Maquette pendule à double hélices

Recherche & Développement

Rapport final

ROBO 5

Esteban Gissinger

22 Janvier 2025

Université Nice Sophia Antipolis

POLYTECH NICE-SOPHIA

Table des matières

I Introduction....3
II Asservissement du système....3
II.1 Préparation....3
II.2 Filtres....4
II.2.a Filtre Complémentaire....4
II.2.b Filtre de Kalman....5
II.3 Contrôleurs....8
II.3.a PID....8
II.3.b Cascade....9
II.3.c LQR....9
II.4 IHM....10
III Conclusion....13

I Introduction

Le projet s'articule autour d'un bras monté sur un pivot sur lequel deux moteurs brushless équipés d'hélices ont été montés (Figure 1). Un accéléromètre ainsi qu'un gyromètre sont également fixés sur le bras. Une carte équipée d'un STM32F411 contrôle le tout. Le but du projet est de stabiliser le bras à l'horizontale. Pour cela, il faut dans un premier temps pouvoir estimer la position du bras précisément. La seconde étape consiste à développer un contrôleur qui s'occupe d'ajuster la puissance des moteurs pour atteindre l'objectif fixé.

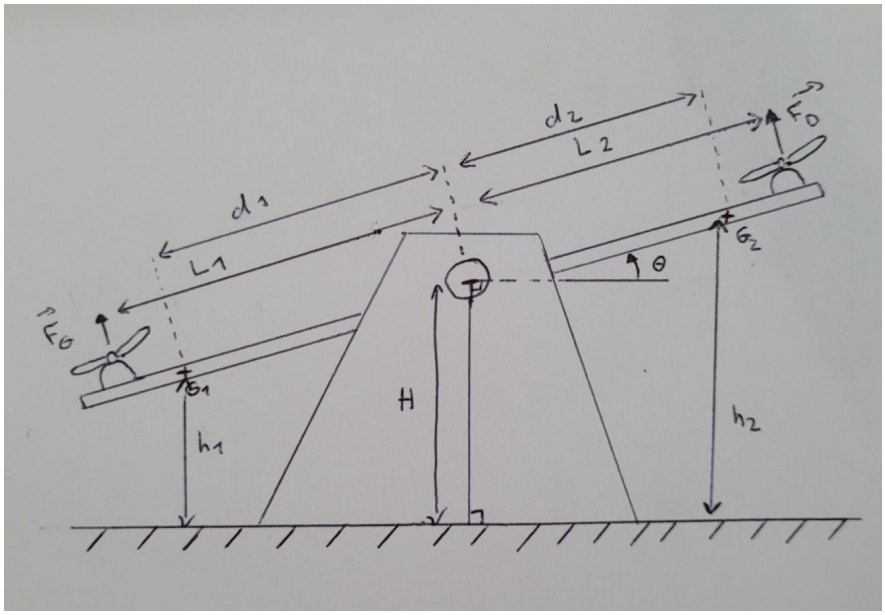

Figure 1: Schéma du système étudié

II Asservissement du système

II.1 Préparation

Avant de pouvoir asservir le système, il faut pouvoir mesurer ses variables d'états, en particulier l'angle ici. Pour estimer la position du bras, on utilise la centrale inertiel (gyromètre + accéléromètre, le magnétomètre n'est pas utilisé car il n'a pas d'intérêt pour notre utilisation) intégrée à la carte de développement. Cela permet d'obtenir la vitesse angulaire ainsi que l'accélération du bras. Si le bras est immobile, alors l'accélération mesurée est celle de la gravité, ce qui permet d'estimer l'angle du bras.
Cependant, lorsque le bras est en mouvement ou plus généralement à cause des vibrations des moteurs, on ne mesure plus uniquement l'accélération du a la gravité. C'est pourquoi on utilise essentiellement la vitesse angulaire que l'on intègre pour obtenir une position. En revanche, celui-ci n'est pas parfait non plus puisqu'il possède un biais qui fait diverger l'estimation et il faut connaître la position de départ. C'est pourquoi on fait du sensor fusion pour combiner les deux capteurs afin d'obtenir une meilleure estimation. Deux filtres seront comparés dans ce projet : le filtre complémentaire et le filtre de Kalman.

L’accéléromètre et le gyromètre sont compris dans un seul circuit intégré, le LSM303DLHC qui communique avec le microcontrôleur (MCU) via I2C, un protocole de communication standardisé. Il suffit donc de lire certains registres pour récupérer les valeurs qui nous intéressent.

Il faut ensuite être capable de contrôlé les actionneurs de notre système. Les deux moteurs brushless sont contrôlés par des ESCs. Il suffit donc d’envoyer un signal PWM (Pulse Width Modulation) aux ESCs pour contrôler les moteurs. Ce signal doit avoir une fréquence de 50 Hz ainsi que des impulsions d’une durée comprise entre 1 ms (0% vitesse) et 2 ms (100% vitesse). Pour générer ce signal, il faut utilisé un timer du MCU avec les bons paramètres, c’est-à-dire adapté la taille du compteur en fonction de la fréquence du MCU et configurer les sorties. Pour se faciliter la tâche, on utilise STM32CubeIDE (Figure 2) qui dispose d’une interface permettant de rapidement générer le code utilisant le HAL (librairie qui abstrait la gestion des registres bas niveau) nécessaire à l’initialisation des différents périphériques.

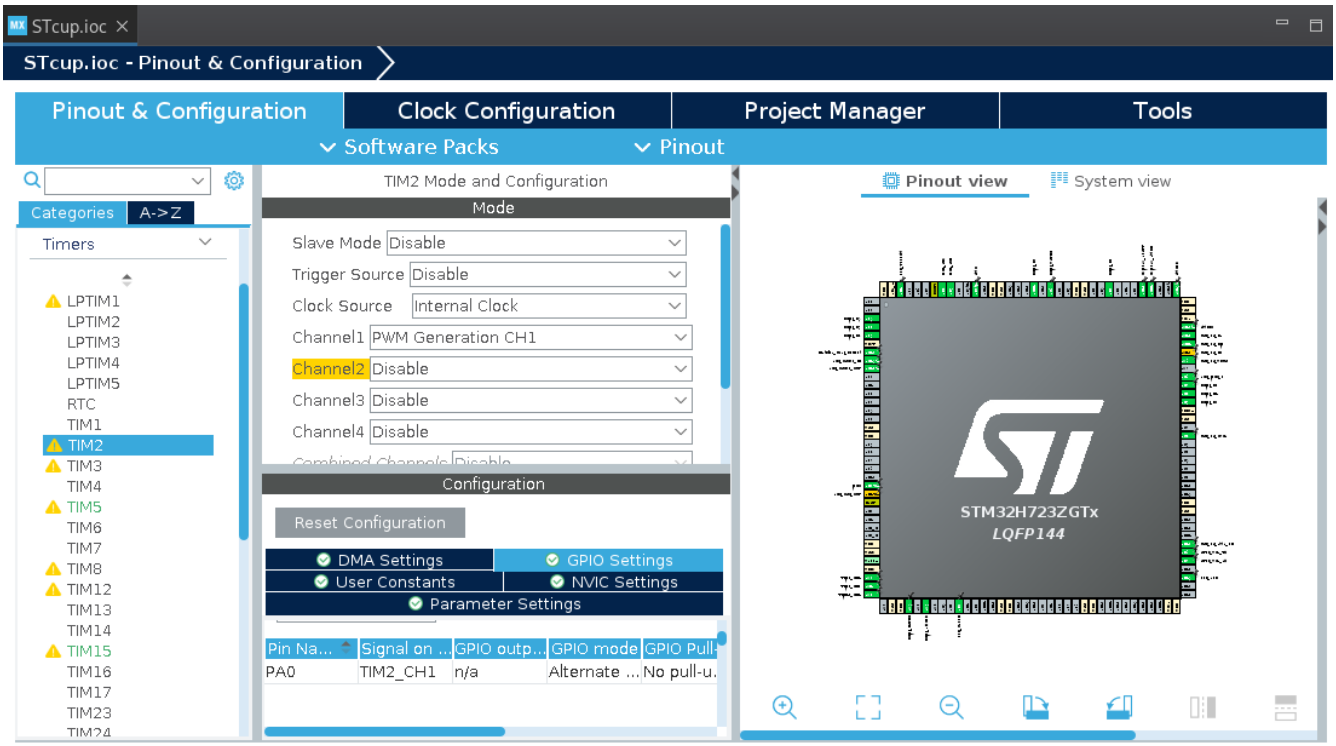

Figure 2: Initialisation des timers sur STM32CubeIDE

II.2 Filtres

II.2.a Filtre Complémentaire

Le premier filtre implémenté est également le plus simple : le filtre complémentaire. Celui-ci consiste simplement à appliquer un filtre passe bas sur l’accéléromètre et un filtre passe haut sur le gyromètre puis additionner les deux (Figure 3).

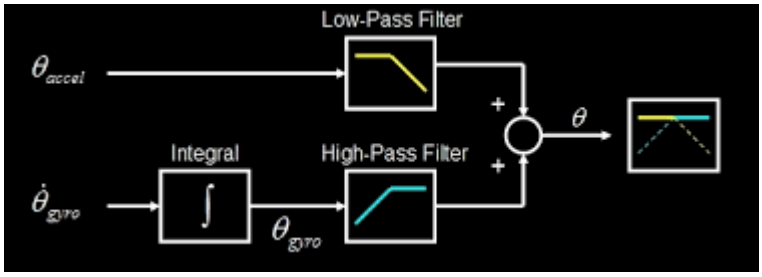

Figure 3: Schéma filtre complémentaire

En pratique, l'implémentation est très simple puisque le schéma ci-dessus se résume aux deux lignes de code suivantes après simplification des expressions du filtre passe haut et passe bas :

```
float theta_gyro = theta + qf * SAMPLE_TIME_S;
theta = theta_gyro * alpha + theta_acc * (1 - alpha);
```

II.2.b Filtre de Kalman

Le filtre de Kalman prend avantage de la connaissance d'un modèle mathématique du système pour obtenir une meilleure estimation. Il consiste en deux étapes : une étape de prédiction grâce au modèle, et une étape de mise à jour de l'incertitude (Figure 4).

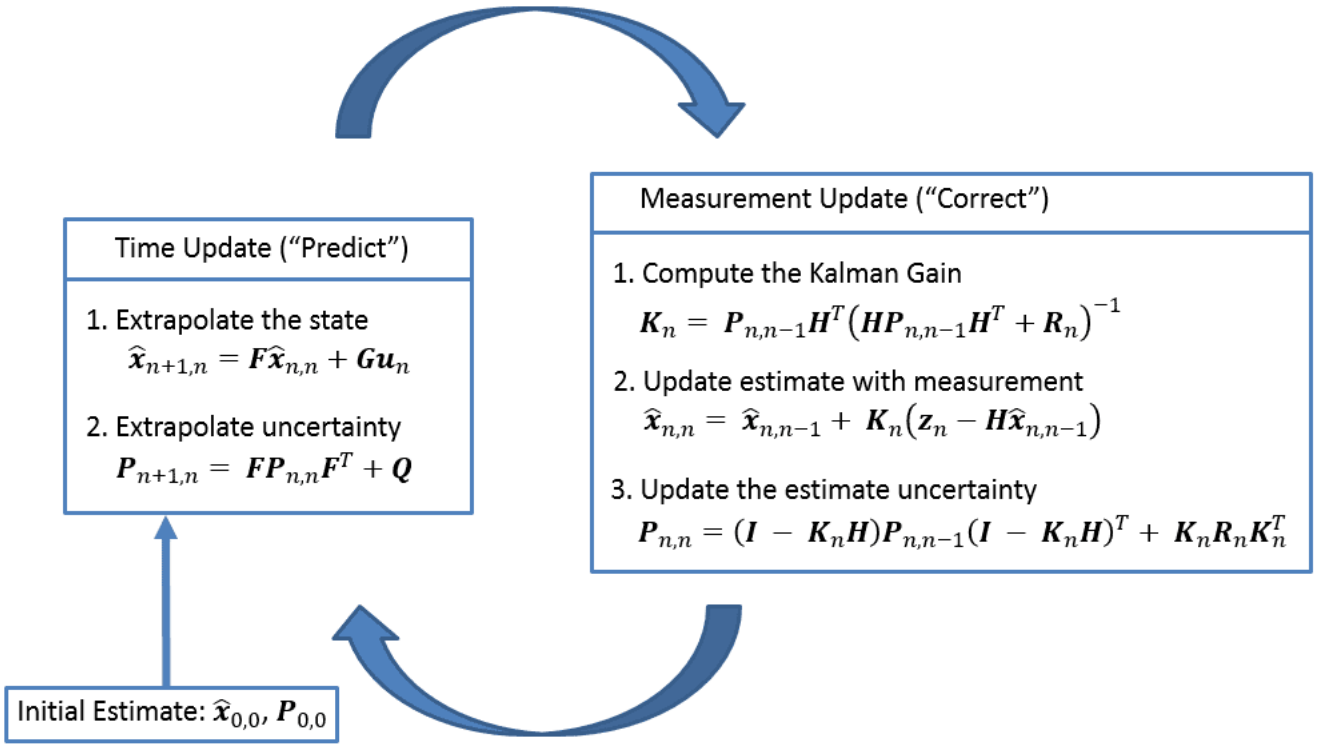

Figure 4: Algorithme filtre de Kalman

Il minimise ainsi l'erreur avec le temps. Dans notre cas, les modèles utilisés sont :

$$\dot{\theta} = \omega - b$$

$$\ddot{\theta} = \frac{L\Delta F}{J}$$

On considère les vecteurs d'état et d'entré suivants :

$$\mathrm{x}= \begin{pmatrix} \theta \\ \dot{\theta} \\ b \end{pmatrix} \quad \mathrm{u}= \begin{pmatrix} \omega \\ \Delta F \end{pmatrix}$$

Ce qui donne les matrices de transition et de contrôle suivantes :

$$\phi_d = I + FT_s = \begin{pmatrix} 1 & 0 & -Ts \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathrm{G}_d = \int_0^{T_s} e^{F\tau} d\tau = (\int_0^{T_s} \phi(\tau) d\tau) G = \begin{pmatrix} T_s & 0 \\ 0 & T_s L/J \\ 0 & 0 \end{pmatrix}$$

Les matrice de mesures et de covariance du bruit sont :

$$\mathrm{H}= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$$\mathrm{R}= \begin{pmatrix} R_\theta & 0 \\ 0 & T_\omega \end{pmatrix}$$

Finalement la matrice de covariance est :

$$\mathrm{Q}= \begin{pmatrix} V_\theta & 0 & 0 \\ 0 & V_{\dot{\theta}} & 0 \\ 0 & 0 & V_b \end{pmatrix}$$

avec V_θ, $\mathrm{V}_{\dot{\theta}}$ et V_b correspondant respectivement au process noise de la position, vitesse angulaire et biais du gyromètre.

Une simulation sur MATLAB est réalisée afin d'estimer les meilleurs paramètres. Les résultats sont visibles sur la Figure 5.

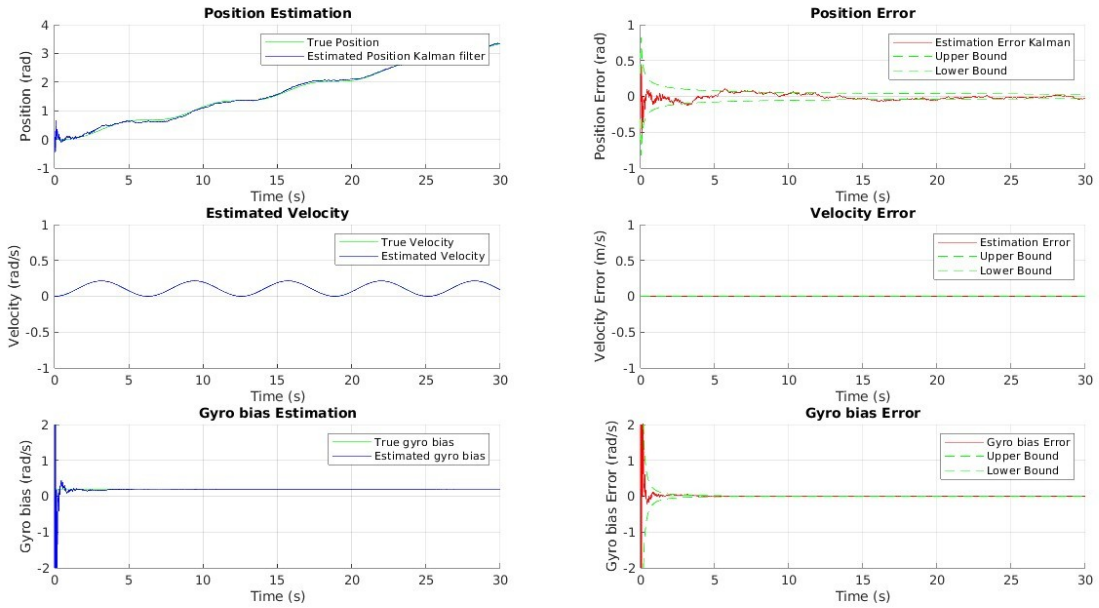

Figure 5: Résutat du filtre de Kalman après simulation sur MATLAB

On observe également que le filtre de Kalman permet d'estimer le biais du gyromètre, ce qui contribue à améliorer la précision. Une autre simulation sur MATLAB permet de comparer le filtre de Kalman avec le filtre complémentaire décris précédemment (Figure 6).

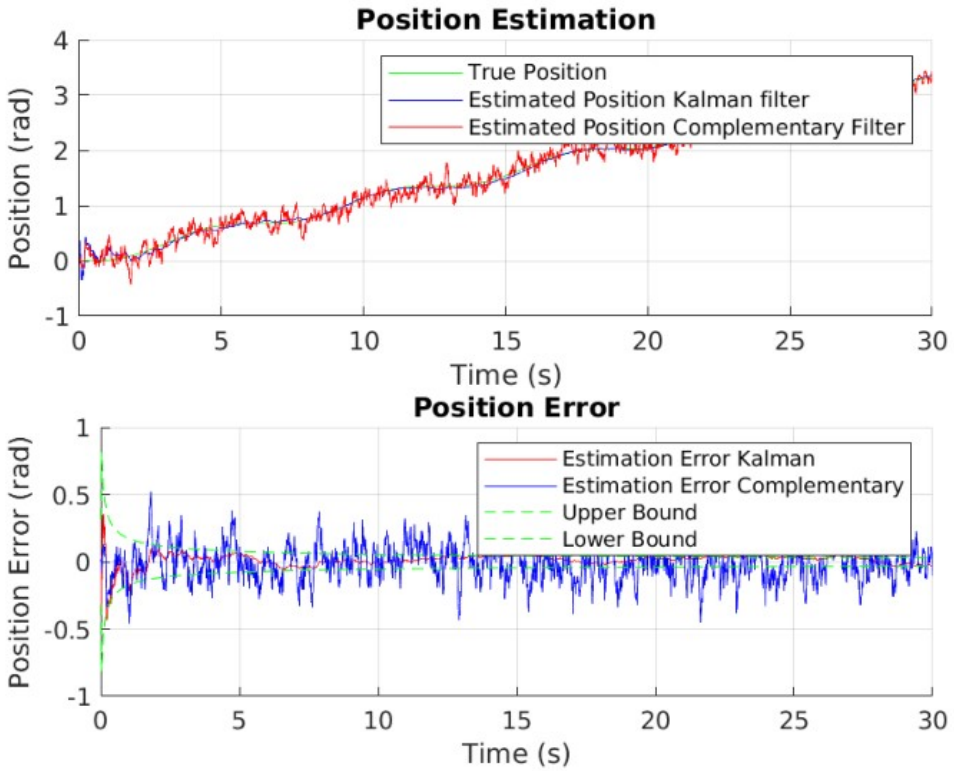

Figure 6: Comparaison des performances entre un filtre de Kalman et un filtre complémentaire sur MATLAB

Cependant, l'implémentation du filtre de Kalman est plus complexe, notamment parce qu'elle nécessite beaucoup plus de puissance de calcul dû aux nombreuses opérations matricielles. Or la puissante du MCU étant limité, il est important d'optimiser le code. C'est pourquoi on utilise le CMSIS, une librairie fournit par Arm© qui contient des fonctions optimisées pour le calcul. En autre, les fonctions *arm_mat_mult_f32*, *arm_mat_add_f32*, *arm_mat_sub_f32*, *arm_mat_inverse_f32*, *arm_mat_trans_f32* qui permettent respectivement de multiplier, ajouter, soustraire, inverser et transposer des matrices. Ces fonctions fonctionnent avec des matrices du type *arm_matrix_instance_f32* et non des tableaux.

II.3 Contrôleurs

II.3.a PID

Le contrôleur PID permet de maintenir un résultat souhaité en calculant l'erreur et en appliquant des actions correctives basées sur des termes proportionnels, intégraux et dérivés (Figure 7). L'avantage de ce contrôleur est qu'il ne nécessite aucun modèle et se règle en ajustant 3 paramètres : *Kp*, *Ki*, *Kd*.

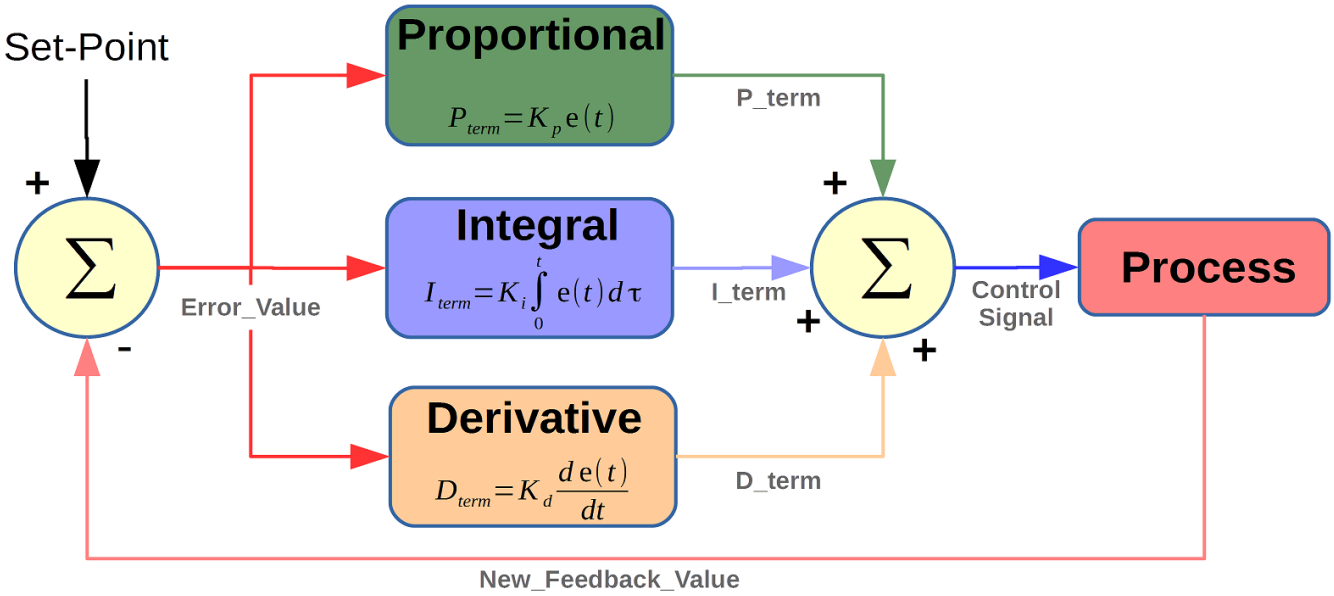

Figure 7: Schéma contrôleur PID

Il ne faut cependant pas oublier de discrétiser le contrôleur avant de l'implémenter sur le MCU. On choisit ici la méthode de Tustin :

$$p[n] = K_p e[n]$$

$$i[n] = \frac{K_i T}{2}(e[n] + e[n-1]) + i[n-1]$$

$$d[n] = \frac{2K_d}{2\tau+T}(e[n] - e[n-1]) + \frac{2\tau-T}{2\tau+T} d[n-1]$$

II.3.b Cascade

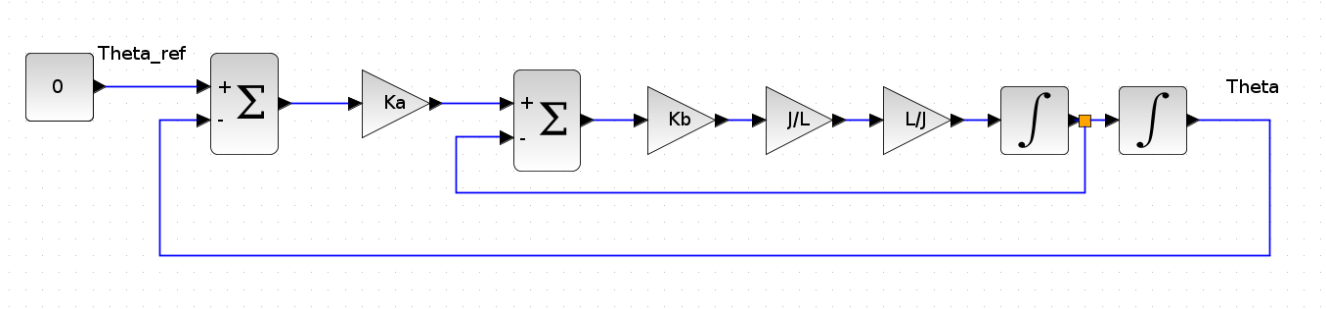

Figure 8: Schéma bloc contrôleur Cascade

$$\theta(s) = \frac{K_\theta K_{\dot\theta}}{s^2 + K_{\dot\theta} s + K_{\dot\theta}} \theta_{ref}(s) = \frac{K\omega_0^2}{s^2 + 2\xi\omega_0 s + \omega_0^2} \theta_{ref}(s)$$

$$\omega_0 = \sqrt{K_\theta K_{\dot\theta}}$$
$$\xi = \frac{K_{\dot\theta}}{4K_\theta}$$
$$\mathrm{K}=1$$

Ce contrôleur permet d'obtenir un système du second ordre. Pour obtenir l'asservissement le plus rapide, il faut choisir les gains pour obtenir $\xi = 0.7$ qui offre le temps de réponse à 5 % le plus faible.

L'implémentation de ce contrôleur est très simple. En effet, il ne fait appel qu'à des multiplications par des constantes, ce qui signifie que la discrétisation du contrôleur est instantanée.

II.3.c LQR

Le contrôleur LQR (Linear Quadratic Regulator) est une stratégie de contrôle optimale qui calcule un gain *K* pour minimiser une fonction de coût, équilibrant les écarts d'état et l'effort de contrôle, pour les systèmes dynamiques linéaires. En cas de non-linéarité, on peut linéariser le système localement pour chaque itération à l'instar d'un filtre de Kalman étendu.

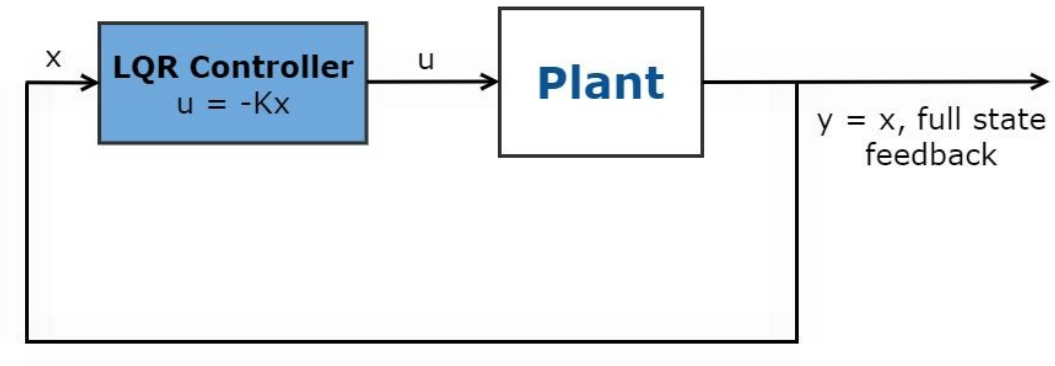

Figure 9: Schéma bloc contrôleur LQR

La fonction de coût est donné par :

$$J = \frac{1}{2}\int_0 \left(x^{\mathrm{T}} Q x + u^{\mathrm{T}} R u \right) \mathrm{d}t$$

avec $u = -Kx$

Ou Q est une matrice qui permet de pénalisé l'écart entre le résultat désiré et celui mesuré et R une matrice qui pénalise l'utilisation des actionneurs (pour économiser de l'énergie par exemple).

K est donné par

$$K = R^{-1}(B^T P(t) + N^T)$$

et P par

$$A^T P(t) + P(t)A - (P(t)B + N)R^{-1}(B^T P(t) + N^T) + Q = -\dot{P}(t)$$

Cette dernière équation est une équation différentielle de Riccati et se résout numériquement avec MATLAB par exemple. Notre système étant linéaire, on peut donc utiliser la solution trouvée pour l'implémentation (Figure 10).

```
float32_t K_data[4] = {
    0.7063, 1.0978,
    -0.7063, -1.0978};

float32_t R_data[2];
arm_matrix_instance_f32 R;
arm_mat_init_f32(&R, 2, 1, R_data);
arm_mat_mult_f32(&K, &S, &R);

dF = R_data[0] * 10;
```

Figure 10: Implémentation LQR sur le MCU

II.4 IHM

Pour faciliter l'expérimentation des différents filtres et contrôleurs, une interface graphique (UI) qui communique avec le MCU est développée en Rust, un langage moderne qui offre les performances du C tout en étant « Memory safe » c'est-à-dire que grâce au système de « borrow checking » du compilateur, tout code qui pourrait produite des bugs du a la mauvaise gestion de la mémoire (déréférencement d'un pointeur NULL en C par exemple) ne compile pas. L'inconvénient est que le langage est plus contraignant que le C et qu'étant encore jeune, l'écosystème est moins développé, en particulier, il n'y a pas d'équivalent comme QT qui permette de développer des UI complète. Toutefois, il existe egui, qui est suffisant pour notre usage. L'application fonctionne selon la Figure 11.

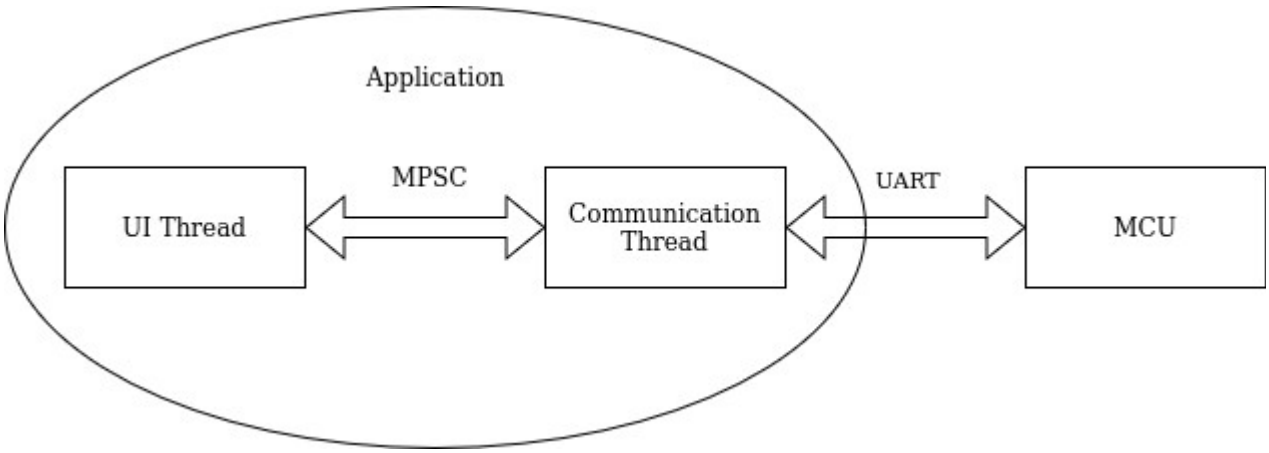

Figure 11: Schéma de l'architecture de l'application

L'utilisation de threads permet de communiquer avec le MCU parallèlement à l'exécution de l'interface utilisateur. Les MPSCs (Multiple Producers Single Consumer) sont des objets Rust permettant la communication entre plusieurs thread.

L'interface est divisée en deux : un menu pour les filtres (Figure 13) et un autre pour les contrôleurs (Figure 12). Dans les deux cas, un graphique est aussi visible permettant de visualiser l'angle du bras en temps réel.

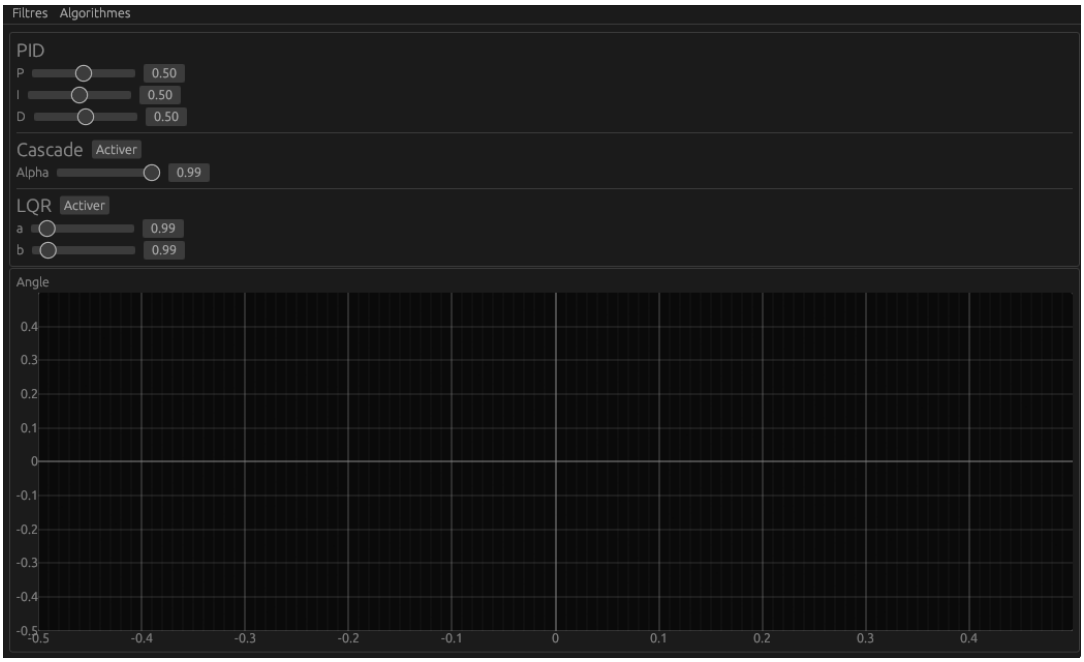

Figure 12: Interface graphique sélection du controlleur

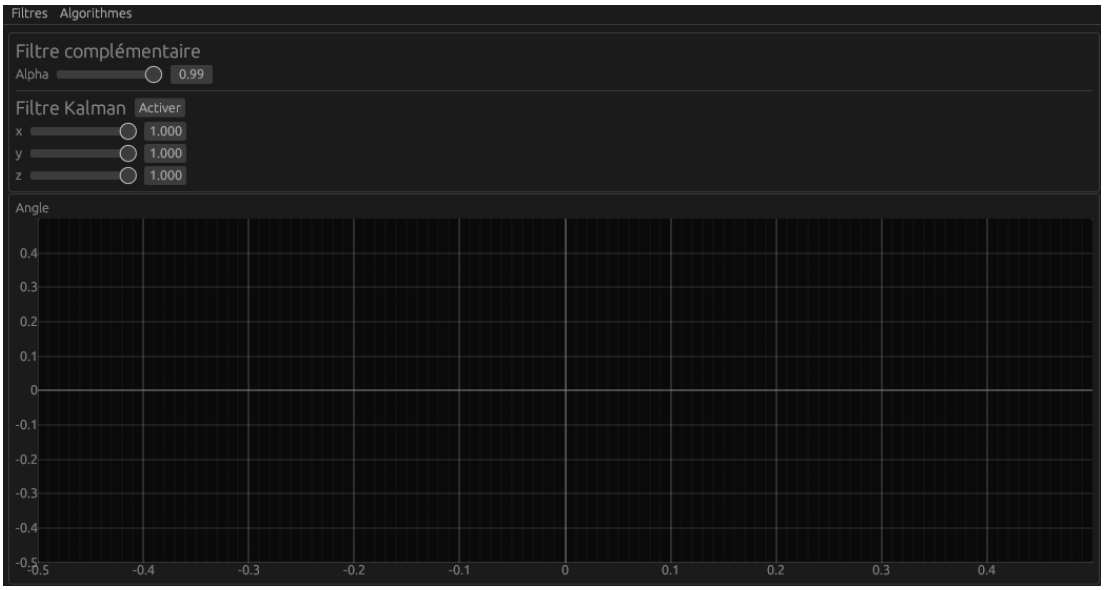

Figure 13: Interface graphique sélection du filtre

La communication entre le MCU et l'ordinateur est assuré part UART. S'il est simple d'envoyer par UART depuis le MCU, la réception est plus compliquée. En effet, celle-ci ne doit pas être bloquante et se faire en parallèle de l'asservissement du système. Pour ce faire, il faut configurer l'UART avec le DMA qui s'occupe de la mise en mémoire des données reçu au lieu du CPU et génère une interruption lorsque le buffer de données est plein. Cependant, les commandes envoyées ont des longueurs variables, un grand buffer doit donc être utilisé mais l'interruption n'est alors jamais appelé. Heureusement, le MCU peut détecter lorsque la ligne RX ne change pas d'état pendant un certain temps (HAL_UARTEx_ReceiveToIdle_DMA). Ainsi, une fois la commande entièrement reçue, on peut exécuter une fonction qui s'occupe de décoder le message. Les messages sont des chaînes de caractères arbitraires comme suit :

« k » : activer le filtre complémentaire
« K » : activer le filtre de Kalman
« P » : activer le contrôleur PID
« C » : activer le controleur Cascade
« L » : activer le contrôleur LQR
« a:b » : mettre à jour la variable « a » avec la valeur « b » (a ne doit pas être le même que pour l'activation d'un filtre ou contrôleur)

La logique implémenté sur le MCU suit le schéma de la Figure 9.

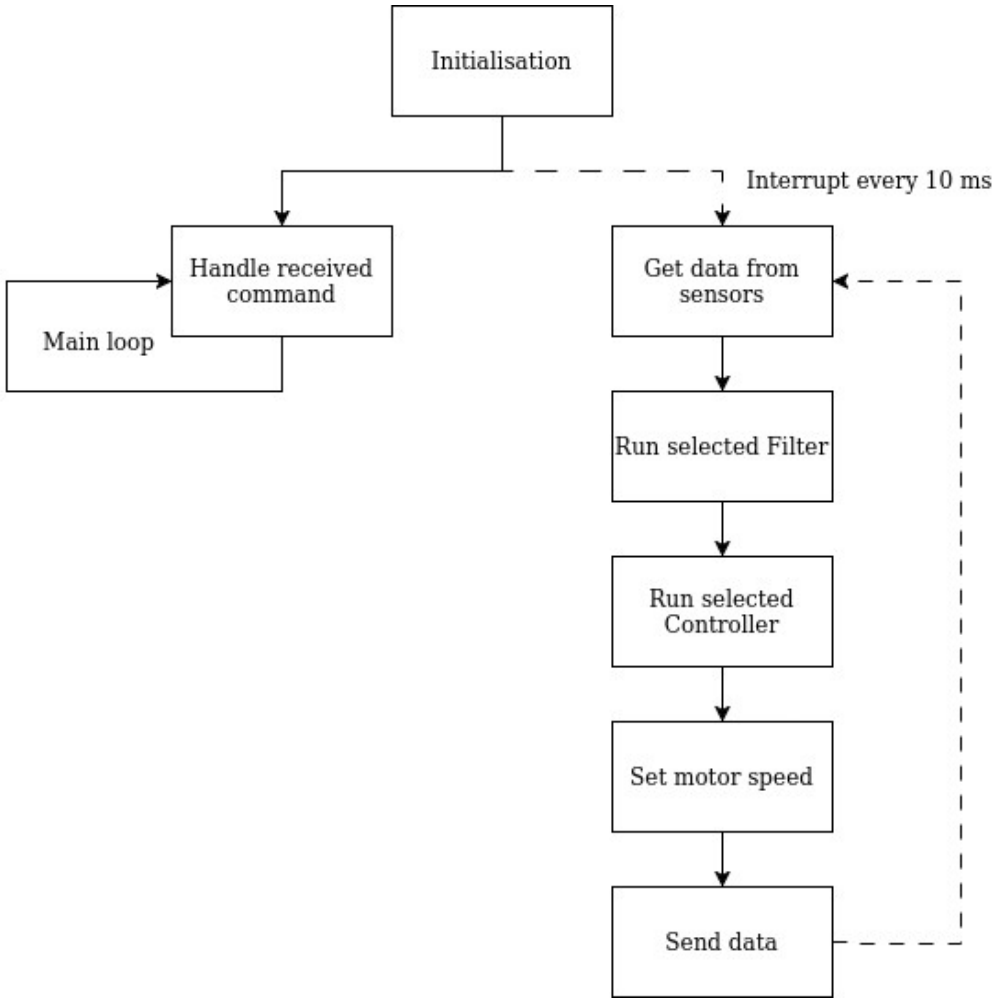

Figure 14: Schéma de l'architecture du MCU

III Conclusion

Cette maquette permet de mettre en évidence que les algorithmes qui théoriquement permettent un meilleur asservissent ne le sont pas toujours en pratique. En effet, l'utilisation d'un modèle limité ou des contraintes sur la puissance de calcul peuvent faire perdre tout avantage face à un simple filtre complémentaire et contrôleur PID. Ci-dessous, se trouvent deux tableaux qui comparent respectivement les performances des différents filtres et contrôleurs.

Table 1: Tableau comparatif des filtres

Filtres	Filtre Complémentaire	Filtre de Kalman
Coût en ressource	Faible	Élevé
Précision	Élevé	Très élevé si modèle valide
Implémentation	Très simple	Compliquée

Table 2: Tableau comparatif des contrôleurs

Contrôleur	PID	Cascade	LQR
Coût en ressource	Moyen	Faible	Faible si le sytème est linéaire
Efficacité	Très bon	lent	lent
Modèle nécessaire	Non	Oui	Oui
Implémentation	Simple	Très simple	Simple si le système est linéaire, plus complexe sinon