Exercices de cours – Corrigés

TD 1 : Corrigé

Points essentiels

1. Sortie numérique

```
gpio_set_dir(pin, GPIO_OUT);
gpio_put(pin, value);
```

Le programme impose un niveau logique à une broche.

2. Entrée numérique

```
gpio_set_dir(pin, GPIO_IN);
bool value = gpio_get(pin);
```

Le programme observe un niveau électrique externe.

3. Pull-up

```
gpio_pull_up(pin);
```

Elle évite une entrée flottante et permet un câblage simple du bouton vers GND.

4. Niveau et événement

```
niveau : « le bouton est actuellement appuyé »
événement : « le bouton vient d'être appuyé »
```

La détection de front nécessite de conserver un état précédent.

5. ADC

```
uint16_t value = adc_read();
```

Une tension analogique est convertie en une valeur numérique.

6. PWM

```
pwm_set_gpio_level(pin, level);
```

Hypothèses et remarques générales

Les solutions suivantes utilisent le brochage du sujet :

```
constexpr uint LED_PIN    = 15;
constexpr uint BUTTON_PIN = 14;
```

Les boutons sont supposés connectés entre le GPIO et GND. On utilise donc une **pull-up interne** :

```
bouton relâché $\rightarrow$ GPIO = 1
bouton appuyé  $\rightarrow$ GPIO = 0
```

Ainsi, il est souvent pratique d'écrire :

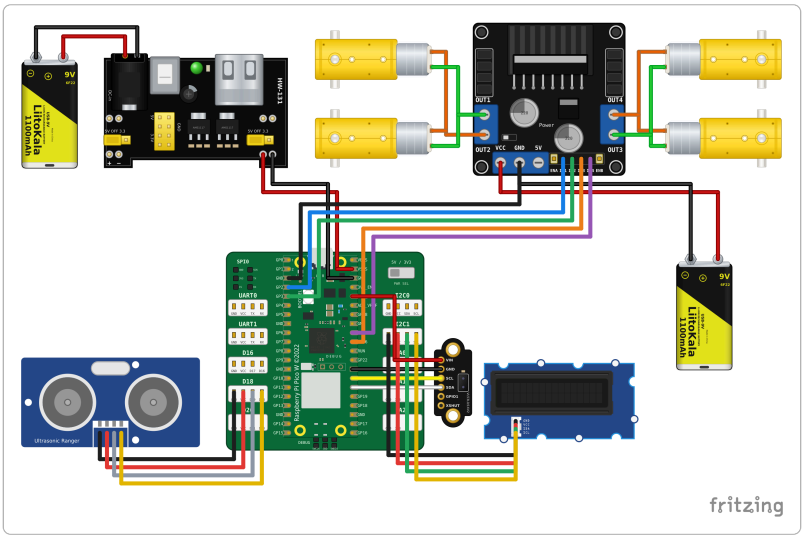
```
bool pressed = !gpio_get(BUTTON_PIN);
```

Pour les exercices utilisant `printf()`, il faut appeler :

```
stdio_init_all();
```

et activer la sortie USB ou UART dans `CMakeLists.txt` selon l'environnement utilisé.

Exercice 1 : Branchements



Exercice 2 : Hello GPIO

Solution minimale

```c
#include "pico/stdlib.h"

constexpr uint LED_PIN = 15;

int main() {
    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);

    while (true) {
        gpio_put(LED_PIN, 1);
        sleep_ms(1000);

        gpio_put(LED_PIN, 0);
        sleep_ms(1000);
    }
}
```

`gpio_init()` initialise le GPIO, `gpio_set_dir()` fixe sa direction et `gpio_put()` fixe son niveau logique.

Clignotement à 5 Hz

Une fréquence de 5 Hz correspond à une période :

```
T = 1 / f = 1 / 5 = 0,2 s = 200 ms
```

Avec un rapport cyclique de 50 %, chaque demi-période dure 100 ms :

```
while (true) {
    gpio_put(LED_PIN, true);
    sleep_ms(100);
    gpio_put(LED_PIN, false);
    sleep_ms(100);
}
```

100 ms ON / 900 ms OFF

```
while (true) {
    gpio_put(LED_PIN, true);
    sleep_ms(100);
    gpio_put(LED_PIN, false);
    sleep_ms(900);
}
```

La période vaut ici 1 s, soit une fréquence de 1 Hz, avec un rapport cyclique de 10 %.

Fonction `blink()`

```
void blink(uint pin, uint duration_ms) {
    gpio_put(pin, true);
    sleep_ms(duration_ms);
    gpio_put(pin, false);
    sleep_ms(duration_ms);
}
```

Utilisation :

```
while (true) {
    blink(LED_PIN, 250);
}
```

Exercice 3 : Lire un bouton

Le **numéro du GPIO** (GPIO14) identifie une broche logique du microcontrôleur ; son **état logique** est la valeur 0 ou 1 sur cette broche.

```
#include <cstdio>
#include "pico/stdlib.h"

constexpr uint BUTTON_PIN = 14;

int main() {
    stdio_init_all();

    gpio_init(BUTTON_PIN);
    gpio_set_dir(BUTTON_PIN, GPIO_IN);
    gpio_pull_up(BUTTON_PIN);

    while (true) {
        bool raw = gpio_get(BUTTON_PIN);
        bool pressed = !raw;
```

```
        printf("button = %d, pressed = %d\n", raw, pressed);
        sleep_ms(100);
    }
}
```

La résistance de pull-up impose un état connu lorsque le bouton est ouvert. Sans elle, l'entrée pourrait être flottante et donc prendre un niveau indéterminé.

Avec ce câblage :

```
bouton relâché : entrée reliée faiblement au 3,3 V -> 1
bouton appuyé  : entrée reliée directement à GND   -> 0
```

D'où :

```
bool pressed = !gpio_get(BUTTON_PIN);
```

La valeur physique « bouton appuyé » est ici représentée par un zéro logique : on parle souvent d'entrée **active à l'état bas**.

Exercice 4 : Utilisation des moteurs

```
programme
   │
   │ consigne : vitesse = -255 ... +255
   ▼
controleMoteur()
   │
   ├── signe ──────────────► choix du sens
   │
   └── valeur absolue ─────► rapport cyclique PWM
                                  │
                                  ▼
                            GPIO A / GPIO B
                                  │
                                  ▼
                              pont en H
                                  │
                                  ▼
                               moteur
```

```
#define LEFT_MOTOR_PIN_A   26
#define LEFT_MOTOR_PIN_B   27
#define RIGHT_MOTOR_PIN_A  2
#define RIGHT_MOTOR_PIN_B  3

void controleMoteur(uint pinA, uint pinB, int vitesse)
{
    int puissance = vitesse < 0 ? -vitesse : vitesse;

    if (vitesse > 0) {
```

```
        pwm_set_gpio_level(pinA, puissance);
        pwm_set_gpio_level(pinB, 0);
    }
    else if (vitesse < 0) {
        pwm_set_gpio_level(pinA, 0);
        pwm_set_gpio_level(pinB, puissance);
    }
    else {
        pwm_set_gpio_level(pinA, 0);
        pwm_set_gpio_level(pinB, 0);
    }
}
void controleMoteurs(int gauche, int droite)
{
    controleMoteur(LEFT_MOTOR_PIN_A, LEFT_MOTOR_PIN_B, gauche);
    controleMoteur(RIGHT_MOTOR_PIN_A, RIGHT_MOTOR_PIN_B, droite);
}
```

```
                          ┌── moteur avant gauche
vitesse gauche ── L298N ──┤
                          └── moteur arrière gauche

                          ┌── moteur avant droit
vitesse droite ── L298N ──┤
                          └── moteur arrière droit
```

Exercice 5 : Missions

1. Écrire un programme qui fait avancer le robot pendant 1 seconde, puis reculer le robot pendant 1 seconde.
2. Écrire un programme qui fait tourner le robot de 90° vers la gauche.

Chaque exercice est traité avec le protocole JSON, puis avec l'API `robot.js` (téléphone) et l'API `Robot.h` (robot).

Tourner de 90°

Le robot n'a ni encodeur ni gyroscope : il ne mesure pas l'angle dont il a tourné. Un virage sur place (roue gauche en arrière, roue droite en avant) dure un temps fixé, et ce temps donne l'angle. La valeur `600` ms à la vitesse `180` est un point de départ, à mesurer sur chaque robot et sur le sol de la salle.

1. Poser le robot sur le sol de la salle, aligné sur un repère (bord de dalle, ruban adhésif).
2. Lancer le quart de tour à gauche avec la durée courante.
3. Relever l'angle obtenu.
4. Corriger la durée en proportion : si 600 ms donnent 75°, essayer `600 * 90 / 75 = 720` ms.
5. Recommencer jusqu'à obtenir 90°, puis vérifier avec quatre quarts de tour d'affilée : le robot doit revenir face au repère.

Remarque. `turnLeft()` dans `robot.js` vaut par défaut 500 ms à la vitesse 200. Cette valeur par défaut n'est pas un quart de tour mesuré.

Attention. La durée mesurée ne vaut que pour une vitesse, un sol et un niveau de batterie. Changer l'un des trois oblige à la mesurer de nouveau.

Protocole JSON

Conventions des échanges ci-dessous :

- → : ligne envoyée au robot (console USB, ou corps de `POST /api/command`).
- ← : ligne envoyée par le robot (console USB, ou `GET /api/lines`).
- Les champs des capteurs (`front_mm`, `front_ok`, `front_q`, `side_mm`, `side_ok`, `side_q`) sont omis des lignes de télémétrie pour la lisibilité.
- Les valeurs de `ms` et de `seq` sont des exemples. La télémétrie part toutes les 200 ms par défaut ; seules les lignes utiles sont montrées.

Question 1 : avancer 1 s, puis reculer 1 s

À la main (console USB ou `curl`). Deux commandes de mouvement, la seconde envoyée après la ligne `done:true` de la première :

```
→ {"seq":1,"left":200,"right":200,"duration":1000}
← {"seq":1,"ms":41230,"left":200,"right":200,"moving":true,"mission":0}
← {"seq":1,"ms":42230,"left":0,"right":0,"moving":false,"mission":0,"done":true}
→ {"seq":2,"left":-200,"right":-200,"duration":1000}
← {"seq":2,"ms":42430,"left":-200,"right":-200,"moving":true,"mission":0}
← {"seq":2,"ms":43390,"left":0,"right":0,"moving":false,"mission":0,"done":true}
```

`"duration":1000` est la valeur par défaut et peut être omis. `seq` est facultatif ; il permet de savoir à quelle commande correspond une ligne `done`.

Avec `curl`, en remplaçant `<robot-ip>` par l'adresse affichée sur l'écran du robot :

```
curl -d '{"seq":1,"left":200,"right":200,"duration":1000}' http://<robot-ip>/api/
command
curl -d '{"seq":2,"left":-200,"right":-200,"duration":1000}' http://<robot-ip>/api/
command
```

Remarque. En salle, les ordinateurs ne sont pas sur le point d'accès du robot. `curl` s'utilise alors contre `client/fakerobot.js`, à l'adresse `http://localhost:8080`.

Erreur fréquente. Envoyer les deux commandes d'un coup, dans le même corps de requête ou sans attendre le `done`. Un mouvement remplace le mouvement en cours : le robot exécute le retour presque aussitôt et l'aller n'a pas lieu.

Avec `robot.js`, classe `Robot`. `forward()` et `backward()` envoient les mêmes lignes. `send()` ajoute `seq` après les autres champs ; les tags partent d'une valeur aléatoire :

```
→ {"left":200,"right":200,"duration":1000,"seq":5731}
← … ligne done:true avec "seq":5731 …
→ {"left":-200,"right":-200,"duration":1000,"seq":5732}
← … ligne done:true avec "seq":5732 …
```

Avec `robot.js`, classe `Mission`. `Mission.run()` encadre les étapes par deux commandes de réglage :

```
→ {"telemetry":100,"seq":5731}
→ {"left":200,"right":200,"duration":1000,"seq":5732}
```

```
← … ligne done:true avec "seq":5732 …
→ {"left":-200,"right":-200,"duration":1000,"seq":5733}
← … ligne done:true avec "seq":5733 …
→ {"stop":true,"seq":5734}
```

`{"telemetry":100}` fait passer la télémétrie à 100 ms pour les conditions `stopWhen()`. `{"stop":true}` final laisse le robot arrêté ; il ne produit pas de ligne `done`.

Avec `Robot.h`. Une seule commande traverse la liaison. Les deux mouvements sont décidés sur le robot et n'apparaissent que dans la télémétrie :

```
→ {"mission":1}
← {"seq":-1,"ms":41230,"left":200,"right":200,"moving":true,"mission":1}
← {"seq":-1,"ms":42430,"left":-200,"right":-200,"moving":true,"mission":1}
← {"seq":-1,"ms":43230,"left":0,"right":0,"moving":false,"mission":0,"done":true}
```

Il n'y a qu'une ligne `done:true`, à la fin du retour : le retour remplace l'aller sans que les roues s'arrêtent. `seq` reste à `-1` tant qu'aucune commande n'a fourni de tag. Depuis le téléphone, `bot.mission(1)` envoie `{"mission":1,"seq":5731}` et la télémétrie porte alors `"seq":5731`.

Question 2 : quart de tour à gauche

À la main. Une seule commande de mouvement, roue gauche en arrière, roue droite en avant :

```
→ {"seq":3,"left":-180,"right":180,"duration":600}
← {"seq":3,"ms":50100,"left":-180,"right":180,"moving":true,"mission":0}
← {"seq":3,"ms":50700,"left":0,"right":0,"moving":false,"mission":0,"done":true}
```

`600` est la durée à mesurer.

Avec `robot.js`, classe `Robot`. `turnLeft(180, 600)` envoie :

```
→ {"left":-180,"right":180,"duration":600,"seq":5735}
← … ligne done:true avec "seq":5735 …
```

Avec `robot.js`, classe `Mission`.

```
→ {"telemetry":100,"seq":5735}
→ {"left":-180,"right":180,"duration":600,"seq":5736}
← … ligne done:true avec "seq":5736 …
→ {"stop":true,"seq":5737}
```

Avec `Robot.h`.

```
→ {"mission":2}
← {"seq":-1,"ms":50100,"left":-180,"right":180,"moving":true,"mission":0}
← {"seq":-1,"ms":50700,"left":0,"right":0,"moving":false,"mission":0,"done":true}
```

`"mission":0` apparaît dès la première ligne : `quarterTurnLeft()` remet `_mission` à 0 sur le tour même où elle lance le virage (voir la remarque de l'exercice 2 côté `Robot.h`).

API `robot.js` (téléphone)

Les méthodes de `Robot` envoient une commande et rendent son `seq` sans attendre. `waitUntilDone(seq)` attend la ligne `done:true` du robot. Le code ci-dessous s'exécute dans un module ES (`await` au premier niveau) ou dans une fonction `async`.

Question 1 : avancer 1 s, puis reculer 1 s

```
import { Robot } from './robot.js';

const bot = new Robot();
bot.start();                                    // lit la télémétrie toutes les 100 ms

const aller = bot.forward(200, 1000);
if (await bot.waitUntilDone(aller) === null) throw new Error('pas de fin d\'aller');

const retour = bot.backward(200, 1000);
if (await bot.waitUntilDone(retour) === null) throw new Error('pas de fin de
retour');
```

`waitUntilDone()` rend `null` après 6 s sans ligne `done`. Le test évite d'envoyer le retour alors que l'aller n'a jamais été confirmé.

Même programme sous forme de mission du téléphone, dans l'objet `missions` de `robot/web/missions.js` :

```
  'avancer-reculer': ({ speed = 200 } = {}) =>
    new Mission('avancer-reculer')
      .forward(speed, 1000)
      .backward(speed, 1000),
```

`Mission.run()` envoie chaque étape, attend son `done` et laisse le robot arrêté à la fin.

Attention. Entre l'aller et le retour, le robot reste immobile un court moment : il arrête ses roues à l'échéance, puis le téléphone doit recevoir la ligne `done` (jusqu'à 100 ms de scrutation) et envoyer la commande suivante (un aller-retour HTTP). Combien de millisecondes séparent le `ms` de la ligne `done` de l'aller et le `ms` de la première ligne avec `"left":-200` ?

Question 2 : quart de tour à gauche

```
import { Robot } from './robot.js';

const QUARTER_TURN_SPEED = 180;
const QUARTER_TURN_MS    = 600;                 // à mesurer, voir « Tourner de 90° »

const bot = new Robot();
bot.start();

const virage = bot.turnLeft(QUARTER_TURN_SPEED, QUARTER_TURN_MS);
if (await bot.waitUntilDone(virage) === null) throw new Error('pas de fin de
virage');
```

`turnLeft(speed, duration)` envoie `drive(-speed, speed, duration)` : la roue gauche recule, la roue droite avance, le robot tourne sur place.

Sous forme de mission du téléphone, avec les deux constantes ci-dessus déclarées en tête de `missions.js` :

```
  'quart-gauche': ({ speed = QUARTER_TURN_SPEED } = {}) =>
    new Mission('quart-gauche')
      .turnLeft(speed, QUARTER_TURN_MS),
```

Erreur fréquente. Passer le curseur de vitesse de la page à cette mission sans recalibrer : `QUARTER_TURN_MS` n'est juste qu'à `QUARTER_TURN_SPEED`.

API `Robot.h` (robot)

Un programme du robot est une mission : une méthode de la classe `Robot` appelée à chaque tour de `Robot::update()`. Elle ne doit jamais attendre (`sleep_ms()` est interdit) : elle lit l'heure avec `nowMs()`, bouge avec `drive()` et rend la main tout de suite. Elle garde son état dans des membres déclarés à côté d'elle.

Le code ci-dessous part du firmware de départ, où aucune mission n'existe : il utilise les numéros 1 et 2. Si les missions du programme des TP sont déjà en place (1 à 5), prendre 6 et 7 et ajouter les noms à la fin de `MISSION_NAMES`.

Déclaration des deux missions

Dans `robot/Robot.h` :

```
  static constexpr const char* MISSION_NAMES = "avancer-reculer,quart-gauche";

  void mission() {
    switch (_mission) {
      case 0: break;
      case 1: forwardThenBack(); break;
      case 2: quarterTurnLeft(); break;
      default: break;
    }
  }
```

Dans `execute()`, remettre l'exercice 1 à son début à chaque sélection de mission. Un `{"mission":1}` envoyé après un arrêt en cours de route repart alors de l'aller :

```
    if (command.has("mission")) {
      _mission  = (int)command.integer("mission");
      _backStep = 0;
    }
```

Question 1 : avancer 1 s, puis reculer 1 s

```
  // Avancer 1 s, puis reculer 1 s. Étapes : 0 à lancer, 1 aller, 2 retour.
  int      _backStep  = 0;
  uint32_t _backUntil = 0;

  void forwardThenBack() {
    if (_backStep != 0 && (int32_t)(nowMs() - _backUntil) < 0) return;  // étape en
cours

    if (_backStep == 0) {
      drive(200, 200, 1000);
      _backStep = 1;
```

```
    } else if (_backStep == 1) {
      drive(-200, -200, 1000);
      _backStep = 2;
    } else {
      _backStep = 0;
      _mission  = 0;              // fini : enforceSafety() arrête les roues sur ce
tour
      return;
    }
    _backUntil = _moveDeadline;   // l'étape finit avec le mouvement
  }
```

Lancement depuis la console USB ou par le bouton « avancer-reculer » du téléphone :

```
{"mission":1}
```

`mission()` s'exécute avant `enforceSafety()` dans `Robot::update()`. Sur le tour où l'aller arrive à échéance, `forwardThenBack()` a déjà lancé le retour avec une nouvelle échéance : les roues passent de +200 à -200 sans arrêt intermédiaire. Sur le tour où le retour arrive à échéance, la mission se désactive, puis `enforceSafety()` arrête les roues et envoie la ligne `done:true`.

Attention. La comparaison `(int32_t)(nowMs() - _backUntil) < 0` reste juste quand l'horloge de 32 bits repasse à zéro ; `nowMs() < _backUntil` ne l'est pas. Voir `API.md` — Time.

Attention. Passer directement de +200 à -200 sollicite les moteurs et la batterie. Si le robot redémarre au changement de sens, ajouter une étape d'arrêt de 200 ms entre l'aller et le retour avec `drive(0, 0, 200)`.

Question 2 : quart de tour à gauche

Dans la classe `Robot`, à côté de la méthode :

```
  static constexpr int      QUARTER_TURN_SPEED = 180;
  static constexpr uint32_t QUARTER_TURN_MS    = 600;   // à mesurer, voir « Tourner
de 90° »

  // Un seul mouvement : enforceSafety() l'arrête à l'échéance.
  void quarterTurnLeft() {
    drive(-QUARTER_TURN_SPEED, QUARTER_TURN_SPEED, QUARTER_TURN_MS);
    _mission = 0;
  }
```

Lancement :

```
{"mission":2}
```

Le quart de tour est un seul mouvement. La mission le lance une fois, puis se désactive pour ne pas le relancer au tour suivant. L'échéance armée par `drive()` arrête les roues et produit la ligne `done:true`, comme pour une commande tapée dans la console.

Remarque. Pendant le virage, la télémétrie indique déjà `"mission":0`. Pour garder `"mission":2` jusqu'à la fin, attendre que `motors.isMoving()` soit faux avant de remettre `_mission` à 0, avec un booléen qui retient que le virage a été lancé.

Erreur fréquente. Écrire `QUARTER_TURN_MS` dans `Config.h` avec `#define` alors qu'il est aussi déclaré dans `Robot.h` : le préprocesseur remplace le nom dans la déclaration et la compilation échoue. Le déclarer à un seul endroit.

Comparaison des deux versions

Question	`robot.js`	`Robot.h`
Où s'exécute le programme ?	Dans le navigateur du téléphone	Sur le Pico
Qui décide de lancer le retour ?	Le téléphone, après la ligne `done`	La mission, à l'échéance de l'aller
Arrêt entre l'aller et le retour ?	Oui, au moins un aller-retour réseau	Non
Modifier le programme demande…	Un rechargement de la page[1]	Une recompilation et un flash

[1] Contre `client/fakerobot.js`. Sur le vrai robot, la page est servie par le firmware : la modifier demande aussi une recompilation et un flash.

TD 2 : Corrigé

Rappel de la formule PID discrète

$$u[n] = K_p \cdot e[n] + K_i \cdot \sum_{k=0}^{n} e[k] + K_d \cdot (e[n] - e[n-1])$$

avec $e[n] = distance_{cible} - distance_{mesurée}$.

Remarque. Discrétisées avec une période Δt, l'intégrale devient $\sum e[k] \cdot \Delta t$ et la dérivée $\frac{e[n]-e[n-1]}{\Delta t}$. Si Δt est fixe, on pose $K_i = {K'}_i \cdot \Delta t$ et $K_d = {K'}_d / \Delta t$ (${K'}_i$, ${K'}_d$: gains du PID continu). À 50 Hz, ${K'}_i = 5$ et ${K'}_d = 0.004$ donnent $K_i = 0.1$ et $K_d = 0.2$. Changer la fréquence de la boucle oblige à recalculer K_i et K_d.

- K_p : gain proportionnel → réagit à l'erreur actuelle.
- K_i : gain intégral → corrige les erreurs accumulées dans le temps.
- K_d : gain dérivé → réagit à la vitesse de variation de l'erreur et amortit les changements rapides.

Exercice 1 : composante proportionnelle (P)

Distance cible = 10 cm, distance mesurée = 6 cm

1. **Que signifie $e[0]$? Calculez sa valeur.**

   $$e[0] = 10 - 6 = 4 \text{ cm}$$

   C'est l'erreur au premier pas de temps : le robot est 4 cm trop près du mur. Une erreur positive signifie « trop près ».
2. **Si $K_p = 0.5$, quelle est la correction appliquée ?**

   $$u[0] = K_p \cdot e[0] = 0.5 \cdot 4 = 2$$

   Le robot applique une correction de 2 unités de commande (par exemple, un écart de vitesse entre les roues) pour s'écarter du mur.
3. **Que se passe-t-il si $K_p = 2$?**

   $$u[0] = 2 \cdot 4 = 8$$

   La correction est quatre fois plus forte. Le robot réagit plus vite, mais risque de dépasser la cible (sur-correction) puis d'osciller.

Exercice 2 : composante intégrale (I)

$e[0] = 4, e[1] = 4, e[2] = 4$: l'erreur persiste.

1. **Commande totale après 3 mesures (avec $K_p = 0.5$ et $K_i = 0.1$) :**
   - Correction proportionnelle = $0.5 \cdot 4 = 2$
   - Correction intégrale = $K_i \cdot \sum e[k] = 0.1 \cdot (4 + 4 + 4) = 1.2$

- Commande totale $u[2] = 2 + 1.2 = 3.2$

Remarque. Le terme dérivé est nul ici : $e[2] - e[1] = 0$.

2. **Quelle est la part due à l'intégrale ?**

   1.2 sur 3.2, soit 37,5 % de la commande. Tant que l'erreur reste à 4, cette part augmente de 0.4 à chaque pas de temps.

3. **Que se passe-t-il si K_i est trop grand ?**

   - La somme accumulée pèse trop dans la commande → sur-correction.
   - L'intégrale ne diminue que lorsque l'erreur change de signe : le robot dépasse la cible et oscille, même quand l'erreur actuelle est faible.

Exercice 3 : composante dérivée (D)

$e[2] = 4$, $e[3] = 1$: l'erreur diminue rapidement, le robot s'éloigne du mur.

1. **Variation d'erreur :**

$$e_d = e[3] - e[2] = 1 - 4 = -3$$

2. **Correction dérivée :**

$$u_d = K_d \cdot e_d = 0.2 \cdot (-3) = -0.6$$

   Le terme dérivé est négatif : il **freine** la commande car l'erreur diminue rapidement.

3. **Commande totale $u[3]$ (avec $K_p = 0.5$, $K_i = 0.1$ et $K_d = 0.2$) :**

$$u[3] = 0.5 \cdot 1 + 0.1 \cdot (4 + 4 + 4 + 1) + 0.2 \cdot (-3) = 0.5 + 1.3 - 0.6 = 1.2$$

   Le robot **réduit** sa correction pour ne pas dépasser la consigne.

   Erreur fréquente. Multiplier deux fois par K_d : $0.2 \cdot (-0.6)$. La correction dérivée -0.6 contient déjà le gain.

Exercice 4 : pertinence des composantes

1. **Analyse des comportements :**

Situation	Interprétation	Réglage probable
A : oscille autour de la cible	K_p trop grand, K_i trop grand ou K_d trop faible	Diminuer K_p ou K_i, augmenter K_d
B : corrige trop lentement	K_p trop faible, ou K_d trop grand (amortissement excessif)	Augmenter K_p, diminuer K_d
C : dépasse systématiquement	K_i trop grand (somme accumulée pendant l'approche), K_p trop grand ou K_d trop faible	Diminuer K_i, augmenter K_d

2. **Comment corriger ?** Régler les gains un par un, sur le robot :

1. Mettre $K_i = 0$ et $K_d = 0$.
2. Augmenter K_p jusqu'à ce que le robot oscille autour de la cible, puis le diviser par deux.
3. Augmenter K_d jusqu'à ce que les oscillations restantes s'amortissent.
4. Si une erreur persiste à l'équilibre, ajouter un petit K_i.
5. Borner la somme $\sum e[k]$ (*anti-windup*) pour qu'elle ne s'accumule pas quand le robot est bloqué ou loin de la cible.

Attention. Ne modifier qu'un gain à la fois, sinon on ne sait pas lequel a produit l'effet observé.

TD 3 : Corrigé

Exercice 1 : temps de calcul et fréquence de lecture

1. **Fréquence maximale de la boucle** :
   - Le capteur mesure pendant que le processeur lit et calcule : la boucle attend seulement la mesure suivante.
   - Période de la boucle = 20 ms, soit **50 Hz**.
   - Élément limitant : le capteur.

   Erreur fréquente. Additionner 20 ms + 0,5 ms + 0,02 ms. Ce calcul vaut pour un capteur qui attend la fin de la lecture avant de lancer la mesure suivante (mode « une seule mesure ») : on obtiendrait alors $\frac{1}{0.02052} \approx 48.7$ Hz.
2. **Part du temps processeur** :
   - Lecture + calcul = 0,5 + 0,02 = 0,52 ms toutes les 20 ms
   - $\frac{0.52}{20} = 2.6\,\%$
   - Le reste sert notamment à la pile Wi-Fi (pilote CYW43439 et TCP/IP), au serveur HTTP, à la télémétrie, à l'afficheur et au télémètre à ultrasons.
3. **Boucle à 100 Hz** :
   - Le VL53L1X ne produit que 50 mesures par seconde : la boucle lit chaque mesure deux fois.
   - Une fois sur deux, $e[n] - e[n-1] = 0$; la fois suivante, toute la variation arrive d'un coup. Le terme dérivé alterne entre 0 et le double de sa valeur.
   - Le terme intégral compte chaque mesure deux fois.
   - Solution : ne calculer le PID que lorsqu'une nouvelle mesure est disponible.
4. **Fréquence raisonnable à 0.5 m/s** :
   - À 50 Hz, le robot avance de $\frac{0.5}{50} = 1$ cm entre deux mesures ; à 25 Hz, de 2 cm.
   - Le coût en calcul étant faible (question 2), **50 Hz** est adapté. 25 Hz reste acceptable si l'on accepte 2 cm entre deux mesures.

Exercice 2 : occupation mémoire

1. **Mémoire utilisée pour 100 erreurs** :
   - 100×4 octets = **400 octets**
   - SRAM = 264 kio = 270 336 octets, soit $\frac{400}{270336} \approx 0.15\,\%$
   - Sur le Pico WH, cet historique n'est pas une contrainte.
2. **Ce que le PID doit conserver** :
   - La somme des erreurs $S[n] = S[n-1] + e[n]$, mise à jour à chaque pas
   - L'erreur précédente $e[n-1]$
   - Soit 2 `float` = **8 octets**. L'historique complet n'est nécessaire que pour une intégrale sur fenêtre glissante ou un filtre.
3. **Historique de télémétrie** :
   - Un échantillon : $4 + 4 \times 2 = 12$ octets (pas d'octets de remplissage : le `uint32_t` est en tête)
   - À 50 Hz : $12 \times 50 = 600$ octets/s
   - Tampon de 32 kio : $\frac{32768}{600} \approx 54.6$ s, soit **environ 54 s**

4. **Réduire l'empreinte** :
   - Tampon circulaire : on garde les N dernières secondes et on écrase les plus anciennes.
   - Sous-échantillonner l'historique à 10 Hz : 5 fois moins de mémoire, le PID continue à 50 Hz.
   - Stocker un écart de temps sur un `uint16_t` au lieu de `ms` sur un `uint32_t` : 10 octets au lieu de 12.
   - Ne rien stocker sur le robot : le téléphone reçoit déjà la télémétrie et garde l'historique, où la mémoire ne manque pas.

Exercice 3 : énergie et efficacité

1. **Autonomie des moteurs** :
   - Sans oscillations : 1000 mAh / 100 mA = **10 heures**
   - Avec oscillations : 1000 mAh / 150 mA ≈ **6,67 heures** (6 h 40 min)
2. **Gain en autonomie** :
   - 10 h − 6,67 h = **3,33 heures** (3 h 20 min)
   - Soit 50 % d'autonomie en plus ($\frac{10}{6.67} = 1.5$)
3. **Stratégie PID pour réduire les oscillations** :
   - Diminuer K_p s'il est trop agressif
   - Augmenter K_d pour amortir les variations
   - Réduire K_i et borner la somme si elle s'accumule trop

Exercice 4 : précision et bruit de mesure

1. **Erreur moyenne** :
   - Moyenne = (198 + 202 + 195 + 205 + 200) / 5 = **200 mm**
   - Erreur moyenne (biais) = 200 − 200 = **0 mm**
   - Erreur absolue moyenne = (2 + 2 + 5 + 5 + 0) / 5 = **2,8 mm**

   Écart-type :
   - Écarts à la moyenne : −2, +2, −5, +5, 0 ; carrés : 4, 4, 25, 25, 0 ; somme = 58
   - Écart-type de la population : $\sqrt{58/5} = \sqrt{11.6} \approx$ **3,4 mm**
   - Écart-type de l'échantillon : $\sqrt{58/4} = \sqrt{14.5} \approx$ **3,8 mm**
2. **Influence du bruit sur le terme dérivé** :
   - Erreurs $e[n] = 200 - mesure$: 2, −2, 5, −5, 0
   - Variations $e[n] - e[n-1]$: −4, +7, −10, +5 mm
   - Avec $K_d = 0.2$: termes dérivés −0,8, +1,4, −2,0, +1,0, alors que le robot ne bouge pas.
   - À 50 Hz, une variation de 10 mm en 20 ms correspond à une vitesse apparente de 0,5 m/s, la vitesse du robot dans les exercices 1 et 5.
   - La différence de deux mesures bruitées a un écart-type $\sqrt{2}$ fois plus grand que celui d'une mesure. Le terme dérivé amplifie le bruit : commandes saccadées, oscillations.
3. **Méthodes de filtrage** :
   - Moyenne glissante sur N mesures : retard de $\frac{N-1}{2}$ périodes, soit 40 ms pour $N = 5$ à 50 Hz.
   - Médiane sur 3 mesures : élimine une valeur aberrante isolée, retard d'une période (20 ms).

- Filtre exponentiel : $y[n] = \alpha \cdot x[n] + (1 - \alpha) \cdot y[n-1]$; plus α est petit, plus le filtre lisse et plus il retarde.
- Ne filtrer que le terme dérivé, pour garder la réactivité des termes P et I.

Attention. Le retard du filtre s'ajoute à la latence de la boucle (exercice 5).

Exercice 5 : latence dans une architecture distribuée
La chaîne entre la mesure et la commande :

- La mesure attend la ligne de télémétrie suivante : de 0 à 20 ms (la ligne porte la dernière mesure, prise au plus 20 ms plus tôt).
- La ligne attend la prochaine requête `GET /api/lines` du téléphone : de 0 à 100 ms.
- La réponse traverse le Wi-Fi jusqu'au téléphone.
- Le téléphone calcule le PID : 1 ms.
- Le `POST /api/command` traverse le Wi-Fi jusqu'au robot.

1. **Latence, canal libre** :
   - Meilleur cas : 0 + 0 + 5 + 1 + 5 = **11 ms**
   - Pire cas : 20 + 100 + 5 + 1 + 5 = **131 ms**
   - À comparer avec le PID local : au plus 20 ms + 0,52 ms.
2. **Latence, canal encombré** :
   - Meilleur cas : 0 + 0 + 50 + 1 + 50 = **101 ms**
   - Pire cas : 20 + 100 + 50 + 1 + 50 = **221 ms**
3. **Distance parcourue à 0.5 m/s** (0,5 mm par milliseconde) :

Cas	Latence	Distance
Canal libre, meilleur	11 ms	5,5 mm
Canal libre, pire	131 ms	6,6 cm
Canal encombré, meilleur	101 ms	5,1 cm
Canal encombré, pire	221 ms	11,1 cm
PID local, pire	20,5 ms	1,0 cm

4. **Est-ce acceptable pour des obstacles à 20 cm ?**
   - Dans le pire cas, le robot parcourt plus de la moitié des 20 cm avant que la commande n'arrive, et il lui faut encore une distance de freinage.
   - La latence varie d'un facteur 20 entre le meilleur et le pire cas : le réglage du PID ne peut pas en tenir compte.
   - Acceptable pour des corrections lentes de trajectoire. Inacceptable pour l'arrêt devant un obstacle.
5. **Canaux Wi-Fi 2,4 GHz** :
   - **3 canaux** de 20 MHz sans chevauchement : 1, 6 et 11. En Europe, où les canaux 1 à 13 sont autorisés, on peut en placer 4 : 1, 5, 9 et 13.
   - Une douzaine de robots se partagent donc 3 ou 4 canaux, avec le Wi-Fi de l'établissement, le Bluetooth et les autres équipements de la bande.

- Sur un même canal, une seule station émet à la fois : chacune attend que le canal soit libre (CSMA/CA), et les collisions provoquent des retransmissions. La latence augmente et devient irrégulière.

6. **Stratégies pour réduire l'impact de la latence** :
   - **Calcul local** : faire tourner le PID sur le Pico à 50 Hz ; il n'occupe que 2,6 % du temps processeur (exercice 1).
   - **Hybridation** : le téléphone envoie des consignes lentes (distance au mur, vitesse, mission) ; le robot exécute le PID et l'arrêt devant un obstacle. C'est le rôle de la mission `stopAtObstacle()` du TP 1.
   - **Prédiction** : le champ `ms` date chaque ligne sur l'horloge du robot. Le téléphone estime l'âge de la mesure et extrapole la distance : $d_{prédite} = d_{mesurée} - v \cdot$ âge.
   - **Sécurité** : chaque commande porte une `duration`. Si le téléphone n'envoie plus rien, le robot s'arrête de lui-même à l'échéance (`enforceSafety()`).
   - **Réseau** : choisir un canal peu occupé pour le point d'accès du robot ; réduire la période de scrutation, au prix de plus de trafic sur le canal.