INSA ROUEN
À taille humaine,
à l'échelle du monde

Projet de Physique P6
STPI/P6/2014

QUADRICOPTER : ASSERVISSEMENT D'UNE BRANCHE

Enseignant responsable
Corentin JOUEN

Étudiants :
Chloé DARDARD
Tanguy MOREAU
Nicolas THÉRON
Antoine JEANMOUGIN
Leslie RIALET

Date de remise du rapport : 16/06/14

Référence du projet : STPI[1]/P6/2014

Intitulé du projet : Quadricopter : asservissement d'une branche

Type de projet : expérimental

Objectifs du projet :

Le but de notre projet se résume à l'asservissement d'une branche d'un quadricopter. Plus précisément cela consiste à maintenir cette branche à un état d'équilibre, c'est-à-dire parfaitement parallèle par rapport au sol, et ce même après qu'un effort ait été appliqué sur celle-ci pour la déséquilibrer.

Pour ce faire, il nous a fallu nous familiariser avec les différents composants et acquérir des connaissances en électronique, en programmation et en automatisme, de façon à pouvoir maîtriser au mieux cet appareil et nous rapprocher de notre objectif.

Remerciements :

Nous tenons à remercier notre professeur, M.JOUEN, ainsi que les techniciens de STPI M.WILLIAMS et Mme RADE, qui nous ont aidé tout au long de notre projet.

1. INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE ROUEN
DÉPARTEMENT SCIENCES ET TECHNIQUES POUR L'INGÉNIEUR
685 AVENUE DE L'UNIVERSITÉ BP 08- 76801 SAINT-ETIENNE-DU-ROUVRAY
TÉL : 33 2 32 95 66 21 - FAX : 33 2 32 95 66 31

Table des matières

Acronymes

CC : Courant Continu

PWM/MLI : Pulse-Width Modulation, ou modulation de largeur d'impulsion

FIFO : First In, First Out, ou premier entré, premier sorti

UART : Universal Asynchronous Receiver Transmitter, ou émetteur-récepteur asynchrone universel

MEMS : MicroElectroMechanical Systems, ou systèmes microélectromécaniques

Introduction

Dans le cadre de notre cursus de deuxième année à l'INSA de Rouen, nous devions réaliser un projet de physique mettant en application les compétences acquises pendant nos deux premières années de cycle préparatoire.

Nous avons choisi comme sujet « Quadricopter : asservissement d'une branche ». Les quadricopter sont des aéronefs équipés de quatre moteurs, quatre hélices et de deux branches. Notre groupe est constitué de cinq étudiants dont l'objectif est d'équilibrer une branche, c'est-à-dire que lorsque l'on applique une force sur une branche, le quadricopter doit revenir dans sa position d'équilibre.

Pour mener à bien notre projet, nous nous réunissions le mercredi de 8h à 9h30, en compagnie de M.JOUEN. Afin d'avancer le plus efficacement possible, nous consacrions notre jeudi après-midi à nos recherches sur les différents composants d'un quadricopter qui nous étaient alors inconnus, au principe de régulation par la méthode PID, puis à nos expériences dont le but était d'améliorer l'équilibre de la branche.

Ce projet s'est décomposé en plusieurs phases : tout d'abord une phase de documentation, notamment sur le fonctionnement de la carte Arduino, puis une importante phase de programmation. Il s'inscrit parfaitement dans notre formation d'ingénieur puisqu'il associe travail de groupe, autonomie, répartition des tâches et respect des échéances imposées.

Nous présenterons, dans un premier temps, les différents composants de notre quadricopter, la régulation PID ainsi que notre programme. Puis nous exposerons les problèmes que nous avons pu rencontrer et les solutions trouvées.

Chapitre 1

Méthodologie, organisation du travail

1.1 Méthodologie

Au début du projet, nous n'avions qu'une idée vague de ce que nous allions devoir faire. Nous étions tous curieux de découvrir ce que signifiait l'intitulé du projet : « asservissement d'une branche ».

Quand l'encadrant nous a expliqué qu'il fallait que nous réussissions à faire en sorte que le quadricopter revienne à sa position d'équilibre après un déséquilibre, nous nous sommes dit que nous devrions réussir en peu de temps. Mais après trois ou quatre semaines, nous nous sommes rendus compte que ce ne serait pas si simple que cela.

Nous avions déjà fait effectué tous les branchements et fait quelques tests quand nous avons réalisé que nous allions avoir besoin d'une régulation, et plus particulièrement d'une régulation PID, dont le principe sera expliqué plus loin dans ce rapport. Cela ne nous a pas paru très compliqué au premier abord. Il suffisait de recopier un bout de code que nous avions trouvé sur internet, et de trouver les valeurs de trois constantes permettant la régulation. Nous avons décidé de chercher ces valeurs expérimentalement, car il suffisait de faire des tests pour trouver les valeurs adéquates.

Or, nous nous sommes vite rendus compte que les constantes n'allaient pas être si simples que cela à trouver. Cette recherche nous a occupé pendant quasiment tout le reste de notre projet. En effet, nous détections des erreurs dans le code assez souvent, qui nous ont empêché d'avancer aussi vite que nous le souhaitions. Même en nous réunissant en dehors des heures de projet, nous n'avons pas réussi à trouver les constantes idéales, même si nous avons fait des progrès depuis le début de nos recherches.

Nous avions espéré avancer beaucoup plus dans ce projet, mais être motivés ne nous a pas aidé à trouver les constantes. Et, après être restés bloqués autant de temps sur ce point, nous n'avons pas pu réaliser tout ce que nous voulions. Nous aurions au moins aimé réussir à faire voler notre quadricopter, mais pour cela il aurait fallu asservir la deuxième branche, et ensuite effectuer d'autres réglages pour faire en sorte que le quadricopter arrive à décoller. Nous avions aussi espérer au tout début du projet pouvoir le télécommander, mais nous nous sommes vite aperçus que nous n'aurions pas assez de temps.

1.2 Organisation du travail

Nous avons créé un groupe sur Facebook pour pouvoir communiquer en dehors des heures de projet. Celui-ci nous a également permis d'échanger des informations ou des documents, comme les comptes-rendus que nous rendions chaque semaine et que tout le monde lisait avant qu'il ne soit envoyé à notre professeur. Il nous a aussi permis de corriger les différentes parties du rapport au fur et à mesure que chacun rédigeait sa partie.

De plus, nous nous sommes souvent réunis le jeudi après-midi, car nous n'aurions jamais autant avancé si nous nous étions juste retrouvés sur les heures de projet initialement prévues.

Nous nous sommes réparti le travail comme l'explique l'organigramme suivant, afin d'être le plus efficace possible.

FIGURE 1.1 – Organigramme

Chapitre 2

Travail réalisé et résultats

2.1 Présentation des composants

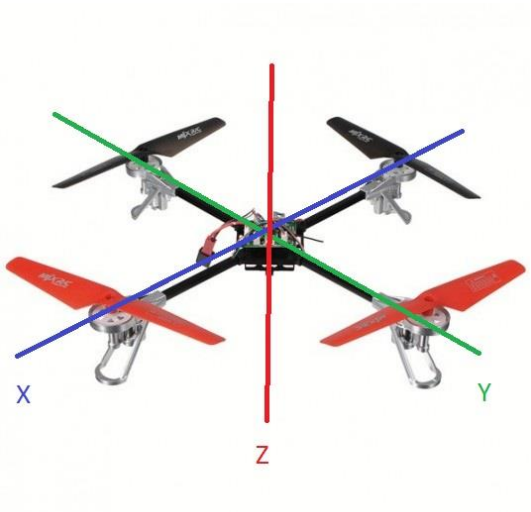
2.1.1 Le quadricopter

Le quadricopter est un aéronef qui a pour particularité de posséder quatre rotors qui lui permettent de voler, de se diriger et de se stabiliser. Contrairement à un hélicoptère, les hélices de cet appareil sont contenues dans un plan parallèle à celui du sol.

Mais avant de se lancer dans l'explication du fonctionnement d'un quadricopter, il est nécessaire de donner quelques définitions générales concernant les mouvements possibles qu'il peut effectuer.

Les mouvements en question sont ceux définis par trois axes (représentés sur la photo ci-contre) :

- Le Roulis décrit la rotation selon l'axe X (c'est-à-dire l'axe de vol), et permet à l'appareil d'effectuer un mouvement vers les côtés (à gauche ou à droite) ;
- Le Tangage permet au quadricopter de s'incliner vers l'avant ou bien vers l'arrière, cette inclinaison se fait par rapport à l'axe Y ;
- Enfin, le Lacet autorise un mouvement de rotation selon l'axe Z (axe vertical au sol) et induit donc l'aéronef à changer de direction en pivotant sur lui-même.



Remarque : il est à noter que l'avant et l'arrière du quadricopter ont été choisis de façon arbitraire selon l'axe X sur ce schéma afin de faciliter la compréhension des définitions.

Maintenant nous pouvons nous intéresser au principe de fonctionnement d'un tel engin. Un quadricopter simple est composé d'une base au centre de laquelle partent quatre branches au bout desquelles est rattaché un moteur qui entraîne une hélice. Ces branches sont perpendiculaires les unes aux autres, et les hélices fonctionnent en couple ; plus précisément les deux hélices contenues sur un même axe (X ou Y) tournent dans le même sens. Ainsi un des couples d'hélices tournent dans le sens horaire et l'autre dans le sens anti-horaire. Cela permet au quadricopter de rester stable en compensant les effets des moteurs,

et crée une force de sustentation qui le fait voler.

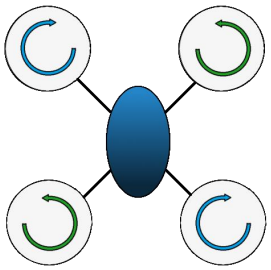


FIGURE 2.1 – Sens de rotation des hélices

Tout ceci étant dit, nous pouvons à présent expliquer comment un quadricopter arrive à se stabiliser et de quelle manière il arrive à effectuer les trois mouvements décrits précédemment. Dans les deux cas, cela est possible grâce à des accéléromètres et des gyroscopes qui sont en général placés au niveau de la base. Ces derniers sont capables de capter des informations concernant la position du quadricopter ainsi que sur les mouvements qu'il effectue. Dans le cas qui nous intéresse en premier lieu, ces informations vont être utilisées par le programme pour faire en sorte que l'appareil se stabilise parfaitement.

Pour ce faire, le principe consiste à jouer sur la puissance délivrée aux différents moteurs qui vont enclencher les mouvements désirés afin d'arriver à terme à l'objectif final (dans le cadre de notre projet à la stabilisation horizontale).

Par exemple pour créer un mouvement de tangage (selon l'axe Y), il va falloir modifier les vitesses des hélices contenues sur l'axe X, en diminuant celle d'une des deux hélices et en accélérant l'autre, ainsi le quadricopter penchera soit vers l'avant soit vers l'arrière, ou bien s'il est déjà penché cela lui permettra de revenir vers sa position d'équilibre. Il existe donc pour chacun des trois mouvements des spécificités sur le jeu de couple des moteurs.

Enfin, sur tout le quadricopter il existe un système de régulation qui permet d'obtenir de meilleurs résultats par rapport aux attentes souhaitées, mais nous en parlerons dans la suite de ce rapport.

2.1.2 La carte Arduino

La carte Arduino est le composant principal de notre quadricopter pour effectuer toutes sortes d'actions (démarrer, voler, s'équilibrer...). La carte Arduino entre dans la catégorie des microcontrôleurs, c'est-à-dire d'ordinateur miniature capable de stocker des données, de recevoir des informations et d'en envoyer. Notre modèle de carte Arduino est le MEGA 2560. C'est elle qui nous permet, grâce à différents branchements, de communiquer avec les moteurs et le gyroscope. Par le biais de l'ordinateur nous téléversons un programme composé par notre équipe dans la carte Arduino. Ce programme récupère les données envoyées par le gyroscope (mesures d'angle) et en fonction de ces données, envoie des informations aux moteurs dans le but de stabiliser le quadricopter.

La carte Arduino a cependant de nombreuses autres utilités, elle est composée de 54 entrées/sorties digitales (dont 15 utilisables en PWM), 16 entrées analogiques, 4 UARTs, un oscillateur de 16 MHz, une connexion USB et une jack, ainsi qu'un bouton « reset ».

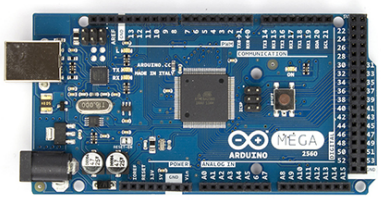


FIGURE 2.2 – La carte Arduino Mega 2560

Nous utilisons le système PWM qui permet d'envoyer des données non pas binaires (on/off) mais modulables (entre 0 et 255) ce qui a pour effet de faire tourner les moteurs à des vitesses très variables (et pas seulement deux états : allumé/éteint) afin d'atteindre un état d'équilibre beaucoup plus rapidement.

2.1.3 Le gyroscope

Le gyroscope MPU 6050 possède 6 axes. Ce composant nous permet de repérer le quadricopter dans l'espace. Sa puce MEMS est très précise avec une conversion analogique-digitale sur 16 bits, et une interface de 400 kHz. Le composant est contrôlé par l'Arduino. Le gyroscope contient un registre FIFO de 1024 octets que l'Arduino peut lire si un signal d'interruption est donné. Le gyroscope nous donne à la base une vitesse angulaire de rotation en degrés/seconde selon les 3 axes cartésiens de l'espace. Or, dans notre cas nous nous intéressons aux angles d'orientation. Ces angles s'obtiennent en intégrant dans le temps la vitesse angulaire.

FIGURE 2.3 – Le gyroscope

Ce composant est très complet, cependant nous n'utiliserons pas toutes ses capacités. En effet il se compose aussi d'un accéléromètre qui renvoie une accélération en m^2/s. Dans notre cas il est inutile puisque notre quadricopter reste sur le portique, mais il aurait pu nous permettre de connaître la vitesse de notre quadricopter en intégrant dans cette accélération le temps.

De plus, avec les 3 axes restants, le gyroscope aurait pu contrôler un magnétomètre 3 axes (mesure du champ magnétique terrestre) et ainsi nous aurions été en mesure d'orienter notre quadricopter dans l'espace (Nord Sud Est Ouest).

2.1.4 Le contrôleur de moteurs

Le contrôleur contient une double commande permettant de contrôler deux moteurs CC à partir d'une sortie PWM d'un microcontrôleur (ex : carte Arduino). Ce module est basé sur un L298N monté sur un refroidisseur permettant une grande puissance de sortie. Le L298N est en double pont en H, il permet d'inverser le sens de rotation du moteur, en inversant la tension aux bornes du moteur et la variation de la vitesse du moteur en modulant la tension aux bornes du moteur. Il convient pour les charges inductives tels que les solénoïdes, les relais ou les moteurs. La carte possède un connecteur d'alimentation avec une alimentation pour la logique, une autre pour les moteurs et une masse commune.

Ce contrôleur ne convient que pour les moteurs à courant continu.

FIGURE 2.4 – Le contrôleur de moteurs

2.2 Corps du projet

2.2.1 Préparation du quadricopter

Avant de pouvoir procéder aux premiers tests, nous avons dû en quelque sorte préparer le quadricopter, c'est-à-dire lui fabriquer un portique, effectuer les soudures entre les différents composants puis faire les branchements.

Pour le portique, nous avons demandé aux techniciens s'ils avaient du matériel à nous donner pour que nous puissions le construire. Quand nous sommes arrivés en cours la semaine d'après, ils nous avaient gentiment fabriqué notre portique. Cependant, celui-ci avait été endommagé lorsque nous sommes revenus la semaine d'après, ils nous ont donc fabriqué un autre portique, plus solide que le premier.

FIGURE 2.5 – Portique

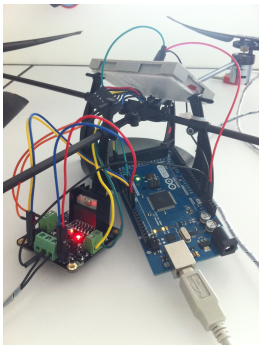


FIGURE 2.6 – Branchements entre les composants

Nous en avions besoin pour pouvoir maintenir une branche du quadricopter immobile pendant que nous faisions les tests sur l'autre.

Il a également fallu que nous fassions des soudures entre différents petits composants.

Ensuite il a fallu effectuer tous les branchements entre la carte Arduino, le gyroscope et les contrôleurs de moteurs.

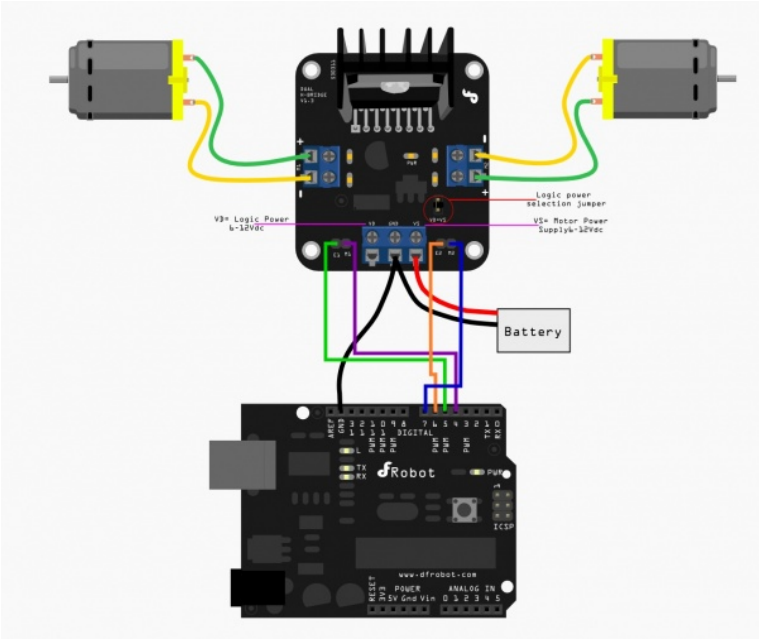


FIGURE 2.7 – Schéma des branchements avec la carte Arduino

Toutes ces étapes étaient nécessaires avant de pouvoir commencer à effectuer des tests.

2.2.2 Régulation PID

Pourquoi doit-on réguler ?

Nous nous sommes aperçu au cours de notre projet qu'il était difficile, et même en réalité impossible, d'obtenir un équilibre parfait et satisfaisant. Le fait d'ordonner simplement au programme de faire varier la puissance des moteurs d'une branche en fonction de l'angle qu'il peut y avoir ne suffit pas à parvenir au résultat souhaité et crée une erreur ou imper-

fection. Par exemple le système peut continuer à osciller autour de la valeur d'angle recherchée. Le but de la régulation est donc de prendre en compte ces imperfections par rapport à la consigne de départ pour y apporter une correction : c'est la définition même d'un asservissement !

Dans le cas de notre projet, nous travaillions en boucle fermée. Ce type de contrôle consiste à considérer les informations antérieures sur l'état du système pour les utiliser afin d'ajuster la commande, c'est-à-dire de prendre en compte les valeurs en sortie du système pour les réinjecter en entrée. Ce contrôle en boucle fermée s'oppose à celui en boucle ouverte qui lui ne prend pas en compte la réponse du système : par exemple un système d'arrosage automatique qui ne considère pas l'humidité ambiante fonctionne en boucle ouverte, il arrosera même s'il pleut !

Pourquoi avoir choisi la méthode du PID ?

La régulation PID est ce que l'on appelle un « organe de contrôle » dans un système. Celui-ci s'utilise en boucle fermée et convient tout à fait à nos besoins. De plus, c'est la méthode de régulation la plus utilisée dans l'industrie aujourd'hui car elle permet de résoudre un nombre important de problèmes : trouver de la documentation sur cette méthode n'est donc pas difficile. Mais la principale raison qui fait que nous avons choisi d'utiliser ce type de régulation se base sur le fait qu'elle s'applique de manière empirique : c'est-à-dire que l'on peut, une fois que l'algorithme a été intégré au code, l'améliorer de façon expérimentale. Ainsi aucune connaissance poussée et précise n'est nécessaire pour pouvoir réguler son système et parvenir à l'objectif principal.

Comment fonctionne cette méthode ?

Avant de se lancer dans l'explication directe de la régulation PID nous trouvons intéressant de vous montrer le coté concret et intuitif qu'elle présente à travers un exemple souvent utilisé pour introduire le sujet.

Cet exemple décrit la façon dont un conducteur opère pour stabiliser sa vitesse à 130 km/h par exemple. Sa première action va être d'accélérer franchement pour atteindre cette vitesse, en relâchant tout de même la pression lorsqu'il se rapproche des 130 km/h, et arrête d'appuyer sur la pédale au moment où la vitesse est atteinte (sinon il aura une amende...). Que se passe-t-il alors ? La voiture va voir sa vitesse diminuer quelque peu jusqu'à ce que le conducteur enclenche à nouveau une accélération très faible de façon à stabiliser le véhicule à 120 km/h (pourquoi pas !). Pour améliorer cela, l'idée est de se dire que plus le conducteur reste longtemps en-dessous de la vitesse recherchée, plus il va accélérer. En quelque sorte il garde en mémoire l'état dans lequel il se trouve et agit en conséquence : si l'on reprend là où l'on s'est arrêté, le conducteur à 120 km/h va donc continuer à accélérer jusqu'à 130 km/h. A cette vitesse exactement il va continuer d'accélérer car même si l'erreur est nulle, il a en mémoire une erreur due aux instants précédents où il était en- dessous de la vitesse. Ainsi au moment où il aura dépassé la limite, cette erreur s'inversera et donc il va décélérer, et ainsi de suite avant de se stabiliser à 130 km/h. Cependant, pour des raisons de légalité (dans notre pays en tout cas) il va voir apparaître des oscillations non souhaitées. Ainsi, pour éviter cela, le conducteur va prendre en compte le fait qu'il se rapproche de la vitesse voulue et donc il va accélérer de moins en moins. Les oscillations vont diminuer, et le résultat est atteint !

Cet exemple est souvent utilisé pour introduire l'explication. En effet, nous pouvons féliciter le conducteur, car intuitivement il a réalisé une régulation de type « PID » de sa vitesse. Malheureusement si l'Homme est capable de le faire sans même en être conscient, l'informatique ne l'est pas... Nous allons donc à présent expliquer comment fonctionne précisément cette méthode.

A l'aide du graphique ci-dessous nous allons pouvoir définir visuellement les notions suivantes :

— La Consigne : valeur souhaitée ;
— Le Dépassement : plus grosse amplitude entre la consigne et la valeur mesurée en régime transitoire ;
— L'Erreur Statique : écart entre la consigne et la valeur réelle mesurée en régime permanent ou stationnaire ;
— Le Temps de montée : durée que met le signal pour passer de 10% à 90% de sa valeur
— Le temps d'établissement du régime stationnaire : correspond au temps que met le système pour atteindre le régime permanent.

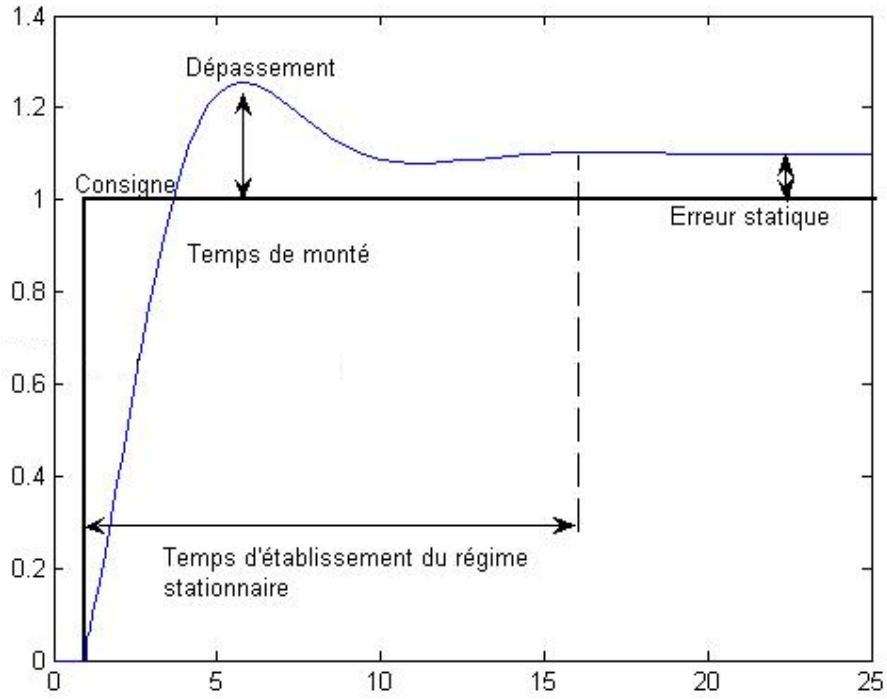


FIGURE 2.8 – Graphique d'explication du PID

La régulation PID est désignée par ses trois initiales (P : Proportionnel, I : Intégral, D : Dérivé) et fait intervenir trois constantes qui vont avoir un rôle bien précis dans la régulation. Nous les noterons *Kp*, *Ki* et *Kd*.

Le régulateur Proportionnel :

Le coefficient *Kp* a une importance majeure dans le sens où il constitue la base de la régulation. Il va permettre au système d'avoir de la puissance. Dans le cas de l'exemple ci-dessus, cela peut être comparé à l'accélération que produit le conducteur en appuyant sur la pédale pour atteindre la vitesse. Dans le cadre de notre sujet cela donne aux moteurs la possibilité d'avoir de la puissance et donc de la vitesse. Il existe une relation entre la commande et le coefficient *Kp*.

La commande peut alors s'écrire :

$$commande = Kp * erreur$$

Il est donc évident de s'apercevoir que lorsque l'on augmente *Kp* cela va avoir pour effet d'augmenter la commande et donc la rapidité de la réponse, sauf qu'il y aura des oscillations et l'apparition d'une erreur statique. A l'inverse, un *Kp* trop faible aura comme effet de ne pas pouvoir lancer les moteurs.

Le régulateur Proportionnel Intégral :

Ce terme est aussi important dans la régulation car il permet en premier lieu d'éliminer l'erreur statique amenée par le terme Proportionnel. Il fait intervenir la notion de temps dans le sens où il intègre les erreurs précédant un instant précis. L'addition de ces erreurs multipliée par *Ki* en fait une moyenne qui est prise en compte dans la régulation. En trouvant cette moyenne on est capable de savoir quel est l'écart avec la valeur de la consigne .
La commande s'écrit alors :

$$commande = Kp * erreur + \underbrace{Ki * \underbrace{sommeDesErreurs}_{\to 0}}_{\text{Le terme tend à se stabiliser}}$$

Dans le cas de notre exemple, il est utile pour faire en sorte que le conducteur ne se stabilise pas en-dessous de la valeur souhaitée, mais bel et bien à 130 km/h. Dans le cas de notre projet ce terme permet de ne pas avoir d'erreur statique, et de bien se stabiliser autour d'un angle nul avec l'horizontale !
Il reste tout de même un problème : en faisant ceci, il reste des oscillations...

Le régulateur Proportionnel Dérivé :

Faisant également intervenir le temps, l'action dérivée a comme effet d'anticiper l'état de l'erreur dans le futur. Elle est un indicateur de l'erreur qu'il peut y avoir d'un échantillon à un autre.
La commande s'écrit :

$$commande = Kp * erreur + Kd * (erreur - erreurPrcdente)$$

Cela a pour effet de limiter les dépassements ou les oscillations.

Le régulateur Proportionnel Intégral Dérivé :

Finalement, une fois que tous les termes sont ajoutés les uns aux autres, la commande s'écrit :

$$commande = Kp * erreur + Ki * sommeDesErreurs + Kd * (erreur - erreurPrcdente)$$

En conclusion, la méthode de régulation PID s'appuie sur trois constantes Kp, Ki, Kd qui permettent à un système d'atteindre un état souhaité. Nous expliquerons plus loin dans ce rapport comment trouver ces coefficients.

2.2.3 Programmation

Le programme est au cœur du problème de la stabilisation du quadricopter. C'est lui qui va envoyer les ordres, recevoir les informations et réagir en conséquence. C'est lui le « cerveau » de l'opération qui nous permet d'agir sur le quadricopter. Il doit être parfait pour assurer une stabilité sans faille au quadricopter présumé en vol. Le moindre degré de trop entraînerait irrémédiablement une chute et des dommages non négligeables.

Nous avons donc passé la plupart de notre temps de projet sur ce programme, nous avons tout d'abord utilisé comme base l'exemple donné dans la bibliothèque Arduino sur l'utilisation du gyroscope. Nous avons ensuite couplé ce programme avec celui, déjà écrit par nos soins, de contrôle des moteurs. Arrivés à ce point, nous avons essayé différents parcours, modifiant le code au fil des problèmes sans cesse rencontrés. Et nous pouvons facilement dire que d'une séance à l'autre, le code a toujours été en constante évolution.

Le but général du code est donc à chaque instant de récupérer l'angle d'inclinaison, en degrés, du quadricopter puis, à l'aide de l'implémentation PID, de corriger l'erreur en jouant sur la vitesse de rotation des moteurs afin d'équilibrer à 0 degré le quadricopter.

Approchons-nous maintenant de plus près du fonctionnement du code.

Il commence par définir les outils dont nous aurons besoin dans la suite du programme, ici la bibliothèque permettant l'utilisation du gyroscope (MPU6050) et la récupération des différentes valeurs d'angle converties en degrés (Yaw, Pitch, Roll).

```
MPU6050 mpu;

// uncomment "OUTPUT_READABLE_YAWPITCHROLL" if you want to see the yaw/
// pitch/roll angles (in degrees) calculated from the quaternions coming
// from the FIFO. Note this also requires gravity vector calculations.
// Also note that yaw/pitch/roll angles suffer from gimbal lock (for
// more info, see: http://en.wikipedia.org/wiki/Gimbal_lock)
#define OUTPUT_READABLE_YAWPITCHROLL
```

FIGURE 2.9 – Extrait code 1

Puis l'initialisation de toutes les valeurs de nos constantes, du PID, des branchements moteurs, et de toutes les autres manipulations que nous avons dû utiliser.

Dans la suite du programme nous allons seulement détailler la partie concernant le PID et la réception/envoi des valeurs.

Voilà cependant un rapide résumé de ce que font les autres parties (moins intéressantes pour notre projet) :

— La partie *Initial Setup* lance la procédure de réception des valeurs du gyroscope, elle prévoit les cas où le gyroscope pourrait avoir des problèmes. Généralement tout se passe bien et nous n'avons plus qu'à presser une touche et à appuyer sur entrée pour que la suite du programme démarre.

— La deuxième partie de *Main Program Loop (Second part)* sert à vider la mémoire si elle est trop pleine, et la réinitialiser vide mais aussi à enlever les valeurs qui ont déjà été lues. Enfin la dernière partie permet la lecture des valeurs envoyées par le gyroscope : Le lacet, le tangage et le roulis (Yaw, Pitch, Roll en anglais).

```
//Motors variables
int E1 = 5;
int M1 = 4;
int E2 = 6;
int M2 = 7;
float velocity1 =0;
float velocity2 =0;
float erreur = 0;
float somme_erreurs = 0;
float variation_erreur = 0;
float erreur_precedente = 0;
int mapage = 257;
unsigned char RealVelocity1 =0;
unsigned char RealVelocity2 = 0;
float test = 0;
float value2 = 0;
float commande = 0;
float Kp =  0.0035;
float Ki = 0.00001; //0.000029;
float Kd = 0.07;
float value =0;
long dure, dure_prec=0;
char boucle=0;
```

FIGURE 2.10 – Extrait code 2

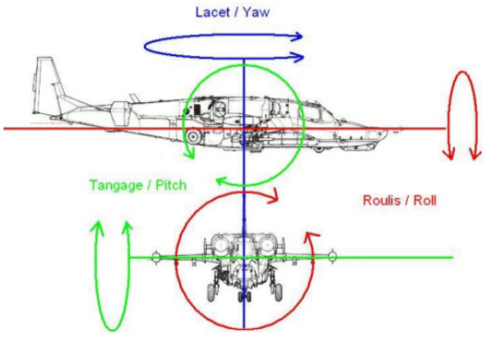


FIGURE 2.11 – Informations gyroscope

Intéressons-nous maintenant au cœur de la stabilisation de notre quadricopter, l'implémentation du PID et son usage. La valeur nommée value correspond à l'angle reçu par notre gyroscope et préalablement convertie en degrés. La valeur ypr[2] est la valeur d'angle détectée par notre gyroscope, ici le chiffre 2 correspond au roulis qui est l'angle qui nous intéresse.

```
value = ypr[2]* 180/M_PI;
```

FIGURE 2.12 – Extrait code 3

Implémentation PID :

```
erreur = value;
somme_erreurs += erreur;
variation_erreur = erreur - erreur_precedente;
commande = Kp*erreur + Ki*somme_erreurs + Kd*variation_erreur;
erreur_precedente = erreur;
```

FIGURE 2.13 – Extrait code 4

Nous attribuons donc la valeur d'angle à notre erreur qui est dès lors notre fonction f. Ainsi la somme_erreurs est l'intégrale de cette fonction F ; et la variation_erreur est la dérivée de cette fonction : f′ . La valeur finale de commande incrémentée des coefficients adéquats (Kp, Ki, Kd) est donc la valeur de la régulation PID, c'est elle qui va modifier la vitesse des moteurs dans la suite du programme.

Les valeurs velocity1 et velocity2 correspondent aux moteurs gauche et droit du quadricopter. L'une se verra attribuer « plus la commande » tandis que l'autre aura « moins la commande ».

```
velocity1 -= commande;
velocity2 += commande;
```

FIGURE 2.14 – Extrait code 5

En fonction du signe de la commande les moteurs vont accélérer et ralentir pour modifier l'angle du quadricopter qui aura alors une nouvelle valeur d'angle. Celle-ci sera détectée par le gyroscope, convertie en degrés, transformée en commande par la réglementation PID et qui amènera les moteurs à changer leurs vitesses afin de trouver une position d'équilibre le plus rapidement possible.

Plusieurs parties du code autour de l'implémentation PID ont été écrites afin d'ignorer certaines valeurs aberrantes liées à des erreurs de lecture du gyroscope, ou bien de ralentir la régulation PID, parce que trop efficace, ce qui nuisait à une bonne stabilité du quadricopter (la régulation était si rapide que les moteurs étaient tour à tour allumé à fond ou éteint).

2.2.4 Résultats expérimentaux

Comme nous l'avons expliqué précédemment, un des avantages de la régulation PID est qu'elle peut être mise en place de façon expérimentale, « à la main ». Pour ce faire, nous avions à disposition le programme au complet contenant bien sûr le bout de code concernant la régulation. Il ne restait alors plus qu'à trouver les valeurs des constantes qui maintenaient le système à l'équilibre.

Au cours de nos recherches, nous avons compris que la méthode la plus efficace consistait à s'occuper de chaque paramètre (proportionnel, intégral, dérivé) chacun son tour. En effet, puisque chacun d'entre eux joue un rôle particulier, il est donc logique de penser qu'en ne s'occupant que d'un à la fois nous allons pouvoir régler un des problèmes que nous voulons solutionner pour arriver au final au résultat attendu.

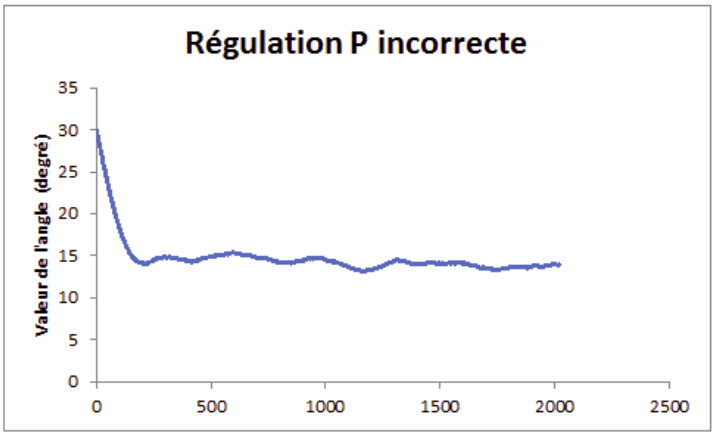
Cependant, nous nous sommes aperçus que cela n'est pas entièrement vrai. Effectivement, chaque terme de la régulation peut avoir des effets secondaires en quelque sorte, et peut perturber le système. Il a donc paru évident que nous devions procéder à des ajustements des valeurs des constantes de façon à améliorer nos résultats.

Paramètre	Temps de montée	Erreur statique	Dépassement
Kp	Augmente	Diminue	Augmente
Ki	Diminue	Elimine	Augmente
Kd	Change faiblement	Change faiblement	Diminue

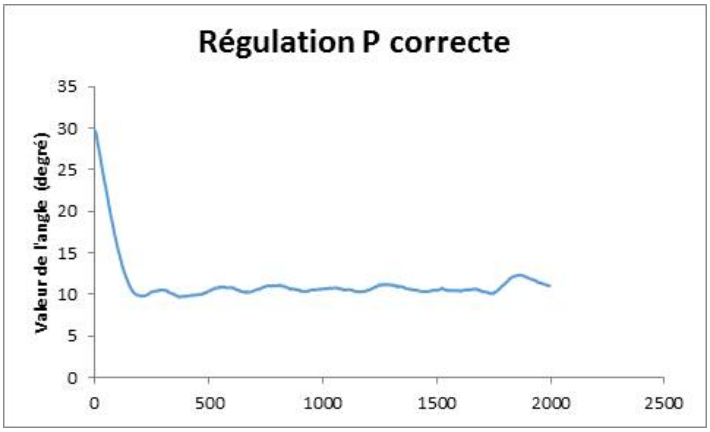
FIGURE 2.15 – Effets de l'augmentation des paramètres

Maintenant intéressons-nous plus précisément à la façon d'obtenir ses constantes.

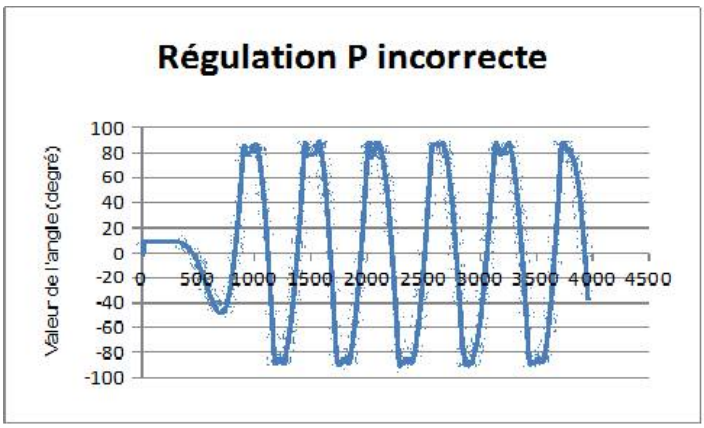
La première étape consistait déjà à donner l'impulsion aux moteurs pour qu'ils puissent tourner et entrainer les hélices. Pour cela nous devions jouer sur les valeurs du coefficient Kp, étant donné que c'est lui qui s'occupe de donner de la puissance ! Nous avons donc initialisé les deux autres constantes à des valeurs nulles, puis nous avons effectué plusieurs essais en changeant le terme proportionnel. Nous souhaitions donc en faisant ceci obtenir un système qui ait assez de force pour démarrer, mais en même temps nous ne voulions pas que le système ne s'emballe et qu'il y ait des oscillations trop importantes.



Kp= 0,01
Ki= 0
Kd= 0



Kp= 0,015
Ki= 0
Kd= 0

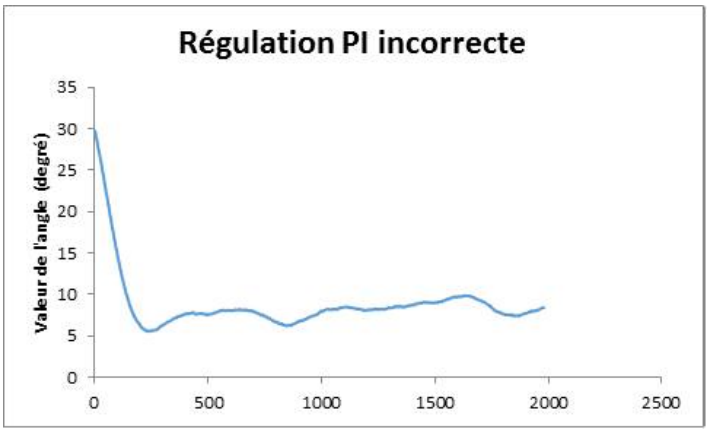


Kp= 1
Ki= 0
Kd= 0

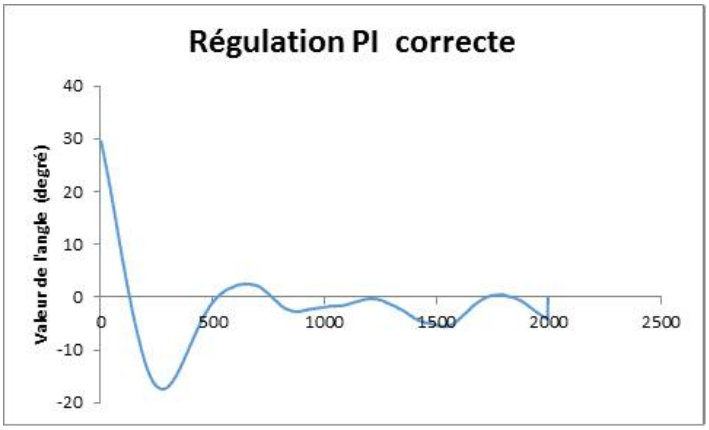
Sur les trois courbes ci-dessus représentant différentes réponses du sytème en fonction du coefficient Kp, on peut voir que lorsque ce dernier est trop faible le système peine à démarrer, mais que lorsqu'il est trop grand l'ensemble devient incontrôlable. Nous avons donc choisi une valeur intermédiaire, même si la branche du quadricopter se stabilise autour d'un angle de 10 degrés.

Ensuite nous nous sommes aperçus que l'appareil (après quelques secondes) arrivait à se stabiliser à peu près correctement, mais pas autour de la valeur recherchée, c'est-à-dire qu'il ne se stabilisait pas horizontalement : c'était notre erreur statique. En plus du coefficient Kp, il fallait donc utiliser les effets positifs du terme Intégral de façon à inhiber complètement cette erreur. Cependant, il fallait faire attention à ne pas faire diminuer le temps de montée de la réponse, car nous voulions que le système s'équilibre assez vite !

Nous avons donc commencé par initialiser Ki à une valeur très petite, puis nous l'avons augmenté progressivement de façon à obtenir une erreur statique qui soit inférieure à 5 degrés.



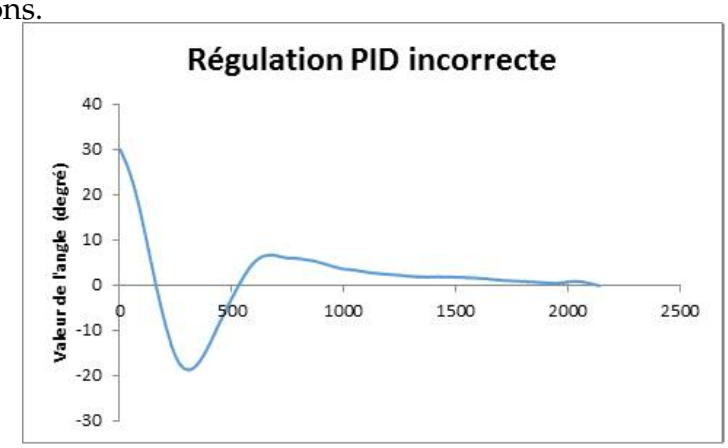
Kp= 0,015
Ki= 0,00001
Kd= 0



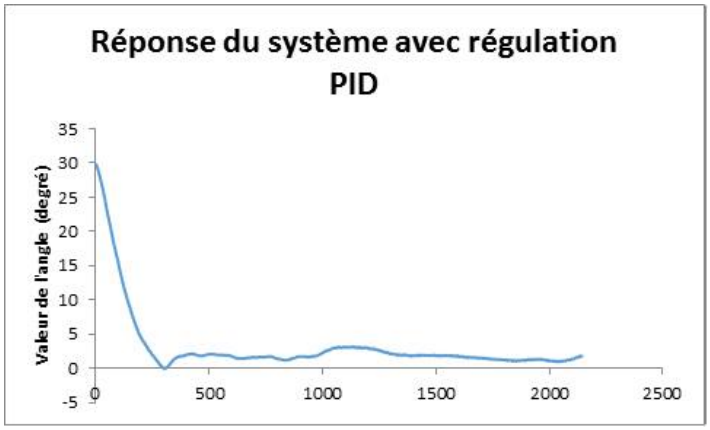
Kp= 0,015
Ki= 0,000029
Kd= 0

Enfin, en incorporant le terme Intégral, des oscillations importantes sont apparues autour de l'état d'équilibre. Il était donc temps d'ajouter le terme Dérivé qui a permis d'atténuer ces

oscillations.



Kp= 0,015
Ki= 0,000029
Kd= 0,02



Kp= 0,015
Ki= 0,000029
Kd= 0,2

2.3 Problèmes rencontrés et solutions

L'électronique était quelque chose de nouveau pour tous les membres du groupe. Nous avons dû apprendre par nous-mêmes et les obstacles ont été nombreux.

Nous avons tout d'abord rencontré quelques problèmes pour brancher la carte Arduino à l'ordinateur, celle-ci n'était pas détectée par l'ordinateur. Nous l'avons donc branchée à l'ordinateur portable de Tanguy Moreau. De plus, des soudures se sont défaites et il y avait des faux contacts, nous les avons donc refaites pour partir sur une base saine.

Après avoir effectué les premiers tests, nous nous sommes rendus compte que les deux moteurs ne tournaient pas à la même vitesse pour une même valeur d'entrée. Nous avons donc modifié au début le programme afin de tenir compte de ce problème en initialisant les deux moteurs à des valeurs différentes ; puis nous avons pensé que le régulateur PID règlerait ce déséquilibre.

De plus, une maladresse de l'ensemble du groupe lors d'un essai mal contrôlé a causé l'endommagement d'une pièce qui a fondu, elle fut vite remplacée et le projet a pu continuer normalement.

Pendant la séance 6, alors que nous tracions quelques courbes donnant l'angle du quadricopter en fonction du temps, nous nous sommes aperçus qu'il y avait une erreur dans la façon dont le programme lisait ou interprétait les valeurs des angles données par le gyroscope. Nous avons donc rajouté un correcteur d'erreurs dans le code. C'est une simple boucle selon laquelle si la différence entre deux angles lus est supérieure à un certain nombre alors

la dernière valeur n'est pas prise en compte. Nous n'avons plus jamais rencontré ce type de problème par la suite.

Après ces problèmes de codage nous avons subi des problèmes techniques dont les causes n'ont jamais été trouvées : ordinateur qui s'éteint, téléversement bloqué, programme qui stoppe son exécution brusquement...

Conclusion et perspectives d'amélioration

Nous pouvons tout d'abord rappeler que l'objectif premier de notre projet était l'asservissement d'une branche d'un quadricopter. Nous avons pour ceci implanté une régulation PID qui fonctionne. Le but de faire voler ou au moins d'équilibrer le quadricopter était une motivation pour le groupe qui s'est ainsi impliqué. Bien que le quadricopter ne vole pas, nous avons appris beaucoup sur la régulation PID qui pourra nous resservir pour tout autre système ayant besoin d'être régulé.

De plus, c'est notre premier projet en groupe s'étalant sur un semestre entier. Nous avons tous apprécié travailler au sein du groupe et avons appris à faire connaissance. Nous avons acquis des qualités essentielles au sein d'un groupe, comme l'écoute, la confiance et le sérieux, qui nous ont été indispensables pour mener notre projet à son terme.

Cependant, bien que l'asservissement d'une branche de notre quadricopter fonctionne, la régulation est loin d'être parfaite. Celle- ci pourrait être perfectionnée, notamment par le codage sur 16 bits au lieu de 8 bits qui améliorerait la précision de notre système. L'asservissement de la seconde branche avec ses propres constantes aurait par la suite permis de faire des vols tests. Nous aurions ainsi pu déplacer notre quadricopter dans une ou plusieurs directions. Au lieu d'équilibrer la branche nous aurions pu créer un petit déséquilibre provoquant le déplacement du quadricopter. Nous aurions également pu diriger le quadricopter avec une télécommande bluetooth.

Enfin, notre ignorance initiale concernant l'Arduino et la régulation par PID nous a contraint à nous informer par nous-même et à nous autoformer sur un domaine totalement nouveau, ce qui fut enrichissant et intéressant pour notre carrière future en tant qu'ingénieurs.

Bibliographie

[1] http://www.generationrobots.com/fr/401203-kit-robotique-dfrduino_mega-pour-4-moteurs.html (Valide à la date du 05/06/14)

[2] http://www.gotronic.fr/art-commande-de-2-moteurs-dri0002-19344.htm (Valide à la date du 05/06/14)

[3] http://www.dfrobot.com/wiki/index.php/MD1.3_2A_Dual_Motor_Controller_%28SKU:_DRI0002 (Valide à la date du 13/05/14)

[4] http://www.robotshop.com/en/6-dof-gyro-accelerometer-imu-mpu6050.html (Valide à la date du 03/06/14)

[5] http://www.linuxcnc.org/docs/2.4/html/motion_pid_theory_fr.html (Valide à la date du 05/06/14)

[6] http://fr.wikipedia.org/wiki/R%C3%A9gulateur_PID (Valide à la date du 05/06/14)

[7] http://www.ferdinandpiette.com/blog/2011/08/implementer-un-pid-sans-faire-de-calculs/ (Valide à la date du 05/06/14)

[8] http://www.telecom-robotics.org/wiki/Tutoriels/AsservissementPid/WebHome (Valide à la date du 05/06/14)

[9] http://www.google.fr/url?sa=t&rct=j&q=&esrc=s&source=web&cd=13&ved=0CHwQFjAM&url=http%3A%2F%2Ffreddy.mudry.org%2Fpublic%2FNotesApplications%2FNAPidAj_06.pdf&ei=X-SEU4S0OpSw7AaAh4CIDQ&usg=AFQjCNGlZMFm2BAMl0StDUXNKHjhNfYnFA&bvm=bv.67720277,d.ZG4 (Valide à la date du 05/06/14)

Annexe A

Le programme

```
// Arduino Wire library is required if I2Cdev I2CDEV_ARDUINO_WIRE
   implementation
// is used in I2Cdev.h
#include "Wire.h"

// I2Cdev and MPU6050 must be installed as libraries, or else the .cpp/.h
    files
// for both classes must be in the include path of your project
#include "I2Cdev.h"

#include "MPU6050_6Axis_MotionApps20.h"
//#include "MPU6050.h" // not necessary if using MotionApps include file


MPU6050 mpu;

// uncomment "OUTPUT_READABLE_YAWPITCHROLL" if you want to see the yaw/
// pitch/roll angles (in degrees) calculated from the quaternions coming
// from the FIFO. Note this also requires gravity vector calculations.
// Also note that yaw/pitch/roll angles suffer from gimbal lock (for
// more info, see: http://en.wikipedia.org/wiki/Gimbal_lock)
#define OUTPUT_READABLE_YAWPITCHROLL


// uncomment "OUTPUT_TEAPOT" if you want output that matches the
// format used for the InvenSense teapot demo
#define OUTPUT_TEAPOT



#define LED_PIN 13 // (Arduino is 13, Teensy is 11, Teensy++ is 6)
bool blinkState = false;

// MPU control/status vars
bool dmpReady = false; // set true if DMP init was successful
uint8_t mpuIntStatus; // holds actual interrupt status byte from MPU
uint8_t devStatus; // return status after each device operation (0 =
   success, !0 = error)
uint16_t packetSize; // expected DMP packet size (default is 42 bytes)
uint16_t fifoCount; // count of all bytes currently in FIFO
```

```
uint8_t fifoBuffer[64]; // FIFO storage buffer

// orientation/motion vars
Quaternion q;     // [w, x, y, z]   quaternion container
VectorInt16 aa;   // [x, y, z]      accel sensor measurements
VectorInt16 aaReal; // [x, y, z]    gravity-free accel sensor
   measurements
VectorInt16 aaWorld; // [x, y, z]   world-frame accel sensor measurements
VectorFloat gravity; // [x, y, z]   gravity vector
float euler[3];   // [psi, theta, phi] Euler angle container
float ypr[3];     // [yaw, pitch, roll] yaw/pitch/roll container and
   gravity vector

// packet structure for InvenSense teapot demo
uint8_t teapotPacket[14] = {
  '$', 0x02, 0,0, 0,0, 0,0, 0,0, 0x00, 0x00, '\r', '\n' };

//Motors variables
int E1 = 5;
int M1 = 4;
int E2 = 6;
int M2 = 7;
int velocity1 =0;
int velocity2 =0;
int erreur = 0;
int somme_erreurs = 0;
int variation_erreur = 0;
int erreur_precedente = 0;
int mapage = 257;
unsigned char RealVelocity1 =0;
unsigned char RealVelocity2 = 0;
float test = 0;
float value2 = 0;
float commande = 0;
float Kp = 0.015;
float Ki = 0.000029;
float Kd = 0.2;
float value =0;
long dure, dure_prec=0;
char boucle=0;



// ================================================================
// ===          INTERRUPT DETECTION ROUTINE     ===
// ================================================================

volatile bool mpuInterrupt = false; // indicates whether MPU interrupt
   pin has gone high
void dmpDataReady() {
  mpuInterrupt = true;
}
```

```
// ================================================================
// ===                INITIAL SETUP                             ===
// ================================================================

void setup() {
  // join I2C bus (I2Cdev library doesn't do this automatically)
  Wire.begin();
  pinMode(M1, OUTPUT);
  pinMode(E1, OUTPUT);
  pinMode(M2, OUTPUT);
  pinMode(E2, OUTPUT);
  // initialize serial communication
  // (115200 chosen because it is required for Teapot Demo output, but it's
  // really up to you depending on your project)
  Serial.begin(115200);
  while (!Serial); // wait for Leonardo enumeration, others continue immediately

  // initialize device
  Serial.println(F("Initializing I2C devices..."));
  mpu.initialize();

  // verify connection
  Serial.println(F("Testing device connections..."));
  Serial.println(mpu.testConnection() ? F("MPU6050 connection successful")
      : F("MPU6050 connection failed"));

  // wait for ready
  Serial.println(F("\nSend any character to begin DMP programming and demo: "));
  while (Serial.available() && Serial.read()); // empty buffer
  while (!Serial.available());       // wait for data
  while (Serial.available() && Serial.read()); // empty buffer again

  // load and configure the DMP
  Serial.println(F("Initializing DMP..."));
  devStatus = mpu.dmpInitialize();

  // make sure it worked (returns 0 if so)
  if (devStatus == 0) {
    // turn on the DMP, now that it's ready
    Serial.println(F("Enabling DMP..."));
    mpu.setDMPEnabled(true);

    // enable Arduino interrupt detection
    Serial.println(F("Enabling interrupt detection (Arduino external interrupt 0)..."));
    attachInterrupt(0, dmpDataReady, RISING);
    mpuIntStatus = mpu.getIntStatus();

```

```
    // set our DMP Ready flag so the main loop() function knows it's okay
       to use it
    Serial.println(F("DMP ready! Waiting for first interrupt..."));
    dmpReady = true;

    // get expected DMP packet size for later comparison
    packetSize = mpu.dmpGetFIFOPacketSize();
  }
  else {
    // ERROR!
    // 1 = initial memory load failed
    // 2 = DMP configuration updates failed
    // (if it's going to break, usually the code will be 1)
    Serial.print(F("DMP Initialization failed (code "));
    Serial.print(devStatus);
    Serial.println(F(")"));
  }

  // configure LED for output
  pinMode(LED_PIN, OUTPUT);
}



// ================================================================
// ===               MAIN PROGRAM LOOP                          ===
// ================================================================

void loop() {
  // if programming failed, don't try to do anything
  if (!dmpReady) return;

  // wait for MPU interrupt or extra packet(s) available
  while (!mpuInterrupt && fifoCount < packetSize) {
    if (boucle>126) {
       value = ypr[2]* 180/M_PI;
       test = abs(value2-value);
         if (test > 30){
          value = value2;
         }

     erreur = value;
     somme_erreurs += erreur;
     variation_erreur = erreur - erreur_precedente;
     commande = Kp*erreur + Ki*somme_erreurs + Kd*variation_erreur;
     erreur_precedente = erreur;

     velocity1 -= commande;
     velocity2 += commande;
     if (velocity1 > mapage) velocity1 = mapage;
     if (velocity1 < 1) velocity1 = 1;
     if (velocity2 > mapage) velocity2 = mapage;
     if (velocity2 < 1) velocity2 = 1;
```

```
      RealVelocity1 = map(velocity1,0,mapage,70,255);
      RealVelocity2 = map(velocity2,0,mapage,70,255);
      digitalWrite(M1,HIGH);
      digitalWrite(M2,HIGH);
      analogWrite(E1, RealVelocity1);
      analogWrite(E2, RealVelocity2);

      value2 = value;
      boucle=1;
    }
    boucle++;
  }

  // reset interrupt flag and get INT_STATUS byte
  mpuInterrupt = false;
  mpuIntStatus = mpu.getIntStatus();

  // get current FIFO count
  fifoCount = mpu.getFIFOCount();

  // check for overflow (this should never happen unless our code is too
     inefficient)
  if ((mpuIntStatus & 0x10) || fifoCount == 1024) {
    // reset so we can continue cleanly
    mpu.resetFIFO();

  }
  else if (mpuIntStatus & 0x02) {
    // wait for correct available data length, should be a VERY short wait
    while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

    // read a packet from FIFO
    mpu.getFIFOBytes(fifoBuffer, packetSize);

    // track FIFO count here in case there is > 1 packet available
    // (this lets us immediately read more without waiting for an
       interrupt)
    fifoCount -= packetSize;

#ifdef OUTPUT_READABLE_YAWPITCHROLL
    // display Euler angles in degrees
    mpu.dmpGetQuaternion(&q, fifoBuffer);
    mpu.dmpGetGravity(&gravity, &q);
    mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
#endif
  }

  // blink LED to indicate activity
  blinkState = !blinkState;
  digitalWrite(LED_PIN, blinkState);
}
```