



MICRO-315

**Systèmes embarqués et robotique**

Section de microtechnique, Bachelor BA6  
Printemps 2025

## Rapport mini projet

Louis Grange	341237
Arthur Lassagne	341489

Prof. Francesco Mondada / Dr. Frank Bonnet

## Table des matières

1	Introduction .....	3
2	Méthode de travail .....	3
2.1	Outils .....	3
2.2	Organisation du groupe .....	3
3	Conception et analyse .....	4
3.1	Initialisation du système .....	4
3.2	Gestion du déplacement .....	4
3.3	Détection d'obstacles .....	6
3.4	Communication entre les threads .....	7
3.5	Problèmes rencontrés .....	7
3.5.1	Déplacement saccadé .....	7
3.5.2	Avance après rotation .....	8
4	Résultats obtenus .....	8
4.1	Rétrospective .....	8
4.2	Améliorations possibles .....	9
4.2.1	Gestion de la marche arrière .....	9
4.2.2	Arrêt en pente .....	9
5	Conclusion .....	9

## 1 Introduction

Le but du mini projet est de partir sur la base des éléments que nous avons vus lors des travaux pratiques pour créer plusieurs tâches complexes à résoudre par le robot e-puck2.

La mission que nous avons donnée à notre robot est de trouver le point le plus haut de la surface sur laquelle il se déplace, s'y rendre et de rebondir en cas de collision avec un obstacle. Pour ce faire, nous avons développé une librairie de fichiers .h et .c utilisant les éléments suivants du robot : **les deux moteurs pas à pas, les capteurs de proximité infrarouges et l'IMU**.

## 2 Méthode de travail

Cette section a pour but de présenter les outils utilisés, les méthodologies de travail adoptées et l'approche de développement employée qui nous ont permis de mener à bien ce projet.

### 2.1 Outils

Les fichiers .c et .h ont été écrits sur l'environnement de développement *Visual Studio Code* fourni par l'équipe enseignante.

L'utilisation du système de gestion de version *Git* nous a permis de travailler simultanément sur des fonctionnalités différentes tout en préservant l'intégrité de notre code. De plus, le suivi des modifications nous a été facilité par l'extension *Git Graph*.

La librairie *e-puck2\_main-processor* comme base de code ainsi que l'extension *Serial Monitor* pour le debug furent d'une grande aide également.

### 2.2 Organisation du groupe

Nous avons commencé par planifier les fonctionnalités souhaitées, anticiper les potentielles difficultés que nous allions rencontrer et d'ores et déjà imaginer les solutions que nous pourrions apporter à ces problèmes.

Ensuite, nous avons parcouru la librairie mise à disposition afin d'identifier les fonctions que nous pouvions utiliser d'emblée et celles que nous devions concevoir nous-mêmes.

Pour ce qui est de la répartition du travail, nous avons scindé les tâches à réaliser en deux catégories : d'un côté, le déplacement du robot avec la gestion de l'accéléromètre et des moteurs, de l'autre, la détection d'obstacles et l'action de rebondir (cf. Chapitre 3).

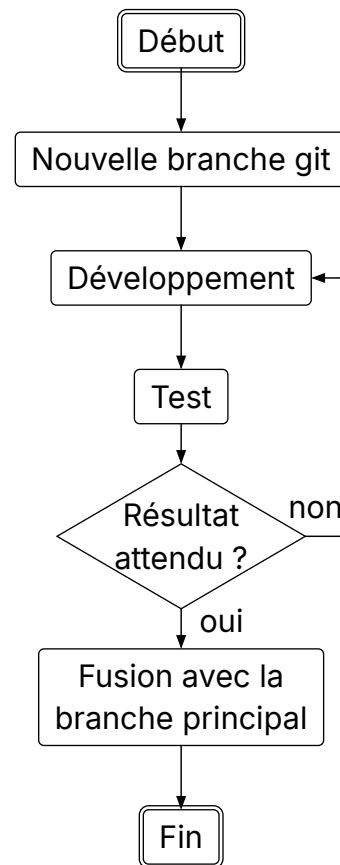


Fig. 1 – Processus de développement d'une nouvelle fonctionnalité.

Comme le montre Fig. 2, nous avons donc travaillé sur plusieurs branches Git distinctes ; l'objectif étant de garder la branche principale `Miniprojet` « propre » et fonctionnelle (cf. Fig. 1). Afin de faciliter le travail de l'autre, nous avons chacun développé une interface `.h` permettant à l'autre d'accéder facilement aux fonctionnalités mises en place.



Fig. 2 – Graphe Git représentant le développement en différentes branches.

A posteriori, nous nous sommes rendus compte du rôle majeur de la communication durant ce projet. C'est cette dernière qui nous a permis de résoudre, à deux, les problèmes rencontrés (cf. Chapitre 3.5) et proposer des solutions innovantes et efficaces.

### 3 Conception et analyse

Notre miniprojet repose sur un RTOS (*Real-Time Operating System*), ChibiOS dans notre cas, utilisant des threads pour gérer parallèlement les différentes tâches à effectuer. Nous avons donc organisé la librairie en trois modules distincts : `main`, `travel` et `detection`.

Tandis que le premier module sert à l'initialisation du système, les deux derniers servent à la gestion de deux threads `WallDetection` et `TravelThread`, qui s'occupent respectivement de détecter les obstacles (cf. Chapitre 3.3) et de gérer le déplacement de l'e-puck2 (cf. Chapitre 3.2).

#### 3.1 Initialisation du système

Le module `main` est utilisé pour initialiser le système ; c'est le point de départ, là où nous appelons les fonctions d'initialisation telles que `chSysInit()`, `mpu_init()` et `motors_init()`. C'est également dans ce module que nous appelons les fonctions de démarrage de nos threads : `travel_thread_start()` et `wall_detection_start()`.

#### 3.2 Gestion du déplacement

Le thread `TravelThread` dicte les trois états possibles du robot (*rebond*, *de-mi-tour* ou *libre* – cf. Fig. 4) et fait notamment appel à l'interface `motors.h` pour actionner les moteurs pas à pas.

Par défaut, l'e-puck2 est en déplacement *libre*, c'est-à-dire que son déplacement est régi par les valeurs de l'accéléromètre. La séquence menant à l'actionnement des moteurs est la suivante :



Fig. 3 – Représentation des axes située sur le dessus de l'e-puck2 (avant du robot vers le haut).

Le thread lit et moyenne les valeurs d'accélération selon l'axe y (cf. Fig. 3). La valeur obtenue est transmise à un régulateur PI qui détermine la vitesse d'avance à adopter. Ensuite, les valeurs d'accélération selon l'axe x sont lues puis comparées à une valeur de seuil ROTATION\_TILT\_THRESHOLD. En cas de dépassement, un facteur de correction de vitesse est calculé puis appliqué aux moteurs de façon à désynchroniser leur vitesse respective, induisant une rotation (cf. Extrait de code 1).

```
speed_correction = ROTATION_TILT_COEFF * (current_tilt_x - GOAL_TILT_X);

// void set_speed(int16_t left_motor_speed, int16_t right_motor_speed);
set_speed(speed + speed_correction, speed - speed_correction);
```

Extrait de code 1 – Calcul du facteur de correction de vitesse et réglage des moteurs.

Cette approche permet la rotation et l'avance en simultané, comme indiqué dans la Fig. 4, garantissant ainsi une certaine fluidité durant le déplacement.

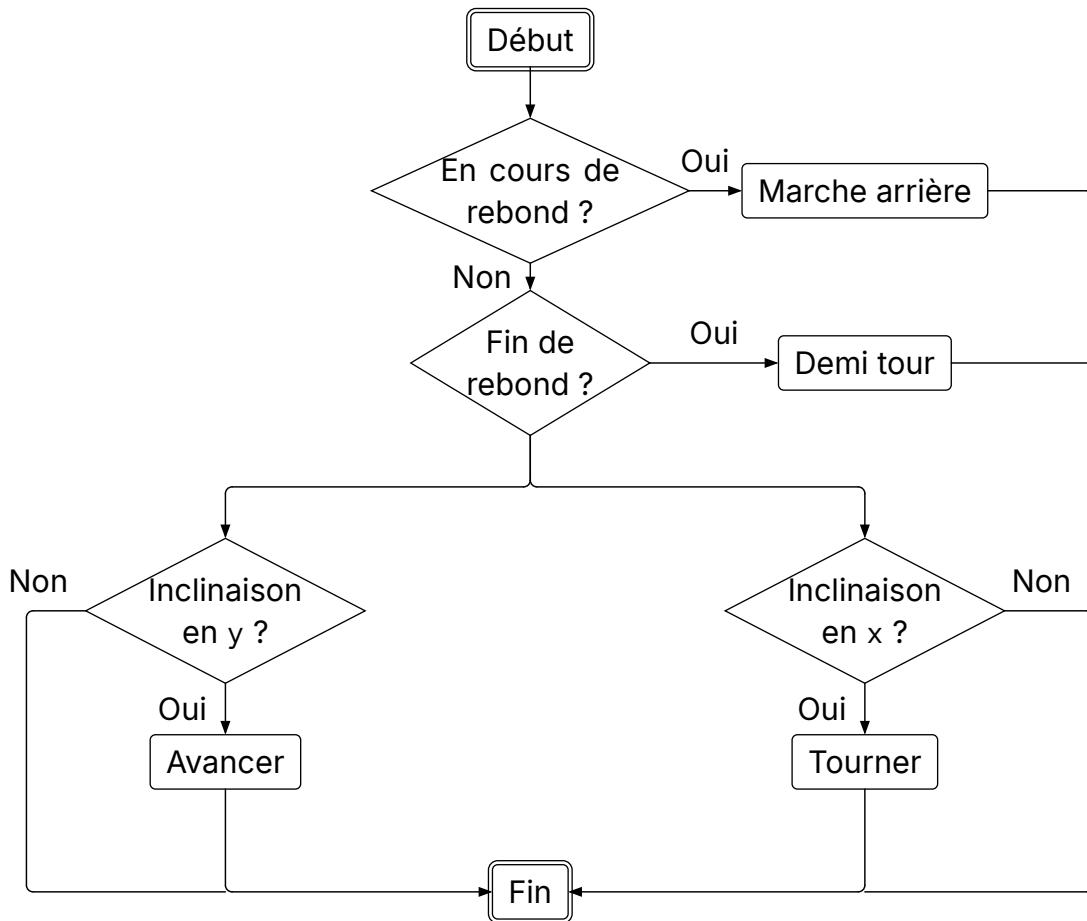


Fig. 4 – Séquence de décisions pour le déplacement gérée par TravelThread.

Lorsque le thread WallDetection communique un état de *rebond* ou de *demi-tour* à TravelThread (cf. Chapitre 3.4), ce dernier actionne alors les moteurs en conséquence.

```

if (in_bounce_procedure() && !achieved_bounce_distance()) {
    set_speed(-800, -800);
} else if (in_turn_procedure() && !achieved_turn_distance()) {
    set_speed(800, -800);
}

```

Extrait de code 2 – Gestion des états *rebond* et *demi-tour*.

### 3.3 Détection d'obstacles

Le thread WallDetection gère la détection d'un mur et le rebondissement une fois que le robot atteint ce dernier. Il se base sur les interfaces sensors/proximity.h et motors.h ainsi que les données fournies par les capteurs infrarouges.

La détection d'un mur est faite grâce à la fonction `is_against_wall()` qui lit les valeurs des capteurs IR1 et IR8 et les compare à une valeur seuil BOUNCING\_DISTANCE.

Dans le cas où un obstacle est détecté et que le robot est dans un état *libre*, le thread enclenche alors l'état *rebond*. L'e-puck2 doit alors reculer sur une certaine distance, avancer de nouveau jusqu'au mur (ceci constitue un rebond) et répéter cette séquence jusqu'à ce que le nombre de rebonds soit supérieur à NBOUNCES.

Pour agrémenter notre rebond de plus de réalisme, nous avons décidé d'implémenter une distance de rebond à décroissance exponentielle, en suivant la formule ci-dessous. Nous avons tenté différentes valeurs de BOUNCING\_DAMPING\_FACTOR, et le plus réaliste semblait être 0.4.

```
NSTEP_BOUNCING * exp(- BOUNCING_DAMPING_FACTOR * bouncing_counter);
```

Extrait de code 3 – Formule de calcul pour la distance de rebond.

Une fois la procédure de rebond terminée, WallDetection enclenche le mode *demi-tour*, ce qui permettra au robot de se retrouver dos à l'obstacle, prêt à repartir.



Fig. 5 – Vue de l'e-puck2 dans son arène, adossé au mur.

### 3.4 Communication entre les threads

Le partage d'informations entre les threads et la clé de cette architecture. Cette communication entre WallDetection et TravelThread s'effectue grâce à une structure static (cf. Extrait de code 4) dont les variables sont accessibles via des getters/setters.

```
struct State {
    bool bouncing_enabled;
    bool turning_enabled;
    int bouncing_counter;
};
```

Extrait de code 4 – Structure permettant la communication entre les threads.

Voici un exemple pour illustrer ce système : lorsque le robot entre en contact avec un mur, le thread WallDetection change `bouncing_enabled` à 1, c'est-à-dire que le robot passe en mode *rebond*. TravelThread perçoit ce changement d'état grâce au getter `in_bouncing_procedure()` (cf. Extrait de code 2) et décidera donc de ne pas avancer mais de reculer.

Voici un autre exemple : lorsque l'e-puck2 est en mode *rebond*, WallDetection vérifie la distance de recul parcourue après le contact avec le mur. Pour ce faire, elle réinitialise la position des moteurs à 0 lors du contact avec l'obstacle. Puis, lorsque le robot a effectué une distance de recul suffisante, c'est-à-dire lorsque `achieved_bounce_distance()` retourne 1, TravelThread reprend l'avance (cf. Extrait de code 2).

### 3.5 Problèmes rencontrés

Durant la conception de notre librairie, nous avons rencontré différents problèmes que nous abordons ici avec les diverses solutions imaginées.

#### 3.5.1 Déplacement saccadé

Initialement, le déplacement en marche avant ne relevait que d'une simple comparaison entre l'accélération perçue selon l'axe y et une valeur seuil empirique. Si l'accélération était supérieure au seuil, alors le robot avance, sinon le robot s'arrête. Cette variation abrupte de vitesse entre l'arrêt et l'avance générait elle-même une accélération qui se trouvait être plus élevée que notre valeur de seuil. Cette accélération parasite provoquait alors une sorte d'à-coups en chaîne qui rendait le déplacement impossible.

La première tentative de résolution fut alors d'ajuster la valeur de seuil empirique. Malheureusement, cette solution ne fut pas d'une grande aide puisqu'elle ne faisait que reporter le problème à des valeurs de pente plus importantes.

Ensuite, de part la morphologie de l'epuck-2, nous nous sommes dit que ces à-coups devaient générer une accélération angulaire qui pourraient être détectés par le gyroscope embarqué. L'idée était alors d'empêcher le changement d'état des moteurs lorsque cette accélération angulaire était mesurée sur l'axe x. Cette solution n'a pas apporté d'amélioration. À ce stade, nous n'avions pas clairement identifié la raison de cet échec. Ce n'est que par la suite que

nous avons réalisé que les valeurs du gyroscope présentaient le même problème que celles de l'accéléromètre : « l'instabilité ».

Après ces deux tentatives infructueuses, nous nous sommes souvenus du TP4 et de l'utilisation d'un régulateur PI. Nous avons donc revu notre cours d'Automatique et Commandes Numériques et implémenté notre `pi_regulator()`. Cette solution a apporté une amélioration visible de l'équilibre, mais de temps à autre, l'e-puck2 rentrait tout de même dans une boucle d'à-coups.

C'est à ce moment, grâce au *Serial Monitor*, que nous avons remarqué ce problème « d'instabilité » des mesures : à inclinaison constante, l'accéléromètre nous renvoyait des valeurs plus ou moins constantes, mais de temps en temps une valeur extrême était lue. C'était cette valeur aberrante qui provoquait l'arrêt soudain des moteurs et qui par une réaction en chaîne provoquait ces à-coups. Pour y remédier, nous avons écrit la fonction `get_imu_average()`<sup>1</sup> qui moyenne les valeurs renvoyées par l'accéléromètre avant de transmettre son résultat à notre régulateur.

Grâce à la combinaison du régulateur PI et de l'utilisation d'une moyenne pour les valeurs de l'IMU, nous avons résolu notre problème de déplacement saccadé.

### 3.5.2 Avance après rotation

Après avoir effectué ses NBOUNCES rebonds, l'e-puck2 passe en mode *demi-tour* et effectue un tour de 180 degrés sur lui-même pour se préparer au départ. Après cette rotation, le robot avançait d'environ un centimètre. Ce comportement que nous souhaitions éviter était du à la moyenne des mesures de l'accélération effectuée par `get_imu_average()`. Par définition, une moyenne mobile comporte une forme d'inertie. Ainsi, lorsque le robot passait du mode *demi-tour* au mode *libre*, la moyenne des dernières valeurs renvoyées par l'accéléromètre était supérieure à notre valeur seuil.

Pour pallier ce comportement, nous avons développé la fonction `reset_imu_averaging()` pour réinitialiser les valeurs d'accélérations à la fin de l'état *demi-tour*.

## 4 Résultats obtenus

### 4.1 Rétrospective

Nous considérons la mission donnée à notre robot comme accomplie : notre e-puck2 parvient à trouver le point le plus élevé d'une surface et à s'y diriger tout en rebondissant en cas de collision.

De plus, nous avons respecté le cahier des charges en utilisant : les deux moteurs pas à pas, les capteurs de proximité infrarouges et l'IMU ; chaque capteur/actionneur est utilisé et géré dans son propre thread ; les conventions du langage C ont été respectées.

---

<sup>1</sup>Nous avons connaissance de la fonction `get_acc_filtered()` qui figure dans `imu.h`. Cependant, pour des raisons propres à notre architecture, il nous était préférable d'écrire une fonction à part entière (cf. Chapitre 3.5.2).

Enfin, nous avons pris soin de structurer notre code de façon claire et efficace en apportant une attention toute particulière à la gestion des ressources (notamment le type des variables utilisées).

## 4.2 Améliorations possibles

Malgré nos efforts, la librairie développée comporte certains points sujets à améliorations que nous avons choisis de ne pas effectuer par soucis de temps. Cela étant dit, nous les avons rassemblés ci-dessous.

### 4.2.1 Gestion de la marche arrière

Actuellement, la marche arrière est gérée par l'algorithme de `travel.c`, avec pour but d'aligner le robot face à la pente. Si le robot est placé parfaitement dos à la pente, celui-ci se dirigera en marche arrière vers celle-ci et n'effectuera pas de rotation, ni de rebond.

Ce comportement, bien que très spécifique, pourrait être rectifié de la façon suivante : premièrement, constater que seule la composante  $y$  du gradient est non-nulle et de signe positif. Ensuite, actionner les moteurs afin d'effectuer une rotation du robot sur lui-même de 180 degrés.

### 4.2.2 Arrêt en pente

Lorsque l'inclinaison de la surface sur laquelle se trouve l'e-puck2 dépasse sa capacité à tenir grâce aux forces de frottement, le robot dérape et glisse.

Il n'est pas impossible que l'arrêt des moteurs en cas d'inclinaison trop élevée résolve ce problème. En effet, l'e-puck2 profiterait du coefficient de frottement statique, plus élevé que le dynamique ( $\mu_s > \mu_d$ ), et s'accrocherait à la surface au lieu de déraper.

## 5 Conclusion

Comme tout projet, nous avons vécu des moments de doute, de frustration et de colère, mais nous avons aussi passé des moments de joie, de rire et de satisfaction. Nous sommes très fiers d'avoir pu mettre en pratique les concepts et principes vus au cours *MICRO-315 Systèmes embarqués et robotique* et remercions l'équipe enseignante d'avoir mis en place les travaux pratiques qui nous ont permis de réaliser ce projet ; il est toujours agréable de pouvoir mettre en application les concepts théoriques enseignés. Nous les remercions également pour l'écoute dont ils ont fait preuve (nous pensons notamment aux vidéos tutorielles Git et au superbe système de demande d'assistant) et à leur réactivité concernant nos remarques.