ECOLE NATIONALE SUPÉRIEURE DES ARTS ET MÉTIERS MEKNES

MISE EN ŒUVRE ET STABILISATION D'UN BRAS DE DRONE QUADRIROTOR

PROJET MÉTIER

AUTHEUR

AHMED AMINE NOUABI

ENCADRANT

MR TALEB

JURY

MR TALEB

MR LAGRIOUI

MR SAADI

MEKNES, FEBRUARY 2025

This page intentionally left blank.

Resume

Ce projet se concentre sur la conception, la mise en œuvre et l'optimisation d'un système de stabilisation pour une barre rotative à un degré de liberté. En utilisant un microcontrôleur Arduino, un capteur IMU MPU6050, un algorithme de contrôle PID et un algorithme d'estimation, l'objectif est de maintenir la barre en position horizontale malgré les perturbations.

This page intentionally left blank.

Contents

List of Figures

Glossary

ECS Electronic Speed Controller is an electronic circuit with the purpose to vary an electric motor's speed, its direction and possibly also to act as a dynamic brake.. *(p. 32)*

I2C Inter-Integrated Circuit is a multi-master, multi-slave, single-ended, serial computer bus invented by Philips Semiconductor (now NXP Semiconductors).. *(p. 5)*

IMU Inertial Measurement Unit is an electronic device that measures and reports a body's specific force, angular rate, and sometimes the magnetic field surrounding the body, using a combination of accelerometers, gyroscopes, and sometimes magnetometers. *(p. 2, 3, 35, 38, 40)*

PWM Pulse-width modulation is a method used to reduce the average power delivered by an electrical signal, by effectively chopping it up into discrete parts.. *(p. 7)*

SPI Serial Peripheral Interface is a synchronous serial communication interface specification used for short-distance communication, primarily in embedded systems.. *(p. 5)*

Acronyms

PID Proportional-Integral-Derivative. *(p. 3)*

SCL Serial Clock Line. *(p. 5, 6)*

SDA Serial Data Line. *(p. 5, 6)*

1

Introduction

1.1 Contexte et motivation

Dans le domaine de l'ingénierie de contrôle, les systèmes de stabilisation sont essentiels pour maintenir l'équilibre de diverses structures et dispositifs mécaniques. Ces systèmes sont largement appliqués dans de nombreux domaines, notamment la robotique, l'aérospatiale, l'automobile et l'automatisation industrielle. La capacité à stabiliser un système de manière efficace peut considérablement améliorer ses performances et sa fiabilité.

Ce projet se concentre sur la stabilisation d'une barre rotative à un degré de liberté, qui sert de modèle simplifié pour des problèmes de stabilisation plus complexes. Le système de la barre rotative, souvent appelé pendule inversé, est un problème classique en théorie de contrôle et fournit une plate-forme précieuse pour tester et développer des algorithmes de contrôle.

1.2 Objectifs

1. Concevoir la structure mécanique.
2. Intégrer un IMU pour obtenir des données d'orientation en temps réel.
3. Développer et mettre en œuvre un algorithme de contrôle PID pour traiter les données de l'IMU et contrôler les rotors.

1.3 Portée

La portée de ce projet comprend la conception et la mise en œuvre des composants matériels et logiciels nécessaires pour le système de stabilisation. Les éléments clés du projet sont :

1. **Etude théorique :** Cela implique la compréhension et l'analyse du système de la barre rotative, y compris les équations de mouvement, principes de contrôle et estimation de l'état.
2. **Conception Mécanique :** Cela implique la conception et la construction de la barre rotative ainsi que le support fixe.
3. **Conception Électronique :** Cela couvre l'intégration de la carte d'aquisition de données et de controle y compris le câblage et la conception des circuits.
4. **Conception Logiciel :** Cela comprend la programmation de l'algorithme de contrôle PID et l'intégration de l'IMU pour obtenir des données d'orientation en temps réel.

2

Bibliographie

2.1 Inertial Measurement Unit

Un IMU est un dispositif électronique qui mesure et rapporte les données d'accélération linéaire, de vitesse angulaire et d'orientation d'un objet. Les IMU sont largement utilisés dans les applications de navigation inertielle, de robotique et de réalité virtuelle.

Les IMU sont généralement composés de trois capteurs principaux : un accéléromètre, un gyroscope et un magnétomètre. L'accéléromètre mesure l'accélération linéaire de l'objet, le gyroscope mesure la vitesse angulaire de l'objet et le magnétomètre mesure le champ magnétique terrestre pour déterminer l'orientation de l'objet.

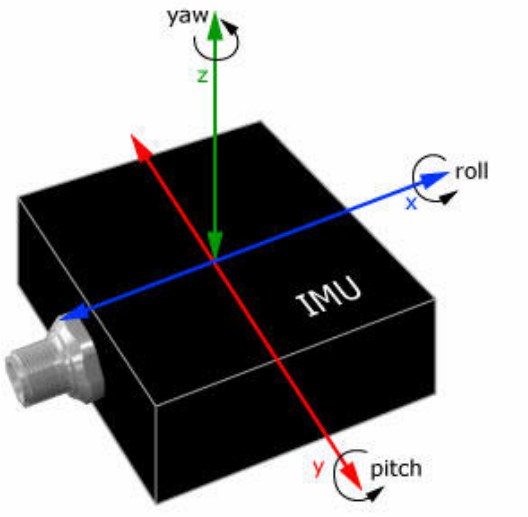

Figure 2.1: *Schéma d'un IMU.*

Les IMU sont souvent utilisés en combinaison avec d'autres capteurs, tels que les GPS et les caméras, pour fournir des données de localisation et d'orientation plus précises. Les IMU sont également utilisés dans les applications de réalité virtuelle pour suivre les mouvements de la tête de l'utilisateur et fournir une expérience immersive.

2.1.1 MPU6050

Le MPU6050 est un IMU à 6 axes qui combine un accéléromètre et un gyroscope dans un seul boîtier. Le MPU6050 est largement utilisé dans les applications de robotique et de contrôle de mouvement en raison de sa petite taille, de sa faible consommation d'énergie et de sa précision élevée. Le MPU6050 est capable de mesurer l'accélération linéaire dans les trois axes et la vitesse angulaire dans les trois axes. Il put communiquer avec un microcontrôleur en utilisant le protocole I2C ou SPI.

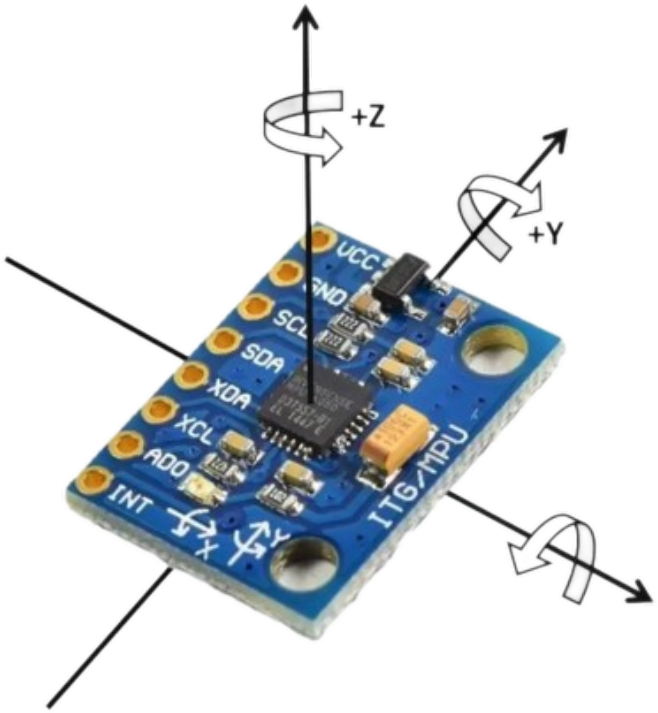

Figure 2.2: *Module MPU6050.*

2.1.2 I2C

L'I2C est un bus de communication série d'architecture Maitre-esclaves qui permet à plusieurs périphériques de communiquer entre eux à l'aide d'un seul bus de données. L'I2C est largement utilisé dans les applications de capteurs et de contrôleurs pour connecter plusieurs périphériques à un microcontrôleur.

L'I2C utilise deux fils pour la communication : un fil de données (SDA) et un fil d'horloge (SCL). Chaque périphérique connecté au bus I2C possède une adresse

unique qui lui permet de communiquer avec les autres périphériques sur le bus. L'I2C prend en charge plusieurs vitesses de communication, allant de 100 kHz à 3,4 MHz.

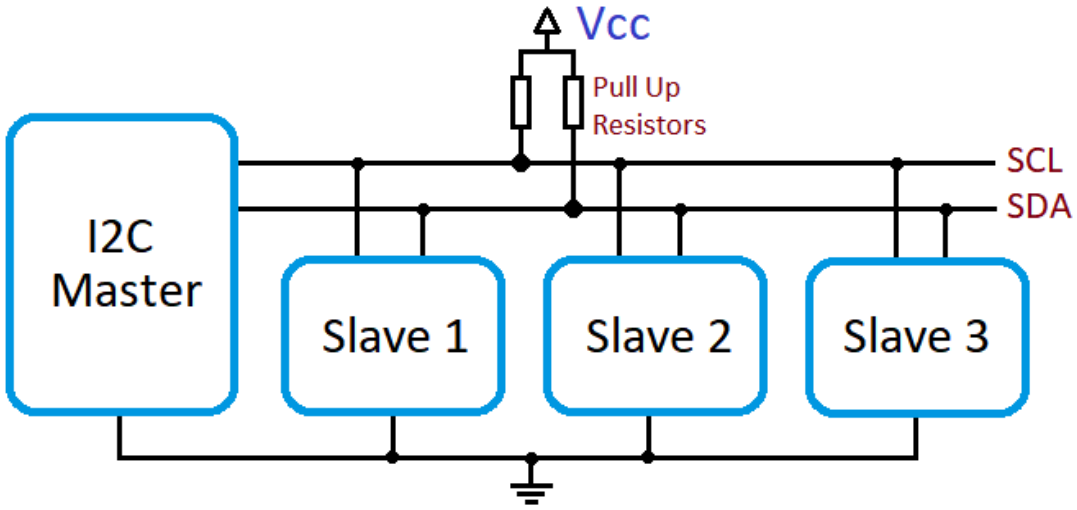

Figure 2.3: *Schéma d'un bus I2C.*

2.1.3 Paquet I2C

Un paquet I2C est composé de :

1. **Condition de demarage :** Un signal de valeur haute sur SDA et SCL indique le début de la communication.
2. **Adresse client :** L'adresse du périphérique esclave auquel le maître souhaite communiquer.
3. **Ecriture/Lecture :** Un bit de lecture/écriture indique si le maître souhaite lire ou écrire des données. (R = 1, W = 0)
4. **Acknowledge :** Un bit de valeur basse sur SDA indique que le périphérique esclave a reçu les données avec succès.
5. **Données :** Les données à écrire ou lire.
6. **Acknowledge :** Un bit de valeur basse sur SDA indique que le périphérique esclave a reçu les données avec succès.
7. **Condition d'arrêt :** Un signal de valeur basse sur SDA et haute sur SCL indique la fin de la communication.

Adresse client est composé de 7 bits d'adresse et d'un bit de lecture/écriture. Alors que les paquets de données peuvent etre sequentiels composés de 8 bits de données et un bit d'acknowledge entre eux.

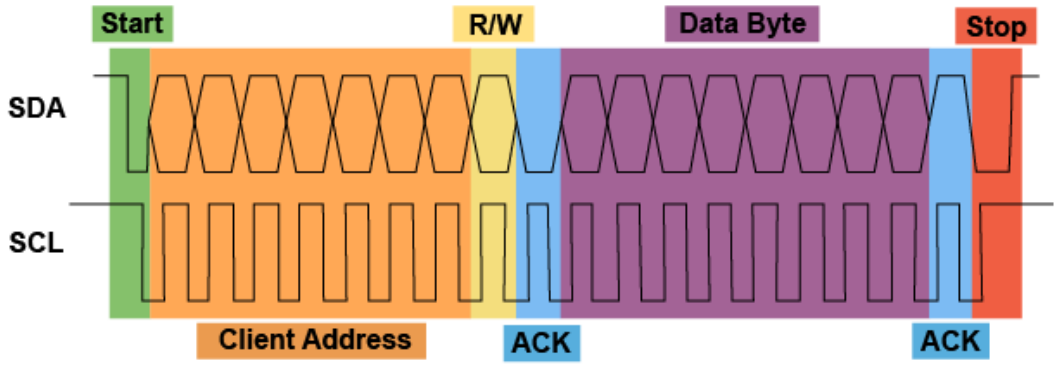

Figure 2.4: *Paquet I2C.*

2.2 Electronic Speed Controller

Un ESC (Electronic Speed Controller) est un dispositif électronique qui convertie DC/AC (DAC) et contrôle la vitesse d'un moteur électrique en ajustant la tension et le courant fournis au moteur. Les ESC sont largement utilisés dans les applications de robotique, de drones et de modélisme pour contrôler la vitesse des moteurs électriques.

On peut controler la vitesse d'un moteur électrique en ajustant la tension et le courant fournis au moteur. pour varier cette derniere on utilise un signal (Pulse Width Modulation) qui permet de controler la vitesse du moteur en ajustant la largeur des impulsions du signal.

On alimente l'ESC avec une tension continue de 5V à 12V, et on controle la vitesse du moteur en ajustant la largeur des impulsions du signal PWM. La largeur de l'impulsion determine la vitesse du moteur, plus l'impulsion est longue, plus la vitesse du moteur est élevée. L'ESC convertit le signal PWM en tension et courant pour controler la vitesse du moteur.

- $T_{on} = T * \alpha$ est la relation du rapport cyclique du signal PWM.

$$1000\mu s \leq T_{on} \leq 2000\mu s \tag{2.1}$$

2.2.1 Moteur A2212/13T

Figure 2.5: *ESC.*

Le moteur A2212/13T est un moteur brushless qui est largement utilisé dans les applications de drone et de modélisme. Le moteur A2212/13T est un moteur à aimant permanent qui utilise un contrôleur électronique (ESC) pour contrôler la vitesse du moteur.

A2212/13T TECHNICAL DATA

No. of Cells:	2 - 3 Li-Poly 6 - 10 NiCd/NiMH
Kv:	1000 RPM/V
Max Efficiency:	80%
Max Efficiency Current:	4 - 10A (>75%)
No Load Current:	0.5A @10V
Resistance:	0.090 ohms
Max Current:	13A for 60S
Max Watts:	150W
Weight:	52.7 g / 1.86 oz
Size:	28 mm dia x 28 mm bell length

Figure 2.6: *Moteur A2212/13T.*

2.2.2 Contrôle PID

Le contrôle PID (Proportionnel, Intégral, Dérivé) est une méthode courante pour contrôler les systèmes dynamiques en utilisant un retour d'information. Le contrôle PID utilise trois termes pour ajuster la commande du système en fonction de l'erreur, de l'intégrale de l'erreur et de la dérivée de l'erreur.

Le terme proportionnel ajuste la commande en fonction de l'erreur actuelle, le terme intégral ajuste la commande en fonction de l'accumulation de l'erreur passée et le terme dérivé ajuste la commande en fonction de la variation de l'erreur. Le contrôle PID est largement utilisé dans les applications de contrôle de mouvement, de robotique et de systèmes de contrôle automatique.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\,d\tau + K_d \frac{de(t)}{dt} \tag{2.2}$$

- $u(t)$ est la commande du système.
- $e(t)$ est l'erreur du système.
- K_p, K_i, K_d sont les coefficients du contrôleur PID.
- t est le temps.
- τ est le temps de l'intégrale.
- $de(t)/dt$ est la dérivée de l'erreur.
- $\int_0^t e(\tau)\,d\tau$ est l'intégrale de l'erreur.

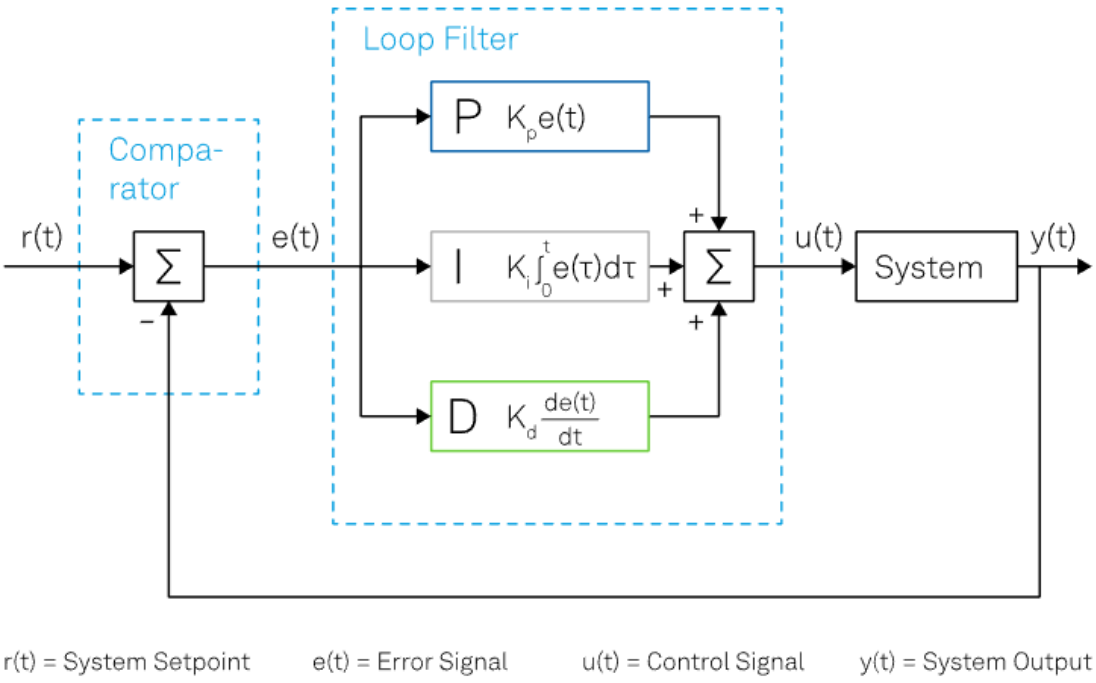

Figure 2.7: *Contrôle PID.*

3

Théorie et Modélisation

3.1 Dynamique du Système

Le système de stabilisation de la barre rotative est un système dynamique non linéaire à un degré de liberté. Le système est composé d'une barre rotative montée sur un support fixe. La barre rotative est contrôlée par deux rotors qui peuvent appliquer un couple sur la barre pour la stabiliser.

Le système est soumis à plusieurs forces et moments qui influent sur son mouvement. Les forces et moments les plus importants sont :

1. **Force de gravité :** La force de gravité agit sur la barre rotative et exerce un couple sur la barre.
2. **Couple des rotors :** Les rotors peuvent appliquer un couple sur la barre pour la stabiliser.
3. **Frottement :** Le frottement entre la barre rotative et le support fixe peut ralentir le mouvement de la barre.

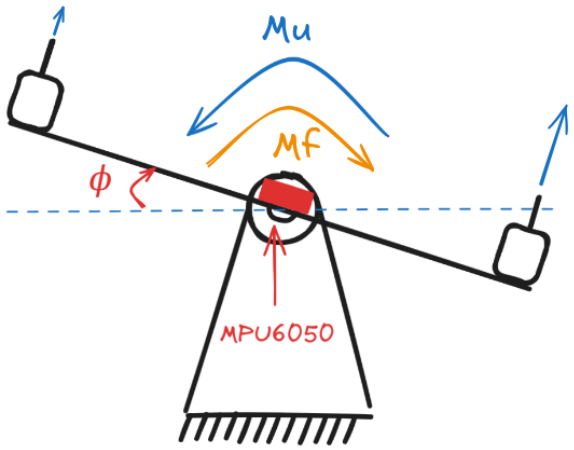

Figure 3.1: *Moments de Force.*

Le système est décrit par l'équation dynamique suivante :

$$J_\Delta * \frac{d^2\phi}{dt^2} = \sum M \tag{3.1}$$

Moments de Force :

- M_f: est le moment de force dû au frottement. $M_f = -f * \frac{d\phi}{dt}$.
- M_g: est le moment de force dû à la gravité (Si centre de masse coincide avec centre geometrique alors $M_g = 0$).
- M_u: est le moment de force dû aux rotors. (Directement proportionnel à l'entrée u).

$$J_\Delta * \frac{d^2\phi}{dt^2} = M_u - f * \frac{d\phi}{dt} \tag{3.2}$$

- ϕ est l'angle de la barre rotative par rapport à l'horizontale 0°.
- J_Δ est le moment d'inertie de la barre rotative.
- L est la longueur de la barre rotative.
- f est le coefficient de frottement.

3.1.1 Commande simplifiée

Pour simplifier la commande du système due à la symmétrie du système, on peut utiliser une commande sous forme scalaire. avec F la force mediane des rotors.

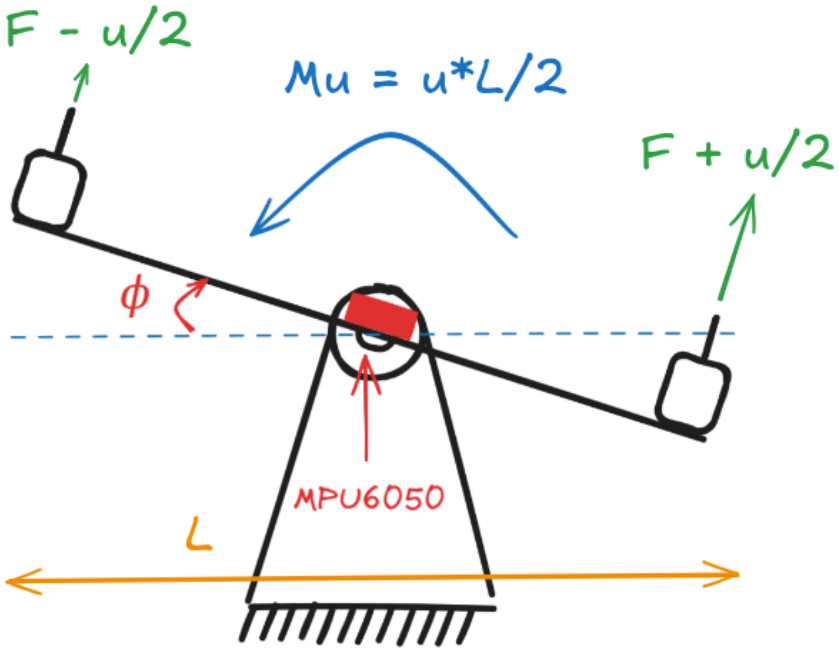

Figure 3.2: *Commande scalaire.*

$$M_u = M_{rotor1} + M_{rotor2} \tag{3.3}$$

$$M_u = \frac{L}{2} * ((F + \frac{u}{2}) - (F - \frac{u}{2})) \tag{3.4}$$

$$M_u = u * \frac{L}{2} \tag{3.5}$$

L'equation dynamique du système devient :

$$J_\Delta * \frac{d^2\phi}{dt^2} + f * \frac{d\phi}{dt} = u * \frac{L}{2} \tag{3.6}$$

Limitation

Le problème le plus pertinent dans cette approche de commande est le faite que les deux motors ne sont pas physiquement les mêmes, donc pour la même consigne les deux rotors ne vont pas tourner à la même vitesse.

3.1.2 Systeme en boucle ouverte

Equation dynamique

$$J_\Delta * \frac{d^2\phi}{dt^2} + f * \frac{d\phi}{dt} = u * \frac{L}{2} \tag{3.7}$$

Transformée de Laplace

$$J_\Delta * s^2 * \Phi(s) + f * s * \Phi(s) = U(s) * \frac{L}{2} \tag{3.8}$$

$$G(s) = \frac{\Phi(s)}{U(s) * \frac{L}{2}} = \frac{1}{J_\Delta * s^2 + f * s} \tag{3.9}$$

Le système est du second ordre mais non standard car le terme constant est nul $\omega_n = 0$ avec deux poles réels:

- **Pole 1 :** $s_1 = 0$
- **Pole 2 :** $s_2 = -\frac{f}{J_\Delta}$

Remarques

- $J_\Delta > 0$, $f > 0$, donc $s_2 < 0$. s_2 dans la region de stabilité.
- Par contre, s_1 est nul, donc le système est marginalement stable.
- Peut être stabilisé en ajoutant un PID.
  - **P :** Pour rendre le système standard mais pas suffisant car il y aura un erreur statique.
  - **I :** Pour éliminer l'erreur statique.
  - **D :** Pour un meilleur amortissement.

Dynamique dyu système sous forme d'espace d'état

$$\frac{dx}{dt} = A * x + B * u \tag{3.10}$$

$$y = C * x \tag{3.11}$$

$$\begin{bmatrix} \frac{d\phi}{dt} \\ \frac{d^2\phi}{dt^2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{f}{J_\Delta} \end{bmatrix} \begin{bmatrix} \phi \\ \frac{d\phi}{dt} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{L}{2J_\Delta} \end{bmatrix} u \tag{3.12}$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} \phi \\ \frac{d\phi}{dt} \end{bmatrix} \tag{3.13}$$

Matrice de commandabilité :

$$A * B = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{f}{J_\Delta} \end{bmatrix} \begin{bmatrix} 0 \\ \frac{L}{2J_\Delta} \end{bmatrix} = \begin{bmatrix} \frac{L}{2J_\Delta} \\ -\frac{fL}{2J_\Delta^2} \end{bmatrix} \tag{3.14}$$

$$M_c = \begin{bmatrix} B & A * B \end{bmatrix} = \begin{bmatrix} 0 & \frac{L}{2J_\Delta} \\ \frac{L}{2J_\Delta} & -\frac{fL}{2J_\Delta^2} \end{bmatrix} = \frac{L}{2J_\Delta} \begin{bmatrix} 0 & 1 \\ 1 & -\frac{f}{J_\Delta} \end{bmatrix} \tag{3.15}$$

$$\text{rang}(M_c) = 2 \Rightarrow \text{commandable} \tag{3.16}$$

Matrice d'observabilité :

$$C * A = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 0 & -\frac{f}{J_\Delta} \end{bmatrix} = \begin{bmatrix} 0 & 1 \end{bmatrix} \tag{3.17}$$

$$M_o = \begin{bmatrix} C \\ C * A \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{3.18}$$

$$\text{rang}(M_o) = 2 \Rightarrow \text{observable} \tag{3.19}$$

3.2 Correcteur PID

D'apres les remarques précédentes, on peut utiliser un correcteur PID pour stabiliser le système.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{de(t)}{dt} \tag{3.20}$$

$$C(s) = \frac{U(s)}{E(s)} = K_p + K_i * \frac{1}{s} + K_d * s \tag{3.21}$$

$$G(s) = \frac{1}{J_\Delta * s^2 + f * s} \tag{3.22}$$

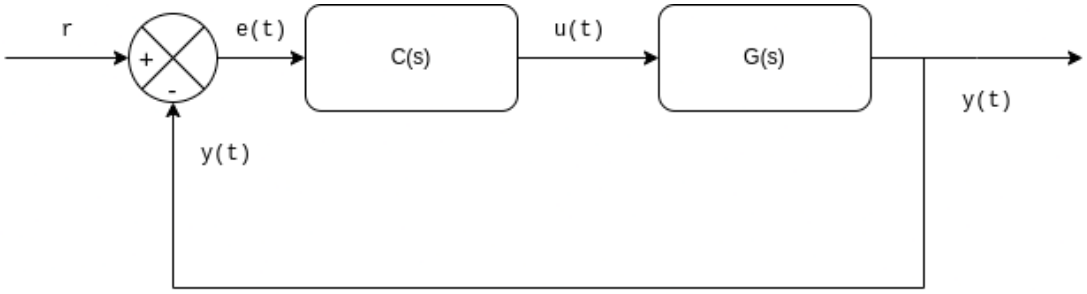

Figure 3.3: *Correcteur PID.*

$$FTBO(s) = C(s) * G(s) = \frac{K_d * s^2 + K_p * s + Ki}{J_\Delta * s^3 + f * s^2} \tag{3.23}$$

$$FTBF(s) = \frac{C(s) * G(s)}{1 + C(s) * G(s)} = \frac{K_d * s^2 + K_p * s + Ki}{J_\Delta * s^3 + (f + K_d) * s^2 + K_p * s + Ki} \tag{3.24}$$

3.2.1 Stabilité

$$D(s) = J_\Delta * s^3 + (f + K_d) * s^2 + K_p * s + Ki \tag{3.25}$$

Tous les coefficients de $D(s)$ doivent être positifs pour que le système soit stable (Pas suffisant mais nécessaire).

- $J_\Delta > 0$, (toujours vrai).
- $f + K_d > 0$,
- $K_p > 0$.
- $Ki > 0$.

Critère de Routh-Hurwitz

$$\begin{array}{c|cc} s^3 & J_\Delta & K_p \\ s^2 & f + K_d & Ki \\ s^1 & \frac{K_p*(f+K_d)-J_\Delta*Ki}{f+K_d} & 0 \\ s^0 & Ki & 0 \end{array} \tag{3.26}$$

- $J_\Delta > 0$, (toujours vrai).
- $f + K_d > 0$,
- $K_p * (f + K_d) > J_\Delta * Ki$.
- $Ki > 0$.

3.2.2 Conclusion

Fonction de tranfert du système avec correcteur PID en boucle fermée :

$$H_{des}(s) = \frac{K_d * s^2 + K_p * s + Ki}{J_\Delta * s^3 + (f + K_d) * s^2 + K_p * s + Ki} \tag{3.27}$$

Le système avec correcteur en boucle fermée est stable si les conditions suivantes sont vérifiées :

- $K_d > -f$.
- $K_p * (f + K_d) > J_\Delta * Ki$.
- $Ki > 0$.
- $K_p > 0$.

3.3 Calcul de l'orientation

Le capteur MPU6050 est un capteur d'inertie à six degrés de liberté qui combine un accéléromètre et un gyroscope pour mesurer l'orientation d'un objet dans l'espace tridimensionnel. L'accéléromètre mesure l'accélération linéaire de l'objet, tandis que le gyroscope mesure la vitesse angulaire de l'objet.

Alors pour savoir l'orientation de l'objet, on doit envisager un algorithme pour calculer l'angle d'orientation.

Les Angles d'Euler

Les angles d'Euler sont une méthode courante pour représenter l'orientation d'un objet dans l'espace tridimensionnel. Les angles d'Euler sont composés de trois angles : l'angle de roulis, l'angle de tangage et l'angle de lacet. Ces angles décrivent la rotation de l'objet autour de ses axes X, Y et Z respectivement d'un système de coordonnées fixe.

1. **Roulis (Roll) :** Rotation autour de l'axe X. θ.
2. **Tangage (Pitch) :** Rotation autour de l'axe Y. ϕ.
3. **Lacet (Yaw) :** Rotation autour de l'axe Z. ψ.

La matrice de rotation

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R_{xyz} = R_x(\theta) \times R_y(\phi) \times R_z(\psi)$$

$$R_{xyz} = \begin{bmatrix} \cos(\phi)\cos(\psi) & \cos(\phi)\sin(\psi) & -\sin(\phi) \\ \sin(\theta)\sin(\phi)\cos(\psi) - \cos(\theta)\sin(\psi) & \sin(\theta)\sin(\phi)\sin(\psi) + \cos(\theta)\cos(\psi) & \sin(\theta)\cos(\phi) \\ \cos(\theta)\sin(\phi)\cos(\psi) + \sin(\theta)\sin(\psi) & \cos(\theta)\sin(\phi)\sin(\psi) - \sin(\theta)\cos(\psi) & \cos(\theta)\cos(\phi) \end{bmatrix}$$

On définit la matrice R de rotation du capteur par rapport au repère fixe.

$$R = R_{xyz} \tag{3.28}$$

3.3.1 Accéléromètre

Modèle Mathématique

L'accélération linéaire mesurée par l'accéléromètre est composée, l'accélération gravitationnelle et de l'accélération linéaire de l'objet, le biais de l'accéléromètre et le bruit de l'accéléromètre.

$$\vec{a} = \frac{\vec{dv}}{dt} - R * \vec{g} + b_a(t) + n_a(t) \tag{3.29}$$

- $\vec{a}$ est l'accélération linéaire mesurée par l'accéléromètre.
- R est la matrice de rotation du capteur.
- $b_a(t)$ est le biais de l'accéléromètre.
- $n_a(t)$ est le bruit de l'accéléromètre.

L'accélération gravitationnelle est définie comme :

$$g = 9.81\, m/s^2$$

$$g = 1 gram$$

Pour une plage de mesure de l'accéléromètre de $\pm 4g$ on a :

$$g = \frac{2^{16}}{8 grams} * 1 gram = 8192 \ \ (raw\ sensor)$$

Dans un premier temps on se pose au repos donc l'accélération linéaire est nulle aussi nous allons négliger le bruit et le biais de l'accéléromètre pour simplifier le modèle.

$$\vec{a} = -R * \vec{g}$$

On deduit trois equations pour les trois axes de l'accéléromètre.

$$
\begin{aligned}
a_x &= g\sin(\phi) \\
a_y &= -g\sin(\theta)\cos(\phi) \\
a_z &= -g\cos(\theta)\cos(\phi)
\end{aligned}
$$

Pour ce projet notre bras posede un seul degré de liberté, donc nous allons utiliser l'angle de tangage pour déterminer la position du bras.

$$\hat{\phi}_{accelo}(n) = \arcsin\left(\frac{a_x(n)}{g}\right) \tag{3.30}$$

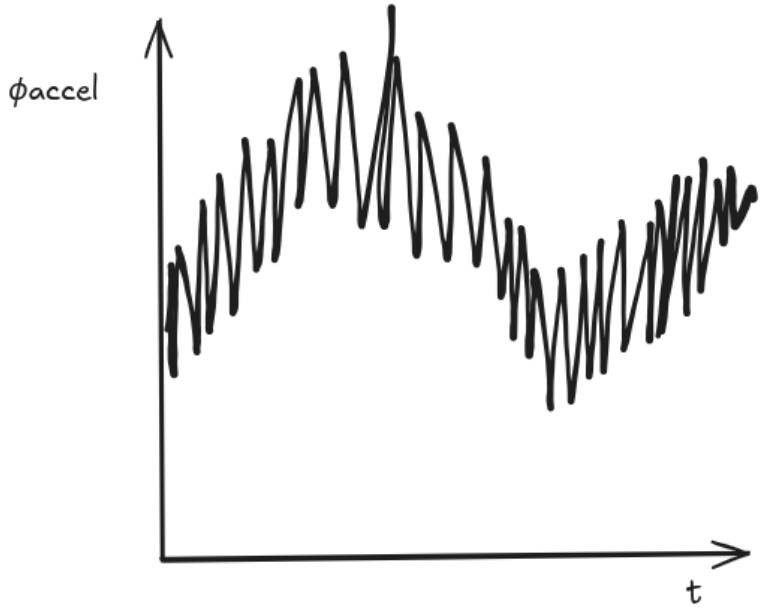

Figure 3.4: *Accéléromètre Limitation.*

Le problème de cette estimation de l'orientation est que l'accéléromètre est bruité et sensible aux vibrations. L'estimation de l'orientation à partir de l'accéléromètre est sujette à des erreurs et des dérives.

3.3.2 Gyroscope

L'intégrale de la rotation est une méthode simple pour estimer l'orientation d'un objet à partir des données d'un gyroscope. L'intégrale de la rotation consiste à intégrer les données du gyroscope pour estimer l'orientation de l'objet en temps réel.

Modèle Mathématique

La vitesse angulaire mesurée par le gyroscope est composée de la vitesse angulaire de l'objet, du biais du gyroscope et du bruit du gyroscope.

$$\omega = \frac{d\phi}{dt} + b_g(t) + n_g(t) \tag{3.31}$$

1. ω est la vitesse angulaire mesurée par le gyroscope.
2. $\frac{d\phi}{dt}$ est la vitesse angulaire reelle.
3. $b_g(t)$ est le biais du gyroscope.
4. $n_g(t)$ est le bruit du gyroscope.

De meme on neglige le bruit et le biais du gyroscope pour simplifier le modèle.

$$\hat{\phi}_{gyro}(n) = \int_0^{n*T} \dot{\phi}(n)\, dt \tag{3.32}$$

$$\hat{\phi}_{gyro}(n) = \sum_{i=0}^{n} \dot{\phi}(i) * T \tag{3.33}$$

- $\hat{\phi}_{gyro}(n)$ est l'angle de tangage estimé à partir des données du gyroscope.
- $\dot{\phi}(i)$ est la vitesse angulaire du gyroscope à l'instant i.
- T est l'intervalle de temps entre deux mesures.

Le problème de cette estimation de l'orientation à partir du gyroscope est la dérive du gyroscope. L'erreur cumulée de l'intégration des données du gyroscope entraîne une dérive de l'angle de tangage au fil du temps.

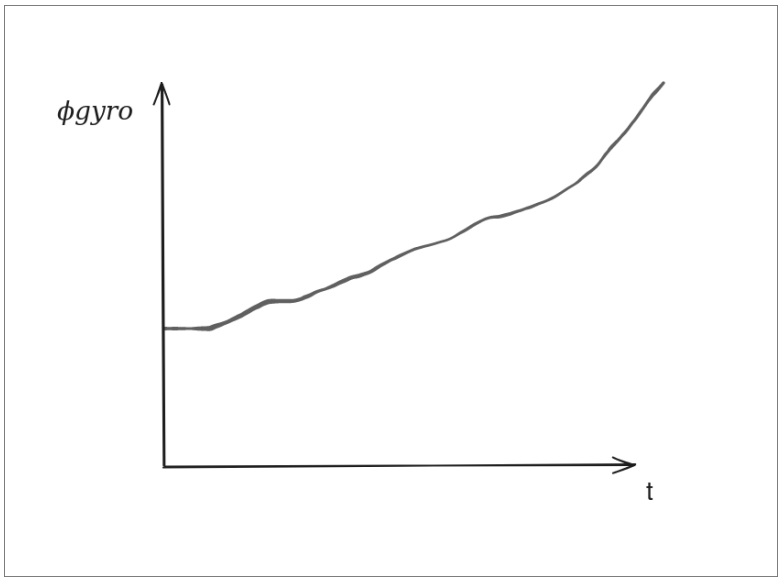

Figure 3.5: *Dérive du gyroscope.*

Illustration des limitations des deux capteurs :

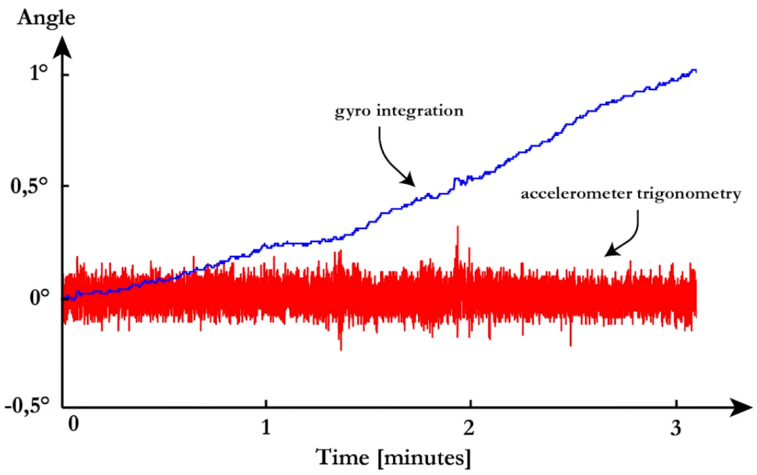

Figure 3.6: *Gyroscope and Accelerometer Limitations.*

3.3.3 Estimation de l'orientation

Dans la derniere partie, on a vu que le calcule de l'orientation n'est pas exacte et susceptible à des erreurs et des bruits. Pour obtenir une estimation plus précise de l'orientation de l'objet, on doit envisager un estimateur robuste.

Filtre Complémentaire (Sensor Fusion)

Le filtre complémentaire est une méthode courante qui combine des données de capteurs différents pour obtenir une estimation plus précise de l'orientation de l'objet.

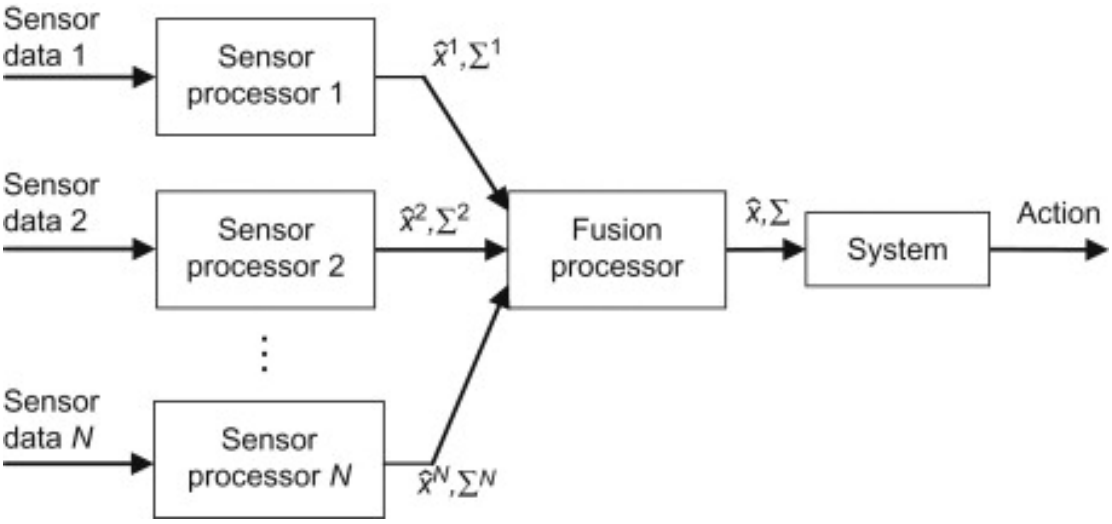

Figure 3.7: *Filtre Complémentaire.*

Donc on peut combiner les deux estimations de l'angle de tangage à partir de l'accéléromètre et du gyroscope en utilisant un filtre complémentaire pour obtenir une estimation plus précise de l'orientation de l'objet.

- **Le gyroscope :** fonctionne mieux avec un filtrage passe-haut sur l'angle ϕ pour remedier aux erreurs cumulées de l'intégration.
- **L'accéléromètre :** est plus efficace lorsqu'un filtre passe-bas élimine les vibrations et autres effets de haute fréquence sur l'angle ϕ.

$\hat{\phi}$ = filtre-passe-haut(ϕ_{gyro}) + filtre-passe-bas(ϕ_{accel})

Equations canoniques filtres de premier ordre :

- Filtre Passe Bas :

$$H_{PB}(s) = \frac{1}{1+\tau s} = \frac{\omega_c}{s+\omega_c} \tag{3.34}$$

- Filtre Passe Haut :

$$H_{PH}(s) = \frac{\tau s}{1+\tau s} = \frac{s}{s+\omega_c} \tag{3.35}$$

En fusionnant les deux estimations, chaqu'une avec son filtre, on obtient une estimation plus précise de l'orientation de l'objet. qu'on peut schématiser par :

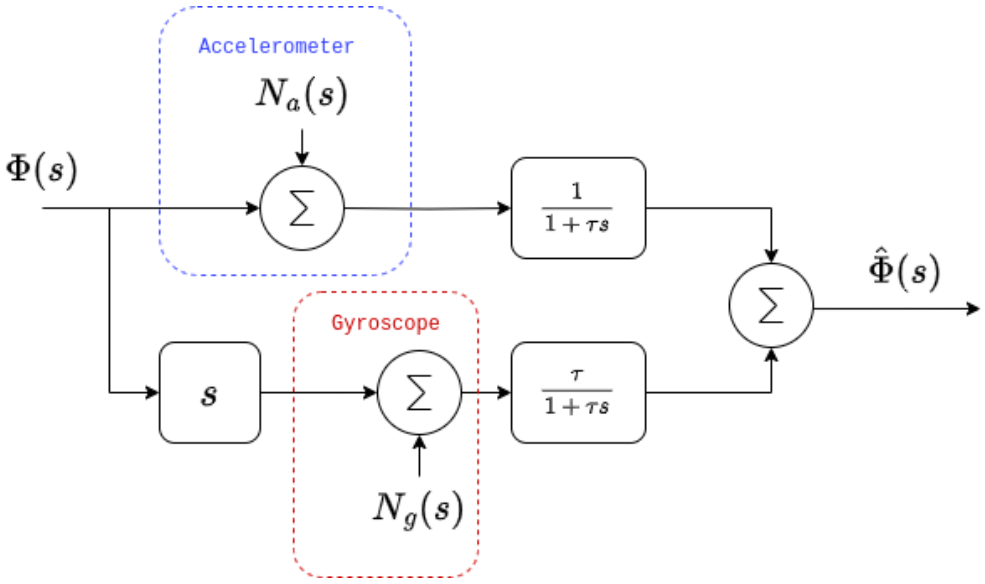

Figure 3.8: *Estimateur de l'orientation.*

La branche du gyroscope doit se comporter comme un filtre passe-haut et sachant que le gyroscope est de nature derivateur (il donne la derive de l'angle de tangage) on termine la fonction du transfert du filtre passe-haut par un filtre passe-bas de gain τ ce qui justifie la fonction de Transfert suivante :

$$\frac{\tau}{1+\tau s} = \frac{1}{s} * \frac{\tau s}{1+\tau s} \tag{3.36}$$

L'equation de l'estimateur de l'angle de tangage devient :

$$\hat{\Phi}(s) = \Phi(s) + N_g(s) * \frac{\tau}{1+\tau s} + N_a(s) * \frac{1}{1+\tau s} \tag{3.37}$$

On remarque qu'on peut remedier aux bruits N_a, N_g de l'accéléromètre et du gyroscope respectivement.

Discrétisation des filtres par bloquer d'ordre 1

$$H(z) = (1-z^{-1}) * TZ(TL^{-1}(\frac{H(s)}{s})) \tag{3.38}$$

- $H_1(s) = \frac{1}{1+\tau s} = \frac{\omega_c}{\omega_c+s}$:

$$H_1(z) = (1-z^{-1}) * TZ(TL^{-1}(\frac{\omega_c}{s(\omega_c+s)})) \tag{3.39}$$

$$H_1(z) = (1-z^{-1}) * TZ(TL^{-1}(\frac{1}{s} - \frac{1}{s+\omega_c})) \tag{3.40}$$

On prend $Z(s) = \frac{1}{s} - \frac{1}{s+\omega_c}$

Transformée de la place inverse :

$$TL^{-1}(Z(s)) = z(t) = (1-e^{-\omega_c t}) * u(t) \tag{3.41}$$

Discretisation de periode d'echantillonnage $T_e = \frac{1}{f_e}$:

$$z(n) = (1-\alpha^n) * u(n) \tag{3.42}$$

Avec $\alpha = e^{-\omega_c T_e}$

$$TZ(TL^{-1}(\frac{H_1(s)}{s})) = TZ(z(n)) = \frac{1}{1-z^{-1}} - \frac{1}{1-\alpha * z^{-1}} \tag{3.43}$$

$$H_1(z) = (1-z^{-1}) * (\frac{1}{1-z^{-1}} - \frac{1}{1-\alpha * z^{-1}}) \tag{3.44}$$

$$H_1(z) = \frac{1-\alpha}{z-\alpha} \tag{3.45}$$

De meme pour $H_2(s)$ on aura la meme forme multipliee par $\tau = \frac{1}{\omega_c}$

- $H_2(s) = \tau \frac{1}{1+\tau s}$:

$$H_2(z) = \tau * \frac{1-\alpha}{z-\alpha} \tag{3.46}$$

Developpement de l'equation de l'estimateur de l'angle de tangage :

$$\hat{\Phi}(z) = H_2(z) * \omega_{gyro} + H_1(z) * \Phi_{accel} \tag{3.47}$$

$$\hat{\Phi}(z) = \tau * \frac{1-\alpha}{z-\alpha} * \omega_{gyro} + \frac{1-\alpha}{z-\alpha} * \Phi_{accel} \tag{3.48}$$

Avec $\tau = \frac{\alpha * T_e}{1-\alpha}$ on aura : $\tau * (1-\alpha) = \alpha * T_e$

On peut deduire l'equation de l'estimateur de l'angle de tangage en fonction des mesures de l'accéléromètre et du gyroscope.

$$\hat{\Phi}(z) * (z-\alpha) = \alpha * T_e * \omega_{gyro} + (1-\alpha) * \Phi_{accel} \tag{3.49}$$

En faisant la transformée inverse de Z on obtient :

$$\hat{\Phi}(n+1) - \alpha * \hat{\Phi}(n) = \alpha * T_e * \omega_{gyro} + (1-\alpha) * \Phi_{accel} \tag{3.50}$$

$$\hat{\Phi}(n+1) = \alpha * (\hat{\Phi}(n) + \omega_{gyro} * T_e) + (1-\alpha) * \Phi_{accel} \tag{3.51}$$

On peut deduire l'equation de l'estimateur de l'angle de tangage en fonction des mesures de l'accéléromètre et du gyroscope.

$$\hat{\Phi}(n) = \alpha * (\hat{\Phi}(n-1) + \omega_{gyro} * T_e) + (1-\alpha) * \Phi_{accel} \tag{3.52}$$

3.4 Schéma final du système

Le schéma final du système de stabilisation de la barre rotative est composé de trois parties principales :

- **Correcteur PID $C(s)$:**

$$C(s) = K_p + \frac{K_i}{s} + K_d * s \tag{3.53}$$

- **FTBO du système $G(s)$:**

$$G(s) = \frac{1}{J_\Delta * s^2 + f * s} \tag{3.54}$$

- **Estimateur d'orientation $\hat{\Phi}(s)$:**

$$\hat{\Phi}(s) = \frac{\Phi(s) + N_a(s)}{1+\tau s} + \frac{\tau(s\Phi(s) + N_g(s))}{1+\tau s} \tag{3.55}$$

Le schéma final du système est illustré par :

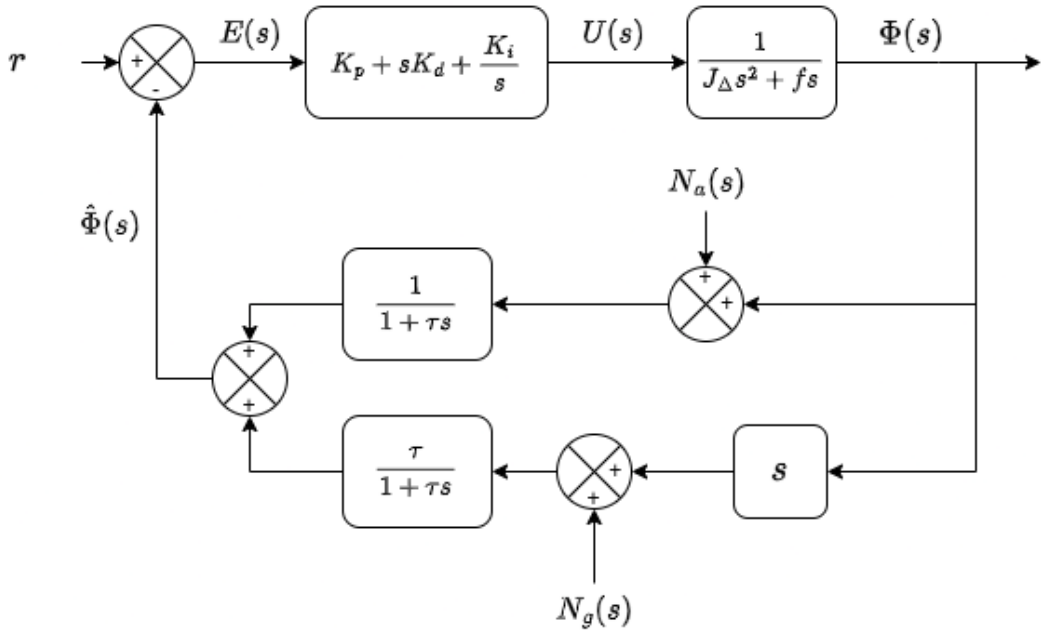

Figure 3.9: *Schéma final du système.*

3.5 Calibration des Capteurs

Apres avoir remedier aux erreurs et bruits des capteurs, les seuls parametres des modeles des capteurs restants sont les biais des capteurs.

Pratiquement on mesure en repos un nomber N de mesures (N = 500) pour chaque capteur et on calcule la moyenne des mesures pour chaque axe. Cette methode est efficace en court terme mais en long terme les biais des capteurs peuvent changer.

3.5.1 Biais de l'Accéléromètre

Pour calibrer le biais de l'accéléromètre, on peut utiliser la mesure de l'accélération gravitationnelle lorsque l'objet est au repos.

$$\vec{a} = -R * \vec{g} + b_a(t) \tag{3.56}$$

$$R = \begin{bmatrix} \cos(\phi)\cos(\psi) & \cos(\phi)\sin(\psi) & -\sin(\phi) \\ \sin(\theta)\sin(\phi)\cos(\psi) - \cos(\theta)\sin(\psi) & \sin(\theta)\sin(\phi)\sin(\psi) + \cos(\theta)\cos(\psi) & \sin(\theta)\cos(\phi) \\ \cos(\theta)\sin(\phi)\cos(\psi) + \sin(\theta)\sin(\psi) & \cos(\theta)\sin(\phi)\sin(\psi) - \sin(\theta)\cos(\psi) & \cos(\theta)\cos(\phi) \end{bmatrix} \tag{3.57}$$

Calibration en repos $\Longrightarrow \phi = 0$ **et** $\theta = 0$ **et** $\psi = 0$

Donc la matrice de rotation devient :

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = I_{\mathbb{R}^3} \tag{3.58}$$

$$\vec{b_a} = \begin{bmatrix} a_x \\ a_y \\ a_z + 9.81 \end{bmatrix} \tag{3.59}$$

On peut deduire les biais de l'accéléromètre en utilisant les equations suivantes :

$$b_{ax} = a_x \tag{3.60}$$
$$b_{ay} = a_y \tag{3.61}$$
$$b_{az} = a_z + 9.81 \tag{3.62}$$

Pour une plage de mesure de l'accéléromètre de $\pm 4g$ on a : g = 8192 (raw sensor)

$$b_{ax} = a_x \tag{3.63}$$
$$b_{ay} = a_y \tag{3.64}$$
$$b_{az} = a_z + 8192 \tag{3.65}$$

3.5.2 Biais du Gyroscope

Pour calibrer le biais du gyroscope, on peut utiliser la mesure de la vitesse angulaire lorsque l'objet est au repos.

$$\omega = \frac{d\phi}{dt} + b_g(t) \tag{3.66}$$

$$\omega = 0 + b_g(t) \tag{3.67}$$

On peut deduire le biais du gyroscope en utilisant l'equation suivante :

$$\vec{b_g} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{3.68}$$

$$b_{gx} = \omega_x \tag{3.69}$$
$$b_{gy} = \omega_y \tag{3.70}$$
$$b_{gz} = \omega_z \tag{3.71}$$

3.6 Conclusion

Ce qu'il faut envisager au pratique et implementation du système :

1. **Calibration des capteurs :** Dans un premier temps en fonction setup() on doit calibrer les capteurs pour obtenir les valeurs des biais des capteurs.
2. **Estimateur :** Implementation et execution en boucle de l'estimateur d'orientation dimensionné dans la partie précédente avec valeur de τ, α ou bien f_c adaptées.
3. **Commande :** Implementation et execution en boucle du correcteur PID pour stabiliser le système.

4

Conception du Système

4.1 Introduction

Dans ce chapitre, nous allons présenter la conception du système. Nous allons commencer par présenter la conception mecanique du système, ensuite nous allons présenter la conception électronique du système.

4.2 Conception Mécanique

4.2.1 Schema Simple

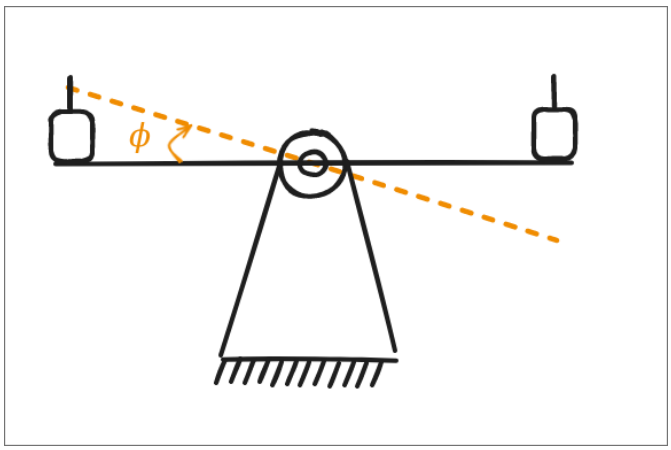

Figure 4.1: *Schéma Simple du Système*

Le système est principalement composé de deux parties : la partie mobile et la partie fixe. La partie mobile est composée de deux moteurs brushless qui sont montés sur une barre rotative. La partie fixe est composée d'une base qui supporte la barre rotative. La figure 4.1 montre le schéma simple du système.

Deux liaisons mécaniques seront utilisées. La première liaison est une liaison pivot qui permet à la barre rotative de tourner autour de l'axe vertical. La deuxième liaison est une liaison encastrement des moteurs à la barre rotative et du support à la base.

4.2.2 Modele 3D

Pour assurer la fixation des moteurs à la barre rotative, on va percer les memes trous existants dans les moteurs dans les deux extrémités de la barre rotative. Ensuite, on va fixer les moteurs à la barre rotative en utilisant des vis.

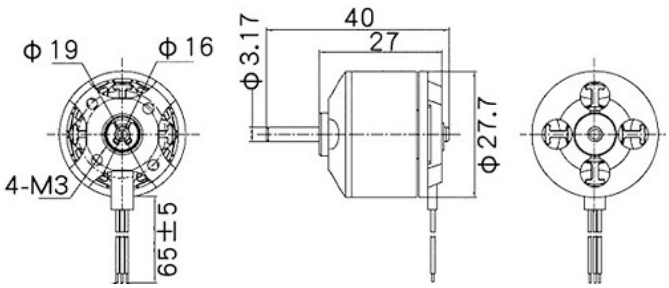

Figure 4.2: *Dessin technique du moteur*

Pour assurer la fluidité de la rotation de la barre rotative, on va utiliser un roulement à billes pour la liaison pivot. Ce roulement à billes sera fixé à la base et à la barre rotative.

La partie fixe est composée d'une base qui supporte la barre rotative et une table de bois.

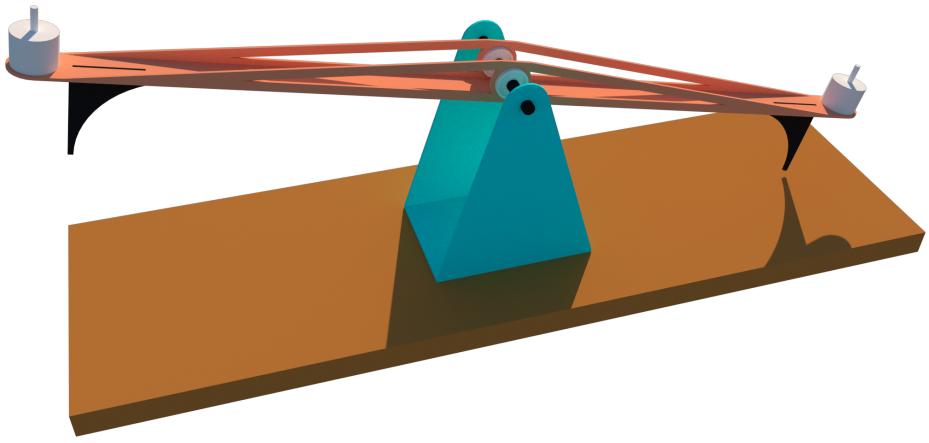

Figure 4.3: *Modele 3D du design mecanique*

Après avoir couper et souder les différentes pièces metaliques pour former le modèle, on va obtenir le modèle mecanique comme montré ci-dessous.

Figure 4.4: *Modèle Mecanique 1*

Figure 4.5: *Modèle Mecanique 2*

4.3 Conception Électronique

4.3.1 Carte d'aquisition et de controle

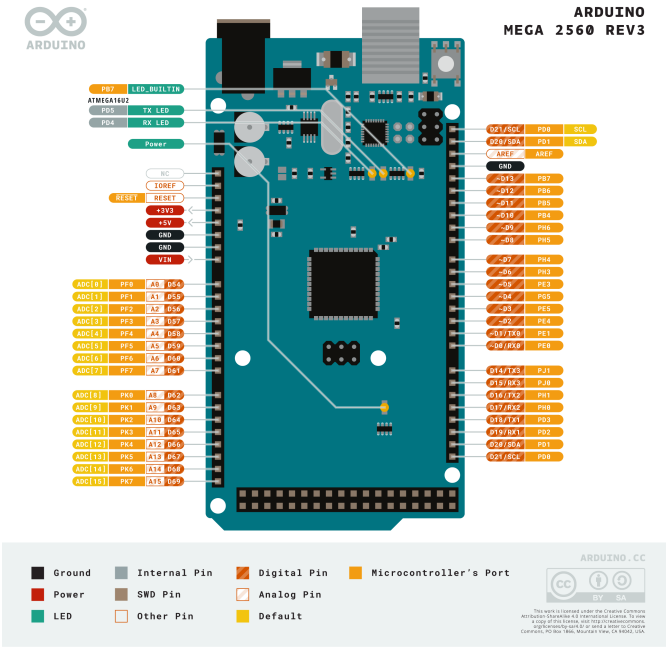

Figure 4.6: *Arduino Mega 2560 Pins Layout*

On va utiliser une carte Arduino ATMega 2560 pour la gestion des capteurs et des actionneurs. Cette carte est équipée de plusieurs entrées/sorties numériques et analogiques qui nous permettent de connecter les différents capteurs et actionneurs.

Cette carte peut communiquer en utilisant plusieurs protocoles de communication comme le protocole I2C ou SPI qui nous seront utile pour communiquer avec l'IMU. Aussi, cette carte contient des pins PWM qui nous permettent de contrôler les deux moteurs controlleurs.

4.3.2 MPU 6050

Le MPU 6050 est un capteur d'inertie qui combine un accéléromètre et un gyroscope. Il est utilisé pour mesurer l'accélération et la vitesse angulaire. Il communique avec la carte Arduino en utilisant le protocole I2C ou SPI.

Plages de mesures

Le MPU 6050 peut mesurer l'accélération selon les trois axes X, Y et Z avec une plage de mesure de $\pm 2g$, $\pm 4g$, $\pm 8g$ **et** $\pm 16g$. **Il peut aussi mesurer la vitesse angulaire selon les trois axes X, Y et Z avec une plage de mesure de** $\pm 250\ °/s$, $\pm 500\ °/s$, $\pm 1000\ °/s$ **et** $\pm 2000\ °/s$.

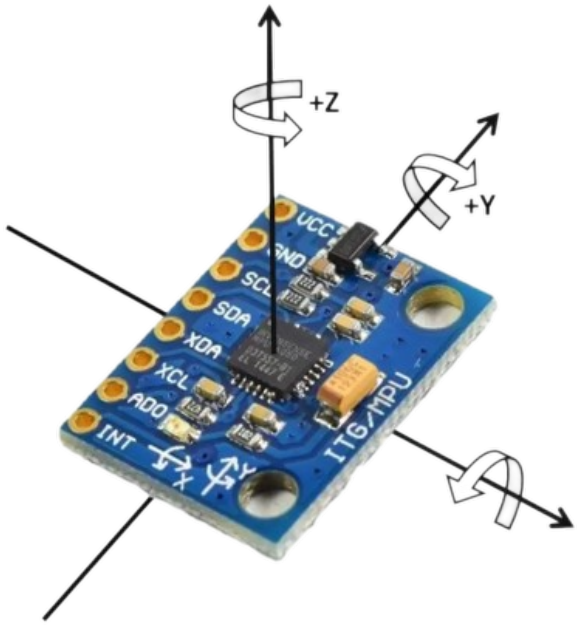

Figure 4.7: *MPU 6050*

Pins

Le MPU 6050 contient 8 pins qui sont :

- VCC : Alimentation 3.3V ou 5V
- GND : Masse
- SDA : Données I2C
- SCL : Horloge I2C
- XDA : Données I2C
- XCL : Horloge I2C
- AD0 : Adresse I2C
- INT : Interruption

Pour utiliser le MPU on peut utiliser les pins SDA et SCL pour communiquer en utilisant le protocole I2C. Et on laisse AD0 flottant pour utiliser l'adresse x68 par défaut pour l'MPU 6050. Donc on va utiliser les pins SDA, SCL et VCC pour alimenter le MPU 6050 et GND.

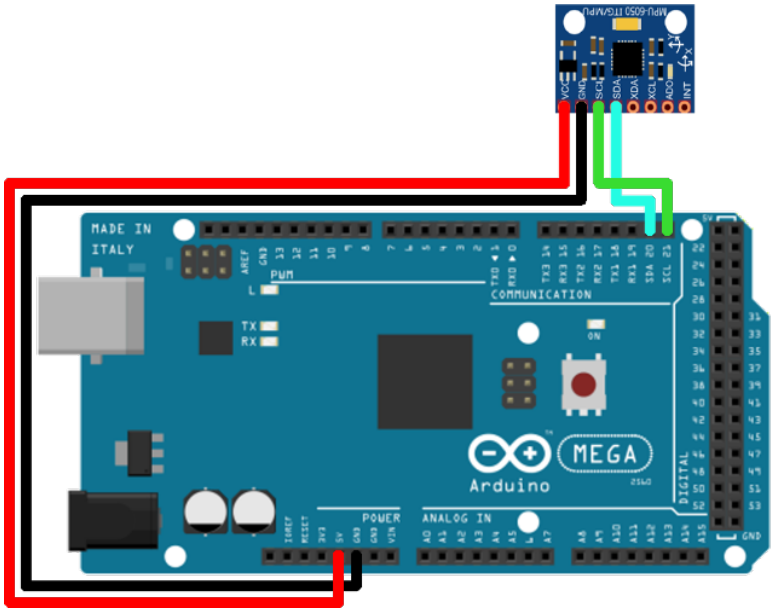

Figure 4.8: *MPU 6050 Pins*

4.3.3 Moteurs et leurs drivers

On va utiliser deux moteurs brushless 1000KV pour la propulsion. Ces moteurs sont alimentés et commander par des ESC (Electronic Speed Controller) qui sont des drivers pour les moteurs brushless.

Ces moteurs contients 3 fils (triphase) qui seront connectés aux ESC.

Les ECS sont connectés à la carte Arduino en utilisant les pins PWM pour contrôler la vitesse des moteurs. et alimenter par une batterie LiPo 11.1V.

Pinning

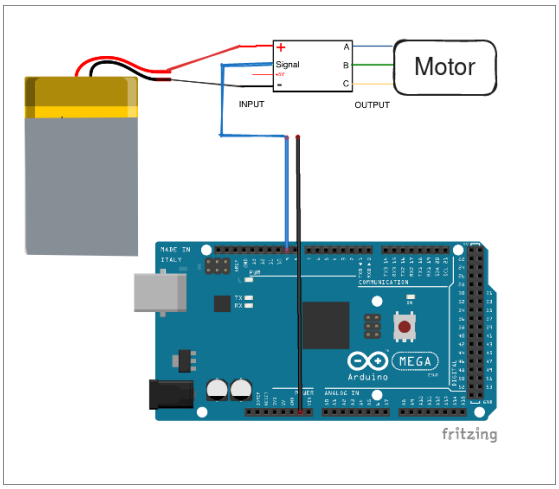

Figure 4.9: *ESC Pinning*

4.3.4 Circuit Electrique

Apres avoir vu chaque composant du système, nous allons maintenant voir comment les connecter ensemble. La figure 4.10 montre le cablage du système.

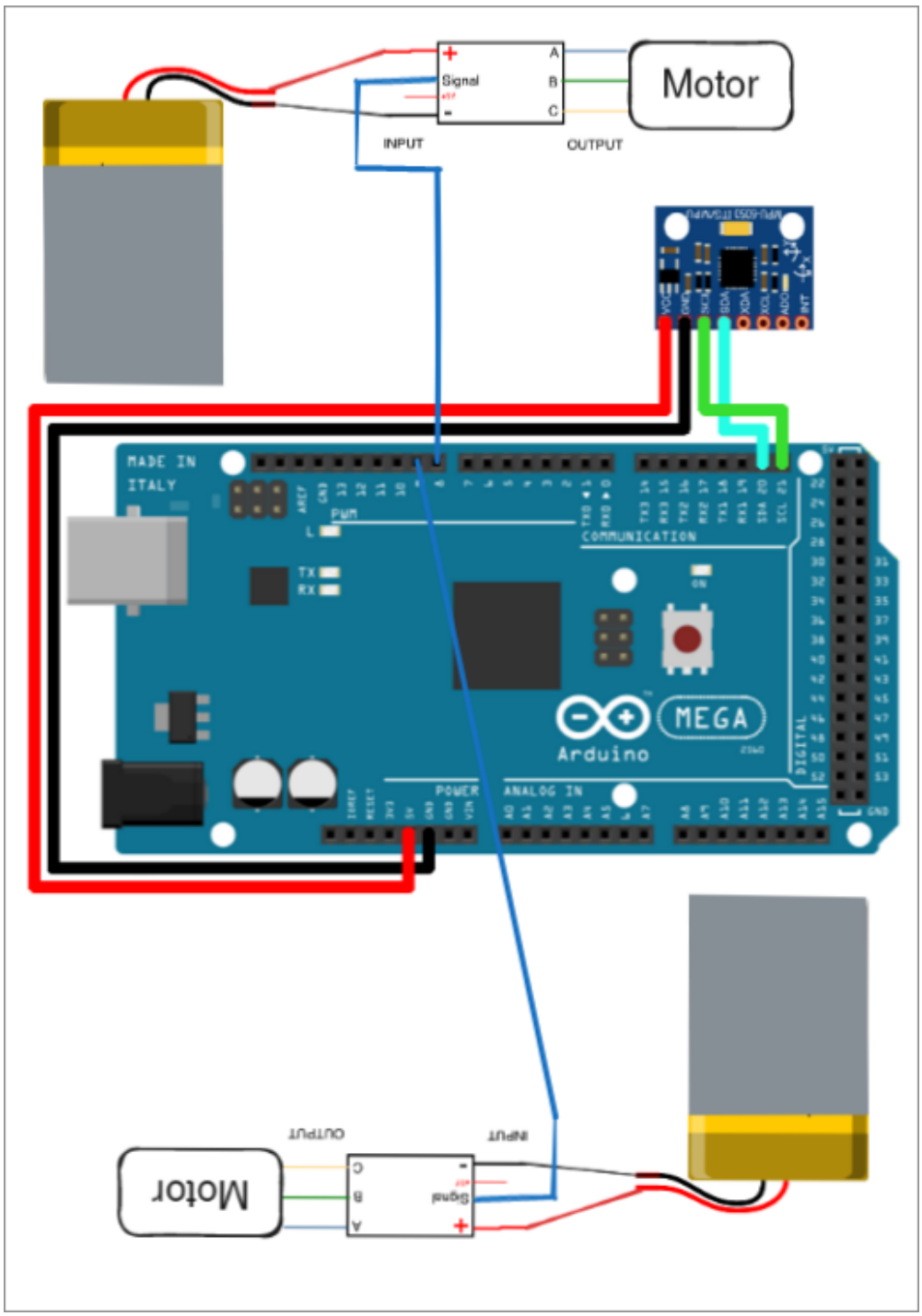

Figure 4.10: *Circuit Final du Système*

5

Development Logiciel

5.1 Introduction

Ce chapitre présente le développement logiciel du projet. il compote deux parties principales :

- **Logiciel Embarqué (Arduino) :** Implémentation du code Arduino pour le système de stabilisation.
- **Tableau de bord (Web) :** Implémentation du tableau de bord Web pour la supervision du système en temps réel.

Le code Arduino est utilisé pour contrôler les capteurs et les actionneurs du système de stabilisation. Il est basé sur le Framework d'Arduino pour la gestion des capteurs et des actionneurs en utilisant la bibliothèque Wire pour la communication I2C et la bibliothèque ESC pour le contrôle des moteurs.

Le tableau de bord Web est utilisé pour visualiser les données du système en temps réel. Il est constitué d'un Serveur Web utilisant UART Serial pour communiquer avec la carte arduino et d'une interface Web qui se connecte au serveur Web avec WebSockets pour afficher les données en temps réel.

Le code source complet est disponible sur le dépôt GitHub du projet :

`https://github.com/amineNouabi/arm-stabilizer`

5.2 Logiciel Embarqué

Pour faciliter la tache de développement, On definit des constantes pour tous les adresses registres de l'IMU utilises et les valeurs des plages de mesure. Par exemple pour les registres d'acceleration, de vitesse angulaire et de temperature.

```
1     ...
2     ...
3  #define MPU6050_RA_ACCEL_XOUT_H     0x3B
4  #define MPU6050_RA_ACCEL_XOUT_L     0x3C
5  #define MPU6050_RA_ACCEL_YOUT_H     0x3D
6  #define MPU6050_RA_ACCEL_YOUT_L     0x3E
7  #define MPU6050_RA_ACCEL_ZOUT_H     0x3F
8  #define MPU6050_RA_ACCEL_ZOUT_L     0x40
9  #define MPU6050_RA_TEMP_OUT_H       0x41
10 #define MPU6050_RA_TEMP_OUT_L       0x42
11 #define MPU6050_RA_GYRO_XOUT_H      0x43
12 #define MPU6050_RA_GYRO_XOUT_L      0x44
13 #define MPU6050_RA_GYRO_YOUT_H      0x45
14 #define MPU6050_RA_GYRO_YOUT_L      0x46
15 #define MPU6050_RA_GYRO_ZOUT_H      0x47
16 #define MPU6050_RA_GYRO_ZOUT_L      0x48
17    ...
18    ...
```

Listing 5.1: *Registers Addresses*

5.2.1 Composants Logiciels

Le code est divisé en plusieurs fichiers pour faciliter la gestion et la maintenance du code. Les fichiers principaux sont :

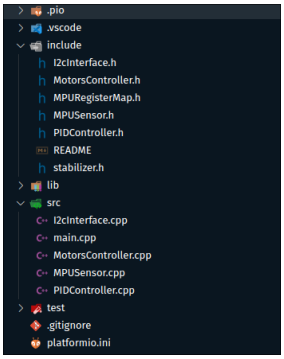

Figure 5.1: *Structure du Dossier*

L'approche orientée objet est utilisée pour structurer le code en classes. Cela permet de regrouper les fonctions et les variables associées dans des classes, ce qui facilite la gestion et la maintenance du code.

Diagramme UML des classes :

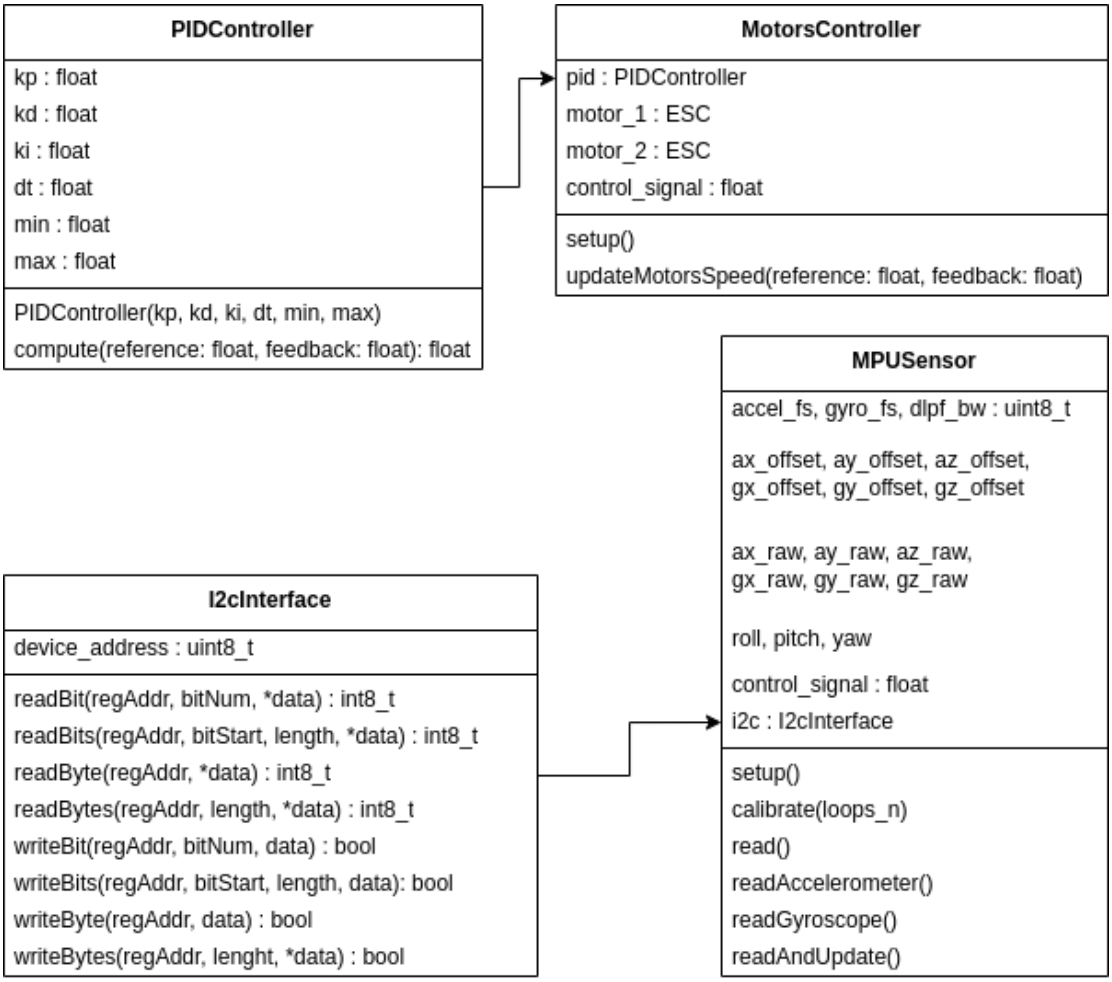

Figure 5.2: *Diagramme de Classes*

5.2.2 I2cInterface Class

La classe I2cInterface est utilisée pour communiquer avec les périphériques I2C. Elle contient les fonctions pour lire et écrire des données sur le bus I2C.

```
#ifndef _I2C_INTERFACE_
#define _I2C_INTERFACE_

#include "stabilizer.h"

class I2cInterface
{
private:
    uint8_t deviceAddress;
    uint16_t readTimeout;

public:
    I2cInterface(uint8_t deviceAddress, uint16_t readTimeout);

    int8_t readBytes(uint8_t regAddr, uint8_t length, uint8_t *data);
    int8_t readByte(uint8_t regAddr, uint8_t *data);
    int8_t readBit(uint8_t regAddr, uint8_t bitNum, uint8_t *data);
    int8_t readBits(uint8_t regAddr, uint8_t bitStart, uint8_t length, uint8_t *data);

    bool writeBytes(uint8_t regAddr, uint8_t length, uint8_t *data);
    bool writeByte(uint8_t regAddr, uint8_t data);
    bool writeBit(uint8_t regAddr, uint8_t bitNum, uint8_t data);
    bool writeBits(uint8_t regAddr, uint8_t bitStart, uint8_t length, uint8_t data);
};

#endif /* _I2C_INTERFACE_ */
```

Listing 5.2: *Classe I2cInterface*

La fonction la plus pertinente readBytes est utilisée pour lire plusieurs octets à partir d'un registre. Elle prend en entrée l'adresse du registre, le pointeur vers le tampon de données et le nombre d'octets à lire.

```
int8_t I2cInterface::readBytes(uint8_t regAddr, uint8_t length, uint8_t *data)
{
    if (length >= BUFFER_LENGTH)
    {
#ifdef DEBUG_LOGS
        Serial.println("I2C error: Uimplemented feature reading more than 32 bytes at
        ↪  once.");
#endif
        return (-1);
    }

    uint8_t count = 0;
    uint32_t initialTime = millis();

    Wire.beginTransmission(deviceAddress);
    Wire.write(regAddr);
    Wire.endTransmission();
    Wire.requestFrom(deviceAddress, length);

    while (Wire.available())
    {
        if (millis() - initialTime >= readTimeout)
        {
#ifdef DEBUG_LOGS
            Serial.println("I2C error: Read Timeout");
#endif
            return (-1);
        }
        data[count++] = Wire.read();
    }

    if (count < length)
    {
#ifdef DEBUG_LOGS
        Serial.println("I2C error: mismatch in read length");
#endif
        return (-1);
    }

    return count;
}
```

Listing 5.3: *Implementation de la fonction readBytes*

5.2.3 MPUSensor Class

La classe MPUSensor implemente l'estimateur de l'orientation et utilise La classe I2cInterface pour communiquer avec l'IMU MPU6050. Elle contient les fonctions pour lire les données de l'IMU, initialiser la configuration de l'IMU, calibrer l'IMU.

```
class MPUSensor
{
private:
    uint8_t accel_fs, gyro_fs, dlpf_bw;
    long ax_offset, ay_offset, az_offset, gx_offset, gy_offset, gz_offset;
    I2cInterface i2c;

    const float _fusion_alpha = 0.9996f;
    const double _g_scale[4] = {131.0, 65.5, 32.8, 16.4};
    const double _a_scale[4] = {16384.0, 8192.0, 4096.0, 2048.0};

public:
    const double _r2d = 180.0f / 3.14159265359f;

    long ax_raw = 0, ay_raw = 0, az_raw = 0;
    long gx_raw = 0, gy_raw = 0, gz_raw = 0;

    double gx_rad_s = 0.0, gy_rad_s = 0.0, gz_rad_s = 0.0;
    double gx_deg_s = 0.0, gy_deg_s = 0.0, gz_deg_s = 0.0;

    double ax_g = 0.0, ay_g = 0.0, az_g = 0.0;

    double roll_deg = 0.0, pitch_deg = 0.0, yaw_deg = 0.0;
    double roll_rad = 0.0, pitch_rad = 0.0, yaw_rad = 0.0;

    double a_roll = 0.0, a_pitch = 0.0;
    double g_roll = 0.0, g_pitch = 0.0;
    double g_yaw = 0.0;

    float delta_t;

    MPUSensor(
        float delta_t,
        uint8_t gyro_fs = MPU6050_GYRO_FS_250,
        uint8_t accel_fs = MPU6050_ACCEL_FS_4,
        uint8_t dlpf_bw = MPU6050_DLPF_BW_42,
        uint8_t mpu_address = MPU6050_DEFAULT_ADDRESS);

    int8_t getDeviceId();
    void calibrate(uint16_t loops_number);
    void setup();
    void read();
    void readAccelerometer(bool is_calibrating = false);
    void readGyroscope(bool is_calibrating = false);
    void readAndUpdate();
};
```

Listing 5.4: *Classe MPUSensor*

La creation d'objet de la classe MPUSensor est modulaire et reglable en utilisant le constructeur de la classe.

Par exemple pour les valeurs suivantes :

- **Periode d'echantillonnage :** 0.004s
- **Plage de mesure de l'accélération :** $\pm 4g$
- **Plage de mesure de la vitesse angulaire :** $\pm 250°/s$
- **Bande passante du filtre passe-bas :** 42Hz
- **Adresse I2C de l'IMU :** 0x69

```
/**
 * @brief Create a new MPUSensor object with the specified parameters
 *
 * @param delta_t Periode d'echantillonnage =  0.004s
 * @param accelRange Accelerometer full scale range = 4g
 * @param gyroRange Gyroscope full scale range = 250°/s
 * @param dlpfBandwidth Digital low pass filter bandwidth = 42Hz
 * @param address I2C address of the MPU6050

 */

 MPUSensor mpu(
    0.004f,
    MPU6050_ACCEL_FS_4,
    MPU6050_GYRO_FS_250,
    MPU6050_DLPF_BW_42,
    MPU6050_ADDRESS_AD0_HIGH
);
```

Listing 5.5: *Creation d'objet de la classe MPUSensor*

L'implementation des 3 fonctions pertinentes de la classe MPUSensor est presentée ci-dessous :

- **setup :** Initialiser la configuration de l'IMU
- **calibrate :** Calibrer l'IMU
- **readAndUpdate :** Lire les données de l'IMU et mettre à jour l'orientation

```
void MPUSensor::setup()
{
    while (getDeviceId() != MPU6050_WHO_AM_I)
        delay(5000);

#ifdef DEBUG_LOGS
    Serial.println("Successfully Found MPU Sensor ...");
    Serial.println("Configuring MPU Sensor ...");
#endif
    // By default the MPU-6050 sleeps. So we have to wake it up.
    // We want to write to the PWR_MGMT_1 register (6B hex) 0 at SLEEP bit
    if (!i2c.writeBit(MPU6050_RA_PWR_MGMT_1, MPU6050_PWR1_SLEEP_BIT, 0))
    {
#ifdef DEBUG_LOGS
        Serial.println("Error: Could not wake up MPU Sensor.");
#endif
    }
    // Set the Clock Source to PLL with X axis gyroscope reference
    if (!i2c.writeBits(MPU6050_RA_PWR_MGMT_1, MPU6050_PWR1_CLKSEL_BIT,
    ↪  MPU6050_PWR1_CLKSEL_LENGTH, 0x01))
    {
#ifdef DEBUG_LOGS
        Serial.println("Error: Could not set the clock source to PLL with X axis
        ↪  gyroscope reference.");
#endif
    }
    // Set the full scale of the gyro to (default = +- 250°/s).
    if (!i2c.writeBits(MPU6050_RA_GYRO_CONFIG, MPU6050_GCONFIG_FS_SEL_BIT,
    ↪  MPU6050_GCONFIG_FS_SEL_LENGTH, gyro_fs))
    {
#ifdef DEBUG_LOGS
        Serial.println("Error: Could not set the full scale of the gyro.");
#endif
    }
    // Set the full scale of the accelerometer to (default = +- 4g).
    if (!i2c.writeBits(MPU6050_RA_ACCEL_CONFIG, MPU6050_ACONFIG_AFS_SEL_BIT,
    ↪  MPU6050_ACONFIG_AFS_SEL_LENGTH, accel_fs))
    {
#ifdef DEBUG_LOGS
        Serial.println("Error: Could not set the full scale of the accelerometer to +/-
        ↪  4g.");
#endif
    }
    // Set some filtering to improve the raw data. BandWidth (default = 42Hz)
    if (!i2c.writeBits(MPU6050_RA_CONFIG, MPU6050_CFG_DLPF_BIT,
    ↪  MPU6050_CFG_DLPF_LENGTH, dlpf_bw))
    {
#ifdef DEBUG_LOGS
        Serial.println("Error: Could not set some filtering to improve the raw data.");
#endif
    }
    calibrate(CALIBRATION_LOOP_N);
    delay(1000);
}
```

Listing 5.6: *Implementation de la fonction setup de la classe MPUSensor*

```
void MPUSensor::calibrate(uint16_t loops_number)
{
#ifdef DEBUG_LOGS
    Serial.println("-----------------------------------");
    Serial.println("Starting MPU Sensor calibration ...");
#endif
    ax_offset = ay_offset = az_offset = gx_offset = gy_offset = gz_offset = 0;
    uint16_t counter;
    for (counter = 0; counter < loops_number; counter++)
    {
        readAccelerometer(true);
        ax_offset += ax_raw;
        ay_offset += ay_raw;
        az_offset += az_raw;
#ifdef DEBUG_LOGS
        Serial.print(".");
#endif
        delayMicroseconds(3700);
    }
    ax_offset /= loops_number;
    ay_offset /= loops_number;
    az_offset /= loops_number;
#ifdef DEBUG_LOGS
    Serial.println("");
    Serial.println("Accelerometer Calibration done.");
    Serial.println("ax_offset:\t" + String(ax_offset));
    Serial.println("ay_offset:\t" + String(ay_offset));
    Serial.println("az_offset:\t" + String(az_offset));
    Serial.println("Starting Gyroscope Calibration ...");
#endif
    for (counter = 0; counter < loops_number; counter++)
    {
        readGyroscope(true);
        gx_offset += gx_raw;
        gy_offset += gy_raw;
        gz_offset += gz_raw;
#ifdef DEBUG_LOGS
        Serial.print(".");
#endif
        delayMicroseconds(3700);
    }
    gx_offset /= loops_number;
    gy_offset /= loops_number;
    gz_offset /= loops_number;
#ifdef DEBUG_LOGS
    Serial.println("");
    Serial.println("Gyroscope Calibration done.");
    Serial.println("gx_offset:\t" + String(gx_offset));
    Serial.println("gy_offset:\t" + String(gy_offset));
    Serial.println("gz_offset:\t" + String(gz_offset));
    Serial.println("Calibration done.");
#endif
}
```

Listing 5.7: *Implementation de la fonction calibrate*

```
void MPUSensor::readAndUpdate()
{
    read();
    // Roll calculation
    a_roll = atan2(-ay_g, sqrt(ax_g * ax_g + az_g * az_g)) * _r2d;
    g_roll = roll_deg + gx_deg_s * delta_t;
    roll_rad = (roll_deg = _fusion_alpha * g_roll + (1 - _fusion_alpha) * a_roll) /
    ↪  _r2d;
    // Pitch calculation
    a_pitch = atan2(ax_g, sqrt(ay_g * ay_g + az_g * az_g)) * _r2d;
    g_pitch = pitch_deg + gy_deg_s * delta_t;
    roll_rad = (pitch_deg = _fusion_alpha * g_pitch + (1 - _fusion_alpha) * a_pitch) /
    ↪  _r2d;
    // Yaw calculation
    g_yaw = yaw_deg + gz_deg_s * delta_t;
    yaw_rad = (yaw_deg = g_yaw) / _r2d;
}

void MPUSensor::read()
{
    readAccelerometer();
    readGyroscope();
}

void MPUSensor::readAccelerometer(bool is_calibrating = false)
{
    uint8_t buffer[6];
    if (i2c.readBytes(MPU6050_RA_ACCEL_OUT, 6, buffer) != 6)
    {
#ifdef DEBUG_LOGS
        Serial.println("Error: Could not read the accelerometer data.");
#endif
        return;
    }
    ax_raw = ((((int16_t)buffer[0]) << 8) | buffer[1]);
    ay_raw = ((((int16_t)buffer[2]) << 8) | buffer[3]);
    az_raw = ((((int16_t)buffer[4]) << 8) | buffer[5]);
    if (is_calibrating)
        return;

    ax_raw -= ax_offset;
    ay_raw -= ay_offset;
    az_raw -= az_offset + _a_scale[accel_fs];

    ax_g = ax_raw / _a_scale[accel_fs];
    ay_g = ay_raw / _a_scale[accel_fs];
    az_g = az_raw / _a_scale[accel_fs];
}
```

Listing 5.8: *Implementation de la fonction readAndUpdate*

5.2.4 PID

La classe PID est utilisée pour implémenter l'algorithme de contrôle PID. Elle contient les fonctions pour initialiser les paramètres du PID, calculer la commande de contrôle et ajuster les paramètres du PID.

```
#ifndef _PID_CONTROLLER_H_
#define _PID_CONTROLLER_H_

#include "stabilizer.h"

class PIDController
{
public:
    PIDController(const float kp, const float ki, const float kd, const float dt,
    ↪  const float min, const float max);
    ~PIDController();
    float compute(const float ref, const float feedback);

private:
    /* Control law constants */
    const float kp_ = 0, ki_ = 0, kd_ = 0, dt_ = 0, min_, max_;
    /* PID responses */
    float yp_ = 0, yd_ = 0;
    /* Derivative state */
    float prev_error = 0;
    /* Integrator state */
    float istate_ = 0;
    /* Output */
    float y_;
    /* Clamping */
    float anti_windup_ = 1;
};

#endif // _PID_CONTROLLER_H_
```

Listing 5.9: *Classe PID*

La fonction compute est utilisée pour calculer la commande de contrôle en utilisant l'algorithme de contrôle PID. Elle prend en entrée la consigne et la valeur actuelle, et retourne la commande de contrôle.

```
float PIDController::compute(const float ref, const float feedback)
{
    float error = ref - feedback;
    /* Proportional error */
    yp_ = kp_ * error;
    /* Derivative error */
    yd_ = kd_ * (error - prev_error) / dt_;
    prev_error = error;
    /* Compute output */
    y_ = yp_ + ki_ * istate_ + yd_;
    /* Saturation and anti-windup */
    if (y_ > max_)
    {
        y_ = max_;
        if (ki_ * error > 0)
            anti_windup_ = 0;
        else
            anti_windup_ = 1;
    }
    else if (y_ < min_)
    {
        y_ = min_;
        if (ki_ * error < 0)
            anti_windup_ = 0;
        else
            anti_windup_ = 1;
    }
    else
        anti_windup_ = 1;
    /* Integrator state */
    istate_ += dt_ * error * anti_windup_;
    return y_;
}
```

Listing 5.10: *Implementation de la fonction compute*

5.2.5 MotorsController

La classe MotorsController est utilisée pour contrôler les moteurs. Elle contient les fonctions pour initialiser les moteurs, contrôler la vitesse des moteurs et arrêter les moteurs. elle utilise la classe PID pour commander le moteur.

```cpp
#ifndef MOTORSCONTROLLER_HPP
#define MOTORSCONTROLLER_HPP

#define MOTORS_MIN_SPEED 1050.0
#define MOTORS_MAX_SPEED 1400.0

#define MOTORS_ARM_SPEED 1000.0
#define MOTORS_MID_SPEED 1200.0

#define MOTORS_MAX_OFFSET 200
#define MOTORS_MIN_OFFSET -150

#define MOTOR_1_PIN 12
#define MOTOR_2_PIN 2

#include "stabilizer.h"

class MotorsController
{
public:
    PIDController pid;
    ESC motor1, motor2;
    float motor1_offset, motor2_offset;

    // MotorsController(const float dt, const float kp = 1.0, const float ki = 0.1,
    ↪  const float kd = 0.01);
    MotorsController(const float dt, const float kp = 1.7, const float ki = 0.05,
    ↪  const float kd = 0.3);

    void setup();
    void updateMotorsSpeed(float ref_pitch, float pitch);
    void updateMotorOffsets(float offset1, float offset2);

    float control_signal, motor1_speed, motor2_speed;
};

#endif
```

Listing 5.11: *Classe MotorController*

La fonction setup est utilisée pour initialiser les moteurs en utilisant la bibliothèque ESC.

Lafonction update execute le calcul PID et met à jour la vitesse des moteurs.

```
void MotorsController::setup()
{

#ifdef DEBUG_LOGS
    Serial.println("Starting motors setup ....");
    Serial.println("Calibrating 1st motor ...");
#endif
    motor1.calib();

#ifdef DEBUG_LOGS
    Serial.println("Calibrating 2nd motor ...");
#endif
    motor2.calib();

#ifdef DEBUG_LOGS
    Serial.println("Successful Motors setup");
#endif
}
```

Listing 5.12: *Implementation de la fonction setup*

```
void MotorsController::updateMotorsSpeed(float ref_pitch, float pitch)
{
    // Compute the control signal using the PID controller
    control_signal = pid.compute(ref_pitch, pitch);

    // Update motor speeds based on the control signal
    motor1_speed = constrain(MOTORS_MID_SPEED + control_signal - motor1_offset,
    ↪  MOTORS_MIN_SPEED, MOTORS_MAX_SPEED);
    motor2_speed = constrain(MOTORS_MID_SPEED - control_signal - motor2_offset,
    ↪  MOTORS_MIN_SPEED, MOTORS_MAX_SPEED);

    // Update the motors speed
    motor1.speed(motor1_speed);
    motor2.speed(motor2_speed);
}
```

Listing 5.13: *Implementation de la fonction updateMotorsSpeed*

5.2.6 Code Arduino

```
#include "stabilizer.h"

const float delta_t = 0.004f;
const uint16_t delta_t_micros = 4000;

const float alpha_fusion = 0.9996f;
uint32_t loop_timer = 0;
const float pitch_ref = 0.0f;

MPUSensor mpuSensor(delta_t);
MotorsController motorsController(delta_t);

void setup()
{
    // Setup the MPU Sensor
    mpuSensor.setup();

    // Setup the motors
    motorsController.setup();

    loop_timer = micros() + delta_t_micros;
}

void loop()
{
    // Read and update the sensor data: output roll, pitch and yaw
    mpuSensor.readAndUpdate();

    // Update the motors speed
    motorsController.updateMotorsSpeed(pitch_ref, mpuSensor.pitch_deg);

    // Wait for the next loop time
    while (micros() < loop_timer)
        ;
    loop_timer += delta_t_micros;
}

```

Listing 5.14: *Code Arduino*

6

Conclusion

Le projet démontre la faisabilité d'un système de stabilisation pour une barre rotative à un degré de liberté, utilisant des technologies accessibles comme l'Arduino et l'IMU MPU6050. L'algorithme PID, une fois correctement réglé, permet de maintenir la barre stable et horizontale, répondant ainsi aux objectifs initiaux du projet. Les résultats obtenus montrent que des ajustements précis et des tests rigoureux sont essentiels pour garantir la performance et la fiabilité du système.

This page intentionally left blank.

ECOLE NATIONALE SUPÉRIEURE DES
ARTS ET MÉTIERS MEKNES

MISE EN ŒUVRE ET STABILISATION D'UN BRAS DE DRONE QUADRIROTOR

AHMED AMINE NOUABI

MEKNES, FEBRUARY 2025