ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

MINI PROJET DE SYSTÈMES EMBARQUÉS ET ROBOTIQUE: ROBOT RÉCOLTEUR DE COMMANDE

MICRO 315: SYSTÈMES EMBARQUÉS ET ROBOTIQUE

Oussama Gabouj
Adam Ben SLama

1st June 2025

Contents

1 REMERCIEMENTS

Nous tenions à remercier Monsieur Francesco Mondada, Monsieur Daniel Burnier ainsi que les assistants-étudiants, pour leurs expertises, conseils et aides tout au long de ce projet. Nous tenions aussi à remercier l'EPFL ainsi que GCtronic pour le matériel qu'ils nous ont offert pour mener à bien ce projet.

2 INTRODUCTION

2.1 CONTEXTE

Dans le cadre du cours de "Systèmes embarqués et robotique", nous avons passé six semaines à étudier la robotique. A cela s'ajoute cinq travaux pratiques pour se familiariser avec le robot e-puck. Il était alors nécessaire de conclure ce semestre en mettant en pratique nos connaissances et notre imagination. C'est pourquoi, nous vous présentons notre projet: Un robot qui récolte les commandes des clients d'un restaurant.

2.2 MOTIVATION

Notre motivation principale était de trouver une application pouvant mettre le robot au service de quelqu'un. De là, nous est venue l'idée de faire un robot qui prendrait les commandes des clients d'un restaurant.

2.3 OBJECTIF

Les conditions de ce projet étaient l'utilisation des deux moteurs pas-à-pas, directement lié aux roues, d'un des capteurs de distance (Time-of-flight ou infrarouge) et d'un des trois capteurs restants (Caméra, micro ou IMU). L'objectif de ce mini projet est de combiner tous ces périphériques pour en faire un robot fonctionnel pouvant exécuter des tâches assignées.

3 DESCRIPTION GÉNÉRALE DE L'APPLICATION

L'application consiste en un robot qui suit une ligne noire grâce à un régulateur PID et envoie par bluetooth des informations à un ordinateur qui affiche, grâce à une interface graphique dédiée, le chemin du robot et d'autres informations sur le robot en temps réel. Si le robot détecte un obstacle à gauche, telle qu'une table, il tourne à droite de 90° pour récupérer la commande via une ligne au sol, faisant office de code barre. Une fois que le robot a terminé son premier tour, il passe en mode "WORK" et suit le parcours pour récupérer d'autres commandes sans retracer le chemin. Lorsque l'utilisateur ferme l'interface graphique, toutes les données récupérées du robot sont stockées dans des fichiers *csv* afin de pouvoir l'utiliser dans des versions ultérieures pour analyser les commandes, prédire les revenus, améliorer la productivité. Ces analyses peuvent aider à optimiser les opérations et à prendre des décisions plus éclairées en matière de menu, de marketing et de gestion de stocks. Cette application offre une solution innovante pour les restaurants qui cherchent à améliorer leur processus de commande et de livraison.

4 MODE D'EMPLOI

Grâce à l'interface graphique (Figure 1), l'utilisateur peut contrôler le robot et le suivre en temps réel.

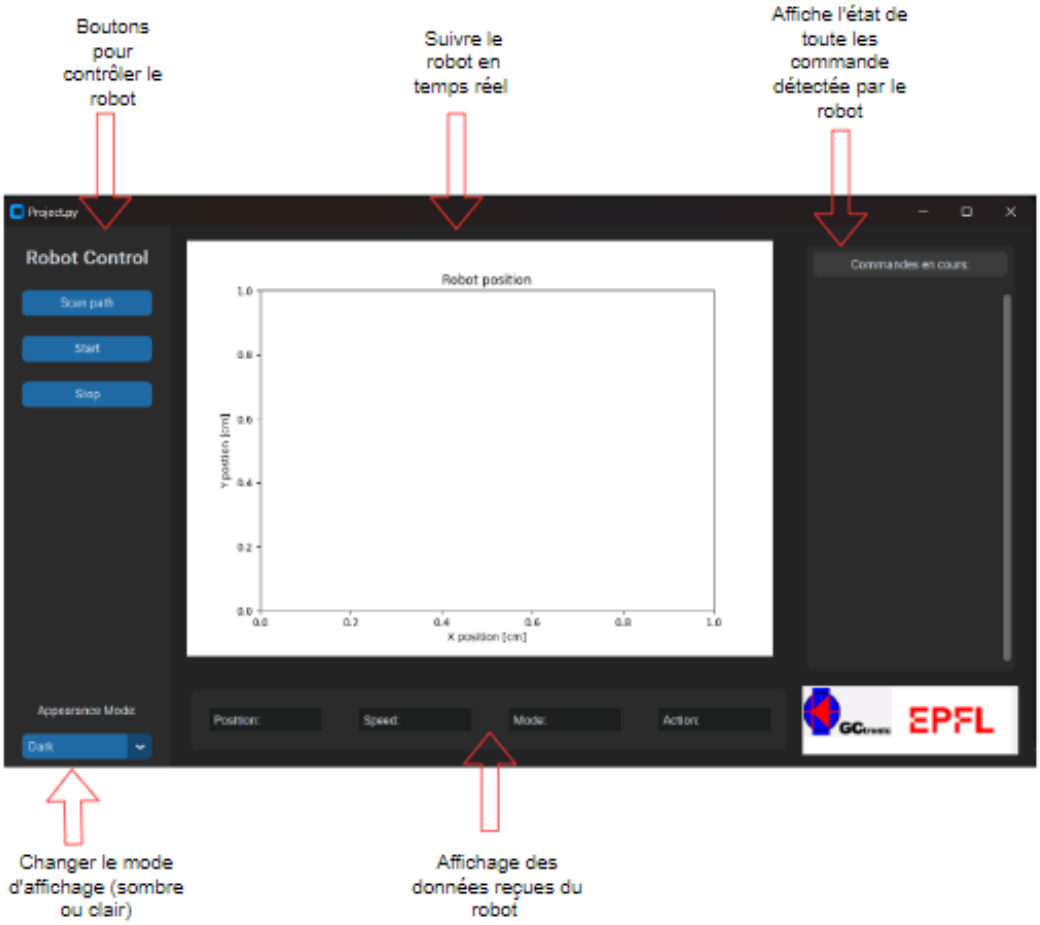

FIGURE 1
Interface graphique python

Le robot a 3 modes :

- **Scan Mode:** Ce mode sert à cartographier le restaurant. Le robot suit la ligne noire et envoie les informations à l'ordinateur qui s'occupe de la reconstruction de chemin parcouru par le robot. Si le robot détecte un obstacle à gauche, tel qu'une table, il tourne à droite pour récupérer la commande.
- **Work Mode:** Le robot suit le même chemin, sans le réimprimer, avec les mêmes fonctionnalités.
- **Stop Mode:** Arrêt d'urgence du robot assuré par le bouton stop dans l'interface graphique ci dessus

La barre en bas de l'interface graphique affiche toutes les données envoyées par le robot:

- **Position:** vecteur indiquant la position du robot en cm selon un repère orthonormé, x et y.
- **Speed:** vitesse moyenne du robot en cm/s
- **Command:** la commande que le robot vient de scanner
- **Action:** la tâche actuelle du robot

La barre à droite (command en cours) indique l'état de chaque commande scannée par le robot. Dès que le robot scanne une nouvelle commande, celle-ci s'affiche dans la progress bar qui s'auto incrémente à la fréquence d'affichage de la GUI. La commande scannée (Burger / pizza / pasta) dépend de la largeur de la ligne noire, au sol, détectée par le robot. Lorsque l'utilisateur ferme l'interface graphique le robot

s'arrête et l'ordinateur génère 3 fichiers *csv* qui continent toutes les informations récoltées par le robot pour des éventuelles analyses plus approfondies de l'environnement scanné ou les performances du robot: *command.csv* qui contient toutes les commandes scannées par le robot, *path_ data.csv*, contenant les coordonnées de tous les points qui constitues le chemin parcourus par le robot et *tables_ position_ data.csv* où sont stockées les positions de toutes les tables détectées par le robot.

5 DESCRIPTION TECHNIQUE DE L'APPLICATION ET DU MATÉRIEL

5.1 ÉQUIPEMENT

En plus des différents périphériques utilisés, pour le bon fonctionnement du projet, nous avons dû développer une solution qui nous permettrait d'utiliser la caméra pour capturer directement des images du sol devant l'e-puck2. C'est pourquoi nous avons imprimé en 3D un support sur lequel nous pouvions coller un miroir à 45° pour refléter l'image.

FIGURE 2
Photo du robot, du support et du miroir

5.2 LISTE DES PÉRIPHÉRIQUES

Comme susmentionné dans l'introduction, plusieurs périphériques, dont la liste est ci-dessous, ont été utilisés pour mener à bien ce projet.

Nom du périphérique	Modèle	Input/Output
Caméra	Omnivision OV7670 CMOS image sensor	Input
Capteur de proximité IR	TSOP36230	Input
Module de communication sans fil	Espressif ESP32	Input et Output
Moteur pas-à-pas	-	Output

TABLE 1
Liste des périphériques utilisés

5.3 APPLICATION TECHNIQUE DES PÉRIPHÉRIQUES

- **La caméra:** Elle fait office de détecteur de chemin et de scanner "code barre" en récupérant l'intensité rouge de chaque pixel d'une ligne horizontale complète afin d'analysant la position de la ligne noire et sa largeur.
- **Capteurs de distance IR:** Seulement le IR5 est utilisé pour détecter un objet à gauche du robot lorsque celui-ci se trouve à une distance inférieure à une certaine limite.

- **Module Espressif ESP32:** Il permet de lui envoyer des informations, via l'interface python, mais également d'en recevoir.
- **Moteurs:** chacun d'eux est relié à une roue et permettent au robot de se déplacer.

6 FONCTIONNEMENT DU PROGRAMME

6.1 PRÉSENTATION GÉNÉRALE DE L'ARCHITECTURE DU PROJET

La figure ci-dessous illustre l'architecture du projet: hiérarchie des modules et la définition des threads assurant le fonctionnent du robot.

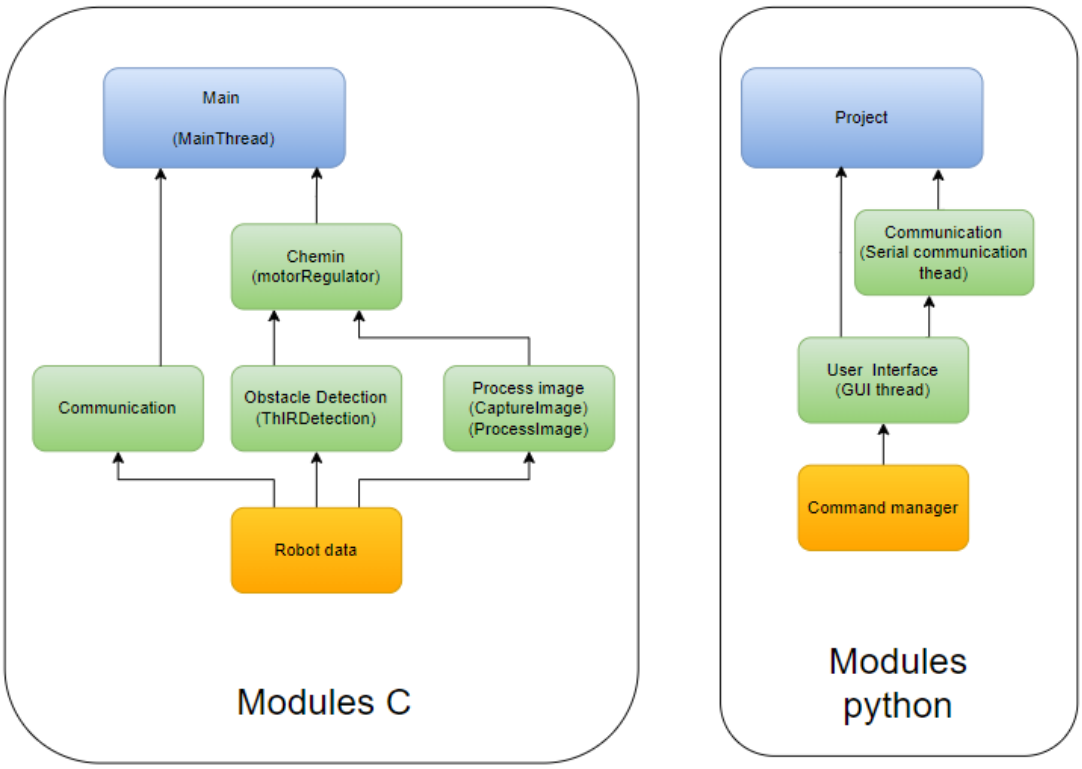

FIGURE 3
Dépendance des différents modules

6.2 PARALLÉLISME ET CONCURRENCE

Dans ce projet, nous avons eu recours à la programmation concurrente et parallèle afin de maximiser la gestion des ressources matérielle. Comme le montre la Figure 3, nous utilisons la concurrence au niveau du robot (réalisée à l'aide des threads dont chibiOs assure leur gestion et leur synchronisation) et aussi au niveau de l'application python qui s'exécute sur l'ordinateur (2 threads: un pour la communication avec le robot et un pour la mise à jour de l'interface graphique). Les deux programmes s'exécutent en parallèle. Étant donné que le ressources matérielle de l'ordinateur est nettement supérieures à celle du robot epuck2, nous avons décidé que le robot envoie les données telles quelles et c'est au programme python de les traitées / convertir et l'afficher sur l'interface graphique. Sur la Figure 3, nous pouvons voir que tous les threads dans les modules C s'exécutent d'une manière concurrente entre eux et de même pour les threads en python. Cependant, tous les threads en C s'exécutent en parallèle avec les threads en python.

6.3 PRÉSENTATION DES THREADS

Le robot a deux fonction: soit il suit la ligne noire, soit il scanne la commande. La figure 4 ci_dessous illustre la gestion des threads contôllant le robot lorsqu'il est suit la ligne noire.

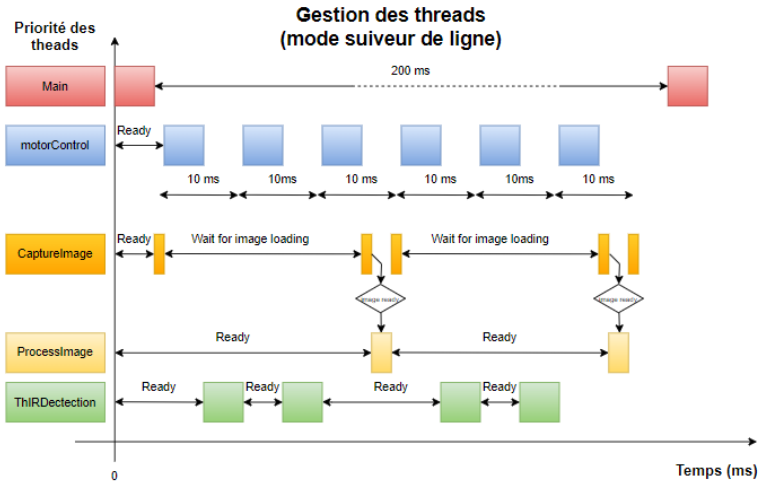

FIGURE 4

Pour le thread **motor_Control**, nous avons utilisé la méthode **chThdSleepUntilWindowed** au lieu d'utiliser un **Sleep** pour fixer la fréquence du thread motor_control à 100Hz afin que le robot puisse réagir plus rapidement indépendamment du temps nécessaire pour la capture de l'image (feedback du régulateur). Le thread "Main" à une priorité plus élevé pour que l'utilisateur puisse stopper le robot à tout moment avec un **Sleep** de 200ms. Nous avons choisi un sleep d'une durée assez longue car augmenter la fréquence de ce thread (qui sert seulement à réceptionner les donnée de l'interface graphique voir 6.3.1) ne sert qu'à surcharger la communication et la synchronisation entre les threads. Lorsque le thread du **motor_control** se termine avant les 10 ms, le thread **CaptureImage** puis le thread **ProcessImage** s'exécutent si la capture de l'image est terminée sinon c'est le thread **thIRDetection** qui sera exécute puis redonne la main au **scheduler** grâce à la fonction **chThdYield**.

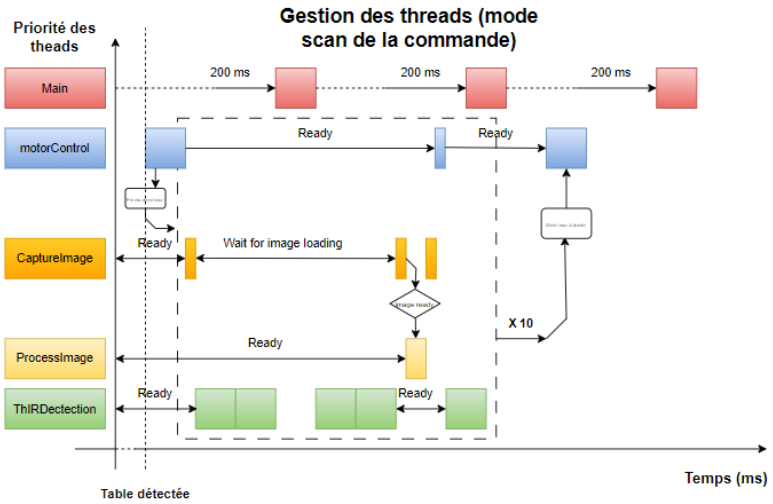

FIGURE 5

La figure 5 ci_dessus illustre la gestion des threads lorsque le robot commence à scanner les commandes. Si le robot détecte une table (voir 6.3.2), le robot tourne de 90° à droite et scanne 10 fois la commande. Pendant ce temps le thread **motor_control** récupère 10 fois la largeur de la ligne scannée pour en calculer la moyenne. Pour ce faire, il doit attendre 10 fois la sémaphore **line_ready** et donc attendre l'exécution des threads **CaptureImage** et **ProcessImage**. Nous avant choisi une telle implémentation pour une meilleur identification de la commande. Étant donné que le thread **motor_control** est en attente et que le thread **Capture_image** attend encore l'image, le thread **ThIRDetection** s'exécute toujours et sera interrompu que si le sleep de la main à pris fin. Ce thread sera toujours en exécution pour que le robot puisse réagir en temps réel à son environnement dans des versions ultérieurs.

6.3.1 MAIN THREAD

Le thread principale consiste à récupérer les informations de l'ordinateur par bluetooth, et d'allumer les *leds* selon l'état du robot envoyé: en mode **STOP**, la *led1* s'allume, en mode **SCAN**, la *led3* s'allume et en mode **WORK**, la *led5* s'allume.

6.3.2 THIRDETECTION

Ce thread permet au robot de détecter s'il y une table à sa gauche. Si une table se trouve à une distance inférieure à 3cm, nous mettons le bit qui correspond à la détection des tables dans la structure robot_ data (voir figure 7) à 1. Étant donnée que les valeurs reçues de ce capteur ne sont pas en cm et par soucis d'optimisation, nous avons fixé le seuil de détection expérimentalement.

6.3.3 MOTORREGULATOR

Ce thread permet au robot d'adapter sa direction selon une ligne noire tracée au sol et de tourner pour récupérer les commandes. Nous avons donc définit deux modes de direction:

- **Suivre la ligne droite:** Le robot avance toujours avec une vitesse constante définie et grâce au régulateur PID dont nous avons fixer ses paramètres d'une manière expérimentale, nous ajustons sa direction pour qu'il reste toujours centré et peut suivre les déviations de la ligne noire.
- **Manoeuvre pour le scan de la commande:** le robot tourne de 90° à droite, récupère 10 fois la commande (voir 6.3 mode scan de la commande) et retourne de 90° à gauche pour suivre de nouveau la ligne.

6.3.4 CAPTUREIMAGE

Ce thread permet de configurer la caméra en fixant le format de l'image à RGB565 ainsi que la position de la ligne et des colonnes à capturer (ligne numéro 100 et 101 qui correspondent au centre de l'image). Ensuite, il prépare le contrôleur DCMI pour la capture. La fonction principale est d'attendre la capture de l'image et de signaler cet événement en utilisant une sémaphore.

6.3.5 PROCESSIMAGE

Ce thread attend jusqu'à ce qu'une image soit capturée afin de récupérer un pointeur vers le tableau rempli avec la dernière image et récupère la valeur de l'intensité rouge de tous les pixels afin d'estimer la positon et la largeur de la ligne noire. Pour ce faire, il calcul l'intensité moyenne de la ligne. Ensuite, il compare en même l'intensité du pixel numéro n et le n+5 avec la moyenne. Si l'intensité du pixel n est plus grand que la moyenne et le n+5 plus petit, cela marque le début de la ligne noire. Pour la fin de la ligne noire, il faut que le pixel n soit plus petit que l'intensité moyenne et le pixel n+5 plus grand. Si le robot ne détecte pas où commence la ligne noire, nous considérons que ça commence à 0 et nous faisons la même chose

pour la fin de la ligne noire. Nous avons choisi une telle implémentation pour améliorer le rendement du régulateur PID et pour que le robot détecte les virages étroits. Nous avant également ajouté un miroir à 45 pour le robot puisse détecter le ligne qui se trouve juste devant lui. En outre le robot allume la frontLed (couleur rouge) afin de saturer tous les pixels rouges pour un objet blanc (augmenter le contraste entre les pixels blancs et les pixels noires) pour une meilleure identification de la ligne noire.

6.4 OPTIMISATION

6.4.1 OPTIMISATION DE MÉMOIRE

Pour optimiser la mémoire utilisée par l'e-puck et ne pas briser le principe d'encapsulation, nous avons éviter l'utilisation de variables globales et nous avons limiter l'utilisation des variables statiques. Pour ce faire, nous avons déclaré une variable locale dans la main nommée robdata, une structure contenant toutes les données récupérées par les capteurs du robots, et nous passons sa référence (un pointeur) à tous les autres threads pour qu'ils puissent modifier ses attributs sans avoir besoins de créer d'autres variables locales. La figure 6) ci_ dessous illustre la mémoire allouée à cette structure dont la taille est de 4 bytes.

FIGURE 6
Mémoire allouée pour la structure robot_data

Cette structure contient:

- **Position:** un tableau de 2 uint_ 8t qui contient la variation de la position du moteur qui actionne les roues du robot.
- **Speed:** un uint_8t contenant la vitesse actuelle du robot en nombre de pas/(4*s) et qui sera convertie dans l'interface graphique en cm/s et multipliée par 4. Vue que la vitesse est stockée sur un seul byte (valeurs entre 0 et 255) et que les 2 LSB ne sont pas assez significatifs pour l'utilisateur, nous avant décidé de stocker la valeur réelle divisée par 4 afin de couvrir un large intervalle de vitesse (valeurs entre 0 et 1023 pas/s) sans avoir besoin de changer le protocole de communication tout en optimisant la mémoire allouée au donnée du robot.
- **État du robot, de l'environnement et de la commande:** (voir figure 7). En effet, au lieu de coder toutes les informations sur 8 boolean et donc 8 bytes, nous avons implémenté un seul entier non signé (1 seul byte) où chaque bit correspond à un boolean comme le montre la figure 7.

FIGURE 7
Informations codées sur le dernier byte de la structure

- **État du robot:** (00) correspond à l'état **STOP**, (01) correspond à l'état **SCAN** et (10) correspond à l'état **WORK**.
- **Scan de l'environnement:** le bit numéro 5 est toujours vrai si le robot est dans le mode SCAN sinon il est mis à faux. Ce bit indique à l'interface graphique que le robot à détecté une ligne noire.

Dans une version ultérieure, il peut aussi servir comme bit de vérification de données envoyées.

- **État du scan de la commande:** (00) indique que le robot est en train de tourner vers la commande, (01) indique que le robot scanne la commande, (10) lorsque le robot tourne vers la ligne et (11) indique que le robot repart pour suivre la ligne.
- **commande:** (00) correspond à aucune commande, (01) signifie que la commande est un *Burger*, (10) signifie que la commande est une *Pizza* et (11) correspond à une commande *Pasta*.

6.4.2 OPTIMISATION DES THREADS

Nous avons également pensé à minimiser le nombre de threads nécessaires pour accomplir les tâches de robot, tout en maximisant l'efficacité de ces threads. En effet, en réduisant le nombre de threads, nous minimisons la surcharge de communication et de synchronisation entre les threads, qui risquent d'entraîner des coûts de performance importants et une utilisation excessive de la mémoire. Dans notre modèle de base, nous avant prévus 2 threads pour la communication, un pour l'envoi des données et un pour la réception. Cependant, après une longue réflexion, nous avons choisi de fusionner ces 2 threads avec d'autres threads.

- **Réception des données:** Nous avons privilégié de placer la méthode de réception de données dans le main (le thread principal) et d'adapter la fréquence de réception à 5Hz (toutes les 200ms) pour ne pas surcharger la permutation entre les threads.
- **Envoie des données:** Cela se fait dans le thread principal qui pilote les moteurs du robot. Étant donnée que le thread pi régulateur s'exécute chaque 10 ms, nous avons implémenté une variable compteur statique afin de régler la fréquence d'affichage à 10Hz (100ms).

6.4.3 OPTIMISATION DU TEMPS DE CALCUL ET DU CHOIX DES VARIABLES

Nous avant également pensé au choix des variables:

- **Variables de type *float*:** Après une longue réflexion, nous avons choisi d'implémenter le régulateur **PID** avec des variables de types entier sur 2 bytes afin de réduire le temps nécessaire pour le calcul. Même si quelques opérations de calcul avec des variables *float* peuvent avoir le même temps que de calcul que celui avec des *entiers* grâce à la **FPU**, l'utilisation de cette unité consomme de l'énergie et nous considérons cela inutile.
- **Choix des constantes du régulateur PID:** Nous avons choisi une valeur seuil de 150 pour le ARW de sorte de pouvoir utiliser des entier de 2 bytes (entre -255 et +255) pour la somme de erreurs au lieu des entiers sur 4 bytes. Nous avons également choisi des constantes entières pour les termes *KP*, *KI* et *KD* toutes en assurant un bon rendement pour le régulateur **PID**.
- **Constante NSTEP_90_DEGREE_TURN:** elle indique le nombre de pas à effectuer par chacun des moteurs pour que le robot puisse tourner de 90° à droite ou à gauche. Nous l'avons calculé manuellement de la manière suivante :

$$\frac{PERIMETER\_EPUCK}{4} * \frac{N\_STEP\_ONE\_TURN}{WHEEL\_PERIMETER} = \frac{5.35}{4} * \frac{1000}{13} = 323, .. = 323 \tag{1}$$

7 PRÉSENTATION DES MODULES

7.1 MODULES EN C

Ces modules servent à programmer le robot afin de pouvoir assurer son bon fonctionnement. Ils permettent d'encapsuler les fonctionnalités et les composants telles que les différents capteurs, la caméra, les moteurs pas-à-pas, le traitement d'image, etc, ainsi qu'une écriture plus lisible et aérée du code.

7.1.1 PROGRAMME MAIN

Ce module permet d'initialiser tous les threads nécessaires. Il contient également le thread pour la réception des données (voir 6.3.1).

7.1.2 FICHIER DE DÉFINITION ROBOT_DATA

Ce fichier contient la structure "Robot_ data" qui définit tous les données nécessaires au fonctionnent du robot (figure 7). Nous définissons également le type Robot_ mode, type qui énumère les 3 états possibles du robot: **STOP, SCAN, WORK** et le type **Environment_ data**: structure avec 2 attributs codés chacun sur un seul bit: **path_ detected** et **table_detected**, ainsi que toutes les constantes nécessaires au fonctionnement du robot.

7.1.3 MODULE COMMUNICATION

Il permet la gestion de protocole de communication bluetooth entre le robot et l'ordinateur:

- **Initiation du protocole communication:** Activation de la communication sérielle pour le périphérique UART3 avec une vitesse de 115200 bits par seconde.
- **Envoi et réception des données:** compression de données récoltées par le robot dans un tableau de 4 entiers non signés et envoi au pc de 10 bytes: 5 bytes pour le mot start, 1 byte pour la taille de donnée et 4 bytes pour les données. Pour la réception des données, il récupère le mot "START" envoyé par le pc et ensuite un byte pour la taille des données qu'il reçoit du pc. Si ce byte est égale à 1, il déballe le dernier byte (lit seulement les 2 MSB) afin de changer l'état de robot (voir figure 7).

7.1.4 MODULE OBSTACLE

Le module obstacle sert à détecter tous les obstacles qui se trouvent à gauche du robot. Il permet d'initier la communication **IPS** via un bus de message, activer le protocole de capture de proximité, calibrer les 8 capteurs en fonction des conditions environnementale et créer le thread de détection d'obstacle (voir 6.3.2).

7.1.5 MODULE CHEMIN

C'est le module où se trouve le thread qui contrôle les moteurs du robot (voir 6.3.3 pour suivre la ligne noire au sol, tourner le robot vers la table lorsque celle-ci est détectée.

7.1.6 MODULE PROCESS_IMAGE

C'est le composant principale du code qui sert à récupérer des images de l'environnement, calculer la position de la ligne noire (feedback pour le régulateur PID), ainsi que la largeur de ligne qui permet de détecter le type de la commande. Ce module fournit une fonction permettant d'initialiser le contrôleur de la caméra DCMI, la caméra PO8030 et le 2 threads CaptureImage et ProcessImage (voir 6.3.4 et 6.3.5). Il fournit également deux fonctions *getter* afin de récupérer la largeur et l'épaisseur de la ligne sans pouvoir les modifier à l'extérieur de ce module.

7.2 MODULES PYTHON

Les modules ci_dessous servent à lancer une interface graphique sur l'ordinateur afin de contrôler le robot, le suivre en temps réel, gérer les commandes et sauvegarder toutes les données récoltées.

7.2.1 MAIN PROGRAMME: PROJECT.PY

C'est le module principal: Si l'utilisateur ne donne pas le numéro du *com* dédié à la communication entre le robot et l'ordinateur le programme se termine, sinon il crée un thread de communication et lance l'interface graphique

7.2.2 COMMUNICATION.PY

Ce module sert à gérer le protocole de communication ainsi que le traitement de données reçues du robot.

- **Initiation le protocole de communication:** Initier une communication sérielle à travers le port précisé par l'utilisateur avec un *timeout* de 0.5 secondes. Si l'ordinateur n'arrive pas à communiquer avec le robot le programme se ferme automatiquement.
- envoie des données: Si l'utilisateur appuie sur l'un des 3 boutons **SCAN**, **START** ou **STOP**, le pc envoie au robot 7 Bytes: 5 Bytes pour le mot START, un byte pour la taille des données et un dernier Byte où les 2 MSB indiquent l'état du robot. Pour des raisons d'optimisation expliqué ci dessus (6.4.1), on utilise seulement 2 bits pour l'état du robot et on garde les 6 autres bits libres et qui peuvent être utilisés dans des versions ultérieures pour renforcer le protocole de communication (par exemple validation bit,...) ou ajouter d'autres commandes au robot.
- **Réception des données:** Cette fonction consiste à lire d'abord le mot *START* et ensuite récupérer 5 Bytes du robot comme expliqué dans le module communication.
- Traitement des données reçues: Cette fonction sert à convertir les données reçues:
  - Estimation de la position du robot en cm: delta_x et delta_y reçues du robot sont converties en cm puis additionnées à leur valeurs précédentes tout en tenant compte de l'angle téta qui indique l'orientation du robot (voir Références [6]).
  - déballage du dernier Byte, bit par bit, afin de récupérer l'état de scan de l'environnement (détection de chemin et de tables) et le scan de la commande (4 bits, 2 pour la commande et 2 pour valider la commande) voir figure 7.
- **Arrêter le protocole de communication**

7.2.3 USER_ INTERFACE.PY

C'est le module principale de la GUI. il permet la création de l'interface graphique (boutons, text_box et le canvas pour dessiner le parcours du robot), la gestion asynchrone des évènements suite à une interaction avec les boutons et l'affichage de la position du robot,l'état de la commande détectée et la tâche du robot en temps réel.

7.2.4 COMMAND_ MANAGER.PY

Ce module sert à gérer les commandes. Il permet la création, la mise à jour et la suppression des commandes en conservant l'intégrité du système par le gestion du cycle de vie des commandes et en fournissant une interface centralisée.

8 CONCLUSION

En conclusion, notre projet de robotique a été une expérience enrichissante pour nous deux. Nous avons réussi à développer un robot fonctionnel qui peut effectuer des tâches précises et complexes. En utilisant les connaissances que nous avons acquises tout au long du semestre grâce au cours mais également au TP, qui ont été très utile, nous avons pu effectuer un robot efficace et performant. Notre collaboration a été fructueuse, et nous avons travaillé en étroite collaboration pour surmonter les difficultés rencontrées tout au long du projet. Nous avons appris à travailler ensemble mais également en distanciel via Github, à échanger des idées et à résoudre des problèmes de manière efficace. Nous sommes fiers des résultats que nous avons obtenus, et nous sommes convaincus que ce projet nous a aidés à développer des compétences clés dans le domaine de la robotique et de la programmation. Nous sommes reconnaissants envers les différents professeur, assistants et GCtronic pour nous avoir donné l'opportunité de réaliser ce projet, qui nous a permis de mettre en pratique les connaissances que nous avons acquises tout au long du semestre.

9 RÉFÉRENCES

[1] Cours et TPs de systèmes embarqués et robotique 2023. (Francesco Mondada)

[2] Dossier électronique du robot e-puck2. (GCtronic).

[3] GCtronic. e-puck2. `https://www.gctronic.com/doc/index.php/e-puck2`

[4] VL53L0X 1/8 inch VGA Single Chip CMOS Image Sensor with 640 X 480 Pixel Array. Pixelplus Co.,Ltd.

[5] Librairie CustomTkinter pour l'interface graphique. `https://github.com/TomSchimansky/CustomTkinter`

[6] wheeled robot control and odometry youtube video `https://www.youtube.com/watch?app=desktop&v=LrsTBWf6Wsc&ab_channel=CCIRobotics`

[7] data sheet capteur IR https://projects.gctronic.com/epuck2/doc/tcrt1000.pdf