TP – Asservissement en distance du mBot avec correcteur PID

🎯 Objectifs - Comprendre le fonctionnement d'un **asservissement en boucle fermée** - Utiliser un capteur ultrason pour mesurer une distance - Découvrir les rôles des termes **P**, **I**, **D** du PID - Expérimenter la stabilité, le dépassement, l'erreur statique - Analyser les effets du bruit et des saturations

—

🧰 Matériel nécessaire - mBot ou carte mCoreLite avec capteur ultrason - Arduino IDE ou mBlock (selon le matériel) - Un obstacle (mur, boîte, livre) - Une règle ou un mètre

—

1) Principe général de l'asservissement

Le robot doit se stabiliser à **20 cm** de l'obstacle. On définit une **erreur** :

$$e = D_{\text{mes}} - D_{\text{cons}}$$

- **e > 0** : le robot est trop loin → il doit avancer - **e < 0** : le robot est trop près → il doit reculer

La commande se calcule via un correcteur PID :

$$v = K_p\, e \;+\; K_i \int e\, dt \;+\; K_d \frac{de}{dt}$$

🔧 Rôle des termes - **P** : corrige proportionnellement à l'erreur - **I** : supprime l'erreur statique mais peut créer des oscillations - **D** : amortit, réduit le dépassement

🧪 Filtrage du dérivé Le capteur ultrason étant bruyant, on utilise :

$$d_{\text{filt}} = \alpha\, d_{\text{filt}} + (1-\alpha)\, \frac{e - e_{\text{prev}}}{dt}$$

🧪 Anti-windup (limitation de l'intégrale)

$$I = \text{clip}(I + e\, dt,\; -I_{\max},\; I_{\max})$$

Et reset si changement de signe :

$$e \cdot e_{\text{prev}} < 0 \;\Rightarrow\; I = 0$$

🧪 Saturation des moteurs

$$v = \begin{cases} V_{\max} & \text{si } v > V_{\max} \\ -V_{\max} & \text{si } v < -V_{\max} \\ v & \text{sinon} \end{cases}$$

🧪 Vitesse minimale

$$|v| < V_{\min} \Rightarrow v = \begin{cases} V_{\min} & \text{si } v > 0 \\ -V_{\min} & \text{si } v < 0 \end{cases}$$

—

2) Travail demandé

Partie A — Analyse conceptuelle 1. Expliquer avec vos mots ce que représente l'erreur *e*. 2. Justifier l'intérêt d'un correcteur en boucle fermée. 3. Décrire séparément les rôles des termes **P**, **I**, **D**. 4. Expliquer pourquoi le dérivé doit être filtré. 5. Expliquer à quoi sert l'anti-windup.

—

3) Partie B — Expérimentations successives

⚙ Étape 1 — Action **P seule** - Mettre **Ki = 0**, **Kd = 0** - Tester plusieurs valeurs de **Kp** : 2, 4, 6, 8… - Observer la stabilisation à 20 cm

Questions : - Y a-t-il une erreur statique ? - Le dépassement existe-t-il ? - Le robot oscille-t-il lorsque Kp est trop grand ?

—

⚙ Étape 2 — Action **PI** - Ajouter une valeur de **Ki** (ex. 0.1 à 0.2)

Questions : - L'erreur statique disparaît-elle ? - Le comportement devient-il plus oscillant ? - Que se passe-t-il si Ki est trop grand ?

—

⚙ Étape 3 — Action **PID** - Ajouter un terme **Kd** (ex. 0.5 ou 1.0)

Questions : - Le dépassement diminue-t-il ? - Le robot est-il plus stable lors de l'approche ? - Le dérivé amplifie-t-il le bruit ? - Le filtrage (paramètre α) améliore-t-il la stabilité ?

—

4) Partie C — Analyse des courbes

Le moniteur série du robot affiche typiquement :

```
#include "mCoreLite.h"

mCoreLite bot;

// ----- CONSIGNE -----
float Dcons = 20.0;      // distance cible (cm)

// ----- PARAMÈTRES PID -----
float Kp = 6.0;          // proportionnel
float Ki = 0.15;         // intégral
```

```
float Kd = 0;          // dérivé (amortissement)

// ----- MEMOIRES PID -----
float integral = 0.0;
float e_prev = 0.0;
unsigned long lastTime = 0;

// ----- SECURITE INTEGRALE (ANTI-WINDUP) -----
float Imax = 50.0;       // limite de l'intégrale

// ----- VITESSE -----
int Vmin = 60;           // évite les sifflements
int Vmax = 200;          // saturation

// ----- FILTRE DERIVÉ -----
// (évite les tremblements dus au bruit US)
float alpha = 0.7;       // 0 = pas de filtre, 1 = filtre fort
float d_filtered = 0.0;

void setup() {
  Serial.begin(115200);
  bot.botSetup();
  lastTime = millis();
}

void loop() {

  // ----- 1) MESURE -----
  float Dmes = bot.distanceCM();

  // ----- 2) ERREUR -----
  float e = Dmes - Dcons;

  // ----- 3) TEMPS -----
  unsigned long now = millis();
  float dt = (now - lastTime) / 1000.0;
  if (dt <= 0) dt = 0.001;   // sécurité
  lastTime = now;

  // ----- 4) ZONE MORTE -----
  if (fabs(e) < 1.0) {
    integral = 0;           // détendre l'intégrale
    bot.motors(0,0);
    debug(Dmes, e, 0, integral, 0);
    return;
  }

  // ----- 5) INTÉGRALE + ANTI-WINDUP -----
  integral += e * dt;
  if (integral >  Imax) integral =  Imax;
  if (integral < -Imax) integral = -Imax;
```

```
  // Réinitialisation si inversion de signe de l'erreur
  if (e * e_prev < 0) integral = 0;

  // ----- 6) DÉRIVÉE (!\ capteur bruyant) -----
  float derivative = (e - e_prev) / dt;
  e_prev = e;

  // Filtre passe-bas sur le dérivé
  d_filtered = alpha * d_filtered + (1 - alpha) * derivative;

  // ----- 7) SORTIE PID -----
  float v = Kp * e + Ki * integral + Kd * d_filtered;

  // ----- 8) SATURATION -----
  if (v >  Vmax) v =  Vmax;
  if (v < -Vmax) v = -Vmax;

  // ----- 9) VITESSE MINIMALE -----
  if (fabs(v) > 0 && fabs(v) < Vmin) {
    v = (v > 0) ? Vmin : -Vmin;
  }

  // ----- 10) ACTION -----
  // v > 0 → avancer (cible trop loin)
  // v < 0 → reculer (cible trop proche)
  bot.motors(-v, v);

  // ----- 11) DEBUG -----
  debug(Dmes, e, v, integral, d_filtered);
}

// ================== DEBUG ======================
void debug(float D, float e, float v, float I, float d) {
  Serial.print("D=");
  Serial.print(D);
  Serial.print("  e=");
  Serial.print(e);
  Serial.print("  v=");
  Serial.print(v);
  Serial.print("  I=");
  Serial.print(I);
  Serial.print("  d=");
  Serial.println(d);
}
```