Projet HIL : Contrôle PID embarqué sur STM32

Moussa Traoré

August 18, 2025

Contents

Introduction

Ce projet vise la conception d'une carte de contrôle-commande sur la base d'un microcontrôleur STM32 pour des applications "Processor in the Loop" (PIL) à coût réduit. La plateforme PIL s'agit d'une Interface Homme Machine (IHM), installée sur un logiciel, qui permet d'émuler le comportement d'un système complexe, afin de valider la stratégie de contrôle embarquée.

D'abord , j'ai conçu un correcteur PID embarqué sur microcontrôleur STM32, relié à Simulink qui simule le système à contrôler et permet la visualisation en temps réel.Après je vais utiliser un deuxieme microcontroleur qui va jouer le rôle d'intetface comme un système réel(echange ADC PWm) synchroniser avec le controleur (une autre STM32) ; par ce que jusque là l'échange se fait en UART .

1 Modélisation du système

J'ai choisi de travailler sur un système du second ordre, car ce type de dynamique est typique de nombreux systèmes physiques (moteur, suspension, circuit électrique RLC, etc.) et suffisamment riche pour mettre en évidence l'intérêt d'un correcteur PID.

La fonction de transfert retenue est :

$$G(s) = \frac{1}{0.25s^2 + 0.8s + 1}$$

Ce modèle présente :

Deux pôles réels (donc un comportement oscillant/amorti selon les coefficients).

Un temps de réponse typique pour une dynamique réelle.

Un gain statique unitaire (pour faciliter l'analyse de la réponse à une consigne).

L'intérêt d'un système d'ordre 2 est qu'il permet d'observer à la fois le dépassement (overshoot), la rapidité (temps de réponse) et l'erreur statique en boucle fermée, ce qui justifie pleinement l'utilisation d'un PID pour la régulation.

Après modélisation dans MATLAB, j'ai discrétisé ce système avec une période d'échantillonnage courte ($T_s = 0.001$ s), adaptée à une régulation rapide et stable sur microcontrôleur. Ce choix garantit que la simulation en temps réel sous Simulink reste fidèle à ce qu'on attend d'un plant réel à contrôler., j'obtiens dans Matlab :

```
Ts = 0.001;
Num = [1];
Den = [0.25 0.8 1];
G = tf(Num, Den);
Gz = c2d(G, Ts, 'zoh');
```

Ce système présente une dynamique classique de type inertiel amorti, bien adaptée à l'illustration des lois PID.

2 Synthèse et réglage du correcteur PID

Pour assurer un suivi précis de la consigne sans erreur statique et avec un temps de réponse court, j'ai conçu un correcteur PID. Le schéma de calcul utilisé est :

```
erreur = consigne - mesure;
integral += erreur * dt;
derivee = (erreur - prev_erreur) / dt;
cmdPID = Kp * erreur + Ki * integral + Kd * derivee;
prev_erreur = erreur;
```

Méthodologie de réglage : la méthode de Ziegler-Nichols

Pour déterminer les coefficients du régulateur PID, j'ai utilisé la **méthode de Ziegler-Nichols**, une approche empirique classique permettant d'obtenir rapidement un premier réglage satisfaisant. Elle consiste à observer la réponse du système en boucle fermée et à en déduire les paramètres PID à partir du comportement oscillatoire.

La procédure est la suivante :

- On place le régulateur en mode purement proportionnel ($K_i = 0$, $K_d = 0$), puis on augmente progressivement le gain proportionnel K_p jusqu'à ce que le système présente une oscillation soutenue (limite de stabilité).
- On note alors la valeur du *gain critique* (K_{cr}) et la *période critique* (T_{cr}), c'est-à-dire la période d'oscillation observée.
- On applique ensuite les formules de Ziegler-Nichols pour déterminer les paramètres PID :

  $$K_p = 0.6\,K_{cr} \qquad T_i = 0.5\,T_{cr} \qquad T_d = 0.125\,T_{cr}$$

  avec $K_i = K_p/T_i$ et $K_d = K_p \times T_d$.

Cette méthode fournit une base de réglage rapide, qu'on affine ensuite par essais-erreurs pour obtenir la performance souhaitée : suppression de l'erreur statique, minimisation du dépassement et temps de réponse court. Dans mon cas, après avoir modélisé le système sous Simulink, j'ai évalué expérimentalement le couple (K_{cr}, T_{cr}) puis appliqué ces formules, avant d'affiner les coefficients en conditions réelles pour tenir compte des éventuelles non-linéarités du système.

Après essais, j'ai retenu des paramètres typiques tels que :

$$K_p = 20.0 \qquad K_i = 10.0 \qquad K_d = 5.0 \qquad dt = 0.001\,s$$

Le réglage s'est fait pour obtenir un temps de réponse de l'ordre de la seconde, sans dépassement excessif.

3 Validation du PID sous simulink

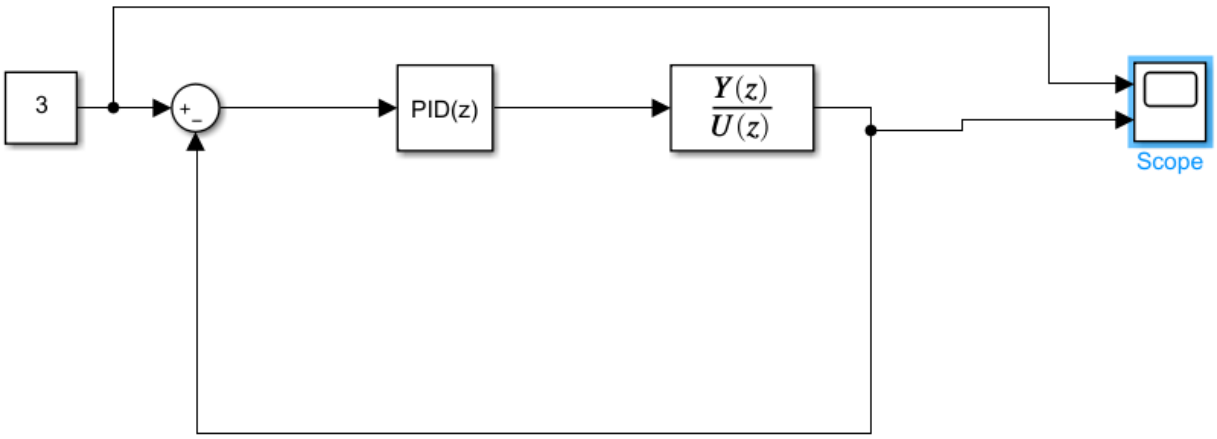


Figure 1: Modèle simulink

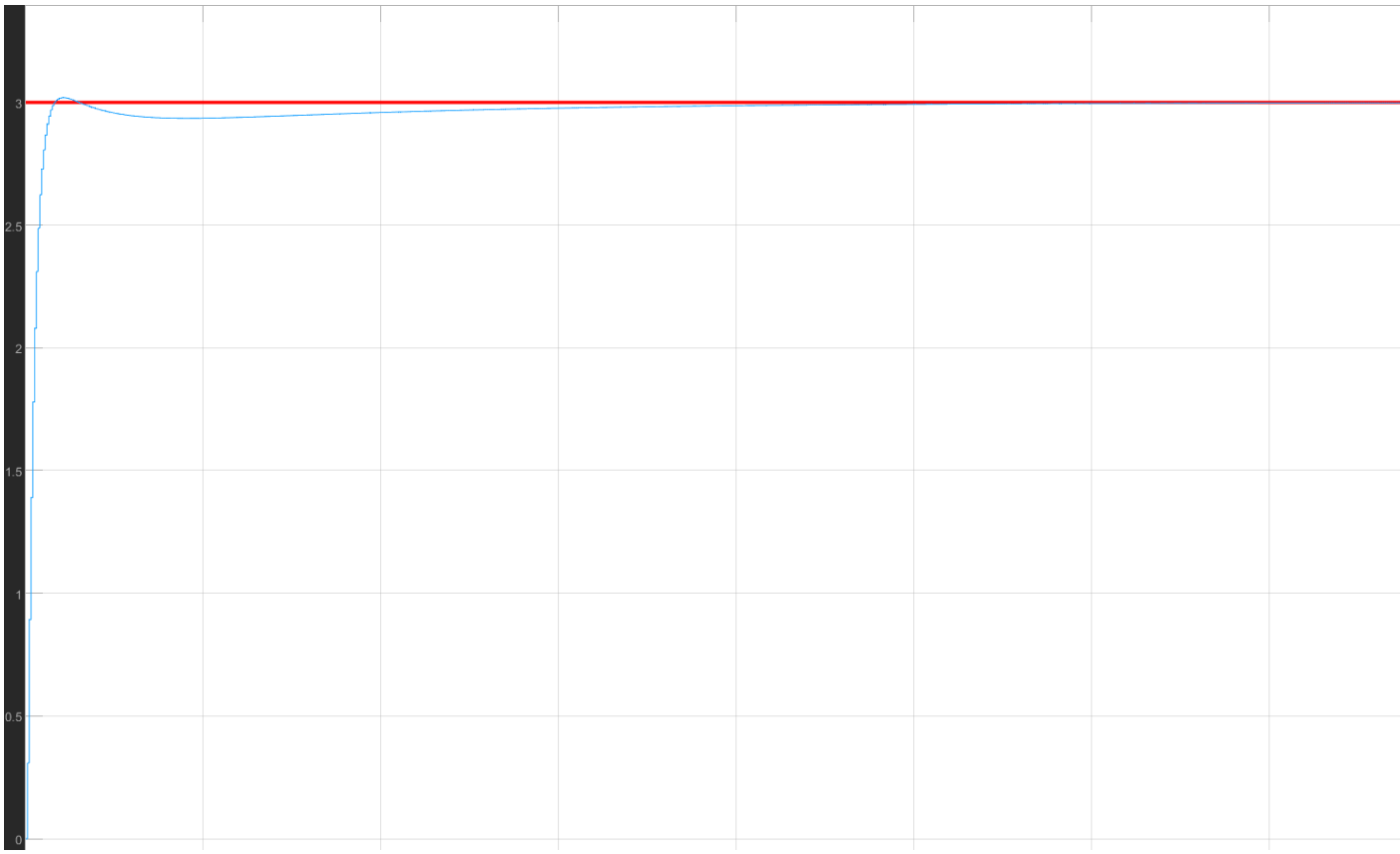


Figure 2: Vue du scope

4 Implémentation sur STM32

Le code embarqué STM32 assure la boucle :

1. Réception UART (8 octets : consigne et mesure, au format `float`)
2. Calcul PID (voir ci-dessus)
3. Transmission UART (4 octets, commande PID au format `float`)

Extrait de code principal :

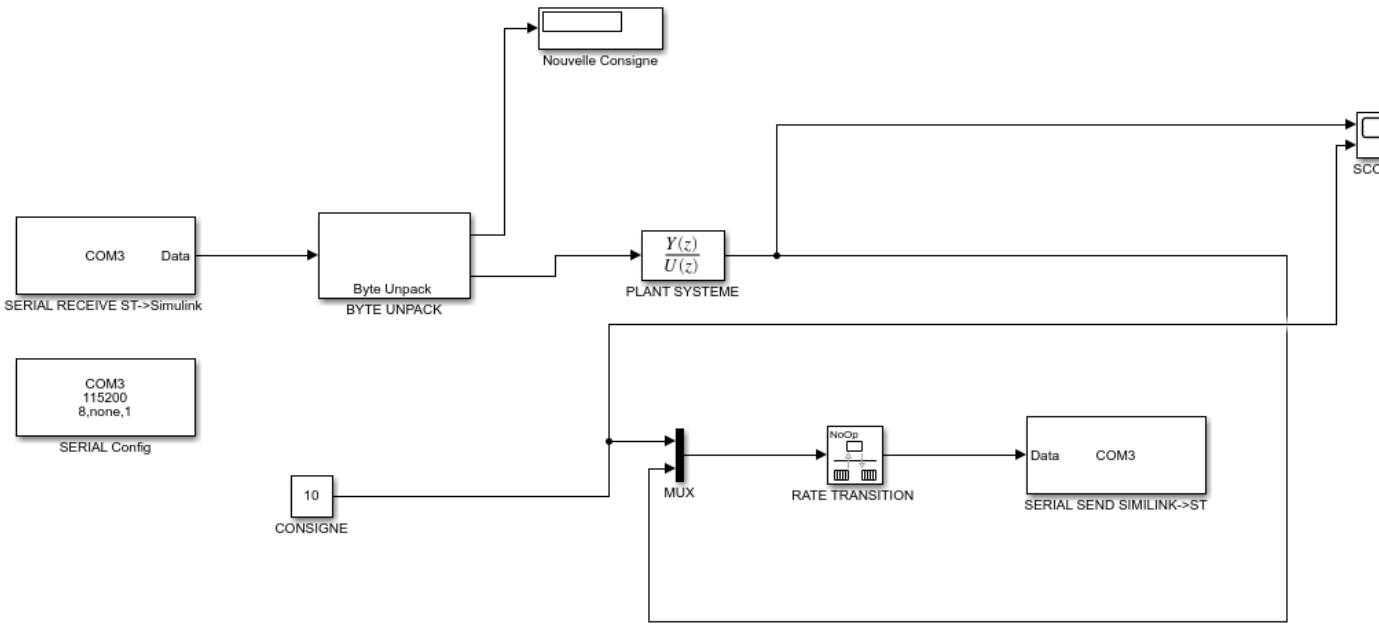
```
while (1)
{
    HAL_UART_Receive(&huart2, rxBuf, 8, HAL_MAX_DELAY);
    memcpy(&consigne, rxBuf, 4);
    memcpy(&mesure,   rxBuf+4, 4);

    erreur = consigne - mesure;
    integral += erreur * dt;
    derivee = (erreur - prev_erreur) / dt;
    cmdPID = Kp * erreur + Ki * integral + Kd * derivee;
    prev_erreur = erreur;

    // Saturation
    if (cmdPID > 4095) cmdPID = 4095;
    if (cmdPID < 0)    cmdPID = 0;

    memcpy(txBuf, &cmdPID, 4);
    HAL_UART_Transmit(&huart2, txBuf, 4, HAL_MAX_DELAY);

    HAL_Delay(1); // dt = 1 ms
}
```

Dans ce code je n'ai pas utilisé les interruptions mais,je vais faire après ; je voulais juste essayer normalement . La communication UART est paramétrée à 115200 bauds, 8 bits, sans parité.

5 Chaîne de simulation sous Simulink

Vue d'ensemble

J'ai construit dans Simulink la chaîne suivante :



(schéma Simulink global.)

Description des blocs principaux

- **SERIAL Config** : Ce bloc initialise et paramètre le port série (`COM3`, 115200 bauds, 8N1) afin d'assurer la liaison UART entre Simulink et le STM32.
- **CONSIGNE (Constant)** : Génère la consigne, c'est-à-dire la valeur de référence à atteindre pour le système (*setpoint*).
- **MUX** : Assemble plusieurs signaux scalaires (ici, la consigne et éventuellement la mesure) en un vecteur pour un envoi groupé sur l'UART.
- **RATE TRANSITION** : Assure la synchronisation des données lorsqu'elles transitent entre des tâches de fréquences différentes, évitant les conflits de mémoire.
- **SERIAL SEND SIMULINK→ST** : Envoie via UART la consigne (et la mesure si besoin) au microcontrôleur STM32, en utilisant la configuration définie.
- **SERIAL RECEIVE ST→Simulink** : Reçoit la commande calculée par le PID embarqué sur le STM32, sous forme de trame binaire.
- **BYTE UNPACK** : Dépaquette les octets reçus pour les convertir en une variable numérique exploitable par Simulink (`float32` ou `double`).
- **PLANT SYSTEME (Discrete Transfer Fcn)** : Modélise le système à contrôler sous forme de fonction de transfert discrétisée (représente la dynamique du plant réel).
- **Nouvelle Consigne (Display)** : Affiche la consigne courante à l'écran, permettant de vérifier facilement les changements de consigne en temps réel.

- **SCOPE** : Visualise en temps réel les principales variables (consigne, sortie du système, etc.) à la manière d'un oscilloscope virtuel.

Résumé des flux : La consigne générée est envoyée au STM32, qui renvoie la commande PID calculée. Simulink simule la réponse du système, visualisée sur le **Scope**.

Synchronisation et gestion des échanges

Tous les blocs sont cadencés à $T_s = 0.001$ s (1 ms). L'assemblage et la conversion des données (byte/float) garantissent la bonne transmission et synchronisation avec le STM32.

6 Résultats et validation

Les essais menés montrent que :

- La réponse suit fidèlement la consigne, avec un temps de réponse de l'ordre de la seconde. Les transitions rapides de consigne sont bien suivies, sans désynchronisation ou pertes de données.

Analyse de la réponse du système en boucle fermée

La figure 3 présente la réponse du système lors de variations brutales de la consigne (paliers successifs). On distingue :

- **Courbe bleue** : la consigne appliquée au système, variant de 30 à 20 puis 10 ;
- **Courbe jaune** : la sortie mesurée, issue du système régulé par le PID embarqué.

Suivi rapide et précis de la consigne À chaque changement de la consigne, la sortie suit très rapidement la nouvelle valeur sans retard notable, démontrant un temps de réponse court et une bonne efficacité du correcteur PID.

Absence d'erreur statique Sur chaque palier, la sortie se stabilise précisément sur la consigne, preuve que le terme intégrateur du PID compense bien l'erreur statique du système.

Amortissement et dépassement Le dépassement lors des transitions reste modéré, ce qui indique un bon compromis entre rapidité et stabilité. Il n'y a pas d'oscillation persistante, ni d'instabilité visible.

Présence de bruit On observe un léger bruit autour de la consigne, typique d'un système réel. Il est dû à la discrétisation du régulateur, au bruit de mesure et aux limitations matérielles du microcontrôleur. Ce bruit reste faible et n'impacte pas la stabilité globale.

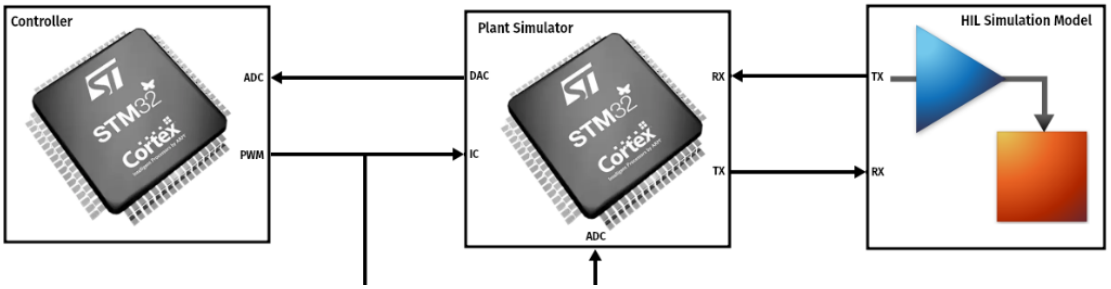
Figure 3: Réponse du système pour des variations de consigne en escalier (sortie jaune, consigne bleue).

7 PARTIE

- Utilisation d'un second microcontrôleur comme interface physique (ADC/PWM)
- Gestion des interruptions UART pour améliorer la robustesse temps réel
- Tests sur un système physique (moteur DC, processus analogique)

8 Simulation HIL sur double microcontrôleur

Après avoir validé la boucle de contrôle en temps réel entre Simulink et un STM32 ("contrôleur"), l'étape suivante consiste à dédier un second microcontrôleur STM32 au rôle de "système à contrôler" (Plant Simulator). L'objectif est que le contrôleur "voit" une vraie réponse physique (par l'intermédiaire du second micro), comme dans une vraie chaîne d'asservissement. Le schéma suivant illustre cette approche :



Description du principe :

- **STM32 Contrôleur :** Exécute le calcul PID en temps réel et génère la commande (signal PWM ou numérique).
- **STM32 Plant :** Reçoit la commande, simule la dynamique du système (fonction de transfert discrète) et renvoie la mesure simulée.
- **Simulink :** Permet de piloter les consignes, visualiser les signaux, et (facultatif) reconfigurer la fonction de transfert à la volée.

Cette architecture permet d'éprouver le contrôleur dans un contexte matériel proche d'un cas industriel, en toute sécurité, avant tout test sur un système réel.

8.1 Protocole de communication et cycle d'échantillonnage

- Les échanges entre microcontrôleurs s'effectuent via liaisons UART (à 115200 bauds), au format float 32 bits (4 octets par variable).
- **Cycle de fonctionnement (tous les 10 ms) :**
  1. Simulink envoie la consigne et la mesure à l'interface STM32 (UART2).
  2. L'interface (plant) réinjecte la mesure sur son DAC (pour être lue par l'ADC du contrôleur), et transmet la consigne reçue au contrôleur via UART3.
  3. Le contrôleur lit la consigne (UART3) et la mesure (ADC), calcule la commande PID, puis transmet la commande au plant (UART3) dans un premier temps pour valider le contrôle et dans un second temps via PWM.
  4. L'interface (plant) transmet la commande vers Simulink via UART2 pour supervision et enregistrement.
- Les LEDs de debug sur chaque carte (LD2) clignotent à chaque échange réussi.

8.2 Code principal de chaque microcontrôleur

8.2.1 STM32 Contrôleur (PID)

Ce microcontrôleur reçoit la consigne sur UART3, lit la mesure sur son ADC (PA0), calcule la commande PID, puis transmet la commande à l'interface via UART3. Le code principal utilisé :

Listing 1: Boucle principale du contrôleur (STM32 PID)

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_ADC1_Init();
    MX_USART3_UART_Init();

    while (1)
    {
        // 1. Attend la consigne depuis l'interface (UART3 RX, 4 octets
        HAL_UART_Receive(&huart3, rxBuf_ctrl, 4, HAL_MAX_DELAY);
        memcpy(&consigne, rxBuf_ctrl, 4);

        // 2. Lis la mesure analogique sur ADC (PA0 reli  au DAC de
        HAL_ADC_Start(&hadc1);
        HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
        uint32_t adc_val = HAL_ADC_GetValue(&hadc1);
        HAL_ADC_Stop(&hadc1);
        mesure = (adc_val * 3.3f) / 4095.0f;

        // 3. Calcul PID
        commande = PID_Update(consigne, mesure);

        // 4. Envoie la commande      l interface   (UART3 TX, 4 octets)
        memcpy(txBuf_cmd, &commande, 4);
        HAL_UART_Transmit(&huart3, txBuf_cmd, 4, HAL_MAX_DELAY);

        // 5. Blink debug LD2 (si connect )
        HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
    }
}
```

8.2.2 STM32 Interface (Plant, Simulateur)

Ce microcontrôleur fait l'interface avec Simulink via UART2, reçoit consigne et mesure, envoie la consigne au contrôleur (UART3), reçoit la commande du contrôleur, puis la retourne à Simulink. Le code principal utilisé :

Listing 2: Boucle principale du simulateur de plant (STM32 Interface)

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART2_UART_Init();
    MX_USART3_UART_Init();
```

```
    MX_DAC1_Init();

    HAL_DAC_Start(&hdac1, DAC_CHANNEL_1);

    while (1)
    {
        // 1. Reoit consigne + mesure de Simulink (UART2 RX, 8 octets
        HAL_UART_Receive(&huart2, rxBuf_sim, 8, HAL_MAX_DELAY);
        memcpy(&consigne, rxBuf_sim, 4);
        memcpy(&mesure,   rxBuf_sim + 4, 4);

        // 2. Met la mesure sur le DAC
        uint32_t dac_val = (uint32_t)(mesure * 4095.0f / 3.3f);
        if (dac_val > 4095) dac_val = 4095;
        HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_1, DAC_ALIGN_12B_R, dac_va

        // 3. Envoie la consigne au contrleur (UART3 TX)
        memcpy(txBuf_ctrl, &consigne, 4);
        HAL_UART_Transmit(&huart3, txBuf_ctrl, 4, HAL_MAX_DELAY);

        // 4. Attend la commande du contrleur (UART3 RX)
        HAL_UART_Receive(&huart3, rxBuf_ctrl, 4, HAL_MAX_DELAY);
        memcpy(&commande, rxBuf_ctrl, 4);

        // 5. Renvoie la commande    Simulink (UART2 TX)
        memcpy(txBuf_sim, &commande, 4);
        HAL_UART_Transmit(&huart2, txBuf_sim, 4, HAL_MAX_DELAY);

        // 6. LED debug : toggle    chaque boucle
        HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
    }
}
```

On peut ainsi suivre le cycle de la consigne, de la mesure et de la commande sur toute la chaîne Simulink → Interface → Contrôleur → Interface → Simulink. Je n'ai pas utiliser d'interruption dans un premier afin de valider la boucle de contrôle , mais le système reste quand même rapide.On pourra utiliser les interruptions pour plus de robustesse .

8.3 Validation expérimentale de la régulation

La figure 4 présente la réponse du système bouclé obtenu lors de la régulation d'un plant simulé par microcontrôleur STM32, avec calcul PID embarqué sur un second STM32 (contrôleur), l'ensemble étant piloté et observé depuis Simulink.

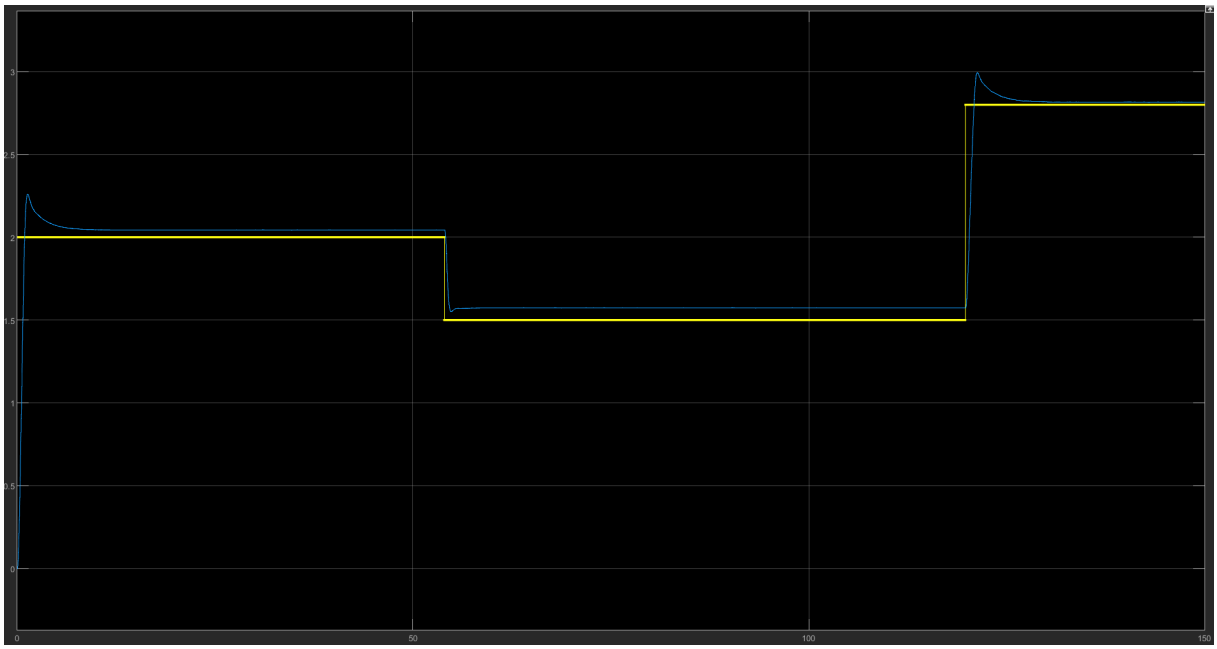


Figure 4: Réponse expérimentale du système contrôlé : consigne (jaune) et sortie mesurée (bleu)

Analyse des résultats :

- **Suivi de consigne :** On observe que la sortie suit bien la consigne pour différents échelons, avec une dynamique rapide et un temps de réponse satisfaisant.
- **Erreur de position :** Malgré le bon suivi, une erreur statique (écart permanent) subsiste entre la consigne et la sortie. Ceci est typique d'un régulateur PID avec un intégral insuffisant ou d'une saturation de la commande (limitation physique du DAC/plant simulé).
- **Robustesse :** Le système reste stable même lors de changements brusques de consigne. Aucun phénomène d'instabilité ou de dépassement excessif n'a été observé, ce qui valide la robustesse de l'implémentation sur la chaîne complète Simulink $\leftrightarrow$ Interface $\leftrightarrow$ Contrôleur.
- **Validation HIL :** Cette expérience valide le fonctionnement de la chaîne Hardware-in-the-Loop, permettant d'envisager le remplacement du plant simulé par un système physique réel (moteur, process analogique, etc.) avec les mêmes principes.

Limites et suite du projet :

- **Erreur statique :** Pour éliminer l'erreur de position, il faudrait augmenter l'action intégrale (K_i), ou bien vérifier l'absence de saturation sur le DAC/ADC et l'absence de bugs de conversion numérique.
- L'envoie de la commande Via PWM au lieu de L'UART
- **Temps réel :** L'utilisation des interruptions UART au lieu du polling pourrait améliorer la robustesse et réduire les délais de transmission.

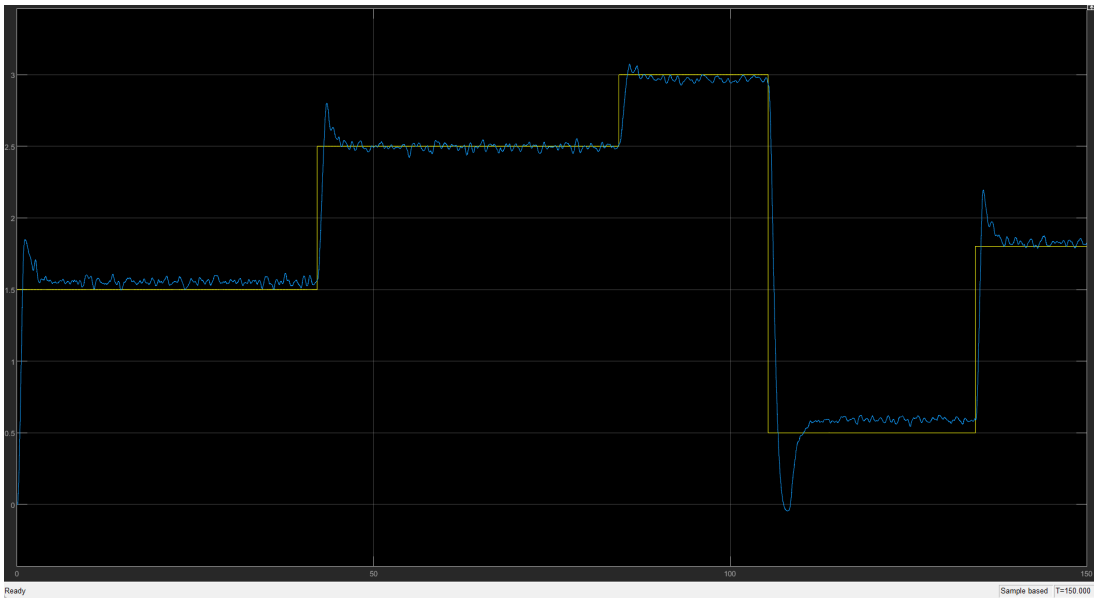
– **Tests avancés :** Il serait pertinent de tester la solution sur un vrai processus analogique ou moteur DC pour valider le contrôleur dans un contexte industriel.

9 Extension : commande réelle par PWM

Après avoir validé la transmission de la commande entre microcontrôleurs via UART, l'étape suivante consiste à générer le signal de commande sous forme de PWM (Pulse Width Modulation). Cette approche permet de piloter un actionneur réel (par exemple, un moteur DC ou une LED de puissance) ou d'attaquer une interface analogique via un filtre RC.

– **Contrôleur STM32 :** Calcule la commande numérique et la convertit en PWM via un timer matériel.
– **Plante STM32 (interface) :** Reçoit la commande PWM, la filtre via un circuit RC, mesure la tension obtenue à l'aide de l'ADC, puis la renvoie à Simulink pour la supervision.

Cette architecture permet de valider la performance du régulateur sur une vraie sortie PWM, au plus proche d'un environnement industriel. La transition entre le signal numérique (PWM) et l'acquisition analogique (ADC via RC) introduit une dynamique supplémentaire qui rend la simulation plus réaliste et permet d'anticiper les comportements d'un système embarqué réel.



La courbe ci-dessus illustre la bonne réponse du système lors de plusieurs changements de consigne, démontrant à la fois la rapidité, la stabilité et la précision de la régulation. Le filtrage RC permet d'obtenir une tension lisse et exploitable pour l'asservissement, tout en conservant la simplicité matérielle du montage.

10 Implémentation de la commande PWM sur STM32

La génération de la commande PWM (Pulse Width Modulation) sur la carte contrôleur STM32 est réalisée à l'aide d'un timer matériel configuré en mode PWM. La valeur de la consigne calculée par le régulateur (par exemple, un PID) est convertie en rapport cyclique (duty cycle), puis affectée au registre de comparaison du timer.

Le code principal permettant la génération du signal PWM est présenté ci-dessous :

Listing 3: Génération de la PWM sur STM32 (extrait du main.c)

```
/* Demarrage du PWM sur TIM2_CH1 */
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);

while (1)
{
    // ... (recuperation consigne & mesure, calcul PID, etc.)

    // Calcul du rapport cyclique (0     3.3V)
    float commande = ...; // Sortie du PID, bornee entre 0 et 3.3V
    uint32_t pwm_period = __HAL_TIM_GET_AUTORELOAD(&htim2);
// typiquement 65535
    uint32_t pwm_val = (uint32_t)(commande / 3.3f * pwm_period);
    if (pwm_val > pwm_period) pwm_val = pwm_period;
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, pwm_val);

    // ... (autres traitements)
}
```

Dans cet exemple, la commande calculée (comprise entre 0 et 3.3V) est proportionnellement convertie en une valeur de rapport cyclique comprise entre 0% et 100% du signal PWM. Ce signal est ensuite transmis vers l'interface (ou le système réel) pour piloter l'actionneur ou le banc de test.

L'utilisation du timer matériel garantit une génération précise et stable du signal PWM, essentielle pour les applications de contrôle temps réel.

11 Synchronisation de l'ADC sur le front montant de la PWM

L'interface STM32 (L476) synchronise l'échantillonnage de la commande avec la **PWM du contrôleur** afin de réduire le jitter de mesure et de garantir une phase d'acquisition constante par rapport au cycle PWM. Le principe est le suivant :

1. La **PWM brute** issue du contrôleur (PA0, TIM2_CH1) est câblée sur **PA6 (TIM3_CH1)** côté Interface.

2. Le **front montant** de cette PWM est détecté par **TIM3** en mode *Input Capture* (IC) et déclenche une **interruption** (NVIC TIM3).
3. Dans le **callback TIM3**, on lance une conversion **ADC1** *logicielle* sur **PC1 (IN2)**, qui reçoit la **PWM filtrée** par RC.
4. Dans le **callback ADC**, on récupère l'échantillon et on le met à disposition de la boucle principale pour l'envoi à Simulink.

11.1 Câblage et signaux

- **Entrée de synchro (front)** : PA6 → TIM3_CH1 (Input Capture, front montant).
- **Mesure analogique** : PC1 → ADC1_IN2 (tension moyenne de la PWM après filtre RC).
- **Sortie DAC (miroir consigne)** : PA4 → ADC du contrôleur (référence analogique).

11.2 Configuration minimale

- **TIM3_CH1 (PA6)** : `ICPolarity = RISING`, `ICSelection = DIRECTTI`, `ICPrescaler = DIV1`, `ICFilter = 0..N`.
- **ADC1** (PC1/IN2) : `ExternalTrigConv = SOFTWARE_START` (on déclenche dans le callback TIM3), `SamplingTime` ajusté selon l'impédance RC.
- **NVIC** : `TIM3 global interrupt` et `ADC global interrupt` *activés*.

11.3 Implémentation (extraits `main.c`)

Listing 4: Demarrage de la synchro et du DAC (Interface)

```
/* Init d ja faite (GPIO/UART/DAC/ADC/TIM3) */
HAL_TIM_IC_Start_IT(&htim3, TIM_CHANNEL_1);
// PA6: synchro front montant PWM
HAL_DAC_Start(&hdac1, DAC_CHANNEL_1);
// PA4: miroir consigne si n cessaire
```

Listing 5: Callback TIM3: déclenchement de l'ADC sur PC1 (IN2)

```
/* Front montant detecte sur PA6 (TIM3_CH1) */
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == TIM3 && htim->Channel == HAL_TIM_ACTIVE_CHAN
    {
        /* D marre une conversion ADC (PC1 = ADC1_IN2) */
        HAL_ADC_Start_IT(&hadc1);
    }
}
```

Listing 6: Callback ADC: acquisition de l'echantillon synchronisé

```
volatile uint32_t last_adc_val = 0;

void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
    if (hadc->Instance == ADC1)
    {
        last_adc_val = HAL_ADC_GetValue(hadc);
// valeur 12 bits [0..4095]
        /* Option: conversion en volts dans la boucle principale
           float cmd = (last_adc_val * 3.3f) / 4095.0f; */
    }
}
```

11.4 Utilisation côté boucle principale

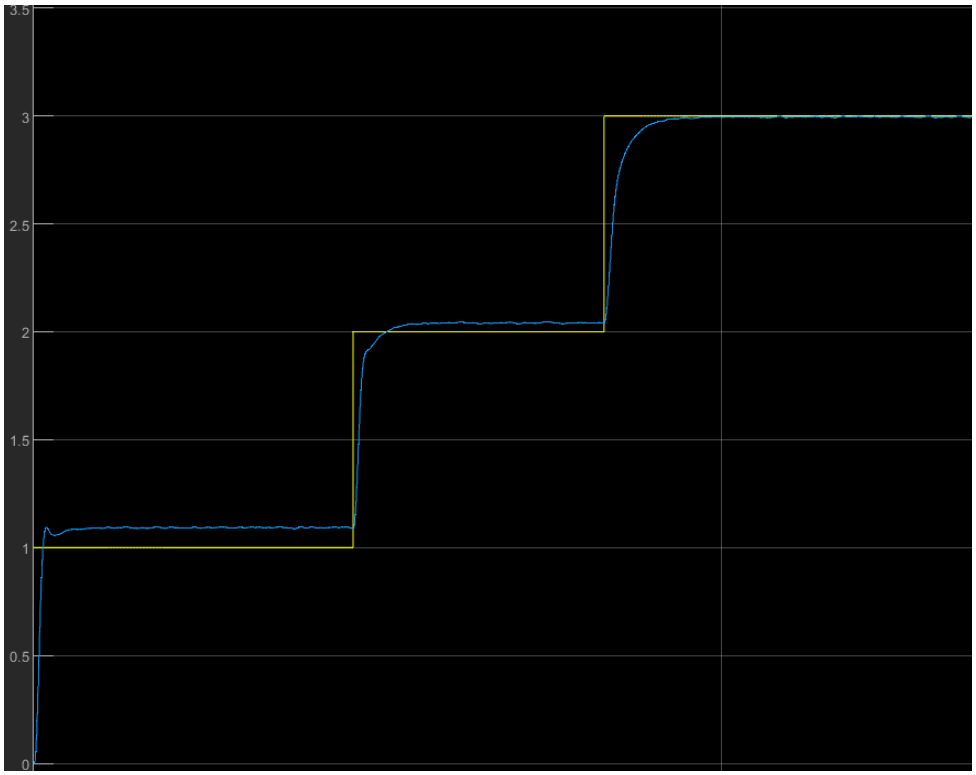
Dans la boucle, on exploite la dernière valeur synchronisée (`last_adc_val`) pour renvoyer la commande à Simulink (UART2) :

Listing 7: Exploitation de la mesure synchronisée

```
/* .reception Simulink, miroir DAC, envoi consigne au contr leur ...
float commande = (last_adc_val * 3.3f) / 4095.0f;  // V
memcpy(txBuf_sim, &commande, sizeof(float));
HAL_UART_Transmit(&huart2, txBuf_sim, sizeof(float), HAL_MAX_DELAY);
```

11.5 Résultat attendu

Cette synchronisation assure que la **commande renvoyée à Simulink** correspond à une mesure analogique (PC1) **prise toujours au même instant relatif** du cycle PWM. L'effet est une **mesure plus stable**, moins sensible aux déphasages et au bruit, et une meilleure cohérence HIL (Simulink $\leftrightarrow$ matériel).



12 Perspectives : essais sur un système réel

L'étape suivante logique de ce travail consiste à remplacer le modèle numérique du système (fonction de transfert implémentée sur la carte interface) par un système physique réel. Cette extension permettra de valider l'architecture de contrôle dans des conditions expérimentales encore plus proches de la réalité industrielle.

- **Carte Contrôleur** : calcule la commande en temps réel et la transmet en PWM à l'actionneur réel.
- **Système réel (ou banc d'essai)** : reçoit la commande PWM, la filtre via un circuit RC, puis la mesure par ADC est renvoyée à la boucle de régulation.

Pour l'instant, ces essais sur système réel n'ont pas encore été réalisés et constituent une perspective directe pour la suite du projet. Ils permettront d'évaluer la robustesse du régulateur face aux non-idéalités réelles (bruit, perturbations, délais, etc.) et de valider pleinement la chaîne d'asservissement complète.

Conclusion

Ce travail a permis de mettre en œuvre et de valider une chaîne d'asservissement numérique temps réel sur une architecture STM32 en mode Hardware-in-the-Loop (HIL). La transition de la commande numérique vers le PWM, puis l'utilisation d'un filtrage RC pour l'acquisition analogique, ont permis de se rapprocher d'un environnement expérimental réaliste et modulaire.

La prochaine étape, naturellement, sera d'étendre cette approche au contrôle d'un système réel afin d'éprouver la robustesse et l'efficacité de la régulation dans des conditions

industrielles concrètes. L'architecture développée se veut évolutive, prête à accueillir de nouveaux modules (actionneurs, capteurs, interfaces de communication), facilitant ainsi la validation rapide de stratégies de commande avancées.