UNIVERSITÉ PARIS-EST CRÉTEIL

Université Paris-Est Créteil (UPEC)
Département / Filière / Parcours ▪ Systèmes embarqués / Automatique

RAPPORT DE PROJET

Commande et régulation PID d'un moteur DC

Mesure RPM par encodeur (INT0) et PWM matériel Timer1

Nom Prénom — Groupe / Matricule **Mouloud Rayan Mallem 2230518010**
Nom Prénom — Groupe / Matricule **Ziane Mohamed Taher**
Nom de l'enseignant — Fonction **Jonathan Caillaiez Programation et application**
Année **2025/2026**

Date **7 février 2026**

PID ▪ Encodeur ▪ ISR ▪ PWM ▪ Timer1

0.1 Contexte et problématique

Un moteur DC commandé en PWM transforme un rapport cyclique en *tension moyenne* appliquée au moteur. En boucle ouverte, la vitesse varie lorsque :

— la charge augmente (couple résistant plus fort) : la vitesse chute,
— la tension d'alimentation baisse : la vitesse chute,
— les frottements changent (température, usure) : la vitesse dérive.

La boucle fermée corrige ces variations en temps réel : **mesure -> erreur -> correction -> action**.

0.1.1 Contraintes embarquées (AVR / Arduino)

Deux contraintes dominent :

— **Acquisition rapide** : les impulsions d'encodeur peuvent arriver très fréquemment. L'ISR doit être courte (pas de calcul flottant, pas de Serial, pas de fonctions lentes).
— **Cohérence des données** : une variable modifiée en interruption peut être lue au mauvais moment dans la boucle principale. Il faut donc une lecture *atomique* (section critique).

0.1.2 du tick à la PWM

Le flot de données est le suivant :

1. À chaque front montant de la voie A, l'ISR met à jour `tickCount` (lignes 21–27).
2. Toutes les 400 ms (test lignes 99), la boucle : copie `tickCount` (lignes 105–109), le remet à zéro, puis calcule `rpm` (ligne 112).
3. Le PID calcule une commande `u` (lignes 114–123).
4. `u` est converti en PWM signé `pwmCmd` (ligne 123) puis appliqué au moteur (ligne 125).

Table 1 – Table de traçabilité : fonctionnalité.

Fonctionnalité	Lignes (Annexe A)
Déclaration paramètres/états (Kp, Ki, Kd, tickCount, etc.)	1–19
Comptage encodeur + sens (ISR INT0)	21–27
Initialisation registres (ports, Timer1, INT0)	29–49
Commande PWM signée (OCR1A/OCR1B)	51–68
Lecture consigne série (ASCII -> float)	75–87
Boucle périodique : dt, lecture atomique, RPM, PID, PWM, logs	95–128

0.2 Déroulement

0.2.1 Étape 0 : démarrage (reset -> setup)

Après reset, Arduino appelle `setup()` une seule fois (lignes 89–93) :

1. Initialisation du port série `Serial.begin(115200)` (ligne 90) pour envoyer/recevoir des données.

2. Configuration matérielle via `setupRegisters()` (ligne 91) : ports, PWM Timer1 et interruption INT0 .
3. Initialisation de la référence temporelle `lastTime = millis()` (ligne 92).

0.2.2 Étape 1 : comptage des impulsions (asynchrone, en interruption)

Pendant que la boucle principale tourne, l'encodeur génère des impulsions. À chaque front montant sur A (D2/PD2), l'ISR s'exécute (lignes 21–27) :

— Elle lit la voie B via `PIND & (1<<PD3)` (ligne 22),
— Puis incrémente `tickCount` (ligne 23) ou le décrémente (ligne 25).

Résultat : `tickCount` contient un nombre signé de ticks sur la fenêtre courante.

0.2.3 Étape 2 : boucle principale (loop) et déclenchement périodique

Arduino appelle ensuite `loop()` en continu (lignes 95–128). À chaque tour de boucle :

1. Lecture non-bloquante de la consigne série (ligne 96).
2. Lecture de l'horloge `now = millis()` (ligne 98).
3. Test du délai `now - lastTime >= SAMPLE_MS` (ligne 99). Tant que ce n'est pas vrai, le programme revient au début (il ne calcule pas la vitesse).

0.2.4 Étape 3 : à chaque période, lecture atomique + calcul RPM

Quand la période est atteinte (ligne 99), la boucle exécute un cycle complet :

1. Calcul de `dt` en secondes (ligne 100) et mise à jour `lastTime` (ligne 101).
2. Lecture atomique de `tickCount` : sauvegarde de SREG (ligne 105), `cli()` (ligne 106), copie dans `ticks` (ligne 107), remise à zéro (ligne 108), restauration SREG (ligne 109).
3. Calcul de la vitesse (ligne 112) :

$$\text{RPM} = \frac{\text{ticks}}{TICKS\_PER\_REV} \cdot \frac{60}{dt}$$

0.2.5 Étape 4 : calcul PID et conversion en PWM

Après la vitesse :

1. Erreur `error = targetRPM - rpm` (ligne 114).
2. Intégrale (lignes 115–116) avec anti-windup par saturation (ligne 116).
3. Dérivée discrète (ligne 118) et mise à jour `prevError` (ligne 119).
4. Sortie PID `u` (ligne 121).
5. Conversion en PWM signé `pwmCmd` (ligne 123).

0.2.6 Étape 5 : application de la commande + logs

— Application sur le moteur : `driveMotorSigned(pwmCmd)` (ligne 125) écrit `OCR1A/OCR1B`.
— Affichage série (lignes 127–130) : format `Target:...,RPM:...` pour observation et tracés.

0.2.7 Schéma fonctionnel et modélisation analytique

Cette partie formalise l'asservissement du projet sous forme de schémas fonctionnels et d'un modèle analytique continu. L'objectif est double : (i) structurer clairement la chaîne *référence* → *régulateur* → *actionneur* → *système* → *capteur*, (ii) établir une fonction de transfert permettant d'expliquer le choix des paramètres et le comportement dynamique.

0.2.8 Schéma fonctionnel général (boucle fermée)

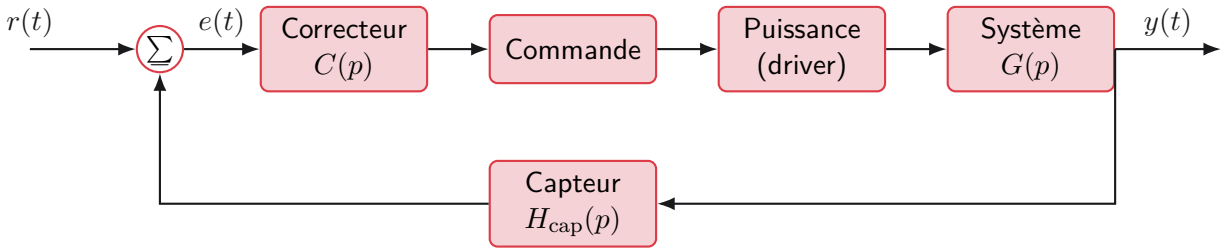

FIGURE 1 – Boucle fermée générique : correcteur $C(p)$, procédé $G(p)$, capteur $H_{\text{cap}}(p)$.

0.2.9 Schéma fonctionnel Arduino + encodeur + moteur DC)

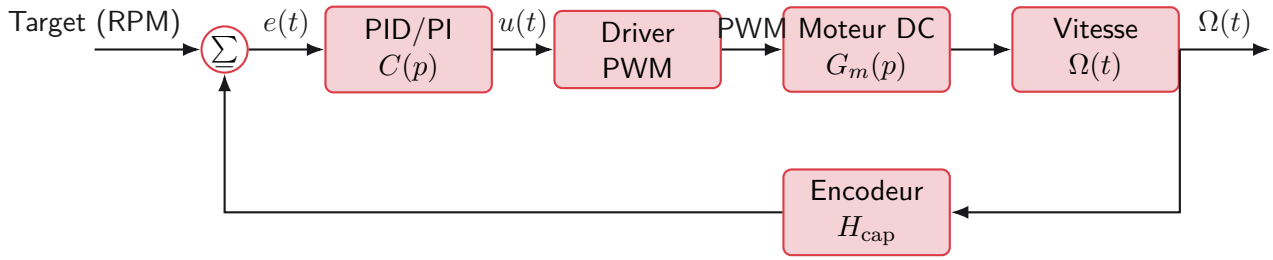

FIGURE 2 – Schéma fonctionnel Target → PI/PID → PWM → moteur → vitesse → encodeur.

0.2.10 Correcteur (PID/PI) : expression continue

le correcteur est de type PID lorsque $K_d = 0$) :

$$C(p) = K_p + \frac{K_i}{p} + K_d\, p$$

$K_d = 0$ donc :

$$C(p) = K_p + \frac{K_i}{p} = \frac{K_p p + K_i}{p}$$

0.2.11 Modèle du moteur

Un modèle simple pour la dynamique vitesse d'un moteur DC commandé en tension moyenne (PWM filtrée) est un **premier ordre** :

$$G_m(p) = \frac{\Omega(p)}{U(p)} = \frac{K_m}{1 + \tau p}$$

où :

— K_m est le gain statique (RPM par unité de commande),
— τ est la constante de temps (inertie + frottements + charge).

0.2.12 Capteur (encodeur) et chaîne de mesure

Le capteur ramène la vitesse à une grandeur mesurable et comparée à la consigne :

$$Y_{\text{mes}}(p) = H_{\text{cap}}(p)\, \Omega(p)$$

Dans un modèle simplifié, on prend $H_{\text{cap}}(p) \approx H_{\text{cap}}$ (gain constant), car la conversion *ticks* → *RPM* est effectuée numériquement.

0.2.13 Fonction de transfert en boucle fermée (cas PI)

Avec retour négatif, on obtient :

$$\frac{\Omega(p)}{\Omega_{\text{ref}}(p)} = \frac{C(p)\,G_m(p)}{1 + C(p)\,G_m(p)\,H_{\text{cap}}}$$

En remplaçant $C(p) = \frac{K_p p + K_i}{p}$ et $G_m(p) = \frac{K_m}{1+\tau p}$:

$$\frac{\Omega(p)}{\Omega_{\text{ref}}(p)} = \frac{K_m\,(K_p p + K_i)}{\tau p^2 + (1 + K_p K_m H_{\text{cap}})\,p + K_i K_m H_{\text{cap}}}$$

0.2.14 l'implémentation avec Arduino

- **Encodeur / H_{cap} :** ISR INT0 incrémente/décrémente `tickCount`, puis conversion en RPM dans la fenêtre `SAMPLE_MS`.
- **PI/PID / $C(p)$:** calcul de l'erreur `error = targetRPM - rpm`, intégrale `integral += error*dt`, puis sortie `u = Kp*error + Ki*integral (+ Kd*derivative)`.
- **Driver PWM :** conversion `u` vers `pwmCmd` puis écriture sur Timer1 (`OCR1A/OCR1B`).
- **Procédé / $G_m(p)$:** dynamique réelle du moteur (inertie, frottements, charge) observée dans les mesures.

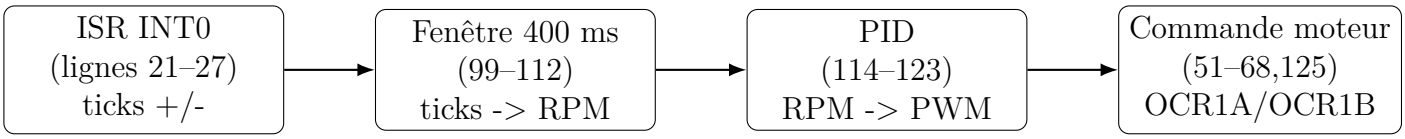

FIGURE 3 – Chaîne de traitement (traçabilité directe avec les lignes du code).

0.3 Analyse fonction des fonction : rôle, étapes internes

0.4 `setupRegisters()` — initialisation matérielle (lignes 29–49)

Rôle : préparer le microcontrôleur avant toute mesure/commande :

- déclarer les directions des broches (entrées/sorties),
- activer les pull-ups des entrées encodeur,
- configurer Timer1 pour produire la PWM sur D9/D10,
- configurer INT0 sur front montant.

Table 2 – Décomposition interne de `setupRegisters()` (étapes).

#	Lignes	Action	Justification (contexte)
1	30	`cli()`	Empêche une ISR pendant la configuration (registre incohérent).
2	32–34	DDRB/ DDRD / PORTD	Sorties PWM sur OC1A/OC1B, entrées encodeur, pull-up pour stabilité.
3	36–40	TCCR1A / TCCR1B	PWM matérielle stable (Timer1) avec prescaler.
4	42–43	OCR1A/OCR1B = 0	Moteur arrêté au démarrage (sécurité).
5	45–48	EICRA / EIMSK	INT0 sur front montant + autorisation INT0.
6	49	`sei()`	Autorise la mesure encodeur après config complète.

0.4.1 Registres importants

- `DDRx` : direction des pins (1=sortie, 0=entrée).
- `PORTx` : écrire 1 sur une entrée => active pull-up.
- `TCCR1A/TCCR1B` : mode PWM + prescaler.
- `EICRA/EIMSK` : mode et activation INT0.

0.4.2 `ISR(INT0_vect)` — comptage ticks + sens (lignes 21–27)

Rôle : compter rapidement les impulsions de l'encodeur. **Entrées :** état logique de B (PD3) au moment du front sur A. **Sortie :** mise à jour de `tickCount`.

Table 3 – Décomposition interne de l'ISR (étapes + lignes).

#	Lignes	Action	Pourquoi c'est fait
1	21	Entrée ISR INT0	Déclenchement matériel précis sur front montant.
2	22	Lecture `PIND & (1<<PD3)`	Lecture registre (très rapide) adaptée ISR.
3	23/25	`tickCount++` ou `tickCount-`	Encode le sens par le signe : simplifie RPM et PID.

0.4.3 `driveMotorSigned(int pwm)` — appliquer la commande (lignes 51–68)

Rôle : transformer une commande *signée* en écritures PWM matérielles :

- saturation à la plage [-255 ;255],
- zone morte pour éviter les micro-commandes,
- seuil minimal pour dépasser le frottement,
- choix du canal OC1A ou OC1B selon le signe.

TABLE 4 – Étapes de `driveMotorSigned()` (avec lignes).

#	Lignes	Action	Effet physique / intérêt
1	52	Saturation `constrain`	Empêche dépassement OCR (8 bits).
2	54–55	Seuil minimal (±40)	Aide au démarrage (frottements, couple de démarrage).
3	56	Zone morte (abs<10)	Supprime petites valeurs (tremblements).
4	58–60	Cas pwm>0 : OCR1A=pwm, OCR1B=0	Sens positif : un seul canal actif.
5	61–63	Cas pwm<0 : OCR1A=0, OCR1B=-pwm	Sens négatif : un seul canal actif.
6	64–67	Cas pwm=0 : OCR1A=OCR1B=0	Arrêt complet.

0.4.4 `readTargetFromSerial()` — consigne dynamique (lignes 75–87)

Rôle : lire un nombre ASCII sur Serial de manière non-bloquante. **Principe :** accumulation des caractères jusqu'à `\n`, puis conversion via `atof` et bornage.

TABLE 5 – Étapes de lecture de consigne .

#	Lignes	Action	Robustesse
1	76	Boucle tant que `Serial.available()`	Ne bloque pas : loop reste fluide.
2	77	`Serial.read()`	Lit un caractère à la fois.
3	78	Ignore `\r`	Compatibilité CRLF.
4	79–82	À `\n` : terminaison + atof + apply	Déclenche mise à jour consigne.
5	83–84	Stockage si pas fin de ligne	Accumule le nombre.
6	84	Protection taille buffer	Évite overflow du tableau.

0.4.5 `applyNewTarget(float v)` — bornage consigne (lignes 70–73)

Rôle : appliquer une consigne sûre (lignes 71–72). Le bornage évite des valeurs absurdes (ex : chaîne mal reçue) et protège le moteur/driver.

0.5 `loop()` — cycle complet RPM+PID+PWM (lignes 95–128)

Rôle : orchestrer la régulation :

- déclenchement périodique (toutes les SAMPLE_MS),
- lecture atomique des ticks,
- calcul RPM,
- calcul PID,
- conversion PWM,
- application + logs.

TABLE 6 – Traçabilité interne du cycle de régulation dans `loop()`.

#	Lignes	Bloc	Pourquoi
1	96	Lecture consigne	Réglage en temps réel sans recompiler.
2	98–101	Période + dt	Cycle fixe pour stabilité/lecture RPM.
3	105–109	Lecture atomique ticks	Évite incohérences ISR/loop.
4	112	Calcul RPM	Transforme compteur en mesure physique.
5	114–122	PID discret	Génère une commande corrigée (P/I/D).
6	123	Conversion PWM	Adaptation matériel (8 bits + signe).
7	125	Application moteur	Écriture OCR1A/OCR1B.
8	127–130	Logs	Diagnostic + graphes.

0.6 Mesure de vitesse : détails, limites et interprétation

Le compteur signé (`tickCount`) encode simultanément :

— **la quantité de rotation** (valeur absolue),
— **le sens** (signe).

Ainsi, `rpm` (ligne 112) hérite automatiquement du signe : un RPM négatif correspond au sens inverse.

0.6.1 Quantification à basse vitesse

Sur une fenêtre de 0.4 s, un seul tick correspond à :

$$RPM_{1tick} = \frac{1}{TICKS\_PER\_REV} \cdot \frac{60}{0.4}$$

Plus `TICKS_PER_REV` est petit et plus la vitesse mesurée est « en escaliers » à faible vitesse. Cette observation justifie le compromis `SAMPLE_MS=400` ms : stabilité et simplicité.

`digitalRead()` est pratique mais lente (surcouche Arduino). Dans une ISR, on cherche une latence minimale : lire `PIND` est une lecture registre 8 bits très rapide, ce qui diminue le temps passé en interruption et limite la perte d'impulsions.

0.7 PID et conversion PWM

0.7.1 PID discret

Dans `loop()` :

— erreur (ligne 114) : $e = r - y$,
— intégrale (115–116) : $I \leftarrow I + e \cdot dt$ (bornée),
— dérivée (118) : $D = (e - e_{prev})/dt$,
— commande (121) : $u = K_p e + K_i I + K_d D$.

La sortie `u` est en unités « PWM » car les gains ont été choisis pour que la valeur finale soit directement comparable à [-255 ;255].

0.7.2 Pourquoi `constrain` + `lroundf` ?

`lroundf` arrondit proprement un float en int. `constrain` impose une saturation strictement compatible avec le Timer1 en PWM 8 bits. C'est une protection indispensable pour éviter des écritures hors plage sur `OCR1A/OCR1B`.

0.8 Tests et validation)

0.8.1 Protocole de test

Pour produire des résultats solides :

1. Fixer une consigne (ex : 150 RPM), noter la réponse (temps de montée, dépassement).
2. Appliquer une perturbation (frein léger sur l'axe / charge) et observer la correction.
3. Tester plusieurs consignes (50, 150, 300 RPM) et comparer.
4. Exporter les logs série au format CSV (Target, RPM).

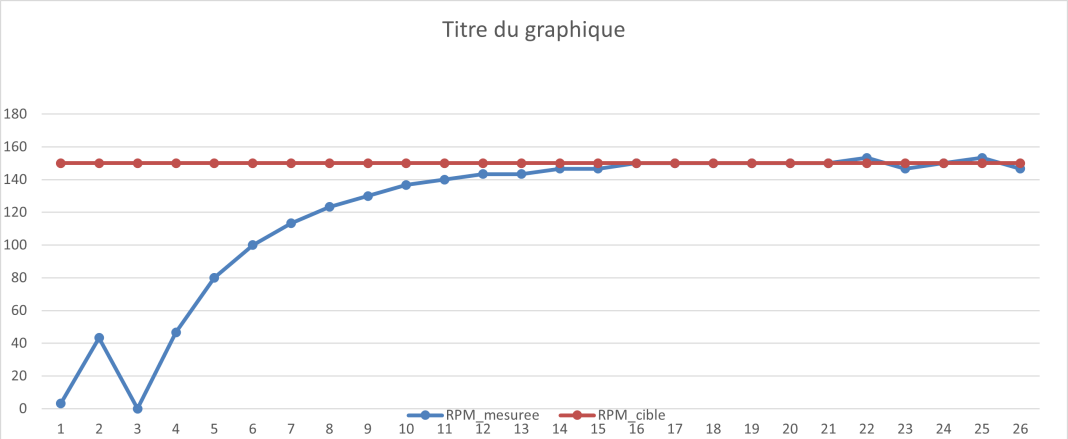

FIGURE 4 – Réponse à échelon

0.9 Conclusion

Ce projet a permis de mettre en place un asservissement complet de la vitesse d'un moteur DC, depuis la mesure jusqu'à la commande. La vitesse est estimée à partir d'un encodeur incrémental, dont les impulsions sont comptées en interruption (INT0). Le choix d'une ISR courte (lecture directe de registre + incrément/décrément) garantit une exécution rapide et réduit le risque de perte d'impulsions lorsque la vitesse augmente. La lecture atomique du compteur dans la boucle principale assure la cohérence des données partagées ISR/`loop()` et fiabilise le calcul de la vitesse en RPM.

La commande est appliquée par PWM matérielle (Timer1), ce qui apporte une sortie stable et indépendante des variations de temps d'exécution du programme. Le correcteur PID discret, avec saturation de l'intégrale, constitue un compromis efficace entre performance et

robustesse : le terme proportionnel corrige rapidement l'erreur, l'intégrale compense les perturbations (charge, frottements) et la limitation de l'intégrale évite l'enroulement (windup) lorsque la commande atteint ses limites. La conversion en PWM signée et le routage sur OC1A/OC1B permettent de gérer proprement le sens de rotation avec une logique de commande simple.

Enfin, la structure du code et la traçabilité lignes→fonctionnalités facilitent la compréhension et la validation expérimentale via les logs série. En perspective, la précision à basse vitesse pourrait être améliorée par une meilleure résolution de comptage (x2/x4) ou un filtrage de la mesure, et le réglage des gains PID peut être systématisé par une identification du modèle (estimation de K et τ à partir d'un essai échelon.

Annexe A

Annexe A)

```
#include <Arduino.h>
#include <avr/io.h>
#include <avr/interrupt.h>

const float TICKS_PER_REV = 90.0f;
const uint32_t SAMPLE_MS = 400;

float Kp = 0.1f;
float Ki = 1.5f;
float Kd = 0.0f;

float targetRPM = 150.0f;
volatile int32_t tickCount = 0;
float integral = 0.0f;
float prevError = 0.0f;
uint32_t lastTime = 0;

char inBuf[20];
uint8_t inLen = 0;

ISR(INT0_vect) {                              // ISR interruption externe INT0 (D2 / PD2)
  if (PIND & (1 << PD3)) {                    // Lecture directe du registre PIND ( tat  pin D3 / PD3)
    tickCount++;                              // Sens 1 (voie B = 1 au front de A)
  } else {
    tickCount--;                              // Sens inverse (voie B = 0 au front de A)
  }
}

void setupRegisters() {
  cli();                                      // D sactive globalement les interruptions

  DDRB |= (1 << PB1) | (1 << PB2);            // DDRB: PB1(D9/OC1A) et PB2(D10/OC1B) en sortie (PWM)
  DDRD &= ~((1 << PD2) | (1 << PD3));         // DDRD: PD2(D2/INT0) et PD3(D3) en entr e (encodeur)
  PORTD |= (1 << PD2) | (1 << PD3);           // PORTD: pull-up internes activ s sur PD2 et PD3

  TCCR1A = (1 << COM1A1) | (1 << COM1B1) | (1 << WGM10);
```

```
37    // TCCR1A: active OC1A/OC1B en PWM non-invers  + bit WGM10 (mode PWM 8
          bits)
38
39    TCCR1B = (1 << CS11) | (1 << CS10);
40    // TCCR1B: prescaler Timer1 = 64 (d marre le timer => PWM sur D9/D10)
41
42    OCR1A = 0;                                  // OCR1A: duty-cycle PWM sur
          OC1A (D9)
43    OCR1B = 0;                                  // OCR1B: duty-cycle PWM sur
          OC1B (D10)
44
45    EICRA |= (1 << ISC01) | (1 << ISC00);    // INT0 sur front montant
46    EIMSK |= (1 << INT0);                       // Autorise INT0
47
48    sei();                                      // R active globalement les
          interruptions
49  }
50
51  void driveMotorSigned(int pwm) {
52    pwm = constrain(pwm, -255, 255);
53
54    if (pwm > 0 && pwm < 40)  pwm = 40;      // PWM minimal pour surmonter
          frottements
55    if (pwm < 0 && pwm > -40) pwm = -40;     // idem sens inverse
56    if (abs(pwm) < 10) pwm = 0;              // zone morte
57
58    if (pwm > 0) {
59      OCR1A = pwm;                              // sens +
60      OCR1B = 0;
61    } else if (pwm < 0) {
62      OCR1A = 0;
63      OCR1B = -pwm;                             // sens -
64    } else {
65      OCR1A = 0;
66      OCR1B = 0;                                // arr t
67    }
68  }
69
70  void applyNewTarget(float v) {
71    v = constrain(v, -1000.0f, 1000.0f);     // bornage s curit
72    targetRPM = v;
73  }
74
75  void readTargetFromSerial() {
76    while (Serial.available()) {
77      char c = (char)Serial.read();
78      if (c == '\r') continue;
79      if (c == '\n') {
80        inBuf[inLen] = '\0';
81        applyNewTarget(atof(inBuf));
82        inLen = 0;
83      } else {
84        if (inLen < sizeof(inBuf) - 1) inBuf[inLen++] = c;
85      }
86    }
87  }
88
89  void setup() {
90    Serial.begin(115200);
```

```
  setupRegisters();
  lastTime = millis();
}

void loop() {
  readTargetFromSerial();

  uint32_t now = millis();
  if (now - lastTime >= SAMPLE_MS) {
    float dt = (now - lastTime) / 1000.0f;
    lastTime = now;

    uint8_t oldSREG = SREG;
    cli();
    int32_t ticks = tickCount;
    tickCount = 0;
    SREG = oldSREG;

    float rpm = (ticks / TICKS_PER_REV) / dt * 60.0f;

    float error = targetRPM - rpm;
    integral += error * dt;
    integral = constrain(integral, -1200.0f, 1200.0f);

    float derivative = (error - prevError) / dt;
    prevError = error;

    float u = Kp * error + Ki * integral + Kd * derivative;
    int pwmCmd = constrain((int)lroundf(u), -255, 255);

    driveMotorSigned(pwmCmd);

    Serial.print("Target:");
    Serial.print(targetRPM);
    Serial.print(",RPM:");
    Serial.println(rpm);
  }
}
```

Listing A.1 – Code Annexe A du projet