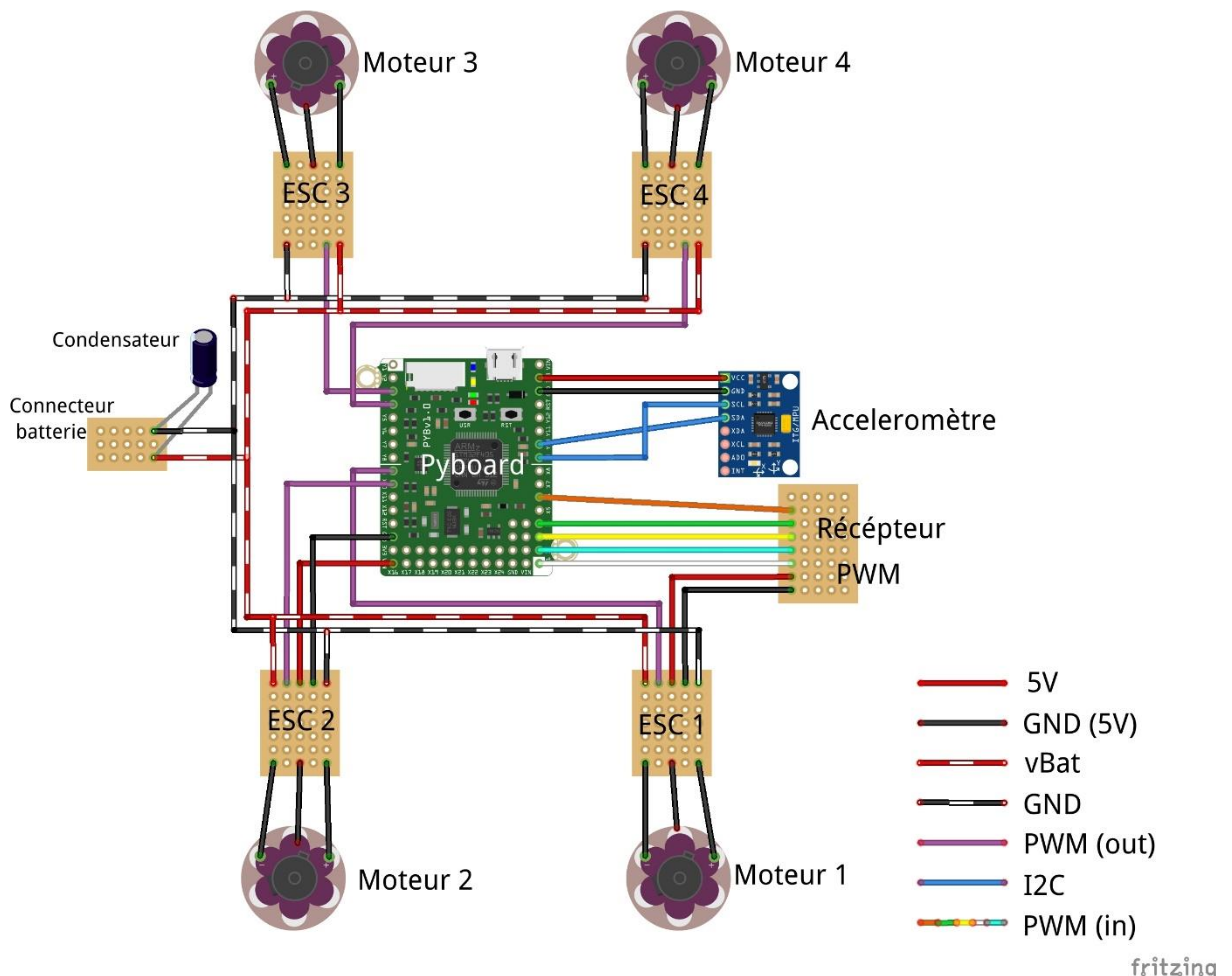
J'ai également ajouté un condensateur[8] au niveau du connecteur de la batterie. Cette pièce permet d'atténuer les variations de tension de la batterie qui sont provoquées par les changements rapides des vitesses des moteurs. Cette pièce n'est pas obligatoire, mais j'ai préféré l'ajouter.

[7] Composant en forme de petit tube en plastique qui se resserre sur lui-même quand il est exposé à de la chaleur. Cette gaine est souvent utilisée pour protéger des soudures fil à fil.
[8] Composant électronique se comportant comme une sorte de batterie temporaire. Il permet de stabiliser une alimentation électrique.

3.3. Mise en place de la carte MicroPython et câblage du « hub »

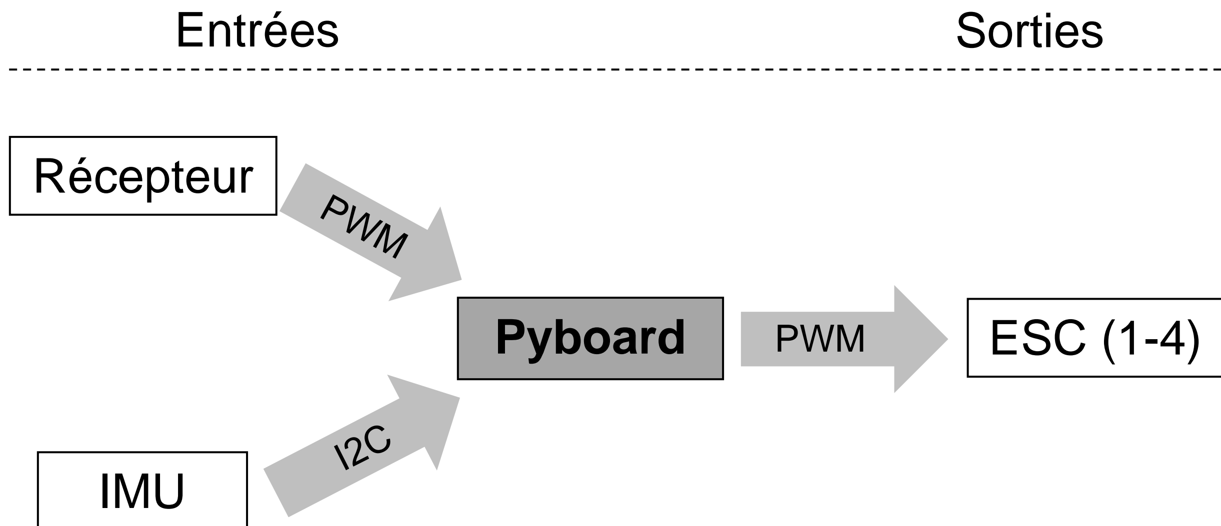
Pour fixer les éléments électroniques, je n'ai pas tout de suite trouvé une solution qui me convenait. Puis j'ai eu l'idée d'utiliser le fond d'une ancienne boite en plastique, où toutes les pièces centrales viendraient se fixer. Je l'ai sobrement nommé le « hub ». Il constitue la partie centrale du drone, que l'on peut comparer à une tête humaine : la Pyboard est le cerveau qui réfléchit, le récepteur est le tympan qui entend les informations et l'IMU est l'oreille interne qui se repère dans l'espace. Les moteurs seraient les muscles, qui traduisent les directives du cerveau en mouvement.

La dernière étape de la construction consiste au branchement de tous les composant électroniques entre eux. J'ai réalisé le schéma ci-dessous (avec un programme appelé *Fritzing*[9]) qui regroupe tous les branchements nécessaires au bon fonctionnement du drone.



[9] https://fritzing.org/

Vous trouverez en annexe une explication concernant les fils qui doivent ne pas être connectés. Et voici un schéma récapitulatif des connexions ainsi que des signaux envoyés et reçus par la Pyboard :



4. Programmation

4.1. Introduction

Pour la partie programmation, j'ai réalisé plusieurs programmes n'ayant qu'une seule fonction. J'ai minutieusement testé chacun de ces « sous-programmes » pour m'assurer de leur fiabilité. J'ai ensuite écrit un programme les regroupant tous. Il y a également du code qui ne venait pas des sous-programmes qu'il a fallu ajouter pour veiller au bon fonctionnement ce celui-ci. Chaque sous-programme possède une partie dans la boucle infinie et une autre est hors de la boucle. Cette dernière sert à l'initialisation des composants et la mise en place de variables et de fonctions. Je vais passer en revue les principaux sous-programmes qui composent le programme final.

Avant de débuter la partie programmation, j'ai dû choisir quel mode de contrôle je voulais programmer. Deux modes étaient possibles : le mode « stabilisé », où la position du joystick de la télécommande est traduite en angle (donc si on lâche les commandes, le drone va se mettre à un angle de 0°) et le mode « manuel », où cette dernière est traduite en vitesse angulaire (si on lâche les commandes, le drone va rester dans sa position actuelle). J'ai finalement opté pour le mode stabilisé plutôt que le mode manuel dont j'avais l'habitude, car bien que plus simple à piloter, il me paraissait également plus complexe et plus intéressant à programmer que le mode manuel, car ce dernier n'utilise pas l'accéléromètre.

4.2. Sortie PWM

Afin de faire tourner les moteurs à la vitesse désirée, il faut indiquer la vitesse désirée à l'ESC via un signal appelé PWM (de l'anglais Pulse Width Modulation, traduit en français par modulation de largeur d'impulsions). Ce type de signal est très courant, car il est aussi utilisé pour contrôler des servo-moteurs[10]. Pour plus d'information par rapport à ce signal, vous trouverez en annexe une explication plus détaillée. En raison de la grande popularité de ce signal, une fonction déjà intégrée dans MicroPython permet de créer un signal à une fréquence donnée.

4.3. Lecture du signal PWM

Le récepteur que j'ai choisi (Flysky X6B), envoi plusieurs sortes de signaux différents par différents connecteurs. J'ai pu donc choisir quel signal je voulais lire avec la Pyboard. Sans hésitation j'ai choisi le plus simple et le plus répandu : le signal PWM. C'est le même type de signal que celui que demande les ESC pour gérer la vitesse des moteurs. Le seul désavantage de ce signal est qu'il faut un fil par canal de transmission. De ce fait, il y a cinq connexions entre le récepteur et la Pyboard (quatre pour les quatre axes de commande, plus un pour la sécurité). Ce sous-programme est un peu plus complexe, car il n'est pas courant de lire un signal PWM. Je me suis inspiré d'un programme de Dave Hylands (partagé sur Github[11] sous le pseudo de *dhylands*) pour créer le mien.

4.4. Lecture du signal de l'IMU

Le lien avec l'IMU se fait par un autre protocole nommé I2C (pour Inter-Integrated Circuit). C'est un signal numérique, ce qui rend son utilisation complexe. L'avantage est que l'ensemble des informations est transmis sur deux fils seulement. Pour ce sous-programme, j'ai utilisé comme base le programme de *sergeLabo* (pseudo Github) que j'ai modifié afin de couvrir mes besoins.

4.5. Filtre complémentaire

Pour bien comprendre le but d'un filtre complémentaire, il faut d'abord se pencher sur le fonctionnement du gyroscope et de l'accéléromètre. L'accéléromètre peut sentir l'accélération normale (la gravité dans un cas stationnaire). Tandis que le gyroscope peut sentir la vitesse angulaire, c'est-à-dire la vitesse de rotation du drone quand celui-ci est en rotation sur lui-même. Il est possible d'obtenir un angle par rapport à l'horizontale en utilisant un seul des

---

[10] Pièce généralement utilisée dans des avions télécommandés pour incliner les surfaces de contrôle. Elle est composée d'un petit moteur et d'un potentiomètre. Elle peut généralement bouger à 180° et connait son orientation.

[11] Plateforme de partage de codes informatiques, https://github.com/

deux capteurs. Mais ces deux méthodes de mesures ont chacune leurs inconvénients et leurs avantages : l'accéléromètre est très précis sur le long terme (il n'a pas besoin d'être calibré) mais il n'est pas précis lorsque qu'il y a des accélérations autres que la gravité (les mouvements du drone ainsi que les vibrations), tandis que le gyroscope est très précis sur le court terme mais peut « dériver » sur le long terme. C'est ici que le filtre complémentaire entre en jeu, en tentant de réunir les avantages des deux capteurs tout en évitant les désavantages. Le fonctionnement est assez simple : il va essayer de recalibrer continuellement le gyroscope avec l'accéléromètre. Voici l'équation d'un filtre complémentaire :

```
AngleFiltré = 0.97 * AngleGyro + 0.03 * AngleAccel
```

(la somme des deux constantes doit valoir 1, il faudra trouver une répartition adaptée à notre système)

4.6. Systèmes de régulation

Pour que le drone puisse voler, il a fallu trouver un moyen d'utiliser l'angle reçu du filtre complémentaire et les instructions reçues par le récepteur (ou autrement dit, la différence entre la position actuelle et la position désirée, que l'on appelle « erreur ») pour calculer la vitesse des moteurs. C'est ce qu'on appelle un système de régulation (*feedback loop* en anglais). C'est la pièce centrale du programme car c'est elle qui va définir les caractéristiques de vol ainsi que la maniabilité du quadricoptère. En réfléchissant à ce projet, trois systèmes me sont venus à l'esprit :

- le premier, un système simple, de type proportionnel. La différence entre l'angle actuel et l'angle désiré, c'est-à-dire l'erreur, est multipliée par une constante pour obtenir la correction à appliquer aux moteurs.

```
moteur[0] += (ch1/toDeg - mpu_6050.AngleX) * K
```

*(augmentationDeLaVitesseD'UnMoteurPourL'AxeX = erreurSurX * constante)*

- le deuxième, un système un peu plus complexe que le système proportionnel que je vais appeler quadratique. L'erreur est cette fois-ci élevée à une puissance avant d'être multiplié par une constante. L'élévation à la puissance pourrait permettre d'améliorer la précision pour les petits mouvements. Cependant, il faut choisir une puissance impaire pour que les erreurs négatives restent négatives, j'ai donc pris 3.

```
moteur[0] += (ch1/toDeg - mpu_6050.AngleX) ** 3 * K
```

*(augmentationDeLaVitesseD'UnMoteurPourL'AxeX = erreurSurX ^3 * constante)*

- et enfin, un système PID (pour Proportionnel Intégral Dérivé), qui est très répandu dans plusieurs domaines dont la robotique, plus complexe et plus difficile à régler mais beaucoup plus précis et plus « rapide » (moins de temps de réaction). L'erreur est traitée de trois

manières différentes (d'où le nom du système). Chacun de ces traitements est dirigé par une constante, qui demande un réglage assez fin.

Voici le système PID d'un axe dans mon code :

```
    error1 = ch1 / toDeg - mpu_6050.AngleX    #calcule de l'erreur

    P1 = error1 * P                                     #traitement proportionnel
    I1 = I1 + error1 * I * mpu_6050.intervalle          #traitement intégral
    D1 = (error1 - error1_old) * D / mpu_6050.intervalle # traitement dérivé

    correction1 = P1 + I1 + D1      #somme des traitements
    if correction1 > 400:       #limitation de la correction
        correction1 = 400
    elif correction1 < -400:
        correction1 = -400

    moteur[0] -= correction1        #application de la correction sur chaque moteur
    moteur[1] -= correction1
    moteur[2] += correction1
    moteur[3] += correction1

    error1_old = error1     #mettre l'erreur dans une variable pour l'utiliser au
prochain cycle
```

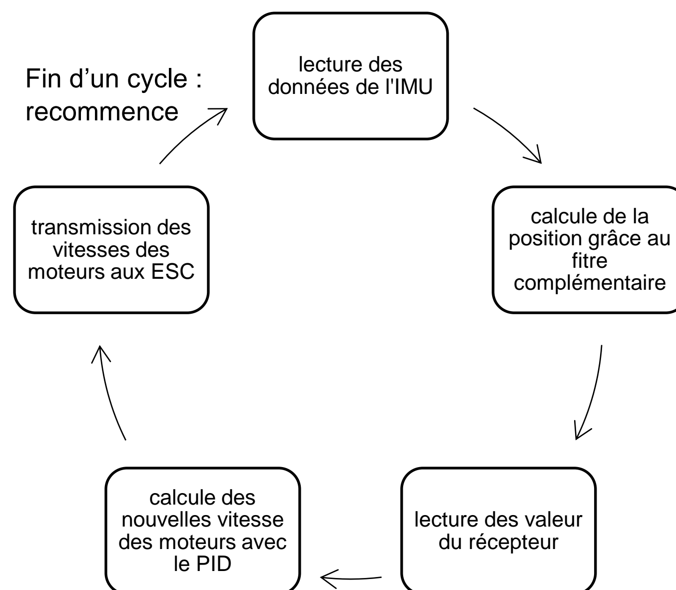
Nous verrons dans le chapitre 4 que le système PID s'est assez vite imposé. Il demande en revanche beaucoup plus temps de réglage.

4.7. Programme final

Pour la mise en commun des programmes, je n'ai pas rencontré trop de problèmes. La plupart des sous-programmes ne demandent pas de modifications majeures. Il y a cependant plusieurs ajouts majeurs qui ne figuraient pas dans les sous-programmes. Tout d'abord, il fallait utiliser le cinquième canal de la télécommande pour intégrer la sécurité. La valeur de ce cinquième canal correspond à la position du commutateur rouge sur la télécommande (cf. photo chapitre 1.2.). Si celui-ci est mis dans la position haute, les moteurs doivent s'arrêter. Cette fonction est indispensable afin de pouvoir manipuler et arrêter le quadricoptère après l'atterrissage. Elle peut aussi être utilisée en cas de perte de contrôle ou d'autre problèmes en vol. J'ai utilisé ce même canal pour détecter une potentielle déconnexion entre le récepteur et la télécommande. Si la Pyboard venait à en détecter une, elle va attendre quelques cycles en tentant de se connecter à nouveau, et dans le cas contraire, elle va immédiatement arrêter les moteurs. En effet, il vaut mieux retrouver l'engin en plusieurs pièces plutôt que de le perdre définitivement.

Il a également fallu ajouter la boucle infinie ainsi que le contrôle du temps de cycle de cette dernière. Il est important de trouver un moyen de sortir de la boucle infinie, afin de pouvoir modifier le code sur la Pyboard. J’ai remarqué qu’il était très probable de provoquer une réinitialisation de la carte si du code était ajouté pendant son fonctionnement. J’ai donc fait en sorte que le programme s’arrête automatiquement si la Pyboard ne détecte aucune connexion avec la télécommande après quelques secondes. Également, pour les premières versions de mon programme, je faisais fonctionner la boucle infinie à une fréquence de rafraichissement fixe, ceci afin d’éviter tout problèmes liés à des temps de cycle différents.

Après ces derniers ajouts, le programme était terminé. Vous pouvez le retrouver en intégralité dans les annexes. En voici un schéma récapitulatif représentant les différentes étapes lors d’un cycle :



5. Tests

5.1. Premiers tests virtuels

Avant de procéder à des tests réels, j’ai d’abord effectué quelques « simulations ». Pour ce faire, je n’ai pas branché les moteurs. J’ai fait tourner le programme dans une version un peu modifiée, en gardant branchée la carte à mon ordinateur, ce qui m’a permis, dans un premier temps, de vérifier que le programme fonctionnait correctement, sans avoir à prendre de risques pour

```
COM6 - PuTTY
MPY: sync filesystems
MPY: soft reboot
MicroPython v1.12 on 2019-12-20; PYBv1.1 with STM32F405RG
Type "help()" for more information.
>>> import mpu6050
>>> i2c = I2C(scl='Y9', sda='Y10', freq=400000)
>>> mpu_6050 = mpu6050.MPU6050(i2c)
>>> mpu_6050.lecture_capteurs()
-0.1627946 0.09802512 0.606145 2.268936
>>> mpu_6050.lecture_capteurs()
-0.6340212 0.3580559 0.7282825 2.162963
>>> mpu_6050.lecture_capteurs()
-0.8916701 1.202377 0.5756107 4.57427
>>> mpu_6050.lecture_capteurs()
-0.8531629 1.667961 0.5756107 1.803962
>>> mpu_6050.lecture_capteurs()
-0.8296955 1.988694 0.6672137 1.061042
>>> mpu_6050.lecture_capteurs()
-1.014524 2.349253 0.6824809 0.747995
>>> mpu_6050.lecture_capteurs()
-0.8327229 2.404781 0.6977481 0.545934
>>> mpu_6050.lecture_capteurs()
-0.5059132 2.47196 0.606145 0.391062
>>> mpu_6050.lecture_capteurs()
-0.3429019 2.616902 0.7282825 0.339145
>>>
```

moi-même ou pour le matériel. Cela permet aussi d'afficher les valeurs lues issues du récepteur, du gyroscope et les valeurs calculées de la vitesse des moteurs sur l'ordinateur. Cela m'a aussi permis de connaître l'orientation du gyroscope (axe x et y), et j'ai pu modifier quelques variables en conséquence. J'ai également pu tester la sécurité. Après ces quelques tests, j'étais prêt à finaliser la construction, et à commencer les premiers tests sans les hélices, qui se sont avérés concluants.

En temps normal, la Pyboard agit comme une clé USB ; elle apparait dans les périphériques de stockage une fois branchée. Pour importer un programme sur la carte, il suffit donc juste de copier le fichier Python dans le répertoire (qu'il faudra renommer en main.py) dans le répertoire et la carte va l'exécuter à son prochain démarrage. Mais dans un cas où l'on veut afficher des données sur l'écran d'un ordinateur, il faut utiliser un programme permettant la communication directe avec la Pyboard. Parmi les plusieurs programmes disponibles, j'ai choisi PuTTY[12] car je l'avais déjà utilisé auparavant et il est assez simple d'utilisation. Ce programme permet en effet d'accéder au REPL (pour *Read, Evaluate, Print and Loop* traduit par lire, évaluer, écrire et recommencer) Python de la carte.

5.2. Tests réels

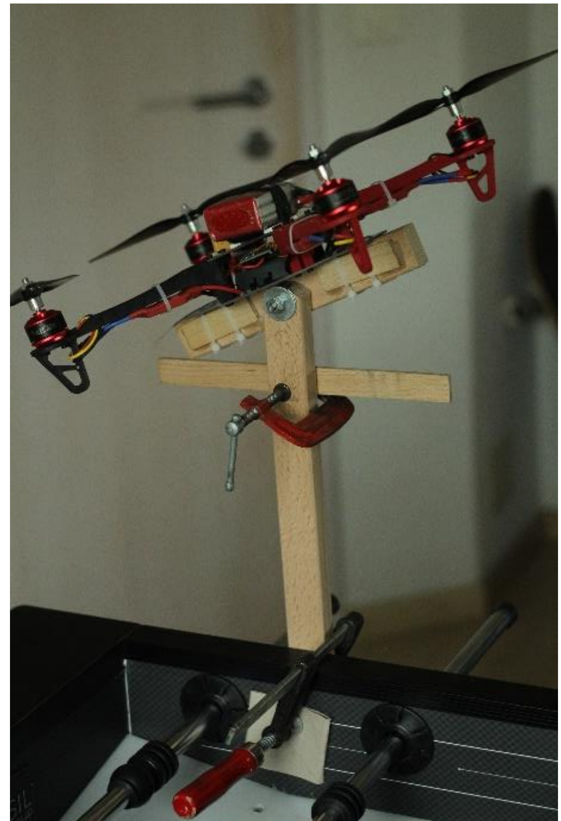
5.2.1. Première tentative

Une fois les essais sans les hélices réalisés, j'ai voulu passer au premier vol en utilisant le système proportionnel. Pour ne pas risquer de perdre mon drone en cas de problème du programme et pour rendre l'étape moins dangereuse, j'ai attaché les quatre pieds du drone à des ficelles elles-mêmes attachées à des sardines. En augmentant que légèrement la puissance des moteurs, le quadricoptère s'est mis à osciller rapidement. Je n'ai donc pas essayé de décoller. Le réglage de la constante n'était pas correct. Après plusieurs autres tests avec des valeurs différentes, je n'arrivais toujours pas à décoller. Il me fallait donc passer mon chemin et tenter de me tourner vers ma deuxième idée de système de régulation : le système quadratique. Après quelques tests supplémentaires, les oscillations se sont quelque peu

[12] https://www.putty.org/

atténuées à faible angle, mais sont colossales à grand angle, ce qui entraine des très grandes vitesses de moteur ce qui rendait le drone encore plus instable qu'en utilisant le système proportionnel. Le programme n'était clairement pas encore prêt à voler, il me fallait passer à un système PID. Cependant, un système PID demande beaucoup de réglages et peut être dangereux s'il est mal réglé. Il me fallait donc trouver un moyen de tester rapidement mes programmes de manière sécurisée, c'est alors que j'ai eu l'idée de concevoir un banc d'essais.

5.2.2. Banc d'essais



L'idée du banc d'essai avait l'avantage de pouvoir effectuer des tests en intérieur (donc par tous les temps), et de manière sécurisée. Je l'ai construit avec une latte en bois déjà trouée que j'ai coupée en deux morceaux. J'ai attaché le drone à la partie la plus petite avec des serre-câbles. J'ai ensuite utilisé une longue vis, des rondelles et deux boulons pour fixer les deux parties entre elles, en laissant du jeu pour que le système puisse tourner librement sans trop de frottements. Comme le drone est presque symétrique sur les axes du roulis et du tangage, le banc d'essais n'a pas besoin de pivoter sur les deux axes ; il suffit de trouver les valeurs pour un axe et de les copier sur le deuxième. Ce système ne représente pas des conditions de vol parfaite (car le point de pivot est plus bas que le centre de gravité, ce qui rend le système naturellement instable), mais il s'en approche. Ce banc d'essais permet de tester plusieurs réglages en très peu de temps car il est possible de garder l'ordinateur juste à côté. Cela m'a permis de trouver l'ordre de grandeur des constantes du système PID. Il restait donc à passer à des tests en extérieur (avec ficelles et sardines) pour affiner le réglage des axes du roulis et du tangage dans des conditions réelles.

Une fois que des valeurs convaicantes étaient atteintes sur les axes du tangage et du roulis, j'ai mis le banc d'essais sur le côté (pivoté de 90°) afin de pouvoir régler l'axe du lacet. Pour contrôler cet axe, j'ai utilisé un système de régulation PI (sans la correction dérivée, car elle n'est pas necéssaire pour ce mouvement). Il n'y avait donc que deux constantes à régler. Je ne vais pas m'étendre sur le régalge de cet axe car un réglage grossier suffit au bon fonctionnement du drone (il est même possible de voler en désactivant la commande du lacet). Je n'ai donc pas passé beaucoup de temps à le régler.

5.2.3. Premiers décollages

Après plusieurs heures de réglage sur mon banc d'essai, je suis arrivé à des réglages qui commençaient à me plaire. J'ai donc décidé de faire un test en extérieur, toujours avec ficelles et sardines comme sécurité. Et là, pour la première fois, j'ai réussi à le faire décoller de façon contrôlée. Il m'a fallu encore régler les valeurs du PID pour les adapter au vol (étant donné qu'elles étaient adaptées au banc d'essai). Après quelque temps, le réglage était meilleur et le drone se comportait mieux, mais il restait encore quelques oscillations. J'ai donc réalisé une dernière version du programme qui tournait avec une vitesse de rafraichissement « dynamique ». C'est-à-dire que la Pyboard tourne à pleine puissance et que le système PID s'adapte à sa vitesse. Mais une fois encore, il a fallu trouver un nouveau réglage. Cette version « dynamique » ne s'est avérée être que légèrement plus performante que la version « fixe », mais il y avait quand même une certaine amélioration.

6. Conclusion

6.1. Résultat

Après plusieurs séances de réglages, mon quadricoptère commençait à atteindre un stade satisfaisant. Je ne m'attendais pas moi-même à arriver à un si bon résultat, même si celui-ci n'est pas totalement parfait comparé aux programmes réalisés dans d'autres langages par des passionnés que j'avais l'habitude d'utiliser (*betaflight*[13] par exemple). Il devrait être possible, en optimisant mon programme ainsi qu'en passant plus de temps sur le réglage, d'obtenir un résultat encore meilleur. Etant plus que satisfait du résultat, j'ai rajouté une caméra et un transmetteur FPV[14] que je n'utilisais pas, afin de pouvoir réaliser des vols en caméra embarquée.

Je pense vraiment avoir atteint mon but premier : comprendre les fondements les plus basiques du fonctionnement des quadricoptères. En plus de cela, ce projet était mon premier de cette ampleur, j'ai appris sur la méthode de travail ainsi que sur l'organisation nécessaire à la réalisation de ce travail (gestion de projet).

Finalement, ce projet m'a fait découvrir un nouveau langage. J'ai tellement aimé le MicroPython que j'ai déjà réalisé plusieurs projets personnels en l'utilisant. L'autre avantage de ce projet est qu'il utilise une grande partie des fonctionnalités de la pyboard, y compris les fonctionnalités plus complexes qui ne sont pas souvent utilisées. J'ai donc beaucoup appris sur son fonctionnement.

6.2. Problèmes rencontrés

Même si la réalisation s'est plutôt bien passée dans son ensemble, elle ne s'est pas déroulée sans son lot de problèmes non plus. En voici la liste non exhaustive de problèmes et d'accros que j'ai pu rencontrer tout au long de ce travail de maturité :

[13] https://betaflight.com/

[14] L'acronyme FPV signifie First Person View (pilotage en immersion en français) désigne le fait de pouvoir piloter un véhicule via un écran connecté à une caméra embarquée.

- Trouver un moyen de positionner l'électronique au centre du quadricoptère. Le châssis n'ayant pas de point d'attache robuste, il était difficile d'attacher toutes les pièces de façon à ce qu'elles soient remplaçables en cas de casse mais aussi bien fixées pour ne pas se détacher en vol ou se mettre à vibrer. Il fallait également que le port micro USB de la Pyboard soit accessible. Le « hub » s'est avéré être une assez bonne solution.

- Choisir les branchements sur la Pyboard. Celle-ci comporte plusieurs horloges (*timers* en anglais) nécessaires à la lecture et à la sortie d'un signal. Ces compteurs ne sont cependant pas accessibles sur tous les emplacements (ou *pins* en anglais) de la carte et ils comportent des caractéristiques différentes les uns des autres (fréquences, période, résolution) et certains sont déjà utilisés par la carte elle-même. Il a donc fallu trouver une configuration fonctionnelle réunissant tous les branchements nécessaires. En lisant la documentation de la carte, je suis finalement parvenu à en trouver une.

- Une petite erreur dans mon programme ralentissait grandement celui-ci, ce qui rendait les réglages de la carte presque impossibles à trouver. Une fois que j'en ai eu conscience, j'ai trouvé avec à l'aide de Sylvain Lovis un moyen de la résoudre.

- Même si ce problème reste inévitable, le fait de devoir à chaque nouvelle modification du programme brancher la Pyboard à un ordinateur prend beaucoup de temps. Si le transfert du programme n'est pas réalisé correctement, la Pyboard peut se corrompre et tout travail réalisé jusque-là est perdu. Je n'ai heureusement jamais perdu de programme car je gardais toujours plusieurs copies sur différents ordinateurs. Également, lorsqu'il y a une erreur dans le code (ce qui arrive assez souvent), il faut reproduire l'erreur en ayant branché la carte à un ordinateur pour connaitre la nature de cette dernière.

- Les premiers ESC que j'avais commandés étaient faits pour être utilisés sur des avions. Ils n'étaient donc pas très réactifs et avaient une fréquence de rafraichissement faible. J'ai donc dû commander des exemplaires plus performants et plus adaptés à mon projet. De plus, le premier récepteur que j'avais commandé ne pouvait pas détecter lorsqu'il était déconnecté, il m'était donc impossible de garantir la sécurité, qui est indispensable quand il y a des pièces mécaniques en mouvement.

6.3. Prix

Pour ce qui en est du coût final, j'ai réalisé la liste ci-dessous regroupant toutes les pièces que j'ai utilisées, ainsi que les dépenses qu'elles ont engendrées.

Article	Prix unitaire	Déjà en possession	Nombre utilisé / acquis	Total payé pour le projet
Télécommande	~40.-	Oui	1	0.-
Batterie (drone)	15-20.-	Oui	3	0.-
Pyboard	10.04.-	Non	1	10.04.-
mpu6050	4.02.- (pack de 3)	Non	1	4.02.-
ESC (rouge)	9.84.-	Non	5	49.2.-
Récepteur (2e)	9.11.-	Non	1	9.11.-
Récepteur (1er)	5.91.-	Non	1	5.91.-
ESC (jaune)	23.30.- (pack de 6)	Non	1	23.30.-
Moteur	5.28.-	Non	5	26.4.-
Châssis	9.79.-	Non	2	19.58.-
Hélices	3.43.- (pack de 4)	Non	2	6.86.-
Caméra FPV	~10.-	Oui	1	0.-
Transmetteur FPV	~15.-	Oui	1	0.-
Ecran FPV	~50.-	Oui	1	0.-
Planches en bois (banc d'essai)	-	Oui	2	0.-
Total				**184.42.-**

Pour les pièces qui pouvaient casser, j'ai préféré prendre quelques unités supplémentaires afin d'éviter les longs temps d'attente des commandes qui sont livrées de Chine, ce qui a légèrement fait augmenter le montant total. Celui-ci s'élève à 184.-. Il prend également en compte les pièces que je n'ai finalement pas utilisés (par manque de fonctionnalités, comme cité plus haut). Il n'inclut cependant pas tous les outils nécessaires à la construction (fer à souder, étain pour soudure, pinces, chargeur pour les batteries, ...) qui étaient déjà en ma possession ainsi que d'autre petits accessoires (câbles divers, serre-câbles, gaine thermo-rétractable, velcro, connecteur batterie, …).

6.4. Temps

Je savais à l'avance que je m'engageais dans un projet conséquent qui allait engendrer un grand nombre d'heures de travail. Je n'ai pas calculé précisément le temps que j'ai passé sur ce projet, car j'ai beaucoup travaillé par petites doses ; quelques essais de programmes en rentrant le soir ou des petites recherches dans les transports publics.

J'ai également préféré ne pas me précipiter dans la réalisation du projet pour ne pas risquer de casser quelque chose. J'ai réalisé beaucoup de tests avant de passer à une étape suivante, ce qui a augmenté le nombre d'heures total.

En tout, je pense que ce travail m'a pris entre 150 et 200 heures. Mais comme le projet venait de mon initiative et me tenait beaucoup à cœur, je ne l'ai pas du tout pris comme une corvée et j'ai eu beaucoup de plaisir à le réaliser. Le temps ne m'a jamais paru long.

7. Déclaration d'authenticité

Je déclare par la présente que j'ai réalisé ce travail de manière autonome et que je n'ai utilisé aucun autre moyen que ceux indiqués dans le texte. Tous les passages inspirés ou cités d'autres auteurs-es sont dûment mentionnés comme tels. Je suis conscient que de fausses déclarations peuvent conduire le Lycée cantonal à déclarer le travail non recevable et m'exclure de ce fait à la session d'examens à laquelle je suis inscrit.

Ce travail de maturité reflète mes opinions et n'engage que moi-même, non pas le professeur responsable de mon travail ni l'expert qui m'ont accompagné dans ce travail.

Lieu et date : .... Signature : ....

8. Autorisation

Le Lycée cantonal requiert votre autorisation afin qu'un exemplaire de votre Travail de maturité soit mis à la disposition des étudiants du Lycée cantonal, par le biais de la médiathèque de l'école

- □ Oui, j'accepte que mon Travail de maturité soit mis à la disposition des étudiants du Lycée cantonal.
- □ Non, je n'accepte pas que mon Travail de maturité soit mis à la disposition des étudiants du Lycée cantonal.

9. Remerciements

J'aimerais tout d'abord commencer par remercier les créateurs des deux programmes dont je me suis inspiré, Dave Hylands (sous le pseudo *dhylands* sur Github) et *sergeLabo* (ceci est son pseudo Github, je n'ai malheureusement pas trouvé son nom) pour avoir partagé leur travail sur Github.

J'aimerais également remercier Sylvain Lovis qui m'a aidé à trouver et à résoudre l'erreur dans mon programme.

Et finalement, je souhaite remercier le créateur de MicroPython, Damien P. George, qui a créé ce langage ainsi que son travail pour la documentation de ce dernier.

10. Bibliographie

Programme de lecture du signal PWM de Dave Hylands :

https://github.com/dhylands/upy-examples/blob/master/ic_test.py (12.10.2019)

Programme de lecture de données de l'IMU :

https://github.com/sergeLabo/MicroPython-mpu-6050 (03.2020)

Documentation du langage MicroPython :

http://docs.MicroPython.org/en/latest/index.html (2020)

https://forum.MicroPython.org/ (2020)

Documentation liée à la Pyboard :

http://docs.MicroPython.org/en/latest/pyboard/quickref.html (2020)

www.st.com/resource/en/datasheet/dm00037051.pdf (2020)

Documentation de l'IMU :

https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf (2020)

Explication du fonctionnement d'un contrôleur PID :

https://pidexplained.com/pid-controller-explained/ (2020)

Explication du filtre complémentaire :

https://www.youtube.com/watch?v=j-kE0AMEWy4 (2020)

Informations concernant l'historique des premiers vols :

https://fr.wikipedia.org/wiki/Quadrirotor (2020)

Images d'une autre source que moi :

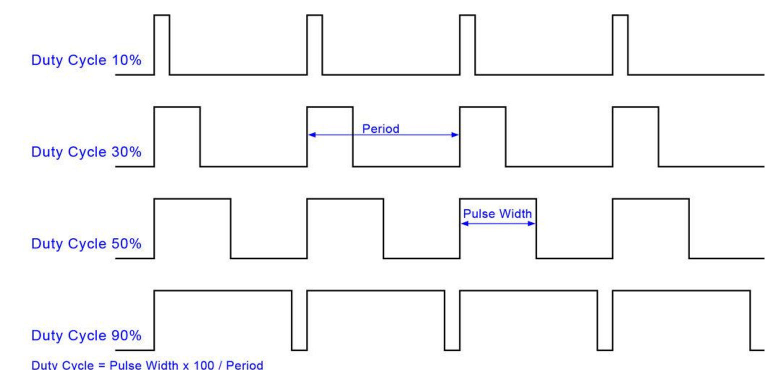
http://www.aviastar.org/helicopters_eng/bothezat.php (2020)

11. Annexes

11.1. Fonctionnement PWM

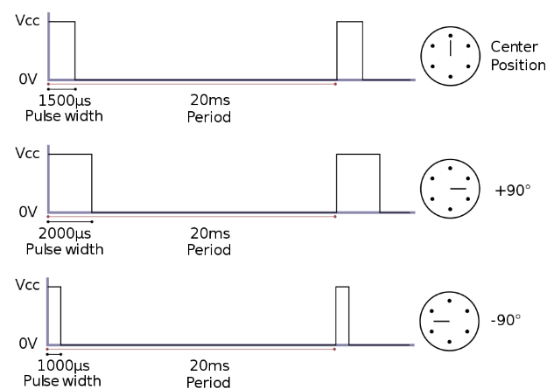
Tout d'abord, qu'est-ce qu'un signal PWM ? C'est un signal de type pseudo analogique. C'est-à-dire que pour transmettre une valeur (la vitesse des moteurs dans notre cas), le signal peut être dans deux états discrets : l'état « bas » (aucune différence de potentiel avec la borne négative) et l'état « haut » (avec différence de potentiel avec la borne négative, soit 5 volts pour la majorité des cas). Ainsi, afin de transmettre une valeur, l'état haut est maintenu pendant un certain laps de temps, et en calculant cette durée, l'on peut transmettre une valeur.

Il y a cependant plusieurs différences entre un signal PWM simple et celui utilisé dans le modélisme. La valeur d'un signal simple n'est que le pourcentage de temps passé en état haut par rapport au temps total. Cette méthode marche très bien lors de transmissions de données de manière filaire mais pose un problème lors de transmissions sans fil : le signal est incapable de faire la différence entre une valeur de 0% et une déconnexion.



[source du schéma : https://protostack.com.au/2011/06/atmega168a-pulse-width-modulation-pwm/]

Le signal PWM utilisé en modélisme se comporte donc de manière un peu différente. Premièrement, il est basé sur une fréquence de transmission fixe ce qui limite le taux de rafraichissement de l'information en laissant un délai entre deux envois. Ce temps non utilisé entre deux mises à jour de valeur vient du fait que les anciennes télécommandes « fusionnaient » les différents signaux sur un seul canal afin de pouvoir le transmettre plus facilement au récepteur. Et deuxièmement, la durée de phase haute pour un certain pourcentage est également fixée. De cette manière, il est maintenant tout à fait possible de détecter lorsqu'il y a une déconnexion entre le transmetteur et le récepteur. Une instruction de 0% se traduira par un temps en phase haute de 1000 microsecondes (1 milliseconde) et une instruction de 100% se traduira par une phase



[SOURCE DU SCHÉMA : HTTPS://EN.WIKIPEDIA.ORG/WIKI/SERVO_CONTROL]

haute de 2000 microsecondes (2 millisecondes), tandis que lors d'une perte de connexion, le signal restera en phase basse.

Il faut donc tenir compte dans notre programme de cette variation. Pour facilement tester le programme de sortie PWM, on peut utiliser des petits servomoteurs de modélisme qui ne sont pas dangereux et qui demandent la même variation du signal que les ESC.

11.2. Images et vidéos

Ce QR code renvoie vers un dossier OneDrive où se trouve toutes les images et les vidéos que j'ai pu prendre tout au long de la réalisation de ce travail de maturité.

11.3. Régulateurs

Petite précision sur une particularité des régulateurs de voltage : les ESC que j'ai choisis possèdent chacune un régulateur 5 volts. Pour des raisons complexes, on ne devrait pas relier ensemble les quatre régulateurs pour avoir un « maximum » de courant à disposition. Un régulateur est donc relié au récepteur, qui demande 5 volts, et un deuxième régulateur est connecté à la Pyboard, qui demande elle aussi 5 volts. C'est elle qui va ensuite fournir les 3.3 volts que demande l'IMU. Il restera donc quatre fils que ne seront pas connectés. Pour qu'ils ne dérangent pas, je les ai coupés à des longueurs légèrement différentes avant de mettre une gaine thermorétractable, comme le montre la photo ci-contre.

11.4. Programmes

11.4.1. main.py

```
#   3       4
#    \     /
#     \   /
#    ¦¦¦¦¦¦¦¦ -->       |--Y-->
#     /   \           X
#    /     \          |
#   1       2        \/
import machine
import pyb
import MicroPython
```

```
from pyb import Pin,Timer,LED
from machine import I2C
import mpu6050
import time



#------valeurs modifiables--------
P = 3.9     # valeurs pour le contrôleur PID de l'angle sur l'axe X et Y
I = 6.2
D = 0.78

angleMax = 40 # angle max sur l'axe X et Y (en deg)


YawP = 2 # constantes du yaw
YawI = 3

YawMax = 300 # vitesse max sur axe Z (en deg/s)


#------créations de variables--------
I1 = 0
error1_old = 0
I2 = 0
error2_old = 0
IYaw = 0

n_fail = 0

toDeg = 500 / angleMax # transpose le PWM en angle
toYaw = 500 / YawMax # transpose le PWM en vitesse angulaire pour yaw

led = LED(4)
led2 = LED(3)
led3 = LED(2)

pyb.delay(200)

#-------output moteurs (1-4)------
tim4 = Timer(4, prescaler=83, period=2499) #50Hz -> 19999 # 400Hz -> 2499

pin1 = pyb.Pin.board.X9
pwm1 = tim4.channel(1, pyb.Timer.PWM, pin=pin1)

pin2 = pyb.Pin.board.X10
pwm2 = tim4.channel(2, pyb.Timer.PWM, pin=pin2)

pin3 = pyb.Pin.board.Y3
pwm3 = tim4.channel(3, pyb.Timer.PWM, pin=pin3)
```

```
pin4 = pyb.Pin.board.Y4
pwm4 = tim4.channel(4, pyb.Timer.PWM, pin=pin4)

pwm1.pulse_width(1001)
pwm2.pulse_width(1001)
pwm3.pulse_width(1001)
pwm4.pulse_width(1001)

pyb.delay(1200)

#-------init accel--------
led3.on() # allumer la led verte pendant la calibration
a = 1
while a:
    try:
        # Set pin: scl on Y9 and sda on Y10
        i2c = I2C(scl='Y9', sda='Y10', freq=400000)
        mpu_6050 = mpu6050.MPU6050(i2c)
        a = 0

    except:
        print("bad init")
led3.off()

#-------input reciver---------
t5 = pyb.Timer(5, prescaler=83, period=0x0fffffff)
t2 = pyb.Timer(2, prescaler=83, period=0x0fffffff) #timer a 1Mhz (84 Mhz / 83+
1)

p1 = pyb.Pin.board.X1
p3 = pyb.Pin.board.X3
p2 = pyb.Pin.board.X2
p4 = pyb.Pin.board.X4
p5 = pyb.Pin.board.X6

ic1 = t5.channel(1, pyb.Timer.IC, pin=p1, polarity=pyb.Timer.BOTH)
ic2 = t5.channel(2, pyb.Timer.IC, pin=p2, polarity=pyb.Timer.BOTH)
ic3 = t5.channel(3, pyb.Timer.IC, pin=p3, polarity=pyb.Timer.BOTH)
ic4 = t5.channel(4, pyb.Timer.IC, pin=p4, polarity=pyb.Timer.BOTH)
ic5 = t2.channel(1, pyb.Timer.IC, pin=p5, polarity=pyb.Timer.BOTH)

ic_start1 = 0
ic_start2 = 0
ic_start3 = 0
ic_start4 = 0
ic_start5 = 0
chan1 = 0
chan2 = 0
chan3 = 0
chan4 = 0
chan5 = 0
```

```
def channel1(tim):
    global ic_start1, chan1
    if p1.value():
        ic_start1 = ic1.capture() # regarde si le signal est montant
    else:
        chan1 = ic1.capture() - ic_start1 & 0x0fffffff # sinon il est descenda
nt -> calculer la dif entre les 2 moments

def channel2(tim):
    global ic_start2, chan2
    if p2.value():
        ic_start2 = ic2.capture()
    else:
        chan2 = ic2.capture() - ic_start2 & 0x0fffffff

def channel3(tim):
    global ic_start3, chan3
    if p3.value():
        ic_start3 = ic3.capture()
    else:
        chan3 = ic3.capture() - ic_start3 & 0x0fffffff

def channel4(tim):
    global ic_start4, chan4
    if p4.value():
        ic_start4 = ic4.capture()
    else:
        chan4 = ic4.capture() - ic_start4 & 0x0fffffff

def channel5(tim):
    global ic_start5, chan5
    if p5.value():
        ic_start5 = ic5.capture()
    else:
        chan5 = ic5.capture() - ic_start5 & 0x0fffffff


ic1.callback(channel1) # demande aux fonctions de s'activer s’il y a un change
ment dans le signal (mode callback)
ic2.callback(channel2)
ic3.callback(channel3)
ic4.callback(channel4)
ic5.callback(channel5)

pyb.delay(200)

#-------------début de la boucle principale---------------
while True:

    #------lecture mpu6050------
```

```
mpu_6050.lecture_capteurs()

#-----lecture recepteur-----
if chan1 < 2100 and chan1 > 900:
    ch1 = chan1 - 1500 # pitch
if chan2 < 2100 and chan2 > 900:
    ch2 = chan2 - 1500 # roll
if chan3 < 2100 and chan3 > 900:
    ch3 = chan3 - 1000 # puissance
if chan4 < 2100 and chan4 > 900:
    ch4 = chan4 - 1500 # yaw

ch5 = chan5 # arming

if ch5 < 1200 or ch5 > 2100 :   #1000=disarm  2000=arm
    n_fail += 1
    led2.off()
else :
    n_fail =0
    led2.on() #led orange => en marche, attention !


#-------calcule des vitesses des moteurs-------

moteur = [ch3]*4 # vitesse moyennne des moteurs = postiton channel 3

#------ch1------(roll)
error1 = ch1 / toDeg - mpu_6050.AngleX

P1 = error1 * P
I1 = I1 + error1 * I * mpu_6050.intervalle
D1 = (error1 - error1_old) * D / mpu_6050.intervalle

correction1 = P1 + I1 + D1
if correction1 > 400:
    correction1 = 400
elif correction1 < -400:
    correction1 = -400

moteur[0] -= correction1
moteur[1] -= correction1
moteur[2] += correction1
moteur[3] += correction1

error1_old = error1


#------ch2------(pitch)
error2 = ch2 / toDeg - mpu_6050.AngleY

P2 = error2*P
```

```
    I2 = I2 + error2 * I * mpu_6050.intervalle
    D2 =(error2 - error2_old) * D / mpu_6050.intervalle

    correction2 = P2 + I2 + D2
    if correction2 > 400:
        correction2 = 400
    elif correction2 < -400:
        correction2 = -400

    moteur[0] += correction2
    moteur[1] -= correction2
    moteur[2] += correction2
    moteur[3] -= correction2

    error2_old = error2

    #------ch4------(yaw)
    errorYaw = ch4 / toYaw - mpu_6050.gyroZ_Vangulaire

    PYaw = errorYaw * YawP
    IYaw = IYaw + errorYaw * YawI * mpu_6050.intervalle

    correctionYaw = PYaw + IYaw
    if correctionYaw > 400:
        correctionYaw = 400
    elif correctionYaw < -400:
        correctionYaw = -400


    moteur[0] -= correctionYaw
    moteur[1] += correctionYaw
    moteur[2] += correctionYaw
    moteur[3] -= correctionYaw

    #------ch5------(failsafe)
    led.off()

    if n_fail > 8:  # disarm ou en cas de failsafe -> arrêter les moteurs
        moteur= [-9999] * 4
        led.on()   # allumer la led bleu = pas de danger
        I1 = 0   # éviter les acumulations d'erreurs -
> moteur à 100% au redémarage
        I2 = 0
        IYaw = 0
        error1_old = 0
        error2_old = 0


    #------calcule output PWM-------
    for i in range (4):
        if moteur[i] == -9999:
```

```
            moteur[i] = 1    # stop les moteurs
        elif moteur[i] > 1000:
            moteur[i] = 1000
        elif moteur[i] < 1:
            moteur[i] = 1
        moteur[i] = int(moteur[i])
        moteur[i] += 1000

    #-------output moteurs(1-4)------
    pwm1.pulse_width(moteur[0])
    pwm2.pulse_width(moteur[1])
    pwm3.pulse_width(moteur[2])
    pwm4.pulse_width(moteur[3])
```

11.4.2. mpu6050.py

```
"""
Adaptation de
itechnofrance
"""
FC = 0.005

import time, math

MPU6050_ADDRESS_AD0_LOW   = 0x68
MPU6050_ADDRESS_AD0_HIGH  = 0x69
MPU6050_REG_POWER_MGMT_1  = 0x6B  # reset
MPU6050_REG_POWER_MGMT_2  = 0x6C  # activate sensors
MPU6050_REG_READ_SENSORS  = 0x3B  # read 14 bytes of sensors
MPU6050_REG_CONFIG        = 0x1A  # configuration du filtre passe bas
MPU6050_REG_SMPRT_DIV     = 0x19  # definition du taux d'echantillonage
MPU6050_REG_CONFIG_GYRO   = 0x1B  # configure gyroscope
MPU6050_REG_CONFIG_ACC    = 0x1C  # configure accelerometer

class MPU6050():

    def __init__(self, i2c):
        self.capteurs = bytearray(14)  # read 14 register
        self.AngleX = 0
        self.AngleY = 0
        self.AngleZ = 0
        self.i2c = i2c
        if self.detect():
           self.reset()
           self.config()
           self.calibration()
           self.temps = time.ticks_us()
```

```
    def detect(self):
        detect_mpu6050 = False
        i2c_peripheriques = self.i2c.scan()
        for i2c_peripherique in i2c_peripheriques:
            if (i2c_peripherique == MPU6050_ADDRESS_AD0_LOW):
                self.adresse = MPU6050_ADDRESS_AD0_LOW
                detect_mpu6050 = True
            if (i2c_peripherique == MPU6050_ADDRESS_AD0_HIGH):
                self.adresse = MPU6050_ADDRESS_AD0_HIGH
                detect_mpu6050 = True
        return detect_mpu6050

    def reset(self):
        self.data = bytearray(2)
        self.data[0] = MPU6050_REG_POWER_MGMT_1
        self.data[1] = 1 << 7  # bit 7 a 1 : reset mpu6050
        self.i2c.writeto(self.adresse, self.data)
        time.sleep(0.100)  # delai de 100ms

    def config(self):
        self.data = bytearray(2)
        self.data[0] = MPU6050_REG_POWER_MGMT_1
        self.data[1] = 1  # desactive le mode sleep et source horloge = X Gyro
        self.i2c.writeto(self.adresse, self.data)
        self.data[0] = MPU6050_REG_POWER_MGMT_2
        self.data[1] = 0  # active tous les capteurs
        self.i2c.writeto(self.adresse, self.data)
        self.data[0] = MPU6050_REG_CONFIG
        self.data[1] = 3  # active le filtre passe bas acc=184Hz gyro=188Hz
        self.i2c.writeto(self.adresse, self.data)
        #self.data[0] = MPU6050_REG_SMPRT_DIV
        #self.data[1] = 32 # taux d'echantillonage environ 1000Hz / 32 = 31Hz
= 32ms
        #self.i2c.writeto(self.adresse, self.data)
        self.data[0] = MPU6050_REG_CONFIG_GYRO
        self.data[1] = 1  # configuration plage de mesure gyroscope +/- 500 de
g/seconde
        self.i2c.writeto(self.adresse, self.data)
        self.data[0] = MPU6050_REG_CONFIG_ACC
        self.data[1] = 1  # configuration plage de mesure accelometre +/- 4g
        self.i2c.writeto(self.adresse, self.data)
       # print('coufig')

    def lecture_capteurs(self):
        self.i2c.readfrom_mem_into(self.adresse, MPU6050_REG_READ_SENSORS, sel
f.capteurs)
        self.acc()
        self.gyro()
        self.angle()
```

```
    def acc(self):
        # octet haut pour l'axe X de l'accelerometre
        self.accX_high_byte = self.capteurs[0]
        # octet bas pour l'axe X de l'accelerometre
        self.accX_low_byte = self.capteurs[1]
        self.accX = self.bytes_to_int(self.accX_high_byte, self.accX_low_byte)
        self.accX_calibre = self.accX #- self.accX_calibration
        # octet haut pour l'axe Y de l'accelerometre
        self.accY_high_byte = self.capteurs[2]
        # octet bas pour l'axe Y de l'accelerometre
        self.accY_low_byte = self.capteurs[3]
        self.accY = self.bytes_to_int(self.accY_high_byte, self.accY_low_byte)
        self.accY_calibre = self.accY #- self.accY_calibration
        # octet haut pour l'axe Z de l'accelerometre
        self.accZ_high_byte = self.capteurs[4]
        # octet bas pour l'axe Z de l'accelerometre
        self.accZ_low_byte = self.capteurs[5]
        self.accZ = self.bytes_to_int(self.accZ_high_byte, self.accZ_low_byte)
        self.accZ_calibre = self.accZ #- self.accZ_calibration

    def gyro(self):
        self.gyroX_high_byte = self.capteurs[8] # octet haut pour l'axe X du g
yro
        self.gyroX_low_byte = self.capteurs[9] # octet bas pour l'axe X du gyr
o
        self.gyroX = self.bytes_to_int(self.gyroX_high_byte, self.gyroX_low_by
te)
        self.gyroX_calibre = self.gyroX - self.gyroX_calibration
        self.gyroY_high_byte = self.capteurs[10] # octet haut pour l'axe Y du
gyro
        self.gyroY_low_byte = self.capteurs[11] # octet bas pour l'axe Y du gy
ro
        self.gyroY = self.bytes_to_int(self.gyroY_high_byte, self.gyroY_low_by
te)
        self.gyroY_calibre = self.gyroY - self.gyroY_calibration
        self.gyroZ_high_byte = self.capteurs[12] # octet haut pour l'axe Z du
gyro
        self.gyroZ_low_byte = self.capteurs[13] # octet bas pour l'axe Z du gy
ro
        self.gyroZ = self.bytes_to_int(self.gyroZ_high_byte, self.gyroZ_low_by
te)
        self.gyroZ_calibre = self.gyroZ - self.gyroZ_calibration

    def bytes_to_int(self, firstbyte, secondbyte):
        if not firstbyte & 0x80:
            return (firstbyte << 8 | secondbyte)
        return - (((firstbyte ^ 255) << 8) | (secondbyte ^ 255) + 1)  # comple
ment a 2

    def calibration(self):
        i = 0
```

```
        self.gyroX_calibration = 0
        self.gyroY_calibration = 0
        self.gyroZ_calibration = 0
        while i < 400:
            self.i2c.readfrom_mem_into(self.adresse, MPU6050_REG_READ_SENSORS,
 self.capteurs)
            self.gyro()
            self.gyroX_calibration += self.gyroX
            self.gyroY_calibration += self.gyroY
            self.gyroZ_calibration += self.gyroZ
            i += 1
            time.sleep(0.01)
        self.gyroX_calibration /= 400
        self.gyroY_calibration /= 400
        self.gyroZ_calibration /= 400

    def angle(self):
        # Calcul en utilisant un filtre complementaire
        # Pour l'utilisation de l'axe Z il est nécessaire d'y adjoindre un mag
netometre
        # si on veut des mesures utilisables
        global AngleX,AngleY

         # intervalle de temps
        self.temps_precedent = self.temps
        self.temps = time.ticks_us()
        self.intervalle = time.ticks_diff(self.temps, self.temps_precedent) /
1000000 # mettre en sec

         # accel
        self.aX = self.accX_calibre / 8192  # 8192 pour le choix plage de mesu
re accelerometre +/- 4g
        self.aY = self.accY_calibre / 8192  # 32767 / 4
        self.aZ = self.accZ_calibre / 8192
        self.accX_angle=0
        self.accY_angle=0
        self.accZ_angle=0
        if self.aX!=0 and self.aZ!=0:
            self.accX_angle = math.degrees (math.atan(self.aY / math.sqrt((sel
f.aX * self.aX) + (self.aZ * self.aZ))))
        if self.aY!=0 and self.aZ!=0:
            self.accY_angle = math.degrees (math.atan(-
1 * self.aX / math.sqrt((self.aY * self.aY) + (self.aZ * self.aZ))))
        #if self.aZ!=0:
        #    self.accZ_angle = math.degrees (math.atan(math.sqrt((self.aX * se
lf.aX) + (self.aY * self.aY)) / self.aZ ))

         # gyro
        self.gyroX_angle = self.gyroX_calibre / 65.5  # 65.5 pour le choix de
mesure +/- 500 deg/s
        self.gyroY_angle = self.gyroY_calibre / 65.5  # 32767 / 500
```

```
        self.gyroZ_Vangulaire = self.gyroZ_calibre / -65.5

         # filtre complementaire
        self.AngleX = (1 - FC) * (self.AngleX + self.gyroX_angle * self.interv
alle) + FC * self.accX_angle
        self.AngleY = (1 - FC) * (self.AngleY + self.gyroY_angle * self.interv
alle) + FC * self.accY_angle
        #self.AngleZ = (1 - FC) * (self.AngleZ + self.gyroZ_angle * self.inter
valle) + FC * self.accZ_angle

        return self.AngleX, self.AngleY, self.gyroZ_Vangulaire, self.intervall
e
```