Travail de maturité

R²Quad :

Figure 1

Conception, fabrication et programmation d'un contrôleur de vol de quadricoptère.

Rafael Riber, Raphaël Flückiger
CECG Madame De Staël – 30.10.17

Table des matières

Introduction

C'est un intérêt commun pour la programmation, l'électronique et les quadricoptères[1] qui nous a donné envie de tenter de fabriquer le nôtre. Un quadricoptère est un véhicule aérien avec quatre rotors fixés à l'extrémité des quatre bras de l'engin. Il possède la capacité de décoller et se poser verticalement. La propulsion de l'engin se fait par la rotation des hélices fixées aux moteurs. Une des particularités des quadricoptères est qu'ils peuvent rester en vol stationnaire sans appel au conducteur, c'est-à-dire qu'ils se stabilisent seuls, sans l'aide continue d'un pilote.

Les mouvements se font par une modulation de la vitesse de rotation des moteurs qui produit un changement de son inclinaison selon les trois axes de l'espace. C'est le contrôleur de vol situé à bord du quadricoptère qui reçoit les signaux du pilote, et effectue les calculs nécessaires pour déterminer à quelle vitesse chaque moteur doit tourner pour suivre la consigne du pilote au mieux.

On voit depuis peu ces engins apparaître sur le marché de consommateurs, alors qu'ils n'étaient réservés jusqu'à récemment qu'aux applications professionnelles ou pour les services de sauvetage. La principale utilisation civile des quadricoptères est la photographie aérienne, qui a été démocratisée par la sortie sur le marché de solutions complètes de qualité professionnelle par des jeunes entreprises déjà multinationales comme DJI, leader des quadricoptères pour l'imagerie aérienne. Il existe cependant de nombreuses autres utilisations des quadricoptères, comme des applications de protection civile et le contrôle d'infrastructures comme des barrages, des éoliennes ou encore des centrales nucléaires, ou encore des applications de loisir comme des courses.

Le but principal de ce travail sera donc de concevoir un contrôleur de vol basé sur la plateforme libre Arduino, qui permet une programmation simplifiée de microcontrôleurs, mettant ainsi à la portée de tous, la programmation en C [2] de systèmes embarqués. Le contrôleur de vol se présentera sous la forme d'un circuit imprimé que nous allons concevoir, sur lequel se brancheront les moteurs du quadricoptère ainsi qu'un récepteur radio pour un contrôle radiocommandé, et qui comprendra le microcontrôleur que nous programmerons, ainsi qu'un circuit intégré permettant la mesure de plusieurs facteurs de position (vitesse angulaire, angle par rapport à l'horizontale, position absolue) qui nous seront utiles dans les calculs effectués par le microcontrôleur.

Petite histoire des quadricoptères

Le premier prototype d'aéronef ressemblant aux quadricoptères d'aujourd'hui est le gyroplane Breguet-Riche, développé en 1907 par Breguet Aviation. Basé sur l'hélicoptère, le gyroplane Breguet-Riche n'était toutefois pas capable de voler sans le soutien de cordes et de plusieurs personnes. Le premier quadricoptère capable de voler (plus de 5 mètres) était le Curtiss-Wright VZ-7, développé pour l'armée américaine en 1958, mais qui n'a pas pu répondre aux besoins de l'armée, et a donc été abandonné. Les développements les plus importants dans le domaine des quadricoptères ont eu lieu seulement dans les deux dernières décennies.

[1] De l'anglais « quadcopter ».
[2] Langage de programmation.

Principe général du fonctionnement d'un quadricoptère

I – Schéma des composants :

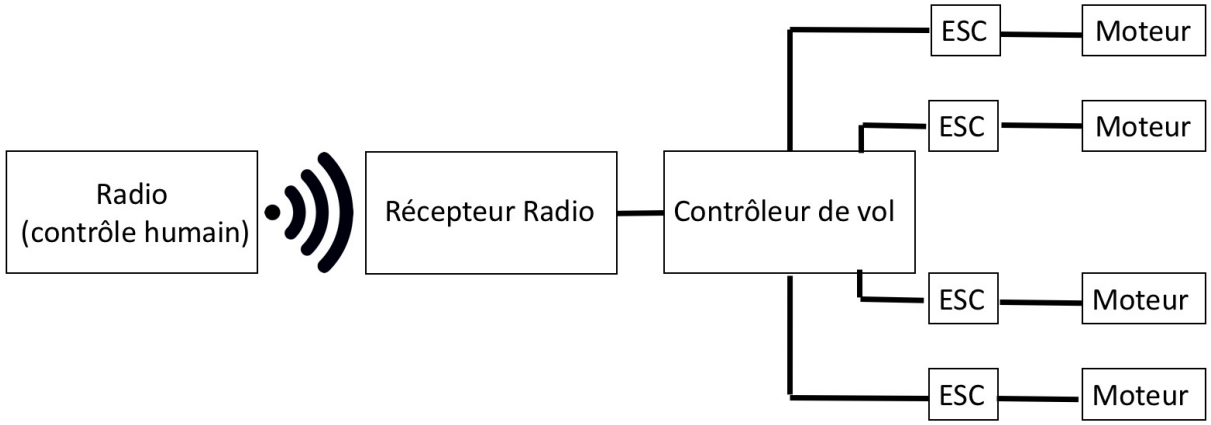

Figure 2

Comme évoqué dans l'introduction le but du contrôleur de vol est de lire les données du récepteur radio et de faire les calculs nécessaires pour écrire les signaux aux ESC[3], qui eux-mêmes calculent comment tourner le moteur à la vitesse demandée par le contrôleur. Chaque composant de ce système mérite une explication détaillée car nous utiliserons beaucoup ces principes de fonctionnement dans notre code. Il est tout de même utile de remarquer que nous n'avons que recherché des informations sur ces sujets et nous n'avons construit aucun composant du système excepté le contrôleur de vol. Ces éléments sont faciles à trouver en magasin et utilisent généralement le même principe de fonctionnement qui ne dépend pas du fabriquant.

[3] De l'anglais « Electronic Speed Controller », contrôleur électronique de vitesse en français.

II – Principe de fonctionnement des moteurs :

Les moteurs utilisés dans les quadricoptères sont pour la plupart des moteurs sans balais [4] à courant continu. Ceux-ci ont un rotor[5] constitué d'aimants permanents, et un stator[6] composé de 3 bobines A, B et C (Figure 3).

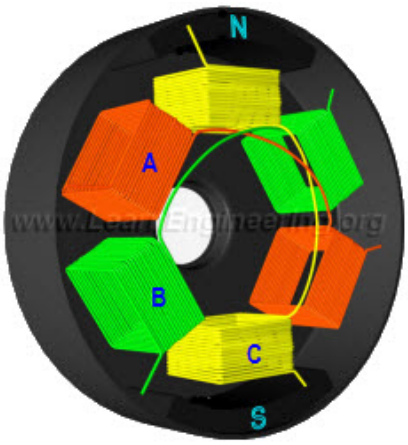

Figure 3

Comme tous les moteurs électriques, son principe de fonctionnement se base sur l'induction magnétique[7]. Lorsqu'un courant traverse un solénoïde (ici les bobines) cela crée un champ magnétique perpendiculaire au sens du courant, celui-ci est en rotation autour du sens du courant et directionnel dans une bobine. Et c'est ce champ magnétique engendré qui va interagir avec les aimants permanents du rotor (deux pôles magnétiques de polarité opposées s'attirent et deux pôles magnétiques de même polarité se repoussent), afin de faire tourner le rotor :

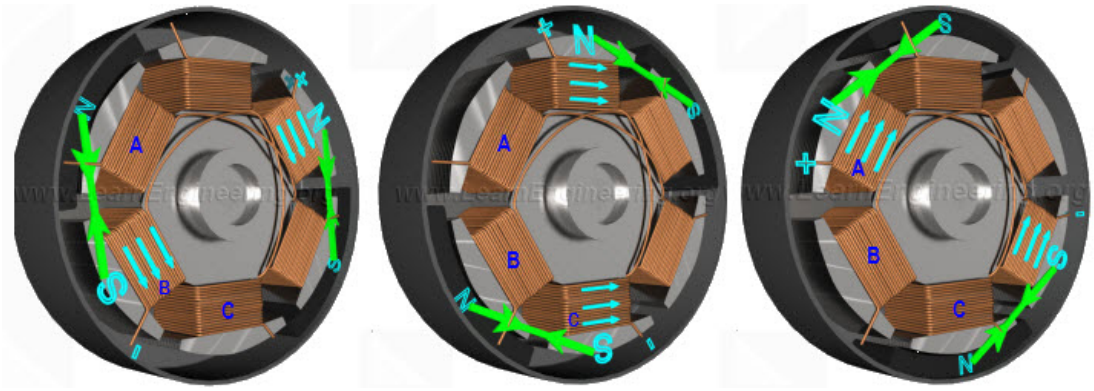

Figure 4

[4] Moteur sans balais : moteur à courant continu n'ayant aucun contact entre le rotor et le stator excepté l'axe de rotation. Ils ont généralement une durée de vie plus grande que les moteurs à balais.
[5] Rotor : Partie en mouvement du moteur. Ici elle contient les aimants Néodymes.
[6] Stator : Partie à l'arrêt du moteur. Ici elle comprend les bobines du moteur ainsi que le senseur Hall.
[7] Induction électromagnétique : Production d'un champ magnétique par un courant passant par un conducteur (généralement un solénoïde).

Il faut par conséquent appliquer un voltage en phase aux différentes bobines pour que le moteur puisse tourner. Il faut donc activer la bobine la plus proche de l'aimant et la désactiver au moment où elle arrive à sa position. De plus, pour un plus grand couple on utilise le fait que deux pôles magnétiques de même polarité se repoussent. Ces inversions de courant se font très souvent, si souvent qu'un microcontrôleur par moteur est nécessaire (les ESC[8]).
Un tour complet se traduirait donc par l'application d'un courant dans les trois pôles A, B et C dans l'ordre illustré ci-contre (Figure 5).

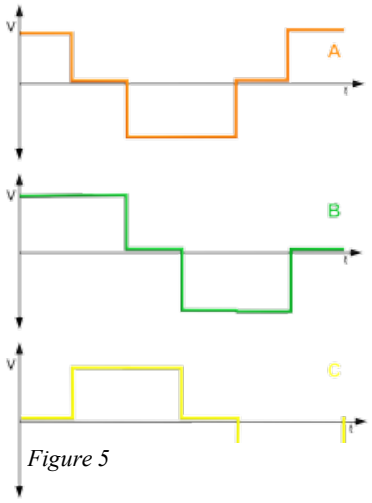

Figure 5

III – Principe de fonctionnement des contrôleurs de vitesse :

Un contrôleur de vitesse, ou ESC [9] a pour fonction de changer la vitesse, la direction du moteur ou même d'agir comme un frein pour celui-ci. Il calcule à quel moment chaque bobine doit être énergisée, afin que le moteur tourne à la vitesse souhaitée. Pour ce faire, il utilise un senseur a effet Hall[10], avec lequel il détermine la vitesse de rotation du rotor ce qui lui permet de savoir à quelle bobine il doit appliquer un courant. Généralement, on trouve à l'intérieur une puce comme celle utilisée dans le contrôleur de vol pour effectuer des calculs rapides. Il est composé d'une part de deux câbles pour recevoir la tension de la batterie et d'un câble qui sert à transmettre le signal PWM[11] de la vitesse de rotation du moteur, et d'autre part les trois câbles de phase qui sont connectés aux moteurs comme vu précédemment (passant le courant aux bobines A, B et C). Dépendant du moteur et de l'ESC la détection de la vitesse de rotation du moteur est faite par un capteur optique, ou dans la plupart des cas un senseur « back EMF[12]»

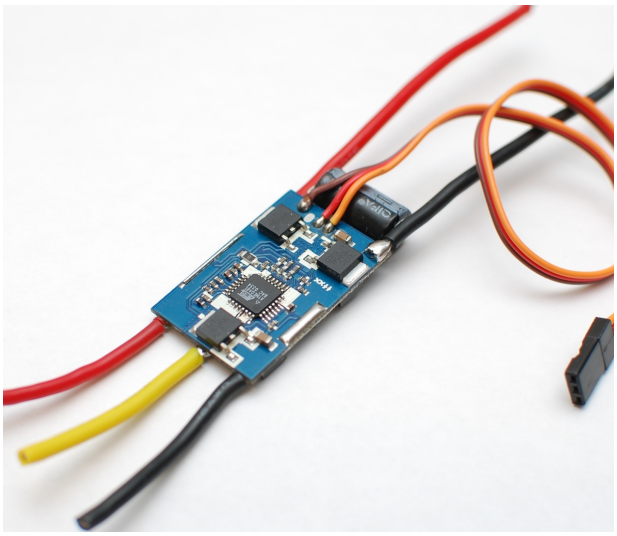
Figure 6

[8] Voir note p. 4
[9] Voir note p. 4
[10] Mesure les champs magnétiques.
[11] De l'anglais « Pulse Width Modulation », modulation de largeur d'impulsion en français.
[12] Force électromagnétique de retour. Permet, selon le voltage venant du moteur, de savoir la vitesse de ce dernier.

IV – Dynamique de vol du quadricoptère :

Le quadricoptère est un engin volant différent des autres et son fonctionnement est assez simple et direct.
Commençons par poser les axes avec lesquels on travaillera :

Généralement les axes présents sur la figure 7 sont appelés Tangage (Pitch), Roulis (Roll) et Lacet (Yaw). De plus, les moteurs sont alternés entre sens horaire et antihoraire, nous expliquerons pourquoi.

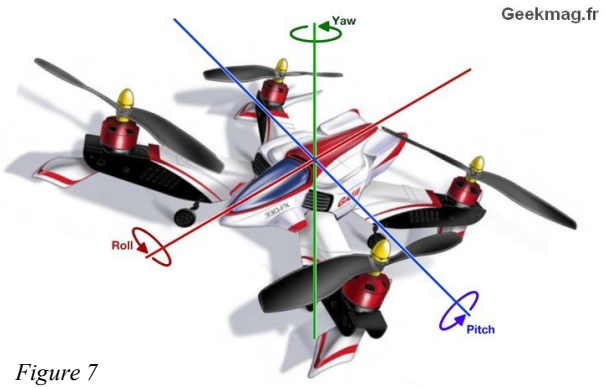

Figure 7

Le quadricoptère fera des manœuvres autour de ses axes lorsqu'on le lui demande. Regardons un exemple de vol pour vous familiariser avec les mécaniques de vol :

Au départ le quadricoptère part à plat sur le sol, ses moteurs ne tournent pas tant qu'on ne leur dit pas de le faire (Figure 8).

Nous indiqueront les moteurs qui tournent plus vite que les autres par des plus et ceux qui tournent moins vite par des moins.

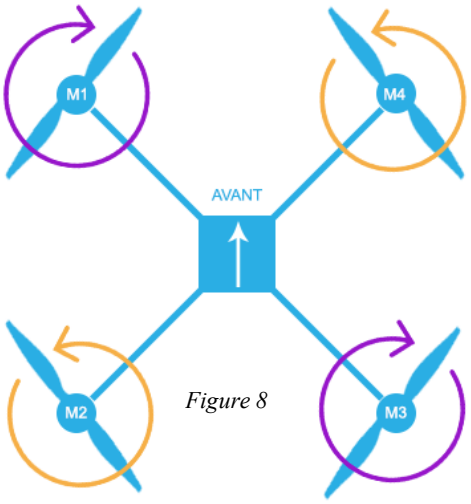

Figure 8

1.Décolage : On active les moteurs tous en même temps et on augmente leur vitesse de rotation ce qui a pour effet de lever le quadricoptère perpendiculairement au sol (le quadricoptère est maintenant stable sur tous les axes) (Figure 9).

Bien sûr, l'aéronef subit des perturbations car les moteurs ne tournent pas exactement à la même vitesse et des facteurs tels que le vent influencent sa position dans l'espace. Notre but est de corriger ces erreurs en contrôlant les moteurs.

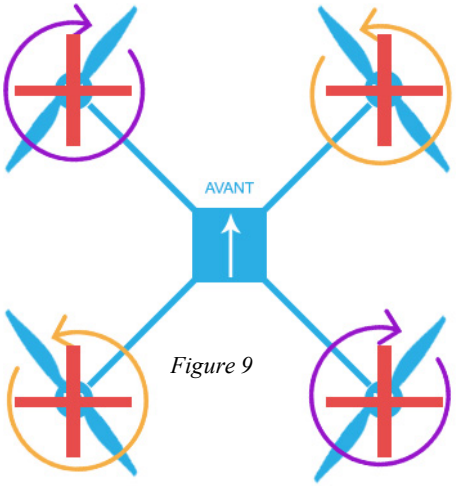

Figure 9

2.Pitch : Pour ce faire, nous allons augmenter la vitesse de rotation des moteurs arrière et diminuer la puissance des moteurs avant (Figure 11). Cela inclinera l'engin dans l'axe pitch (il plongera du nez). Cependant nous ne devons faire ça que jusqu'à ce qu'il soit dans l'angle désiré. Il faut rétablir la position d'équilibre (Figure 10), sinon il continuera d'augmenter son angle autour de pitch (et fera des cabrioles non voulues) :

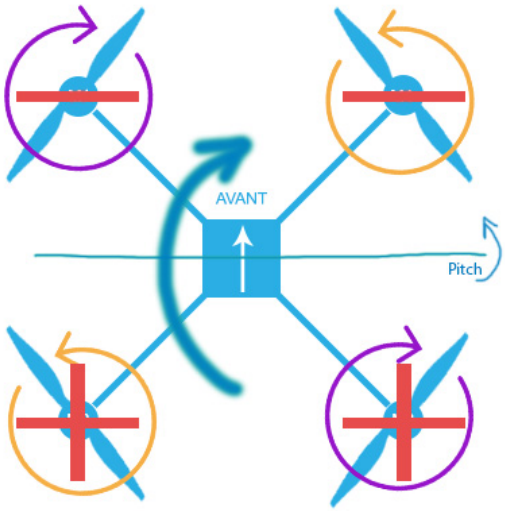

Figure 11

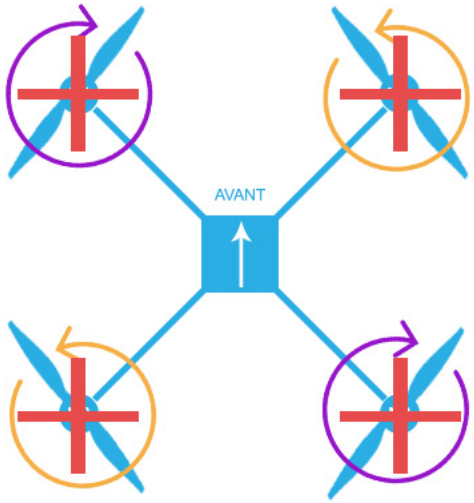

Figure 10

Le quadricoptère est dans une position inclinée (il avance) et le restera tant qu'on ne lui dit pas de faire l'opération inverse. Pour reculer il s'agit de la même opération mais en inversant moteurs plus forts avec moteurs plus faibles.

3.Roll : Pour cela on fonctionne de la même manière que pour le pitch mais sur l'axe roll. C'est-à-dire que l'on va augmenter la puissance des deux moteurs à gauche et diminuer celle des moteurs de droite et ensuite les replacer avec tous les moteurs à la même puissance (Figure 13).

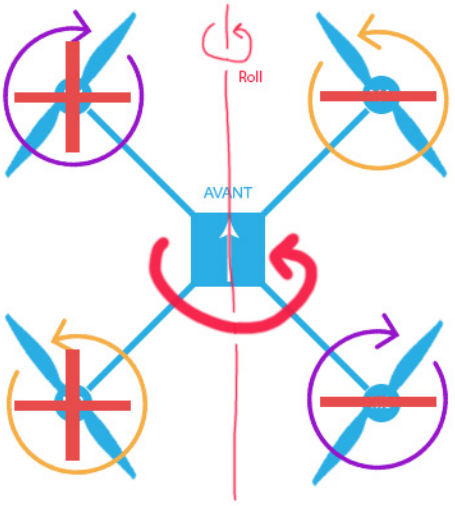

Figure 13

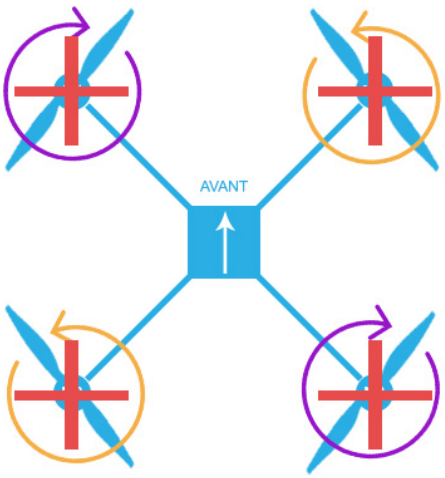

Figure 12

De la même manière que le « pitch », il ira à droite tant qu'on ne lui donne pas l'instruction inverse.

4.Yaw : Ceci est un peu diffèrent car il faut s'intéresser à comment les hélices font que le quadricoptère vole. En effet, une hélice, lorsqu'elle tourne, engendre une force dans l'axe perpendiculaire au plan décrit par l'hélice, mais subit en plus une force résistante causée par le déplacement des pales dans l'air (troisième loi de Newton). Par conséquent une hélice qui tourne dans le sens horaire aura forcément une force opposée à sa rotation c'est-à-dire que le système tournera dans la direction inverse. Donc pour que notre quadricoptère tourne selon l'axe « yaw » il doit augmenter deux moteurs qui tournent dans la direction opposée à celle où l'on veut aller (il est utile de rappeler que toutes les hélices poussent l'air vers le bas et donc que deux d'entre elles sont symétriques aux deux autres) (Figure 15).
Ici on tourne dans le sens horaire, il faudra donc augmenter la vitesse de rotation des moteurs qui tournent dans le sens antihoraire.

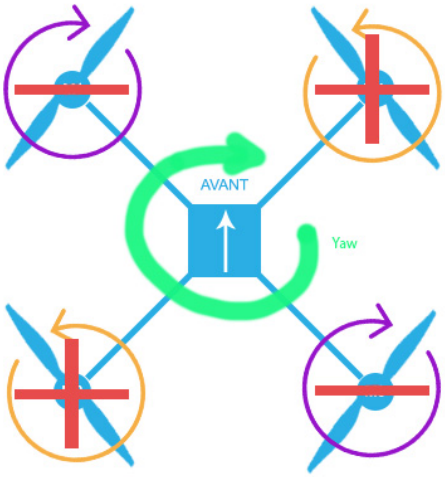

Figure 15

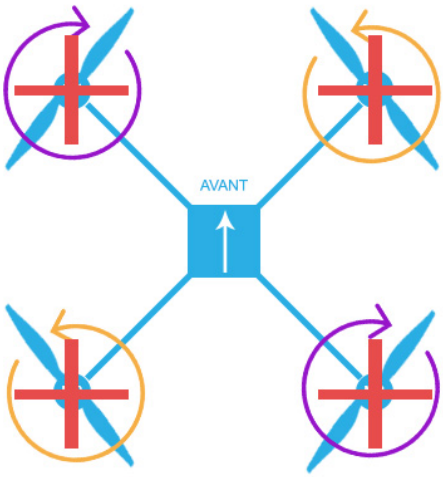

Figure 14

Il faudra ensuite de la même manière pour arrêter la rotation revenir à la vitesse de rotation initiale.

V – Principe de fonctionnement du contrôleur de vol :

Le contrôleur de vol est le « cerveau » du quadricoptère. Son rôle consiste à recevoir les commandes de l'opérateur, et contrôler la vitesse de chaque moteur afin d'assurer une réponse rapide et précise. Pour ce faire, un régulateur PID logiciel est généralement utilisé.

Figure 16

Contrôle radio :

La plupart des quadricoptères sont contrôlés via une radio de modélisme standard, transmettant généralement à une fréquence de 2.4 GHz. Tout système radio comprend un transmetteur, aussi appelé radio (Figure 16), qui est utilisé au sol par l'opérateur, et un récepteur (Figure 17), qui est monté sur l'engin contrôlé, ici un quadricoptère.

Figure 17

Ce système permet la transmission de contrôles sur au minimum quatre canaux ; un pour chaque axe des deux manches à balai de la radio, et un pour chaque interrupteur de la radio. Chaque canal est transmis par radio en utilisant un signal PWM[13] qui traduit la position de chaque axe des manches à balai en une onde carrée, donc binaire, et dont la durée du signal « haut » en microsecondes (entre 1000 et 2000) donne la position de chaque axe du manche à balai, centrés à 1500 microsecondes. Le signal PWM est une onde carrée, dite « haute », ou « high », quand à 3.3V, et « basse » ou « low », quand à 0V. Une illustration de ce signal se trouve en annexe. Sur le récepteur, une sortie pour chaque canal permet de brancher le récepteur au contrôleur de vol, qui reçoit ainsi tous les canaux sur des connexions séparées, et peut ainsi les utiliser dans ses calculs.

La configuration traditionnelle des manches à balai est la suivante (Figure 18) :

- Les gaz (Throttle) sur le manche gauche, axe vertical,
- Le lacet (Yaw), sur le manche gauche, axe horizontal,
- Le tangage (Pitch) sur le manche droit, axe vertical
- Le roulis (Roll) sur le manche droit, axe horizontal

Cette configuration est appelée « Mode 2 », les autres modes changent la configuration des axes.

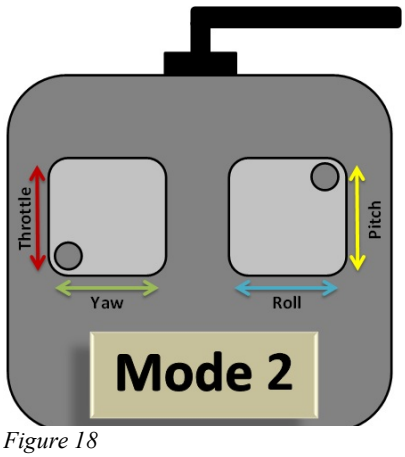

Figure 18

[13] Voir note p. 6

Gyroscope :

En plus du microcontrôleur principal, tout contrôleur de vol possède un gyroscope sous forme de circuit intégré. Ce dernier est connecté au microcontrôleur principal, et lui transmet la vitesse angulaire (« Gyro » sur la figure 19) qu'il mesure, généralement sur trois axes (x, y, z). La puce que nous utilisons fait aussi accéléromètre (« Accel » sur la figure 19) et mesure les champs magnétiques, comme une boussole (« Magnet » sur la figure 19). Toutefois, dans notre application nous n'utilisons pas la boussole. Il est possible de l'utiliser pour bloquer le quadricoptère sur l'axe de lacet.

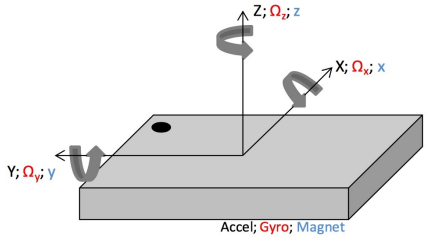

Figure 19

Les mesures données par la partie gyroscope sont en radians par seconde. Nous les convertissons donc en degrés par seconde en les multipliant par $\frac{180}{\pi}$. Pour le calcul des angles d'Euler, la puce combine l'accéléromètre et le gyroscope, qui subissent ensuite une transformation nécessaire pour avoir des angles justes en toutes circonstances. La transformation est nécessaire car, par leur nature, ces angles de rotation se font par rapport à un référentiel à l'arrêt, tandis que dans cette application nous avons besoin d'un référentiel qui tourne pour que les commandes données correspondent bien à ce que l'on veut faire. Si ce n'est pas fait, alors au bout de quelques petites rotations les commandes se substituent les unes aux autres. Nous ne faisons pas cette partie du calcul, car la puce à cette capacité et renvoie directement les valeurs correctes. Le problème majeur avec les angles d'Euler, est le « gimbal lock » : Lorsque l'un des axes bouge de $\frac{\pi}{2}$, alors les deux autres rotations auront le même effet tant l'une comme l'autre. Ceci n'est toutefois pas un problème pour notre quadricoptère car nous n'avons pas l'intention de le stabiliser selon le lacet, donc l'effet n'est plus présent. Par contre, comme on voit sur le schéma ci-dessus les angles Y et Z sont inversées par rapport aux angles que nous avons pris comme standard qui sont comme suit :

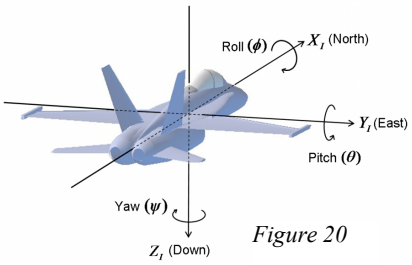

Figure 20

Nous serons donc obligés de les inverser dans le code.

Voici pour les intéressés la matrice de passage d'un référentiel à l'autre :

$$[A] = \begin{pmatrix} \cos(\psi)\cos(\varphi) - \sin(\psi)\cos(\theta)\sin(\varphi) & -\cos(\psi)\sin(\varphi) - \sin(\psi)\cos(\theta)\cos(\varphi) & \sin(\psi)\sin(\theta) \\ \sin(\psi)\cos(\varphi) + \cos(\psi)\cos(\theta)\sin(\varphi) & -\sin(\psi)\sin(\varphi) + \cos(\psi)\cos(\theta)\cos(\varphi) & -\cos(\psi)\sin(\theta) \\ \sin(\theta)\sin(\varphi) & \sin(\theta)\cos(\varphi) & \cos(\theta) \end{pmatrix}$$

Figure 21

Contrôleur PID :

Un contrôleur PID, aussi appelé régulateur ou correcteur, est un système de contrôle qui tourne en boucle. Cela signifie qu'il envoie un signal qui change selon l'erreur du système. L'erreur du système est simplement la différence entre la consigne qu'on lui donne (dans notre cas le contrôle radio du pilote) et de la mesure (ici la vitesse angulaire par le gyroscope). La caractéristique principale du régulateur PID est qu'elle comporte trois termes : la partie proportionnelle, la partie intégrale et la partie dérivée.
Ce diagramme permet de mieux comprendre le fonctionnement de l'algorithme :

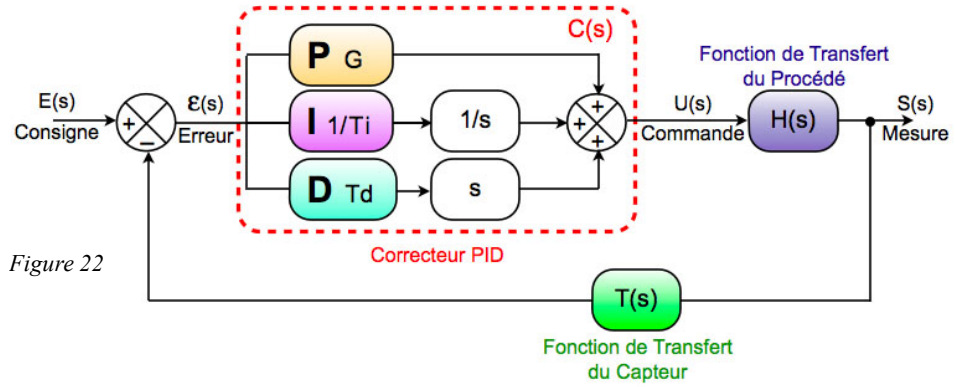

Figure 22

Ce qui revient à la formule suivante :

$$K_\mathrm{p} e(t) + K_\mathrm{i} \int_0^t e(\tau)\,d\tau + K_\mathrm{d} \frac{de(t)}{dt}$$

Figure 23

Tout d'abord, l'erreur est calculée avec la consigne (que nous appelons « setpoint ») et la vitesse angulaire, ensuite, pour la partie proportionnelle, cette erreur est multipliée par un coefficient. Ceci a pour effet de faire tendre le système vers la position désirée mais pour un coefficient trop élevé le système oscille sans contrôle et pour un coefficient trop faible il atteint la consigne trop lentement. On fait aussi la somme de toutes les erreurs passées, ce qui résulte en une intégration numérique, qui va être multipliée par le coefficient de l'intégrale. De plus on tient compte de l'erreur de la boucle précédente, afin de calculer la différence entre celle-ci et celle de la boucle actuelle ce qui résulte en une dérivation numérique, qui va être multipliée par le coefficient de la dérivée. Les effets de chacun de ces termes sont :

Proportionnelle : Fait bouger le système proportionnellement à l'erreur, c'est-à-dire que plus l'erreur est grande plus la réponse va être forte. Les désavantages du terme proportionnel sont que, si le gain est trop faible, le temps de montée du système (le temps que le système met à corriger l'erreur) est trop élevé et, si le gain est trop fort, le dépassement à son tour devient très grand (on appelle cela le « overshoot ») ce qui rend le système très instable. De plus, le terme proportionnel ne corrige pas les erreurs toutes petites ce qui a pour effet de faire dériver (drift) le système vers l'un des côtés.

Intégrale : A le même effet global que le proportionnel mais beaucoup plus lent. Ici le terme intégral est utilisé dans le but de réduire la dérive due au terme proportionnel qui n'est pas assez précis. Si le système dérive vers un côté mais très lentement, les erreurs vont s'additionner à chaque tour de boucle, et le terme va donc corriger ces erreurs toutes petites que le terme proportionnel n'arrive pas à corriger. On dit que l'intégrale corrige l'erreur "statique".

Dérivée : Ce terme permet de résoudre l'autre problème du proportionnel, qui est le dépassement ou « overshoot ». En effet comme la dérivée consiste en une différence de l'erreur passée avec l'erreur de la boucle actuelle, elle ne prend effet que lorsque l'erreur change. Ce terme s'occupe de réduire les changements d'erreur produits par les autres termes, ou des facteurs externes tel que le vent.

Voici un graphique (figure 24) représentant dans notre cas la vitesse angulaire du système en fonction du temps. On y fait aussi figurer la consigne :

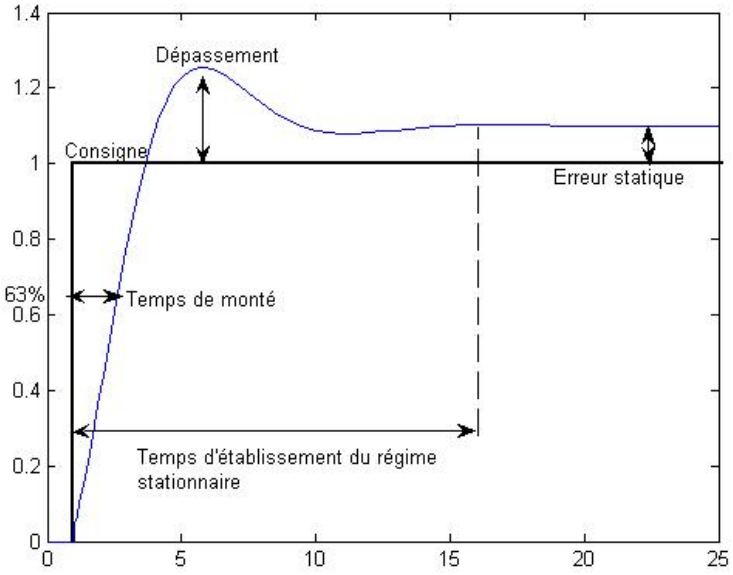

Figure 24

Notre but va être de bien calibrer les constantes devant chaque terme pour que le système réponde vite, sans trop de dépassement, et que l'erreur statique soit nulle.

Réglage du contrôleur :

Il y a plusieurs méthodes pour régler un contrôleur PID. Une des méthodes les plus utilisées dans l'industrie est celle de Ziegler-Nichols. Elle fait entrer en jeu la période d'oscillation du système mais pour notre cas elle n'est pas très efficace car elle engendre beaucoup de dépassements. Cette méthode est principalement utile car elle est simple à utiliser. Voici un tableau montrant comment il faut régler les différents coefficients d'après cette méthode :

Méthode de Ziegler-Nichols[1]			
Type de contrôle	K_p	T_i	T_d
P	$0.5K_u$	-	-
PI	$0.45K_u$	$T_u/1.2$	-
PD	$0.8K_u$	-	$T_u/8$
PID[2]	$0.6K_u$	$T_u/2$	$T_u/8$
PIR[2]	$0.7K_u$	$T_u/2.5$	$3T_u/20$

Figure 44

Dans notre cas, nous règleront les coefficients manuellement car nous ne disposons pas d'une structure afin de mesurer correctement la période comme dans la méthode Ziegler-Nichols.

Matériel

I – Test sur un axe

Avant de nous attaquer à la programmation d'un quadricoptère, nous avons commencé par construire un premier prototype, qui ne possédait que deux moteurs, et ne pouvait pivoter que sur un axe (Figure 25). Ceci nous a permis d'élaborer un algorithme PID pour un axe que nous avons ensuite utilisé pour chaque axe (pitch, roll, yaw).

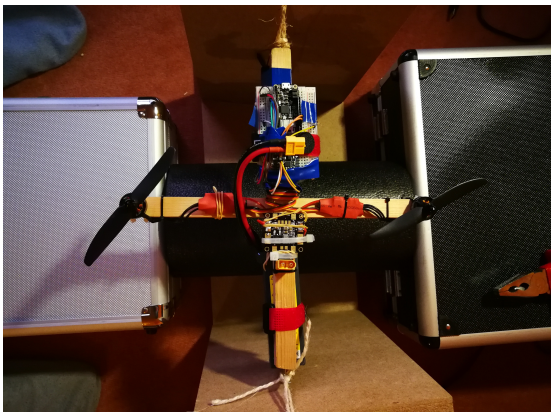

Figure 25

II – Circuit prototype

Afin de pouvoir débuter le développement du programme le plus tôt possible, nous avons d'abord assemblé un premier prototype (Figure 26).

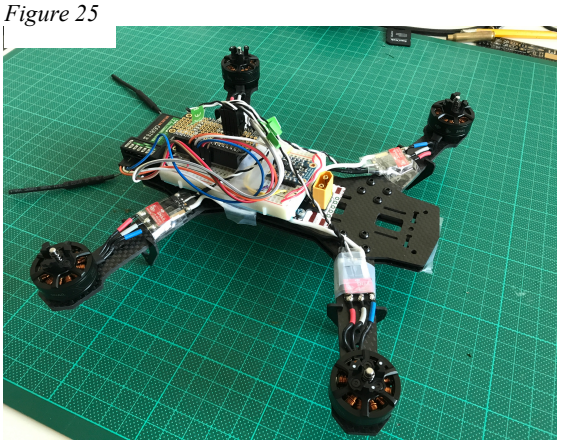

Figure 26

Ce qui peut ressembler à un vulgaire amas de câbles au centre du châssis du quadricoptère est le premier prototype de contrôleur de vol. Ce dernier est compris de plusieurs modules (petits circuits) fabriqués par l'entreprise Adafruit, basée à New York, qui produit de tels modules dans le but de permettre aux gens d'apprendre l'électronique. L'avantage de ces modules est que n'importe qui peut connecter ces modules ensemble et rapidement développer un programme sans avoir à connaître la création de circuits complexes et sans avoir à souder. Chaque module que nous avons utilisé deviendra une section de notre circuit imprimé final.

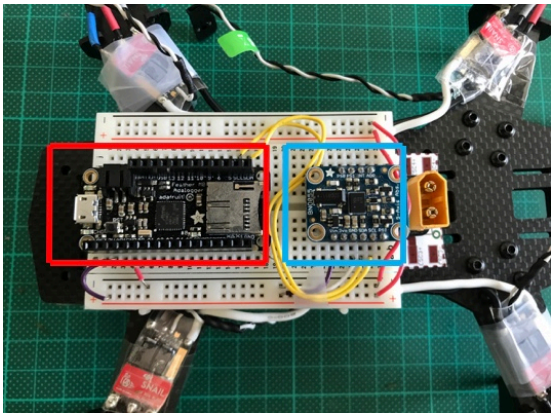

Figure 27

Nous avons donc un module central (Figure 27, encadré rouge) qui comprend le microcontrôleur, un port USB et des connecteurs pour connecter d'autres modules, ainsi qu'un module comprenant le gyroscope (Figure 27, encadré bleu).

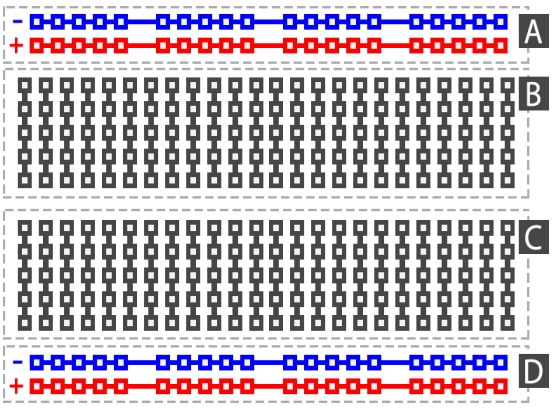

Figure 28

Les différents modules sont connectés sur une « breadboard », qui est une platine d'expérimentation dont les connexions électriques internes permettent la connexion facile des modules. Les connexions internes sont illustrées ci-contre (Figure 28).

III – Structure du circuit imprimé, schéma et composants

Notre contrôleur de vol final se présente sous la forme d'un circuit imprimé. Un circuit imprimé est une plaque qui permet de relier différents composants électroniques de manière compacte. Cette plaque est formée d'une alternance de couches de cuivre gravées de telle manière à former des connexions entre les composants et d'un substrat (FR4 – composé principalement de fibre de verre) entre chacune qui permet de les isoler électriquement, ainsi qu'une couche isolante aux extrémités, appelé masque de soudure, car des trous dans celui-ci permettront la soudure des contacts des différents composants du circuit. Pour relier les couches de cuivre, on utilise un via [14]:

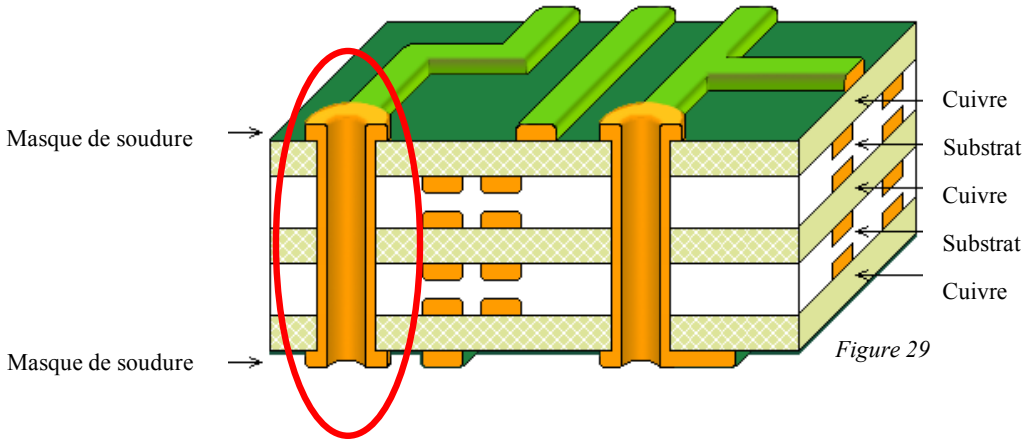

Figure 29

Notre circuit imprimé principal comporte quatre couches de cuivre, et l'inférieur seulement deux, bien qu'un circuit imprimé plus simple puisse en comporter deux, et un circuit plus complexe (comme une carte mère d'ordinateur) puisse en comporter jusqu'à dix.

Voici donc la structure de notre circuit imprimé :

Épaisseur	**Couche**
25,4 µm	Masque de soudure
35,56 µm	Cuivre
170,18 µm	Substrat
17,78 µm	Cuivre
1,2 mm	Substrat Central
17,78 µm	Cuivre
170,18 µm	Substrat
35,56 µm	Cuivre
25,4 µm	Masque de soudure

La première étape dans la conception d'un circuit imprimé est l'établissement du circuit électrique. Ceci est fait dans un programme (EAGLE) qui permet de placer des composants et de dessiner les connexions entre chacun. À la fin de cette opération, on obtient le schéma du circuit. (Voir page suivante).

[14] Trou métallique qui permet de relier électriquement deux couches d'un circuit imprimé

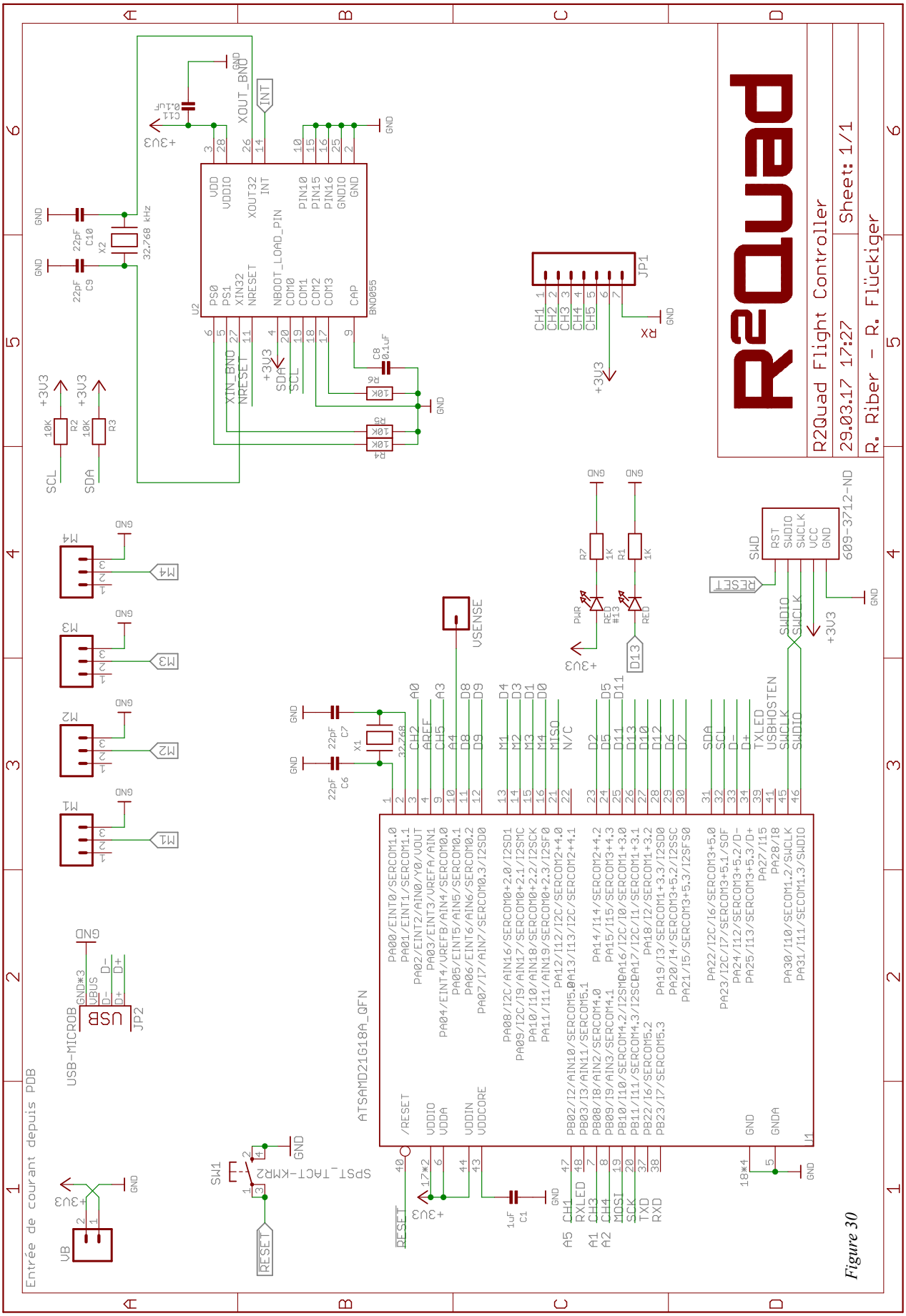
R2Quad
R2Quad Flight Controller
Sheet: 1/1
29.03.17 17:27
R. Riber - R. Flückiger
Entrée de courant depuis PDB
ATSAMD21G18A_QFN
USB-MICROB
SPST_TACT-KMR2
609-3712-ND
JP1
JP2
USENSE
XIN_BND
XOUT_BND
INT
RESET
SWD
SWDIO
SWCLK
+3V3
GND

Figure 30

Le composant central du circuit est le microcontrôleur (voir case C2, figure 30). Le microcontrôleur que nous utilisons est le Atmel ATSAMD21G18. Il s'agit d'un microcontrôleur de type ARM Cortex M0+, qui comprend 256 KB de mémoire de programmation, une interface USB, et fonctionne à une fréquence de 48MHz. Ce microcontrôleur possède aussi une interface I2C (Inter-Integrated Circuit, en anglais), qui est un standard de communication entre puces électroniques qui permet un transfert de données sur seulement deux connexions, une ligne de données (SDA) et une ligne d'horloge (SCL), entre un équipement « maitre » et un équipement « esclave ». Ce protocole nous sera utile pour la transmission des données du gyroscope, qui supporte aussi le protocole I2C. Plusieurs composants passifs sont connectés au microcontrôleur, et permettent le fonctionnement correct du microcontrôleur, tel qu'un quartz oscillant à 32 kHz (voir case B3) qui donne un signal d'horloge au microcontrôleur.

Le deuxième composant crucial du circuit est le gyroscope (voir case B5-B6). Celui-ci est indispensable, comme on l'a vu, aux calculs du microcontrôleur. Nous avons choisi d'utiliser le senseur BNO-055, produit par la branche électronique de Bosch, pour sa précision, son intégration d'un gyroscope et d'un accéléromètre en une seule puce, et sa simplicité d'interface (connexion par I2C, librairie Arduino). Comme le microcontrôleur, un quartz y est connecté pour un signal d'horloge, ainsi que des résistances et condensateurs, nécessaires au bon fonctionnement du senseur.

Le circuit comporte aussi plusieurs connexions en A2-A3-A4 qui permettent la connexion des ESC au contrôleur de vol. Ces connecteurs comportent trois connexions électriques, mais nous n'utilisons que la terre (GND) et le signal (relié à un connecteur du microcontrôleur).
En C5-C6 se situe le connecteur permettant la connexion du récepteur radio au contrôleur de vol. Il possède 7 connexions : 5 pour les canaux de la radio, reliés directement au microcontrôleur, et deux pour l'alimentation du récepteur radio (anode, cathode).

En D4, on trouve le connecteur SWD/JTAG (Serial Wire Debug/Join Test Action Group) qui est un connecteur standard qui permet la connexion d'un programmeur/débugger au microcontrôleur, que nous utilisons pour initialiser le microcontrôleur après l'assemblage du circuit, en chargeant un code spécial appelé « bootloader », chargeur de démarrage en français, qui active la programmation par USB ainsi que le bouton de réinitialisation.

En C4 se trouvent deux diodes électroluminescentes. L'une, connectée entre les contacts de l'alimentation du circuit, permet de savoir si le circuit est sous tension. L'autre, connectée à la sortie #13 du microcontrôleur, permet de charger un code test sur le circuit après assemblage, qui fait clignoter cette diode, afin de vérifier le bon fonctionnement du circuit.

Enfin, en A1 se trouve l'arrivée de courant depuis le circuit inférieur, dont nous parlerons dans les pages qui suivent, en A2 se situe le connecteur micro USB, qui est relié à la terre (GND) et aux deux connexions de l'interface USB du microcontrôleur (D+ et D-, entrée et sortie de données), et en B1 se trouve un bouton de réinitialisation, qui connecte le connecteur de réinitialisation du microcontrôleur à la terre quand appuyé, ce qui relance le code depuis le début.
Le circuit inférieur, dont on a parlé au paragraphe précédent, permet la distribution du courant de la batterie aux moteurs, et de la batterie au contrôleur de vol, après un abaissement de voltage. Voici son schéma :

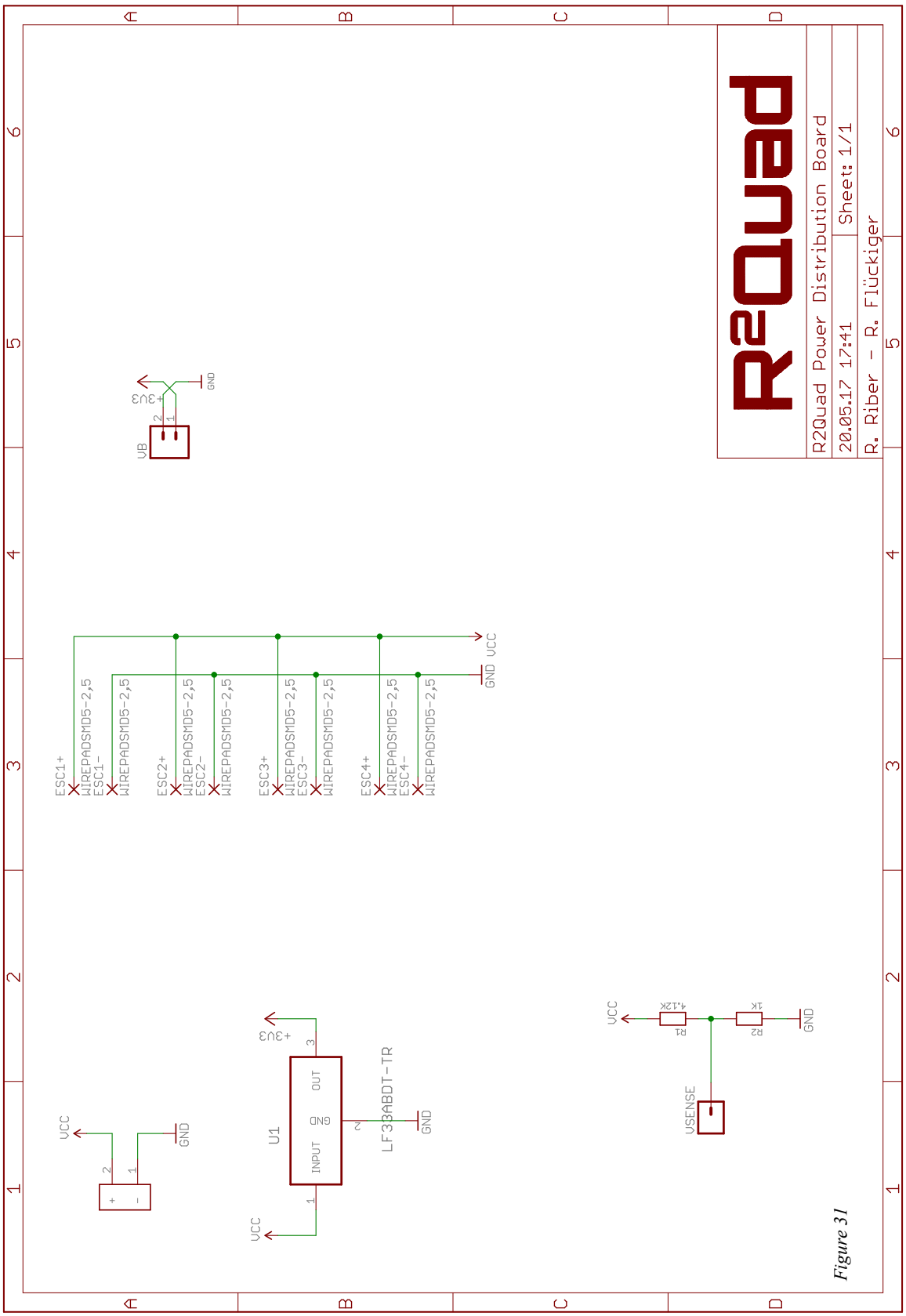

Figure 31

Beaucoup plus simple que le contrôleur de vol, ce circuit (Figure 31) permet de connecter une batterie de modélisme de 16.8V (voir case A1), et fournir le courant aux ESC (connexions en C3-B3-A3), afin d'alimenter les moteurs, ainsi qu'un convertisseur de voltage en B1 et B2, qui convertit la tension de la batterie en une tension de 3.3V, parfaite pour le contrôleur de vol, connecté au circuit inférieur avec le connecteur en A5.

Une fois le schéma établi, on décide dans le même programme de la position physique des composants sur le circuit imprimé, ainsi que des dimensions physiques, trous pour vis, etc.

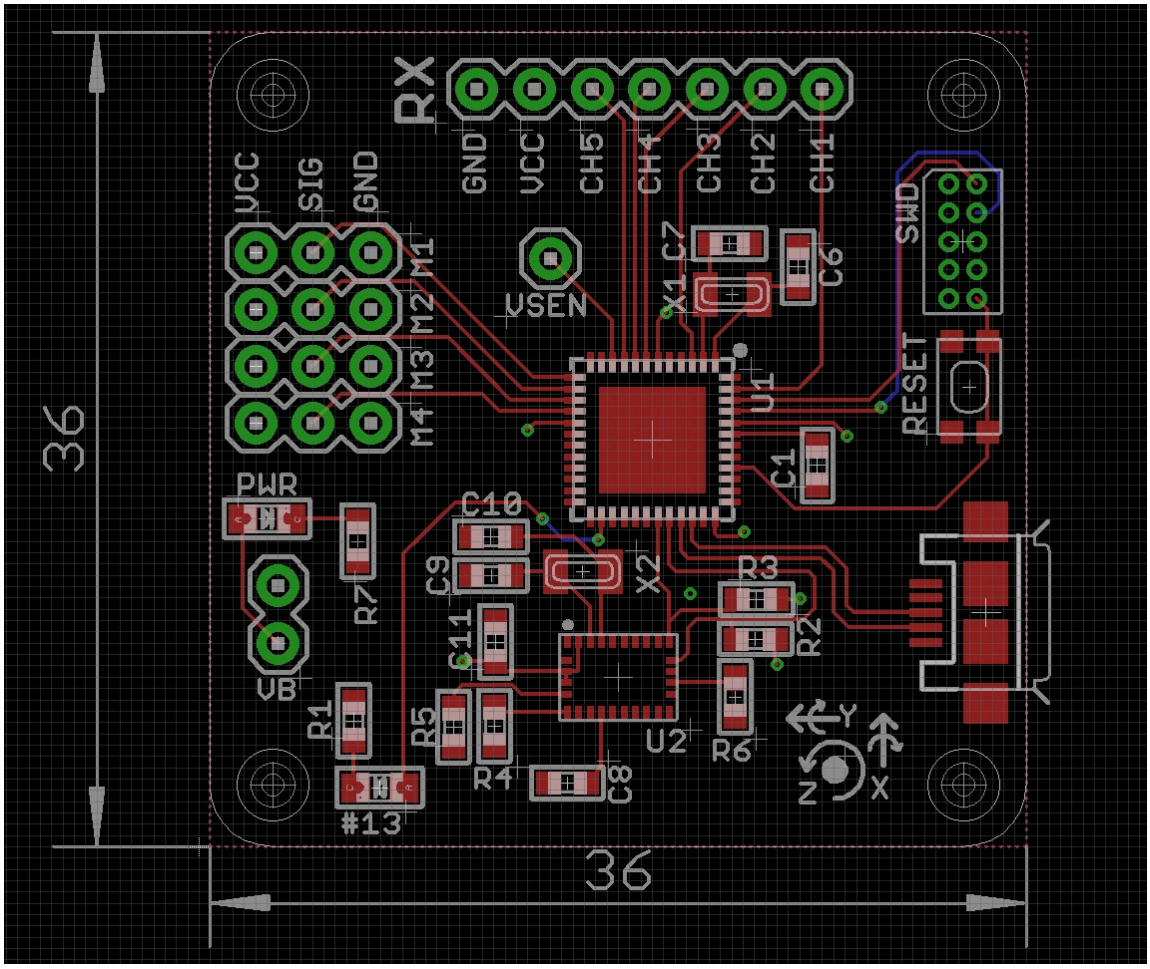

Figure 32

Après avoir placé les composants, les traces de cuivre formant les connexions du circuit sont dessinées manuellement (lignes rouges). Sont ensuite ajoutés les éléments esthétiques tel que le texte et les références des composants, qui faciliteront l'assemblage. On obtient ceci à la fin du processus.

Avant

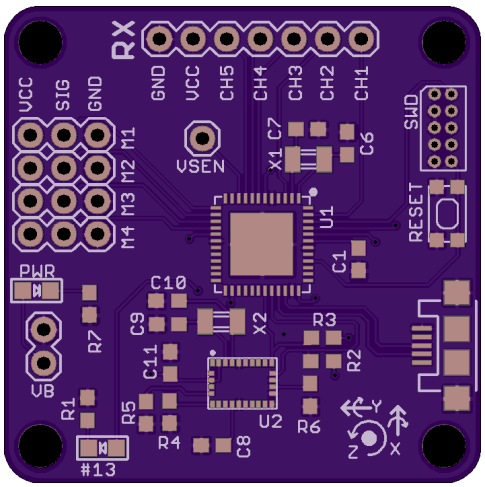

Figure 34

Arrière

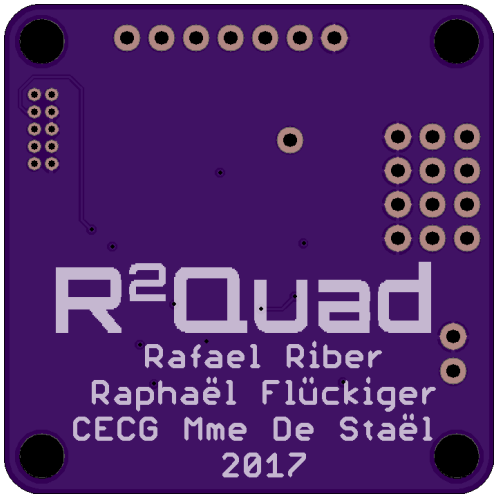

Figure 33

Le fichier produit par le programme est ensuite envoyé pour la fabrication du circuit imprimé. Le service que nous avons utilisé est OSH Park. Il s'agit d'un service de fabrication de circuit imprimés pour les personnes individuelles, qui ne désirent pas une grande quantité de circuits contrairement aux grandes entreprises. Ce service garantit un bas coût (5$ le pouce carré pour 2 couches et 10$ le pouce carré pour 4 couches), en prenant toutes les commandes de ses clients et en les plaçant sur un grand panneau de circuits imprimés, ensuite envoyé à l'usine pour la fabrication. Ce service permet à des personnes individuelles de commander des circuits en petite quantité, sans avoir à payer un panneau de 30 par 45 cm qui ne serait pas rentable pour une petite quantité. Tous les circuits commandés chez OSH Park sont fabriqués aux Etats-Unis.

En parallèle, les différents composants du circuit sont commandés chez un fournisseur de pièces électroniques, DigiKey, basé aux Etats-Unis dans le Minnesota. Voici la liste des composants :

Description	Référence sur le plan	Valeur/Réf. fabriquant
Condensateur 0603	C8, C11	0.1uF
Résistance 0603	R2, R3, R4, R5, R6	10K
Résistance 0603	R1, R7	1K
Condensateur 0603	C1	1uF
Condensateur 0603	C6, C7, C9, C10	22pF
Quartz	X1, X2	32.768 kHz
Microcontrôleur	U1	ATSAMD21G18A_QFN
Gyroscope	U2	BNO055
LED 0805	#13, PWR	RED
Bouton	SW1	SPST_TACT-KMR2
Port micro USB	JP2	10104110-0001LF
Résistance 0603	R1 (Circuit inférieur)	4.12K
Résistance 0603	R2 (Circuit inférieur)	1K
Régulateur de voltage	U1 (Circuit inférieur)	LF33ABDT-TR

IV – Assemblage

Une fois les circuits et les composants reçus, vient l'assemblage du contrôleur de vol. La première étape est de préparer un plan de travail permettant de tenir le circuit en place pendant l'application d'une pâte de soudure. La pâte de soudure remplit la même fonction que la soudure en bobine, mais est très pratique pour des petits composants car elle peut être appliquée à l'aide d'un pochoir, troué là où le contact doit être soudé. On utilise d'autres circuits pour coincer le circuit à assembler, car ils sont de la même hauteur :

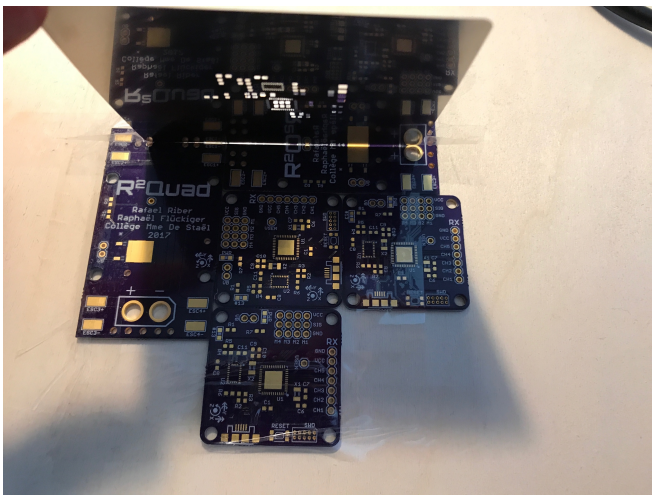

Figure 35

On applique la pâte d'un côté du pochoir, puis on l'étale afin de remplir tous les espaces. Enfin, on place à l'aide de pinces fines chaque composant là où il doit être soudé :

On utilise ensuite une soufflerie à air chaud (250 degrés) afin de faire « fondre » la soudure et que chaque composant soit soudé. On vérifie ensuite minutieusement que des contacts n'ont pas été soudés ensemble, et on corrige les ponts formés au fer à souder.

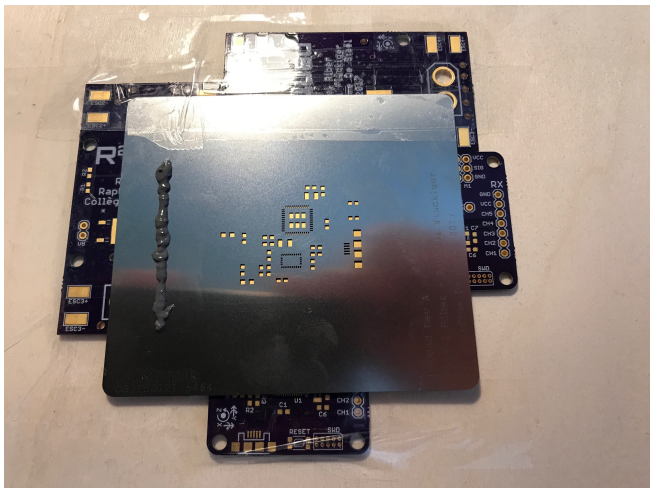

Figure 36

On obtient ainsi un contrôleur de vol assemblé :

On répète la procédure pour le circuit inférieur, mais sans utiliser un pochoir car les contacts sont assez éloignés les uns des autres.

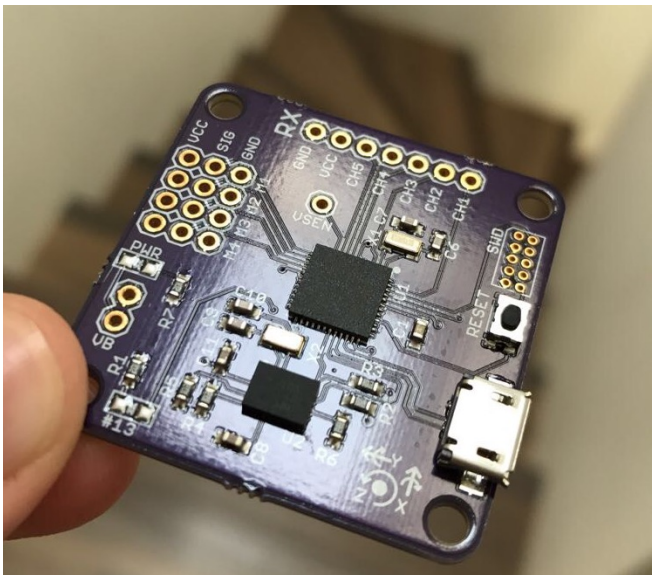

Figure 37

Reste ensuite une dernière étape ; le microcontrôleur vient de l'usine sans aucun programme dans sa mémoire. Il nous faut donc utiliser un programmeur externe afin de charger le fameux « bootloader » (chargeur de démarrage) qui permettra de charger de nouveaux programmes par USB, ce qui est bien plus pratique, et de réinitialiser le microcontrôleur en cas de problème avec le bouton prévu à cet effet.

On connecte donc le programmeur externe aux connecteurs SWD dont on a parlé plus haut (Fig. 38).

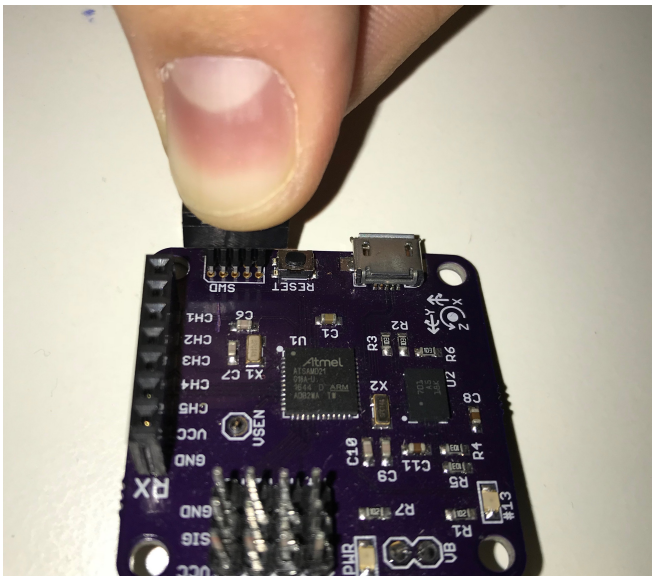

Figure 38

On utilise enfin le programme Atmel Studio, qui est le logiciel du fabriquant du microcontrôleur, afin de charger le fichier hexadécimal du « bootloader » sur la mémoire du microcontrôleur. Ceci termine l'assemblage et le contrôleur de vol peut être connecté à l'ordinateur et programmé avec Arduino (Fig. 39).

Figure 39

Enfin, le circuit supérieur peut être connecté au circuit inférieur (Fig. 40), et le tout est prêt à être connecté aux ESC et un câble est soudé pour la connexion de la batterie (Figure 41).

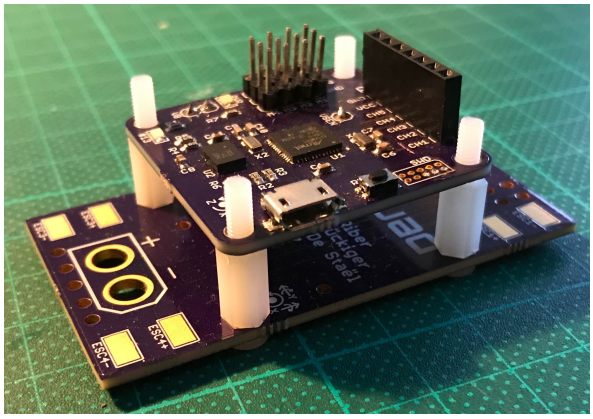

Figure 40

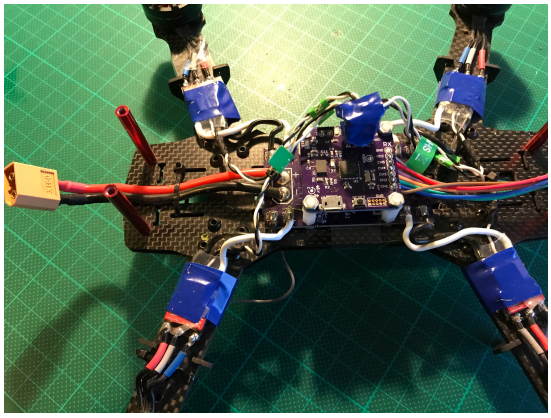

Figure 41

Programme

Nous allons maintenant expliquer chaque partie du code source (voir annexe pour le code dans son intégralité), sa fonction, et pourquoi nous avons choisi de l'écrire de cette manière. A gauche du code sont écrits les numéros de ligne dans le code source.

Avant tout, voici le schéma de fonctionnement du code :
Tout d'abord nous initialisons toutes les variables et exécutons la fonction setup qui ne s'exécutera qu'une seule fois à chaque démarrage.

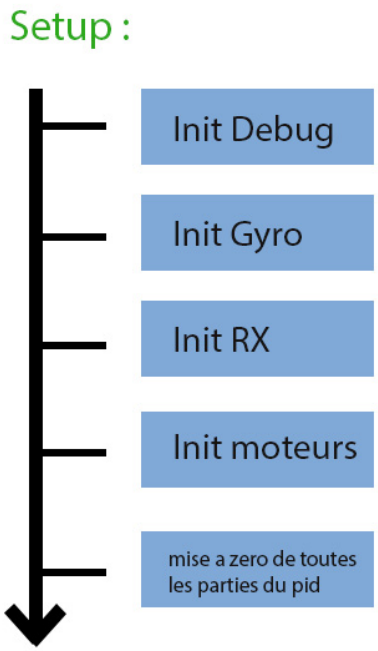

Figure 42

Le but ici est d'initialiser chaque partie du quadricoptère qui sont nécessaires pour son fonctionnement. Setup est une fonction qui se note simplement setup(){…}.

On initialise ici (ligne 363) l'impression par communication USB a une vitesse de 115200 baud[15], qui permet de récupérer différentes valeurs du code pour le débogage.

```
363     Serial.begin(115200);
```

On initialise ensuite le gyroscope, le récepteur et les moteurs, (expliqué plus loin). Ensuite pour être sûr que le quadricoptère débute avec un PID nul, on écrit une valeur nulle dans les variables de sortie du PID, afin d'éviter un saut au démarrage.

```
386     Proportional_roll, Integral_roll, Derivative_roll = 0;
387     Proportional_pitch, Integral_pitch, Derivative_pitch = 0;
388     Proportional_yaw, Integral_yaw, Derivative_yaw = 0;
```

[15] Vitesse de modulation de données

Démarre ensuite la boucle principale, elle s'exécute des centaines de fois par seconde :

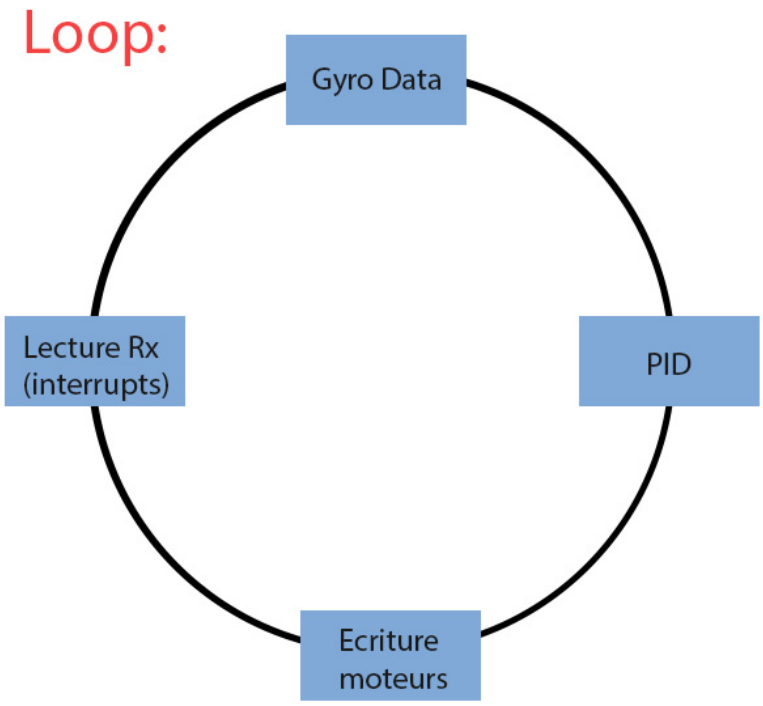

Figure 43

Voici les 4 actions basiques que nous devons effectuer :

- Lire les données du gyroscope
- Lire les données du récepteur
- Calculer le PID
- Écrire les valeurs qui en sortent aux moteurs.

Il y a en plus des subtilités qui permettent une conduite plus sure comme la lecture de l'interrupteur « ARM » qui permet d'activer ou désactiver les moteurs ou encore la conduite en mode niveau, où le quadricoptère retourne à plat si le pilote lâche les commandes.

Dans les premières lignes nous importons les librairies. Celles-ci nous servent à utiliser du code spécifiquement écrit pour remplir une fonction telle que générer un signal PWM pour les moteurs, ou interfacer facilement avec des senseurs tels que le gyroscope, et donc de gagner du temps. Ici nous importons « config.h » qui est un fichier dans lequel on a mis des variables qui ne changeront jamais telles que les gains PID, ou des valeurs par défaut que nous utiliserons.

```
1  #include "config.h"
2  #include <Math.h>
3  #include <Servo.h>
4  #include <Wire.h>
5  #include <Adafruit_Sensor.h>
6  #include <Adafruit_BNO055.h>
7  #include <utility/imumaths.h>
```

La librairie « Math.h » nous sert à faire des calculs comme des sinus et est aussi nécessaire au traitement des données du gyroscope. « Servo.h » nous sert à envoyer plus facilement les signaux aux moteurs. Les autres librairies nous seront utiles pour la communication avec le

gyroscope. Ces dernières sont fournies par Adafruit, fabricant des modules électroniques dont on a parlé précédemment, ainsi que la plateforme Arduino, et sont libres d'utilisation.

I – Accéder aux ESC et leur envoyer un signal de rotation

Ici nous initialisons les 4 objets Servo qu'on appelle esc_1 (resp.2, 3 et 4).

```
10  Servo esc_1; //FRONT RIGHT (Moteur avant droit)
11  Servo esc_2; //BACK LEFT (Moteur arrière gauche)
12  Servo esc_3; //FRONT LEFT (Moteur avant gauche)
13  Servo esc_4; //BACK RIGHT (Moteur arrière droit)
```

Il existe dans la librairie Servo une fonction qui nous permet de définir les bornes en microsecondes du signal PWM, ainsi que les connecteurs sur lesquels ce signal doit être écrit (lignes 376-379).

```
376     esc_1.attach(MOTOR_PIN_FRONT_RIGHT, MOTOR_ZERO_LEVEL, MOTOR_MAX_LEVEL);
377     esc_2.attach(MOTOR_PIN_FRONT_LEFT, MOTOR_ZERO_LEVEL, MOTOR_MAX_LEVEL);
378     esc_3.attach(MOTOR_PIN_BACK_LEFT, MOTOR_ZERO_LEVEL, MOTOR_MAX_LEVEL);
379     esc_4.attach(MOTOR_PIN_BACK_RIGHT, MOTOR_ZERO_LEVEL, MOTOR_MAX_LEVEL);
```

Ici on précise que le signal va de 1000 à 2000 (MOTOR_ZERO_LEVEL à MOTOR_MAX_LEVEL), et les connecteurs sont spécifiés par les constantes « MOTOR_PIN_x_x », définies dans « config.h ».

La fonction esc_1.writeMicroseconds(temps) (lignes 437-440) nous permet d'envoyer un signal PWM d'une longueur voulue à l'objet esc_1 (resp.2, 3 et 4).
Plus tard dans la boucle principale (fonction « loop() »), nous calculerons justement quel sera le paramètre à entrer dans cette fonction.

```
437         esc_1.writeMicroseconds(motorFR);
438         esc_2.writeMicroseconds(motorFL);
439         esc_3.writeMicroseconds(motorBL);
440         esc_4.writeMicroseconds(motorBR);
```

II – Récupération des données du gyroscope

Dans la fonction setup(), qui est exécutée une fois à chaque démarrage, on initialise le gyroscope (Ligne 365), défini au début du code (Ligne 39), et on active l'utilisation d'un quartz externe (Ligne 366).

```
39  Adafruit_BNO055 bno = Adafruit_BNO055(); // Initialisation de l'objet BNO
```

```
365     bno.begin();
366     bno.setExtCrystalUse(true);
```

Nous créons ici « gyroscope » qui est un vecteur contenant les vitesses angulaires des trois axes en radians. Ainsi nous les stockons dans des variables distinctes et les transformons en degrés pars seconde en les multipliant par 180/pi.

```
398     imu::Vector<3> gyroscope = bno.getVector(Adafruit_BNO055::VECTOR_GYROSCOPE);
399     roll_speed = gyroscope.x() * 180 / pi;
400     pitch_speed = -1 * gyroscope.y() * 180 / pi;
401     yaw_speed = -1 * gyroscope.z() * 180 / pi;
```

Similairement pour obtenir les angles d'Euler :

```
403     imu::Vector<3> euler = bno.getVector(Adafruit_BNO055::VECTOR_EULER);
404     roll_angle = -1 * euler.z();
405     pitch_angle = -1 * euler.y();
406     yaw_angle = euler.x();
```

La transformation en angles d'Euler ici se fait implicitement comme expliqué précédemment (section Gyroscope).

III – Réception et traitement du contrôle radio

La lecture des signaux venant du récepteur doit être faite le plus rapidement possible, afin de ne pas ralentir les calculs du PID. C'est pourquoi nous avons utilisé une fonction du microcontrôleur appelée « Interrupt », qui permet de définir des fonctions qui s'exécutent après un changement de voltage sur un connecteur, court-circuitant le code en cours, qui reprend juste après. On initialise donc pour chaque canal une fonction qui mesure la longueur du signal PWM sur un connecteur, et la stocke dans une variable. Aux lignes 57 à 74 se trouve la fonction « interrupt » pour le roulis, les autres fonctions identiques, mais stockent leurs valeurs dans des variables différentes.

```
57  void calcSignalROLL() {
58    if (digitalRead(RX_PIN_ROLL) == HIGH) {
59      chrono_start1 = micros();
60    }
61    else {
62      if (chrono_start1 != 0) {
63        inputROLLprev = inputROLL;
64        inputROLL = ((volatile int)micros() - chrono_start1);
65        if (inputROLL >= inputROLLprev + 200){
66          inputROLL = inputROLLprev + 200;
67        }
68        if (inputROLL <= inputROLLprev - 200){
69          inputROLL = inputROLLprev - 200;
70        }
71        chrono_start1 = 0;
72      }
73    }
74  }
```

Ce code vérifie que le signal PWM soit HIGH (Haut, 3.3V) et il mesure le temps qu'il prend pour revenir en LOW (Bas, 0V). On assigne à la variable inputRoll ce temps. En plus, nous avons remarqué en lisant les signaux que la radio n'était pas très consistante avec les signaux qu'elle envoyait (variations de 2500µs entre une mesure et l'autre) alors nous bornons la différence d'une mesure à l'autre à 200 µs maximum.

On initialise ensuite l'ordre d'interrupt dans la fonction setup(), avec la fonction attachInterrupt(connecteur, fonction à exécuter, condition) (lignes 369-373).

```
369    attachInterrupt(RX_PIN_ARM, calcSignalARM, CHANGE);
370    attachInterrupt(RX_PIN_ROLL, calcSignalROLL, CHANGE);
371    attachInterrupt(RX_PIN_PITCH, calcSignalPITCH, CHANGE);
372    attachInterrupt(RX_PIN_THROTTLE, calcSignalTHROTTLE, CHANGE);
373    attachInterrupt(RX_PIN_YAW, calcSignalYAW, CHANGE);
```

Ici la fonction est appelée, grâce au « CHANGE », lorsque le signal passe de LOW à HIGH que de HIGH à LOW.

IV – PID

L'implémentation du PID en code se fait assez facilement en ayant la formule. De fait, il nous faut une fonction de transfert qui transforme l'entrée utilisateur (transmetteur) en degrés par seconde, nous nommerons le résultat « setpoint ». La manière la plus simple que nous avons trouvée est de soustraire la valeur médiane du transmetteur.
C'est-à-dire : on sait que la valeur donnée par le transmetteur est comprise entre 1000 et 2000 microsecondes. Par conséquent, la valeur du milieu est 1500. Donc si le pilote veut que le quadricoptère ne bouge pas, il ne va pas toucher le manche à balais, ce qui enverra une valeur de 1500 à laquelle nous soustrairons 1500, le résultat est 0 comme espéré. Ceci se traduit pour le roulis comme pour les autres axes par :

```
143       pid_roll_setpoint = 0;
144       if (inputROLL > THROTTLE_RMID + 10)pid_roll_setpoint = (inputROLL - THROTTLE_RMID + 10)/2.0;
145       else if (inputROLL < THROTTLE_RMID - 10)pid_roll_setpoint = (inputROLL - THROTTLE_RMID - 10)/2.0;
```

On multiplie ensuite la valeur par une constante qui déterminera la vitesse angulaire maximale qu'aura le quadricoptère. Si on ne la multiplie pas par cette constante, la vitesse maximale sera de 500 degrés par secondes. On laisse de plus une marge de +-10 microsecondes pour des micromouvements involontaires du manche à balais.
Ensuite en suivant la boucle PID il faut calculer l'erreur, c'est-à-dire la différence entre le setpoint et la vitesse angulaire de chaque axe :

```
147       roll_error = roll_speed - pid_roll_setpoint;
```

Il suffit de calculer ensuite pour chaque terme. Pour le P nous multiplions simplement l'erreur par un gain déterminé Kp :

```
149       Proportional_roll = ROLL_PID_KP * roll_error;
```

Pour le I nous additionnons toutes les erreurs précédentes à la nouvelle, et multiplions par un gain Ki :

```
150       Integral_roll += (ROLL_PID_KI ) * roll_error;
151       if (Integral_roll > ROLL_PID_MAX)Integral_roll = ROLL_PID_MAX;
152       else if (Integral_roll < ROLL_PID_MIN)Integral_roll = ROLL_PID_MIN;
```

De plus nous sommes obligés de contraindre ce terme car il risque d'augmenter trop vite par sa nature (c'est une somme qui se fait à chaque boucle).
Pour le D nous soustrayons simplement l'erreur précédente avec celle de cette boucle et multiplions par un gain :

```
153       Derivative_roll = ROLL_PID_KD * (roll_error - last_roll_error);
```

Finalement, nous additionnons les 3 termes dans une variable appelée pid_AXE_out et assignons l'erreur a la variable de l'ancienne erreur ce qui sert pour les calculs suivants:

```
155       pid_roll_out = Proportional_roll + Integral_roll + Derivative_roll;
156       if (pid_roll_out > ROLL_PID_MAX)pid_roll_out = ROLL_PID_MAX;
157       else if (pid_roll_out < ROLL_PID_MIN)pid_roll_out = ROLL_PID_MIN;
158
159       last_roll_error = roll_error;
```

Nous faisons de même pour les autres axes.

On ne multiplie pas le terme I par « dt » ou divisons D par « dt » car tout se fait à intervalle de temps réguliers, ce qui rend « dt » redondant. Par contre, afin d'être sûr que tout se fait à intervalle régulier on entoure la boucle par

```
137     unsigned long maintenant = millis();
138     int deltaTemps = (maintenant - dernierTemps);
139     if(deltaTemps >= dt){
```

et

```
202         dernierTemps = maintenant;
```

Ainsi la boucle ne s'exécute que si le temps nécessaire est passé.

Il faut maintenant écrire les valeurs aux moteurs. Pour ce faire il faut se rappeler des standards d'axe que l'on a pris qui sont les suivants :

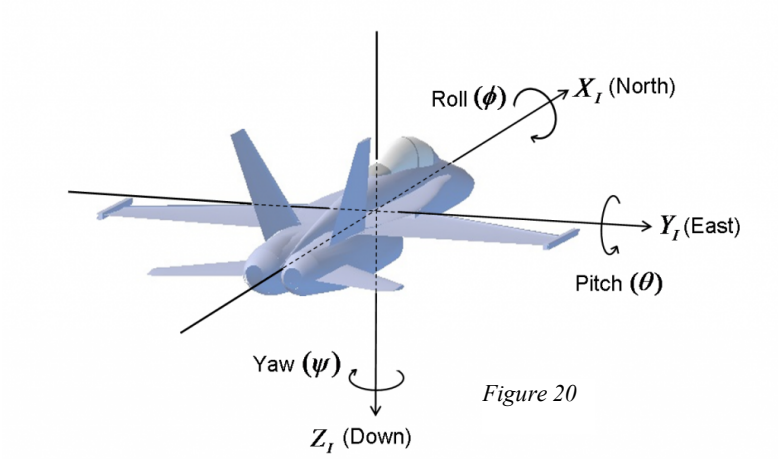

Figure 20

Il suffit de raisonner pour un moteur et faire de manière analogue pour les autres :

Prenons le moteur 1 qui correspond au moteur avant-droit et prenons le cas où le pilote ne veut pas que le quadricoptère bouge. Pour un roulis positif, il faudra augmenter la puissance de ce moteur. Pour un tangage positif, il faudra diminuer la puissance de ce moteur. Finalement pour un lacet positif, il faudra diminuer la puissance de ce moteur car il tourne dans le sens antihoraire. On obtient donc le code suivant :

```
422         motorFR = throttle - pid_pitch_out + pid_roll_out - pid_yaw_out; //1
423         motorBL = throttle + pid_pitch_out - pid_roll_out - pid_yaw_out; //2
424         motorFL = throttle - pid_pitch_out - pid_roll_out + pid_yaw_out; //3
425         motorBR = throttle + pid_pitch_out + pid_roll_out + pid_yaw_out; //4
```

Car les moteurs du même côté d'un axe auront le même signe et les moteurs opposes auront le même signe de lacet.
Nous écrivons ensuite les valeurs aux moteurs comme vu précédemment (voir page 25).

V – Mode Niveau (LEVEL) :

Pour la conception de ce mode nous avons beaucoup hésité à reconstruire le PID, mais cette fois basé sur les angles d'Euler au lieu de la vitesse angulaire. Nous nous sommes finalement dit qu'il serait encore plus simple d'ajouter une modulation au setpoint lui-même, au moment où l'on calcule le PID.

Nous avons choisi de faire comme suit :
Il faut tout d'abord calculer les angles « Pitch » et « Roll » car il s'agit de ceux que l'on veut stabiliser. Ensuite la technique a été de multiplier cette valeur par un nombre tel que celui-ci définisse l'angle maximal et de soustraire ceci au setpoint donc :

$setpoint - angle * N$ ce qui revient a dire que $N = \frac{setpoint}{angle}$, et comme ceci est posé pour un setpoint maximum (qui est de 500 comme vu plus haut) et un angle maximal (disons ~35 degrés), N vaut ici ~15 degrés.

Cela se traduit par :

```
409     roll_angle_adjust = roll_angle * -15;
410     pitch_angle_adjust = pitch_angle * -15;
```

Et ensuite il faut simplement soustraire cette valeur à setpoint :

```
213     pid_roll_setpoint -= roll_angle_adjust;
```

Pour faire que cette opération ne soit appelée que quand le mode est actif, nous avons fait une fonction séparée pour le mode LEVEL :

```
413     if (auto_stabilisation_mode == false)pid_compute();
414     else pid_LEVEL_compute();
```

Conclusion :

Malgré d'innombrables tentatives de calibration des différents coefficients dans le but de faire voler le quadricoptère correctement, nous ne sommes pas parvenus à le faire décoller et le faire tenir de manière stable. Le quadricoptère répond toutefois correctement à des changements d'orientation, comme il est censé le faire, ce qui signifie que, bien que le code soit correct en théorie, les tests pratiques nous ont montré que la calibration est critique au fonctionnement correct du quadricoptère. Des tests plus approfondis nous auraient peut-être permis de le faire décoller, mais nous aurions eu besoin de plus de temps, car la méthode de calibration se faisait plus par tâtonnement qu'en suivant une méthode qui garantirait une calibration correcte.

Notre travail s'est concentré sur l'essence même des quadricoptères, ce qui est indispensable au vol, mais l'avancement de la technologie est si rapide que les quadricoptères commerciaux les plus récents possèdent des fonctions beaucoup plus avancées, telles que la localisation par GPS et l'utilisation de magnétomètres (comme dit précédemment), qui leur permet de rester à une position, une altitude et une orientation extrêmement précises, et d'en plus compenser les facteurs externes tels que le vent afin de garder cette position quoi qu'il arrive. De plus, la localisation GPS permet de faire des vols autonomes, après avoir défini une route au mètre près. Les quadricoptères les plus avancés possèdent même des systèmes de vision intelligents leur permettant d'éviter des obstacles tels que des arbres ou des bâtiments de manière totalement autonome, ainsi que de suivre un sujet tel qu'une personne ou un véhicule automatiquement.

Le futur de ces technologies tend à se préciser, avec un assouplissement des lois pour l'utilisation commerciale de ces engins, tel que la livraison de colis ou de marchandises urgentes telles que les organes à transplanter, ainsi qu'une baisse radicale des prix des quadricoptères grand public ; en effet, un quadricoptère sorti récemment, coûte presque la moitié du prix d'un quadricoptère il y a deux ans, avec des fonctionnalités de pointe telles que celles énoncées plus haut.

Le domaine des quadricoptères ne cesse d'évoluer, et il ne serait pas surprenant de voir une démocratisation totale de ces engins dans les décennies à venir.

Bilan personnel :

Nous avons tous deux pris un très grand plaisir à concevoir notre propre contrôleur de vol, qu'il s'agisse de la programmation, de la recherche, ou de la fabrication. Bien que nous n'ayons pas réussi à faire décoller le quadricoptère, nous considérons tout de même ce travail comme une réussite, car excepté un blocage dans la calibration finale, le contrôleur de vol est totalement opérationnel.

Ce travail nous a permis de mieux comprendre comment des engins qui paraissent si simples sont en fait des systèmes très complexes et instables, où la moindre erreur peut être fatale.

Bibliographie :

LEARN ENGINEERING, *Brushless DC Motor, How it works ?,* [En ligne]. [Consulté en juillet 2017], disponible à l'adresse : http://www.learnengineering.org/2014/10/Brushless-DC-motor.html

WIKIPEDIA, *Electronic Speed Control,* [En ligne]. [Consulté en septembre 2017], disponible à l'adresse : https://en.wikipedia.org/wiki/Electronic_speed_control

BOSCH SENSORTEC, « Data Sheet - BNO055 – Intelligent 9-axis absolute orientation sensor », in *Bosch Sensortec,* [En ligne]. [Consulté en septembre 2017], disponible à l'adresse : https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/BST_BNO055_DS000_14.pdf

WIKIPEDIA, *Régulateur PID,* [En ligne]. [Consulté en septembre 2017], disponible à l'adresse : https://fr.wikipedia.org/wiki/Régulateur_PID

WIKIPEDIA, *PID Controller,* [En ligne]. [Consulté en septembre 2017], disponible à l'adresse : https://en.wikipedia.org/wiki/PID_controller

WIKIPEDIA, *Quadcopter,* [En ligne]. [Consulté en septembre 2017], disponible à l'adresse : https://en.wikipedia.org/wiki/Quadcopter

DRONEADDICT, *Un drone, comment ça vole ?,* [En ligne]. [Consulté en septembre 2017], disponible à l'adresse : http://www.droneaddict.net/comment-ca-vole/

ARDUINO, *Arduino – Reference,* [En ligne]. [Consulté en juillet 2017], disponible à l'adresse : https://www.arduino.cc/en/Reference/HomePage

ADAFRUIT, *Adafruit BNO055 Absolute Orientation Sensor – Absolute orientation without the Ph.D in digital signal processing !,* [En ligne]. [Consulté en octobre 2016], disponible à l'adresse : https://learn.adafruit.com/adafruit-bno055-absolute-orientation-sensor/overview

Table des illustrations :

Déclaration d'authenticité

PLAGIAIRE

[plaʒjɛʀ] n. – plagiere 1584 ; lat. *plagiarius* « celui qui vole les esclaves d'autrui », du gr. *Plagios* « oblique, fourbe » ♦ Personne qui pille ou démarque les ouvrages des auteurs.

PLAGIER

[plaʒje] v.tr. – 1801; de plagiat 1♦ Copier (un auteur) en s'attribuant indûment des passages de son œuvre. => **imiter, piller**.

(Petit Robert I - éd. 1996)

Les élèves
Prénom NOM : Rafael Riber
Prénom NOM : Raphaël Flückiger
Groupe : M408
Maître accompagnant : M. Flumet, Pierre-Alain

attestent avoir conçu et rédigé personnellement, dans son style propre, le travail de maturité ci-joint;

attestent notamment ne pas avoir eu recours au plagiat et avoir systématiquement et clairement mentionné tous les emprunts faits à autrui.

Lieu, date et signature : .....

Lieu, date et signature : .....

Annexes :

Programme principal :

```
#include "config.h"
#include <Math.h>
#include <Servo.h>
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>

// Mise en place des variables
Servo esc_1; //FRONT RIGHT (Moteur avant droit)
Servo esc_2; //BACK LEFT (Moteur arrière gauche)
Servo esc_3; //FRONT LEFT (Moteur avant gauche)
Servo esc_4; //BACK RIGHT (Moteur arrière droit)

// PID
byte dt = 25; // C'est le temps que doit attendre le PID avant de calculer
unsigned long dernierTemps; // Est utile aussi pour le dt du PID
unsigned long maintenant, deltaTemps;
bool auto_stabilisation_mode = false; // Activer ou pas le mode auto-stabilisé
float pid_roll_out,   pid_roll_setpoint,  roll_error, Proportional_roll,  Integral_roll,  Derivative_roll,  last_roll_error, roll_angle_adjust = 0;
float pid_pitch_out,  pid_pitch_setpoint, pitch_error, Proportional_pitch, Integral_pitch, Derivative_pitch, last_pitch_error, pitch_angle_adjust = 0;
float pid_yaw_out,    pid_yaw_setpoint,   yaw_error,  Proportional_yaw,   Integral_yaw,   Derivative_yaw,   last_yaw_error = 0;

// MOTEURS
int motorFR, motorBL, motorFL, motorBR;

// RX
int throttle;
volatile int inputARM, inputROLL, inputPITCH, inputTHROTTLE, inputYAW;
volatile int inputARMprev, inputROLLprev, inputPITCHprev, inputTHROTTLEprev, inputYAWprev;
volatile unsigned long chrono_start0, chrono_start1, chrono_start2, chrono_start3, chrono_start4;
volatile int last_interrupt_time0, last_interrupt_time1, last_interrupt_time2, last_interrupt_time3, last_interrupt_time4;

// IMU
float roll_speed, roll_angle;
float pitch_speed, pitch_angle;
float yaw_speed, yaw_angle;
const float pi = 3.14159265359;
Adafruit_BNO055 bno = Adafruit_BNO055(); // Initialisation de l'objet BNO

////////////////////////////////////////////////////////////////////////
//LECTURE RADIO AVEC INTERRUPTS
////////////////////////////////////////////////////////////////////////
void calcSignalARM() {
  if (digitalRead(RX_PIN_ARM) == HIGH) {
    chrono_start0 = micros();
  }
  else {
    if (chrono_start0 != 0) {
      inputARMprev = inputARM;
      inputARM = ((volatile int)micros() - chrono_start0);
      chrono_start0 = 0;
    }
  }
}

```

```
57. void calcSignalROLL() {
58.   if (digitalRead(RX_PIN_ROLL) == HIGH) {
59.     chrono_start1 = micros();
60.   }
61.   else {
62.     if (chrono_start1 != 0) {
63.       inputROLLprev = inputROLL;
64.       inputROLL = ((volatile int)micros() - chrono_start1);
65.       if (inputROLL >= inputROLLprev + 200){
66.         inputROLL = inputROLLprev + 200;
67.       }
68.       if (inputROLL <= inputROLLprev - 200){
69.         inputROLL = inputROLLprev - 200;
70.       }
71.       chrono_start1 = 0;
72.     }
73.   }
74. }
75.
76. void calcSignalPITCH() {
77.   if (digitalRead(RX_PIN_PITCH) == HIGH) {
78.     chrono_start2 = micros();
79.   }
80.   else {
81.     if (chrono_start2 != 0) {
82.       inputPITCHprev = inputPITCH;
83.       inputPITCH = ((volatile int)micros() - chrono_start2);
84.       if (inputPITCH >= inputPITCHprev + 200){
85.         inputPITCH = inputPITCHprev + 200;
86.       }
87.       if (inputPITCH <= inputPITCHprev - 200){
88.         inputPITCH = inputPITCHprev - 200;
89.       }
90.       chrono_start2 = 0;
91.     }
92.   }
93. }
94.
95. void calcSignalTHROTTLE() {
96.   if (digitalRead(RX_PIN_THROTTLE) == HIGH) {
97.     chrono_start3 = micros();
98.   }
99.   else {
100.          if (chrono_start3 != 0) {
101.            inputTHROTTLEprev = inputTHROTTLE;
102.            inputTHROTTLE = ((volatile int)micros() - chrono_start3);
103.            if (inputTHROTTLE >= inputTHROTTLEprev + 200){
104.              inputTHROTTLE = inputTHROTTLEprev + 200;
105.            }
106.            if (inputTHROTTLE <= inputTHROTTLEprev - 200){
107.              inputTHROTTLE = inputTHROTTLEprev - 200;
108.            }
109.            chrono_start3 = 0;
110.          }
111.        }
112.      }
113.
114.      void calcSignalYAW() {
115.        if (digitalRead(RX_PIN_YAW) == HIGH) {
116.          chrono_start4 = micros();
117.        }
118.        else {
119.          if (chrono_start4 != 0) {
120.            inputYAWprev = inputYAW;
121.            inputYAW = ((volatile int)micros() - chrono_start4);
122.            if (inputYAW >= inputYAWprev + 200){
```

```
123.               inputYAW = inputYAWprev + 200;
124.             }
125.             if (inputYAW <= inputYAWprev - 200){
126.               inputYAW = inputYAWprev - 200;
127.             }
128.             chrono_start4 = 0;
129.           }
130.         }
131.       }
132.
133.       ////////////////////////////////////////////////////////////////////////////////////////////////
134.       //Controleur PID
135.       ////////////////////////////////////////////////////////////////////////////////////////////////
136.       void pid_compute() {
137.         unsigned long maintenant = millis();
138.         int deltaTemps = (maintenant - dernierTemps);
139.         if(deltaTemps >= dt){
140.
141.           //ROLL calculations
142.           //definition du setpoint
143.           pid_roll_setpoint = 0;
144.           if (inputROLL > THROTTLE_RMID + 10)pid_roll_setpoint = (inputROLL - THROTTLE_RMID + 10)/2.0;
145.           else if (inputROLL < THROTTLE_RMID - 10)pid_roll_setpoint = (inputROLL - THROTTLE_RMID - 10)/2.0;
146.
147.           roll_error = roll_speed - pid_roll_setpoint;
148.
149.           Proportional_roll = ROLL_PID_KP * roll_error;
150.           Integral_roll += (ROLL_PID_KI ) * roll_error;
151.           if (Integral_roll > ROLL_PID_MAX)Integral_roll = ROLL_PID_MAX;
152.           else if (Integral_roll < ROLL_PID_MIN)Integral_roll = ROLL_PID_MIN;
153.           Derivative_roll = ROLL_PID_KD * (roll_error - last_roll_error);
154.
155.           pid_roll_out = Proportional_roll + Integral_roll + Derivative_roll;
156.           if (pid_roll_out > ROLL_PID_MAX)pid_roll_out = ROLL_PID_MAX;
157.           else if (pid_roll_out < ROLL_PID_MIN)pid_roll_out = ROLL_PID_MIN;
158.
159.           last_roll_error = roll_error;
160.
161.
162.           //PITCH calculations
163.           //definition du setpoint
164.           pid_pitch_setpoint = 0;
165.           if (inputPITCH > THROTTLE_RMID + 10)pid_pitch_setpoint = (inputPITCH - THROTTLE_RMID + 10)/2.0;
166.           else if (inputPITCH< THROTTLE_RMID - 10)pid_pitch_setpoint = (inputPITCH - THROTTLE_RMID - 10)/2.0;
167.
168.           pitch_error = pitch_speed - pid_pitch_setpoint;
169.
170.           Proportional_pitch = PITCH_PID_KP * pitch_error;
171.           Integral_pitch += PITCH_PID_KI * pitch_error;
172.           if (Integral_pitch > PITCH_PID_MAX)Integral_pitch = PITCH_PID_MAX;
173.           else if (Integral_pitch < PITCH_PID_MIN)Integral_pitch = PITCH_PID_MIN;
174.           Derivative_pitch = PITCH_PID_KD * (pitch_error - last_pitch_error);
175.
176.           pid_pitch_out = Proportional_pitch + Integral_pitch + Derivative_pitch;
177.           if (pid_pitch_out > PITCH_PID_MAX)pid_pitch_out = PITCH_PID_MAX;
178.           else if (pid_pitch_out < PITCH_PID_MIN)pid_pitch_out = PITCH_PID_MIN;
179.
180.           last_pitch_error = pitch_error;
181.
182.           //YAW calculations
```

```
183.           //On calcule le setpoint du yaw ici car il est le meme en stabilise ou en
    acro
184.           pid_yaw_setpoint = 0;
185.           if (inputYAW > THROTTLE_RMID + 10)pid_yaw_setpoint = (inputYAW - THROTTLE
   _RMID + 10) / 2.0;
186.           else if (inputYAW < THROTTLE_RMID - 10)pid_yaw_setpoint = (inputYAW - THR
   OTTLE_RMID - 10) / 2.0;
187.
188.           yaw_error = yaw_speed - pid_yaw_setpoint;
189.
190.           Proportional_yaw = YAW_PID_KP * yaw_error;
191.           Integral_yaw += YAW_PID_KI * yaw_error;
192.           if (Integral_yaw > YAW_PID_MAX)Integral_yaw = YAW_PID_MAX;
193.           else if (Integral_yaw < YAW_PID_MIN)Integral_yaw = YAW_PID_MIN;
194.           Derivative_yaw = YAW_PID_KD * (yaw_error - last_yaw_error);
195.
196.           pid_yaw_out = Proportional_yaw + Integral_yaw + Derivative_yaw;
197.           if (pid_yaw_out > YAW_PID_MAX)pid_yaw_out = YAW_PID_MAX;
198.           else if (pid_yaw_out < YAW_PID_MIN)pid_yaw_out = YAW_PID_MIN;
199.
200.           last_yaw_error = yaw_error;
201.
202.           dernierTemps = maintenant;
203.         }
204.
205.       }
206.       void pid_LEVEL_compute() {
207.
208.         //ROLL calculs
209.         //definition du setpoint angle
210.         pid_roll_setpoint = 0;
211.         if (inputROLL > THROTTLE_RMID + 10)pid_roll_setpoint = inputROLL - THROTTLE
   _RMID + 10;
212.         else if (inputROLL < THROTTLE_RMID - 10)pid_roll_setpoint = inputROLL - THR
   OTTLE_RMID - 10;
213.         pid_roll_setpoint -
   = roll_angle_adjust;            //On soustrait roll adjust pour que le setpoint soit
    change avec langle
214.         pid_roll_setpoint /= 2;
215.
216.         roll_error = roll_speed - pid_roll_setpoint;
217.
218.         Proportional_roll = ROLL_PID_LEVEL_KP * roll_error;
219.         Integral_roll += ROLL_PID_LEVEL_KI * roll_error;
220.         if (Integral_roll > ROLL_PID_MAX)Integral_roll = ROLL_PID_MAX;
221.         else if (Integral_roll < ROLL_PID_MIN)Integral_roll = ROLL_PID_MIN;
222.         Derivative_roll = ROLL_PID_LEVEL_KD * (roll_error - last_roll_error);
223.
224.         pid_roll_out = Proportional_roll + Integral_roll + Derivative_roll;
225.         if (pid_roll_out > ROLL_PID_MAX)pid_roll_out = ROLL_PID_MAX;
226.         else if (pid_roll_out < ROLL_PID_MIN)pid_roll_out = ROLL_PID_MIN;
227.
228.         last_roll_error = roll_error;
229.
230.
231.         //PITCH calculs
232.         //definition du setpoint
233.         pid_pitch_setpoint = 0;
234.         if (inputPITCH > THROTTLE_RMID + 10)pid_pitch_setpoint = inputPITCH - THROT
   TLE_RMID + 10;
235.         else if (inputPITCH < THROTTLE_RMID - 10)pid_pitch_setpoint = inputPITCH -
   THROTTLE_RMID - 10;
236.         pid_pitch_setpoint -= pitch_angle_adjust;
237.         pid_pitch_setpoint /= 2;
238.
239.         pitch_error = pitch_speed - pid_pitch_setpoint;
```

```
240.
241.          Proportional_pitch = PITCH_PID_LEVEL_KP * pitch_error;
242.          Integral_pitch += PITCH_PID_LEVEL_KI * pitch_error;
243.          if (Integral_pitch > PITCH_PID_MAX)Integral_pitch = PITCH_PID_MAX;
244.          else if (Integral_pitch < PITCH_PID_MIN)Integral_pitch = PITCH_PID_MIN;
245.          Derivative_pitch = PITCH_PID_LEVEL_KD * (pitch_error - last_pitch_error);
246.
247.          pid_pitch_out = Proportional_pitch + Integral_pitch + Derivative_pitch;
248.          if (pid_pitch_out > PITCH_PID_MAX)pid_pitch_out = PITCH_PID_MAX;
249.          else if (pid_pitch_out < PITCH_PID_MIN)pid_pitch_out = PITCH_PID_MIN;
250.
251.          last_pitch_error = pitch_error;
252.
253.          //YAW calculs
254.          //On calcule le setpoint du yaw ici car il est le meme en stabilise ou en a
    cro
255.          pid_yaw_setpoint = 0;
256.          if (inputYAW > THROTTLE_RMID + 10)pid_yaw_setpoint = (inputYAW - THROTTLE_R
    MID + 10) / 2.0;
257.          else if (inputYAW < THROTTLE_RMID - 10)pid_yaw_setpoint = (inputYAW - THROT
    TLE_RMID - 10) / 2.0;
258.
259.          yaw_error = yaw_speed - pid_yaw_setpoint;
260.
261.          Proportional_yaw = YAW_PID_KP * yaw_error;
262.          Integral_yaw += YAW_PID_KI * yaw_error;
263.          if (Integral_yaw > YAW_PID_MAX)Integral_yaw = YAW_PID_MAX;
264.          else if (Integral_yaw < YAW_PID_MIN)Integral_yaw = YAW_PID_MIN;
265.          Derivative_yaw = YAW_PID_KD * (yaw_error - last_yaw_error);
266.
267.          pid_yaw_out = Proportional_yaw + Integral_yaw + Derivative_yaw;
268.          if (pid_yaw_out > YAW_PID_MAX)pid_yaw_out = YAW_PID_MAX;
269.          else if (pid_yaw_out < YAW_PID_MIN)pid_yaw_out = YAW_PID_MIN;
270.
271.          last_yaw_error = yaw_error;
272.
273.          maintenant = dernierTemps;
274.        }
275.
276.        // Fonction qui empeche les moteurs de tourner
277.        void motors_set_to_zero() {
278.          esc_1.writeMicroseconds(MOTOR_ZERO_LEVEL);
279.          esc_3.writeMicroseconds(MOTOR_ZERO_LEVEL);
280.          esc_2.writeMicroseconds(MOTOR_ZERO_LEVEL);
281.          esc_4.writeMicroseconds(MOTOR_ZERO_LEVEL);
282.        }
283.        ////////////////////////////////////////////////////////////////////////////////
    ////////////
284.        // Fonctions de debug
285.        ////////////////////////////////////////////////////////////////////////////////
    ////////////
286.        void print_motors() {
287.          Serial.print(motorFR);
288.          Serial.print(",");
289.          Serial.print(motorFL);
290.          Serial.print(",");
291.          Serial.print(motorBL);
292.          Serial.print(",");
293.          Serial.print(motorBR);
294.          Serial.print(",");
295.          Serial.print(roll_speed);
296.          Serial.print(",");
297.          Serial.print(pitch_speed);
298.          Serial.print(",");
299.          Serial.println(yaw_speed);
300.        }
```

```
301.
302.      void print_pid_out() {
303.        Serial.print(pid_roll_out);
304.        Serial.print(",");
305.        Serial.print(pid_pitch_out);
306.        Serial.print(",");
307.        Serial.print(pid_yaw_out);
308.        Serial.print(",");
309.        Serial.println(pitch_speed);
310.      }
311.
312.      void print_pid_setpoints() {
313.        Serial.print(pid_roll_setpoint);
314.        Serial.print(",");
315.        Serial.print(pid_pitch_setpoint);
316.        Serial.print(",");
317.        Serial.println(pid_yaw_setpoint);
318.      }
319.      void print_imu_speed() {
320.        Serial.print(roll_speed);
321.        Serial.print(",");
322.        Serial.print(pitch_speed);
323.        Serial.print(",");
324.        Serial.println(yaw_speed);
325.      }
326.      void print_imu_angle() {
327.        Serial.print(roll_angle);
328.        Serial.print(",");
329.        Serial.print(pitch_angle);
330.        Serial.print(",");
331.        Serial.println(yaw_angle);
332.      }
333.
334.      void print_rx() {
335.        Serial.print(inputARM);
336.        Serial.print(",");
337.        Serial.print(inputROLL);
338.        Serial.print(",");
339.        Serial.print(inputPITCH);
340.        Serial.print(",");
341.        Serial.print(inputTHROTTLE);
342.        Serial.print(",");
343.        Serial.println(inputYAW);
344.      }
345.
346.      void print_roll_pid() {
347.        Serial.print(Proportional_roll);
348.        Serial.print(",");
349.        Serial.print(Integral_roll);
350.        Serial.print(",");
351.        Serial.println(Derivative_roll);
352.      }
353.      void print_pid_to_setpoint() {
354.        Serial.print(roll_speed);
355.        Serial.print(",");
356.        Serial.println(pid_roll_setpoint);
357.
358.      }
359.
360.
361.      // Fonction qui s'execute une fois et au début
362.      void setup() {
363.        Serial.begin(115200);
364.        //BNO init
365.        bno.begin();
366.        bno.setExtCrystalUse(true);
```

```
367.
368.          //RX init
369.          attachInterrupt(RX_PIN_ARM, calcSignalARM, CHANGE);
370.          attachInterrupt(RX_PIN_ROLL, calcSignalROLL, CHANGE);
371.          attachInterrupt(RX_PIN_PITCH, calcSignalPITCH, CHANGE);
372.          attachInterrupt(RX_PIN_THROTTLE, calcSignalTHROTTLE, CHANGE);
373.          attachInterrupt(RX_PIN_YAW, calcSignalYAW, CHANGE);
374.
375.          //Motors init
376.          esc_1.attach(MOTOR_PIN_FRONT_RIGHT, MOTOR_ZERO_LEVEL, MOTOR_MAX_LEVEL);
377.          esc_2.attach(MOTOR_PIN_BACK_LEFT, MOTOR_ZERO_LEVEL, MOTOR_MAX_LEVEL);
378.          esc_3.attach(MOTOR_PIN_FRONT_LEFT, MOTOR_ZERO_LEVEL, MOTOR_MAX_LEVEL);
379.          esc_4.attach(MOTOR_PIN_BACK_RIGHT, MOTOR_ZERO_LEVEL, MOTOR_MAX_LEVEL);
380.
381.          esc_1.writeMicroseconds(MOTOR_ZERO_LEVEL);
382.          esc_2.writeMicroseconds(MOTOR_ZERO_LEVEL);
383.          esc_3.writeMicroseconds(MOTOR_ZERO_LEVEL);
384.          esc_4.writeMicroseconds(MOTOR_ZERO_LEVEL);
385.
386.          Proportional_roll, Integral_roll, Derivative_roll = 0;
387.          Proportional_pitch, Integral_pitch, Derivative_pitch = 0;
388.          Proportional_yaw, Integral_yaw, Derivative_yaw = 0;
389.        }
390.
391.
392.        // Fonction qui s'execute a chaque cycle
393.        void loop() {
394.
395.          //IMU calculs
396.          //vitesse angulaire
397.
398.          imu::Vector<3> gyroscope = bno.getVector(Adafruit_BNO055::VECTOR_GYROSCOPE)
    ;
399.          roll_speed = gyroscope.x() * 180 / pi;
400.          pitch_speed = -1 * gyroscope.y() * 180 / pi;
401.          yaw_speed = -1 * gyroscope.z() * 180 / pi;
402.          //angles d'euler
403.          imu::Vector<3> euler = bno.getVector(Adafruit_BNO055::VECTOR_EULER);
404.          roll_angle = -1 * euler.z();
405.          pitch_angle = -1 * euler.y();
406.          yaw_angle = euler.x();
407.
408.          //maintenant = 0;
409.          roll_angle_adjust = roll_angle * -
    15;  //la constante multipliee est a ajuster
410.          pitch_angle_adjust = pitch_angle * -
    15;//selon l'angle maximal que l'on veut
411.
412.          //PID
413.          if (auto_stabilisation_mode == false)pid_compute();
414.          else pid_LEVEL_compute();
415.
416.          //MOTORS
417.          throttle = inputTHROTTLE;
418.
419.          //ARM switch
420.          if (inputARM > 1500) {
421.            if(throttle > THROTTLE_WMAX)throttle = THROTTLE_WMAX; //Afin de laisser u
    n peu de controlle meme en full throttle ca fait 1850
422.            motorFR = throttle - pid_pitch_out + pid_roll_out - pid_yaw_out; //1
423.            motorBL = throttle + pid_pitch_out - pid_roll_out - pid_yaw_out; //2
424.            motorFL = throttle - pid_pitch_out - pid_roll_out + pid_yaw_out; //3
425.            motorBR = throttle + pid_pitch_out + pid_roll_out + pid_yaw_out; //4
426.
427.            if (motorFR > MOTOR_MAX_LEVEL) motorFR = MOTOR_MAX_LEVEL;
```

```
428.           if (motorFL > MOTOR_MAX_LEVEL) motorFL = MOTOR_MAX_LEVEL; //on ne veut pa
    s ecrire aux esc une valeur
429.           if (motorBL > MOTOR_MAX_LEVEL) motorBL = MOTOR_MAX_LEVEL; //plus grande q
    ue 2000
430.           if (motorBR > MOTOR_MAX_LEVEL) motorBR = MOTOR_MAX_LEVEL;
431.
432.           if (motorFR < MOTOR_ZERO_LEVEL + 95) motorFR = MOTOR_ZERO_LEVEL + 95;
433.           if (motorFL < MOTOR_ZERO_LEVEL + 95) motorFL = MOTOR_ZERO_LEVEL + 95; //h
    istoire que les moteurs tournent
434.           if (motorBL < MOTOR_ZERO_LEVEL + 95) motorBL = MOTOR_ZERO_LEVEL + 95; //q
    uand meme quand on arme
435.           if (motorBR < MOTOR_ZERO_LEVEL + 95) motorBR = MOTOR_ZERO_LEVEL + 95;
436.
437.           esc_1.writeMicroseconds(motorFR);
438.           esc_2.writeMicroseconds(motorBL);
439.           esc_3.writeMicroseconds(motorFL);
440.           esc_4.writeMicroseconds(motorBR);
441.
442.         } else if (inputARM < 1500) {
443.           motorFR = 1000;
444.           motorFL = 1000;
445.           motorBL = 1000;
446.           motorBR = 1000;
447.           esc_1.writeMicroseconds(motorFR);
448.           esc_2.writeMicroseconds(motorBL);
449.           esc_3.writeMicroseconds(motorFL);
450.           esc_4.writeMicroseconds(motorBR);
451.
452.           Proportional_roll, Integral_roll, Derivative_roll = 0;
453.           Proportional_pitch, Integral_pitch, Derivative_pitch = 0; //reset après d
    isarm
454.           Proportional_yaw, Integral_yaw, Derivative_yaw = 0;
455.         }
456.         print_imu_speed();
457.       }
```

config.h :

```
1.  ////////////////////////////////Définition des pins//////////////////////////////////
    /
2.
3.  ////////RX///////
4.  #define RX_PIN_ARM A4
5.  #define RX_PIN_ROLL A0
6.  #define RX_PIN_PITCH A1
7.  #define RX_PIN_THROTTLE A2
8.  #define RX_PIN_YAW A3
9.
10. ////////Motors////////
11. #define MOTOR_PIN_FRONT_RIGHT 4 //ESC1
12. #define MOTOR_PIN_BACK_LEFT 3 //ESC2
13. #define MOTOR_PIN_FRONT_LEFT 1 //ESC3
14. #define MOTOR_PIN_BACK_RIGHT 0 //ESC4
15.
16. ////////////////////////////////Définition des constantes//////////////////////////
    ////////////
17.
18. //////////////PID/////////////////
19. #define ROLL_PID_KP  0.3
20. #define ROLL_PID_KI  0
21. #define ROLL_PID_KD  0.7
22. #define ROLL_PID_MIN  -100.0
23. #define ROLL_PID_MAX  100.0
24.
25. #define PITCH_PID_KP  0.3
26. #define PITCH_PID_KI  0             ////toutes les K a verifier et tester
27. #define PITCH_PID_KD  0.7
28. #define PITCH_PID_MIN  -100.0
29. #define PITCH_PID_MAX  100.0
30.
31. #define YAW_PID_KP  0.7
32. #define YAW_PID_KI  0
33. #define YAW_PID_KD  0
34. #define YAW_PID_MIN  -100.0
35. #define YAW_PID_MAX  100.0
36.
37. ////////////////////////LEVEL PIDS
38. #define ROLL_PID_LEVEL_KP  0.4
39. #define ROLL_PID_LEVEL_KI  0.01
40. #define ROLL_PID_LEVEL_KD  2.2
41.
42. #define PITCH_PID_LEVEL_KP  0.4
43. #define PITCH_PID_LEVEL_KI  0.01             ////toutes les K a verifier et tester
44. #define PITCH_PID_LEVEL_KD  2.2
45.
46. ////////Config RX////////////////////
47. #define THROTTLE_RMIN  1000
48. #define THROTTLE_WMIN  1100
49. #define THROTTLE_SAFE_SHUTOFF 1120
50. #define THROTTLE_RMAX  2000
51. #define THROTTLE_WMAX  1850
52. #define THROTTLE_RMID  1500
53.
54. #define ROLL_RMIN  THROTTLE_RMIN
55. #define ROLL_RMAX  THROTTLE_RMAX
56.
57.
58. #define PITCH_RMIN  THROTTLE_RMIN
59. #define PITCH_RMAX  THROTTLE_RMAX
60.
```

```
61.
62. #define YAW_RMIN  THROTTLE_RMIN
63. #define YAW_RMAX  THROTTLE_RMAX
64.
65. //////////////Variable des moteurs////////////
66. #define MOTOR_ZERO_LEVEL  1000
67. #define MOTOR_ARM_START  1100
68. #define MOTOR_MAX_LEVEL  2000
```

Images du signal PWM (division verticale de 1V, et division horizontale de 1000µs) :

Ici, le signal est haut pendant une division verticale, soit 1000µs.

Ici, le signal est haut pendant deux divisions verticales, soit 2000µs.