École Polytechnique Fédérale de Lausanne
Section de Microtechnique

MICRO-315 - Systèmes embarqués et Robotique

Gilles Regamey (296642) - Mathieu Schertenleib (313318)
Groupe 59 - 15 Mai 2022

Pathfinder

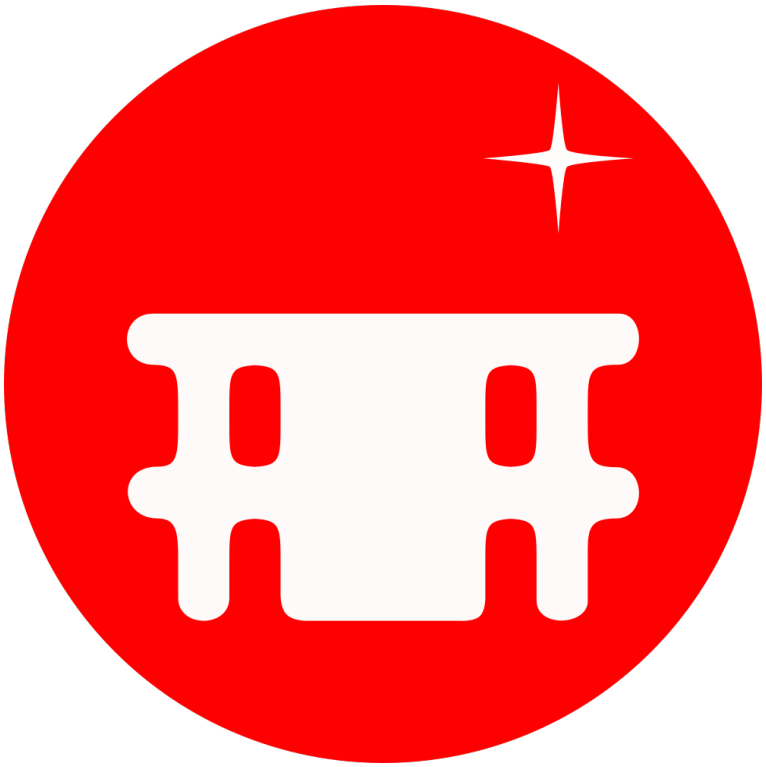


Table des matières

1 Introduction

Ce projet a comme objectif de développer un robot d'exploration, tel un rover à la surface de mars. Son but est de cartographier et étudier une zone de terrain inconnu de manière semi-autonome tout en étant en constante communication avec une base de contrôle.

L'objectif de l'exécution est de placer le robot dans une "arène" qui contient des objets divers puis, sans contact visuel avec le terrain, lui envoyer une suite d'instructions pour qu'il collecte des données sur son environnement. Le résultat final devrait être une carte construite sur l'ordinateur de contrôle, depuis laquelle il est possible de déplacer le robot afin de prendre des photos de son environnement, en trouvant le chemin le plus court qui évite tous les obstacles. L'environnement est considéré comme statique dans le cadre de ce projet, même si la détection d'obstacles imprévus est possible (voir 3.1.2).

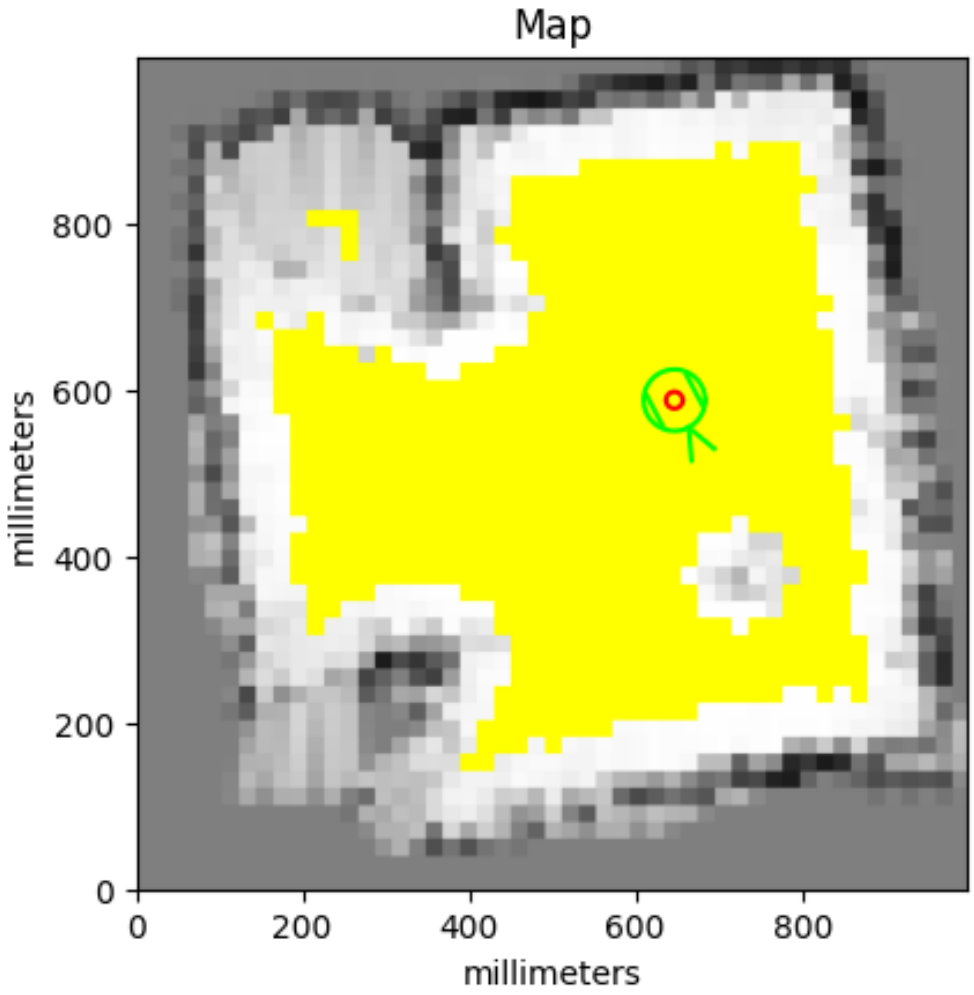


Figure 1 – Carte construite par le robot

2 Utilisation

2.1 Interface sur le PC

Le script Python main.py peut être lancé avec le port à utiliser comme argument sur la ligne de commande, ou alors sans spécifier de port, ce qui va uniquement laisser la possibilité d'utiliser le simulateur et l'éditeur de carte.

La Fig. 2 montre l'interface du programme Python tournant sur le PC. La liste des fonctionnalités des boutons est spécifiée dans la table 1. L'action effectuée par un clic gauche de la souris sur la carte est déterminée par la case d'option située tout à droite et décrite dans

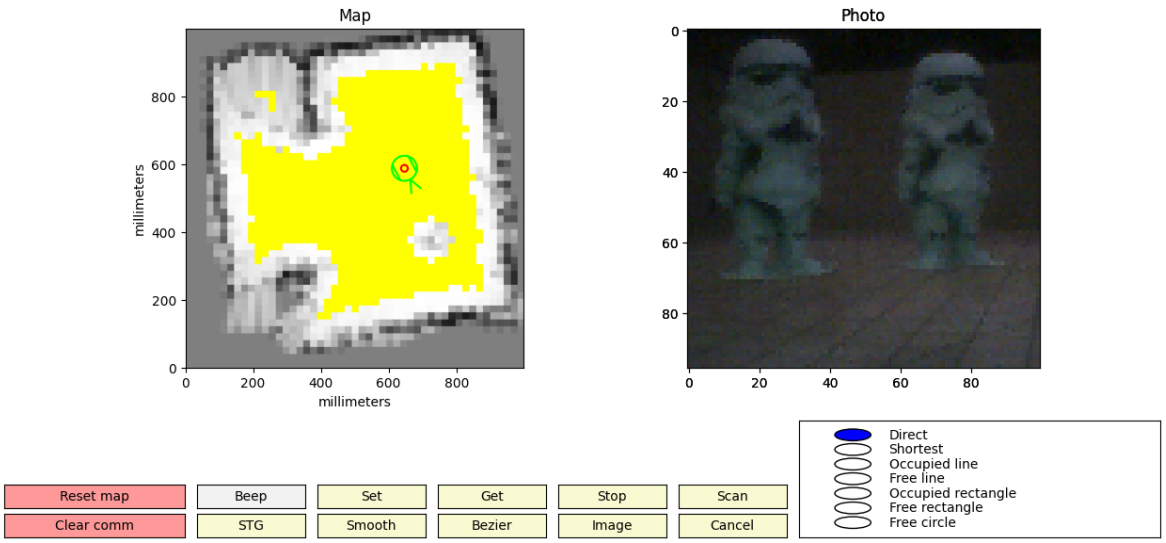


Figure 2 – Interface

la table 2. Les actions de la souris se séparent en deux catégories : celles qui sont utilisées pour définir une trajectoire, et celles utilisées pour dessiner manuellement des obstacles ou des zones libres sur la carte.

Un exemple de carte avec des éléments dessinés manuellement se trouve à la Fig. 5.

Reset map	Remet la carte dans son état initial, c'est-à-dire avec toutes les cases inconnues
Clear comm	Vide le buffer d'entrée de communication du PC
Beep	Demande au robot d'émettre un son à 440Hz pendant 100 ms
Set	Utilise les dernières coordonnées sélectionnées avec la souris pour forcer la position du robot
Get	Demande la position du robot et met à jour la simulation
Stop	Arrête le robot
Scan	Lance un scan de l'environnement pour construire la carte
STG	Lance une commande "Stop-Turn-Go" avec les points sélectionnés (voir 3.6.1)
Smooth	Lance une commande "Smooth turn" avec les points sélectionnés (voir 3.6.2)
Bezier	Lance une commande "Bezier" avec les points sélectionnés (voir 3.6.3)
Image	Capture une image et l'affiche
Cancel	Désélectionne tous les points de contrôle

Tableau 1 – Boutons

Direct	Un clic pose un nouveau point de contrôle
Shortest	Un clic définit une destination qui est utilisée comme but de l'algorithme de recherche de chemin
Occupied line	Un clic suivit d'un deuxième dessine une ligne occupée sur la carte (un mur)
Free line	Un clic suivit d'un deuxième dessine une ligne libre sur la carte
Occupied rectangle	Un clic suivit d'un deuxième dessine un rectangle occupé sur la carte (une enceinte rectangulaire)
Free rectangle	Un clic suivit d'un deuxième dessine un rectangle libre sur la carte
Free circle	Un clic dessine un disque libre sur la carte

Tableau 2 – Actions de la souris

3 Principe de fonctionnement

3.1 Utilisation des capteurs

3.1.1 Time of flight

Le capteur de distance Time of Flight est utilisé lors de l'opération de scan (voir 3.3.7). Il est utilisé dans ce contexte pour déterminer l'éloignement des obstacles autour du robot, et tente de simuler le type de données qu'un lidar fournirait.

3.1.2 Infrarouge

Les capteurs de proximité infrarouges sont utilisés pour la détection à très courte distance d'obstacles imprévus qui pourraient causer une collision. Ils sont ici ajoutés comme mesure de prévention, mais ne contribuent pas à la construction de la carte.

3.1.3 Caméra

La caméra est utilisée pour capturer une image en couleur de son environnement, de la plus grande taille possible.

La détermination des paramètres de largeur et hauteur a été faite en observant la taille maximum du buffer interne du code d'acquisition `MAX_BUFF_SIZE`, définie dans `dcmi_camera.h`. Les pixels sont encodés au format RGB565 puisque nous voulons utiliser une image en couleur la meilleure possible. Étant donné que l'acquisition d'une image n'est qu'occasionnelle, l'utilisation de double buffering n'est pas nécessaire, ce qui permet de capturer une zone maximum de 100 * 96 pixels, représentant 100 * 96 * 2 = 19200 B = `MAX_BUFF_SIZE`. De plus, l'envoi direct des données des pixels au PC depuis le buffer alloué dynamiquement par le driver permet de se débarrasser d'un buffer local au thread de capture.

Finalement, il est à noter qu'un subsampling d'un facteur 4 est utilisé, ce qui permet de capturer une zone quatre fois plus étendue, contre une résolution quatre fois plus faible. Sur le robot, la dimension de l'image est donc spécifiée comme 400 * 384.

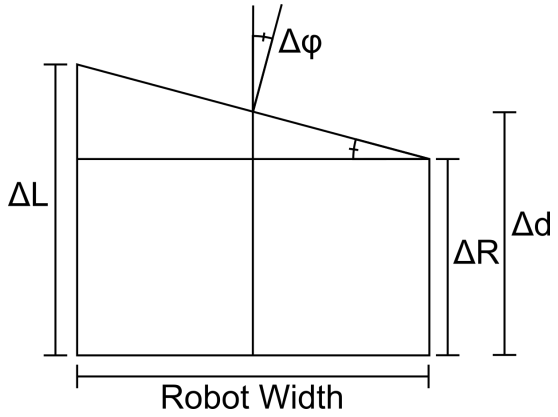
3.2 Déplacement

3.2.1 Commande des moteurs

La commande des moteurs se fait par la librairie du cours.

3.2.2 Odométrie

Pour estimer la position du robot sans se baser uniquement sur la trajectoire idéale envoyée, le robot mesure régulièrement le nombre de pas effectué par chaque moteur. Avec un modèle simple qui approxime le déplacement on peut déterminer quel angle et quel distance il à parcouru entre chaque mesure. Si les mesures sont assez proches, nous pouvons obtenir la position et l'angle du robot en les intégrant. cette technique admet un déplacement sans glissement



$$\Delta d = \frac{(\Delta L + \Delta R)}{2} * cm\_per\_steps\ [cm]$$

$$\Delta\varphi = atan2\left(\frac{(\Delta L + \Delta R) * cm\_per\_steps}{robot\_width}\right)[rad]$$

Figure 3 – Modèle de mesure

des roues.

3.3 Communication avec le PC

Chaque commande envoyée à l'e-puck2 commence par un '!' qui indique le début d'une instruction. Les commandes sont envoyées comme des chaînes de caractères au format ASCII. Si l'instruction reçue (3 ou 4 caractères) est invalide, le robot renvoie la chaîne "INVALID".

Commande	Paramètres	Résumé	Réf.
`!MOVE`	`<size> <data>`	Donne une suite de mouvements à effectuer	3.3.1
`!CLR`	`<posx> <posy> <angle>`	Force la position courante et nettoie le buffer de mouvement	3.3.2
`!BEEP`	`<freq>`	Émet un son	3.3.3
`!POS`		Demande la position courante	3.3.4
`!STOP`		Arrête les moteurs et nettoie le buffer de mouvement	3.3.5
`!PIC`		Capture une image	3.3.6
`!SCAN`		Fait un tour sur lui-même en prenant des mesures de distance	3.3.7

Tableau 3 – Liste des commandes

3.3.1 `!MOVE`

L'instruction `!MOVE` donne au robot une série de mouvements à exécuter.

Le nombre d'instructions de mouvement est envoyé encodé comme un nombre entier sur 16 bits non-signé en little-endian.

Une instruction de mouvement consiste en une vitesse pour le moteur de gauche en steps par seconde, une vitesse pour le moteur de droite en steps par seconde et une durée en millisecondes. Chaque instruction de mouvement est envoyée séquentiellement, encodée comme trois nombres 16-bit non-signés en little-endian.

3.3.2 `!CLR`

L'instruction `!CLR` donne au robot sa position courante et lui demande de vider son buffer interne de mouvement.

La coordonnée x en centimètres, la coordonnée y en centimètres et l'angle en radians sont envoyés comme des nombres en virgule flottante sur 32 bits en little-endian.

3.3.3 `!BEEP`

L'instruction `!BEEP` demande au robot d'émettre un son à la fréquence donnée en Hertz pendant 100 ms.

La fréquence est envoyée encodée en un nombre entier 16 bits non-signé en little-endian.

3.3.4 `!POS`

L'instruction `!POS` demande au robot d'envoyer sa position courante.

Le robot répond en envoyant sa coordonnée x en centimètres, sa coordonnée y en centimètres et son orientation en radians. Les données sont envoyées comme des nombres en virgule flottante sur 32 bits en little-endian.

3.3.5 `!STOP`

L'instruction `!STOP` demande au robot de s'arrêter immédiatement et de vider son buffer interne de mouvement.

3.3.6 `!PIC`

L'instruction `!PIC` demande au robot de capturer une image. Dans notre cas, la taille de l'image est codée sur le robot comme 400 * 384 avec un subsampling de 4, ce qui donne une image de 100 par 96 pixels (voir 3.1.3).

Le robot répond en envoyant premièrement la largeur puis la hauteur de l'image, encodées comme des nombres entiers sur 16 bits en little-endian.

Ensuite, les données des pixels sont envoyées de gauche à droite et de haut en bas, chaque pixel étant codé sur 16 bits au format RGB565. Le nombre d'octets reçu pour les pixels d'une image est donc de largeur * hauteur * 2, ce qui vaut dans notre cas 100 * 96 * 2 = 19200 B.

3.3.7 `!SCAN`

L'instruction `!SCAN` demande au robot de prendre des mesures de distance tout en tournant sur lui-même sur un tour complet.

Le robot répond en envoyant premièrement l'équivalent de `!POS` (voir 3.3.4). Ensuite, il envoie un nombre indéterminé de données de scan, consistant chacune d'une valeur d'angle, codée en virgule flottante sur 32 bits en little-endian, et d'une mesure de distance, codée comme un nombre entier non-signé sur 16 bits en little-endian. La fin de transmission est marquée par une valeure `0xffff'ffffffff`, qui n'est jamais atteignable pour l'angle et la distance, notamment car `0xffffffff` représente -NaN en virgule flottante 32 bits.

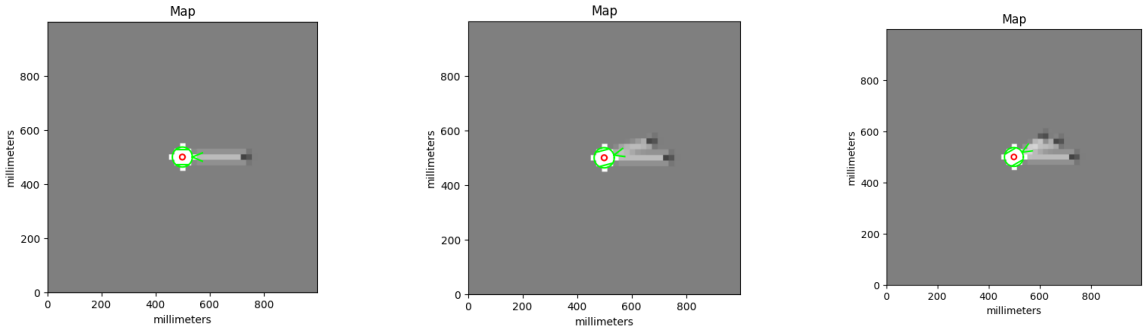
3.4 Construction de la carte

Chaque cellule de la carte enregistre le nombre total d'échantillons qu'elle a vus ainsi que le nombre d'échantillons qui l'ont vue comme libre.

Chaque échantillon du capteur TOF ajoute une ligne d'échantillons libres aux cellules correspondantes de la carte (voir Fig. 4), puis ajoute un échantillon occupé à son extrémité, à condition que la distance mesurée soit plus petite qu'un certain seuil.

L'utilisation des fonctions de dessin de lignes et de cercles de cv2 [1] (le package OpenCV de Python) permet d'ajouter automatiquement de l'antialiasing, ce qui permet d'obtenir un résultat de construction satisfaisant avec un nombre relativement limité d'échantillons.

Les cellules de la carte sont ensuite mises à jour selon la probabilité d'être libres (une cellule blanche a une probabilité de 100% d'être libre) :



(a) Échantillon numéro 1 *(b) Échantillon numéro 2* *(c) Échantillon numéro 3*

FIGURE 4 – Construction de la carte à partir d'échantillons du TOF

$$p = 0.5 + (\frac{F}{T} - 0.5) * (1 - 0.5^T)$$

Où p est la probabilité d'une cellule d'être libre, F est le nombre d'échantillons libres de cette cellule et T est le nombre d'échantillons total de cette cellule.

Cette formule a été choisie afin de réduire la contribution des échantillons lorsque leur total est faible, pour tendre vers $p = F/T$ lorsque le nombre d'échantillons augmente.

Afin de déterminer sur quelles cellules le robot peut se déplacer sans risque de collision, la valeur des probabilités est d'abord comparée à un seuil afin de déterminer si une cellule est considérée comme libre. Pour ce calcul-ci, une cellule libre est représentée par un 0, et une cellule occupée par un 1. Cette image binaire résultante est filtrée avec un kernel constitué d'un disque de '1' entouré de '0'. Toutes les cellules ayant un obstacle plus proche que le rayon de ce disque vont avoir une valeur positive, alors que les cellules qui sont à au moins un rayon de tout obstacle auront une valeur de 0. Cela permet de générer la "walkable map" (représentée en jaune sur l'interface) qui représente toutes les cellules sur lesquelles le robot est assez loin des obstacles. Le rayon du disque du kernel a été pris égal à 2.5 fois celui du robot afin d'avoir une marge de sécurité suffisante.

3.5 Recherche de chemin

L'algorithme de recherche de chemin utilisé est A* [2], en utilisant directement la librairie python-pathfinding [3]. On peut voir sur la Fig. 5 une illustration de la recherche de chemin mise en oeuvre.

3.6 Génération des commandes de mouvement

3.6.1 "Stop-Turn-Go"

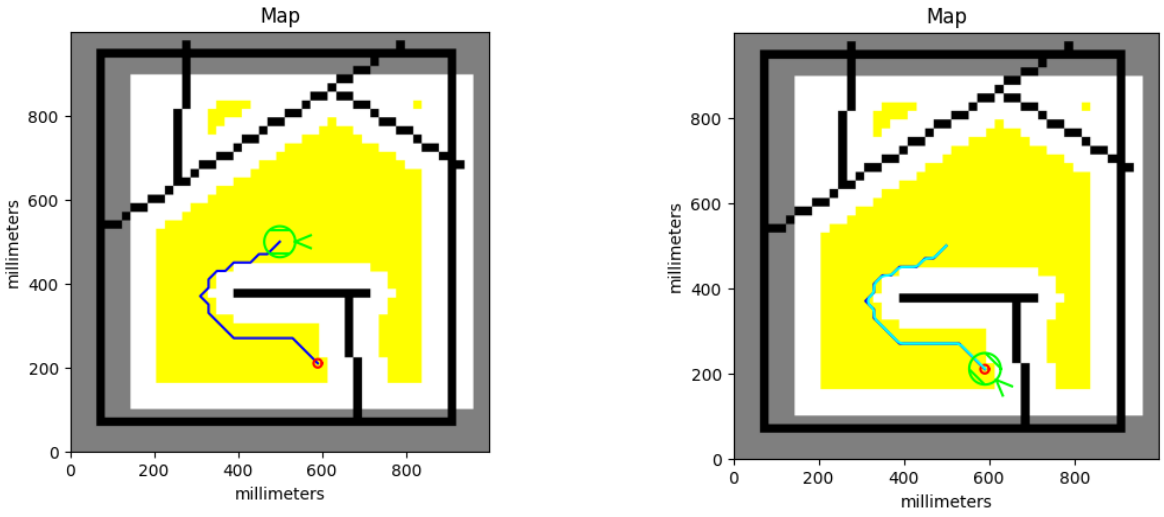
Cette série de commandes très simple fait avancer le robot en ligne droite jusqu'au prochain point de contrôle, puis tourne sur place pour s'aligner au suivant, et ainsi de suite.

3.6.2 "Smooth turn"

Cette série de commandes est en partie identique au "Stop-Turn-Go", mais les angles formés par les points de contrôle sont coupés par un arc de cercle.

3.6.3 "Bezier"

Cette série de commandes, la plus compliquée, forme une courbe de Bezier [4] utilisant les points de contrôle donnés.



(a) Création des points de contrôle

(b) Chemin parcouru en mode "Smooth turn"

FIGURE 5 – *Recherche et exécution du chemin le plus court vers un but*

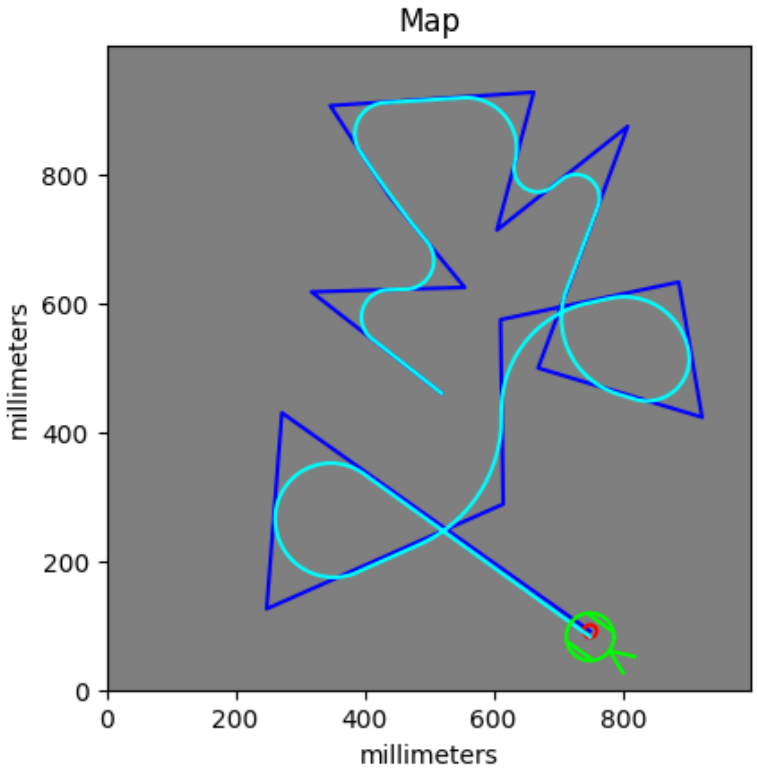


FIGURE 6 – *Example de commande "Smooth turn"*

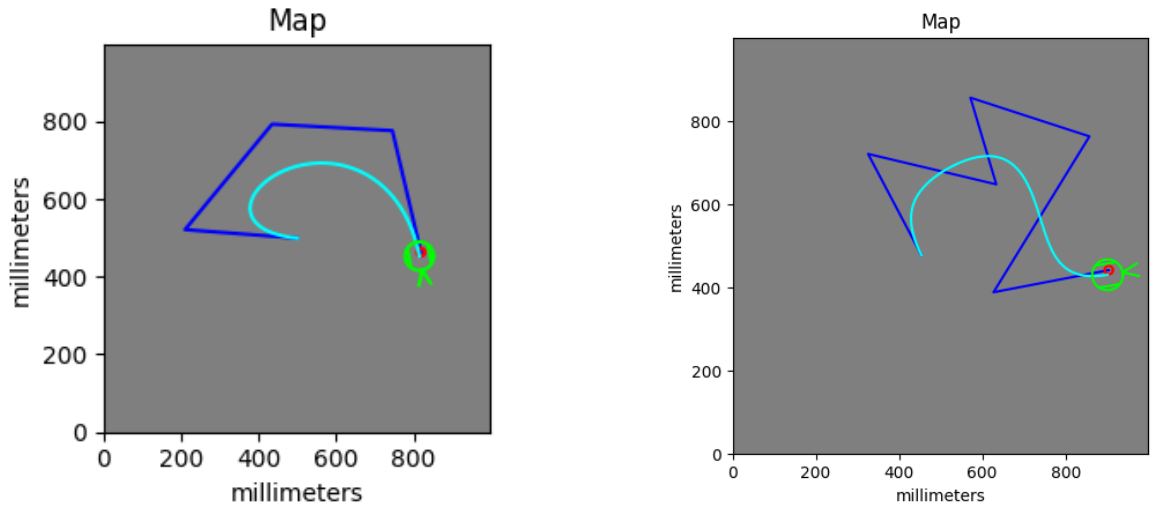


FIGURE 7 – Exemples de courbe de Bezier

4 Organisation du code

4.1 Code source sur GitHub

L'entièreté du code du projet est disponible sur GitHub : https://github.com/mschertenleib/pathfinder

Le code du robot est développé sur la base de la librairie e-puck2_main-processor et ChibiOS, et l'interface Python s'inspire des scripts fournis pour les TPs.

Le code du robot se trouve dans le dossier `robotics_project/robotics_project`. La structure du Makefile suppose que le dossier `lib` contenant `e-puck2_main-processor`, `ChibiOS` et `msgbus` se trouve à la racine du dossier du projet.

Le code de l'interface Python se trouve dans le dossier `robotics_project/python_interface`

4.2 Structure générale du code

Le code du projet se sépare en deux parties principales : premièrement le code en C destiné à être exécuté sur le robot, et deuxièmement le code en Python destiné à être exécuté sur l'ordinateur de contrôle. Le code Python est absolument nécessaire afin de faire fonctionner le robot, puisque c'est lui qui envoie toutes les commandes de déplacement, de scan, de capture d'image, etc. (voir 3). Par contre, il peut être utilisé seul pour tester le simulateur du robot ainsi que les fonctionnalités de construction de carte.

4.3 Code sur le robot : C

4.3.1 main.c

Le main de notre programme est très simple ; il démarre les processus sensibles puis lance les threads. la boucle principale ne fait qu'appeler la fonction `listen(in,out)`.

4.3.2 listen.c

La fonction `listen(in,out)` vide le buffer d'entrée de l'UART à la recherche de commandes à exécuter. Il permet aussi de synchroniser la communication entre le robot et l'interface de contrôle.

4.3.3 move.c

Le module move défini plusieurs fonctions qui agissent ou sur un tableau de données nommé `move_sequence` ou sur les moteurs. `move_sequence` contient des trio sous la forme : `[(speed_left 0,speed_right 0,move_time 0), ...,(speed_left n,speed_right n,move_time n)]` ou n a été choisi plus petit que 100. La taille du tableau est donc 300. Le Thread MoveThread est responsable de l'exécution des instructions de `move_sequence`. Il attend sur `sequence_ready_sem` pour rentrer dans une boucle for qui passe par chaque mouvement de la séquence et assigne les vitesses au moteurs et le temps à attendre jusqu'au prochain mouvement. Ce thread à une priorité NORMALPRIO+1 car le timing des mouvement est important afin de ne pas changer de trajectoire. Seul le thread d'odométrie est plus prioritaire.
Les fonction `ReceiveSpeedInstMove()` et `scan()` sont appelées pour effectuer les commandes `!MOVE` et `!SCAN`. Elles libèrent `sequence_ready_sem`.

4.3.4 Odometrie.c

Ce fichier contient la position du robot dans un tableau de float `[posX [cm], posY [cm], phi [rad]]`. Le thread `odometrieThd` execute le modèle décrit en 3.2.2 de façon régulière. Il a une priorité NORMALPRIO+2 car il faut assurer qu'aucun autre thread le fasse attendre beaucoup plus que son court intervalle, sans quoi les approximations du modèle seraient une source d'erreur importante.

4.3.5 obstacle.c

Ce module permet de détecter si un des 8 capteurs infrarouge dépasse une certaine valeur grâce au thread `obstacleThread`. la détection d'une trop grande proximité est suivie par l'arrêt de la séquence en cours et de l'exécution d'une séquence courte propre à chaque détecteur qui permet de s'éloigner de l'obstacle. La priorité de ce thread est de NORMALPRIO+1 car le timing n'est pas essentiel mais il doit être appelé assé rapidement pour réagir à temps à un obstacle.

4.3.6 process_image

Le module process_image définit deux threads. Le premier, CaptureImage, effectue la configuration de la caméra, puis attend qu'une capture soit demandée par l'intermédiaire du sémaphore `picture`. Le cas échéant, il effectue la capture puis signale grâce au sémaphore `image_ready_sem` qu'une image est prête. Il retourne ensuite dans un état d'attente du sémaphore `picture`.

Le deuxième thread, ProcessImage, attend sur le sémaphore `image_ready_sem` puis envoie la largeur et la hauteur de l'image et les données des pixels (voir 3.1.3) au PC via SD3 (Bluetooth).

4.4 Code sur l'ordinateur de contrôle : Python

4.4.1 EPuck2.py

Ce module gère toutes les données propres au robot e-puck2. Il contient notamment toutes ses dimensions et ses valeurs maximum de vitesse et de mesure. La classe EPuck2 met à disposition des méthodes permettant sa simulation, notamment en lui donnant les mêmes commandes que celles envoyées au robot réel, avec également la possibilité de lire des fichiers textes d'instructions à des fins de debug et de flexibilité.

4.4.2 communication.py

Ce module gère toutes les communications entre le PC et le robot. Il est responsable d'envoyer les commandes telles que définies dans 3, ainsi que de recevoir des données en réponse, dans le cas des commandes de scan (3.3.7), position (3.3.4) et capture d'image (3.3.6).

4.4.3 environment_map.py

Ce module définit une classe Environment_map qui représente la carte bidimensionnelle d'un environnement inconnu. Elle peut être mise à jour avec des échantillons de position et de distance venant du robot (voir 3.4 pour les détails de la mise à jour de la carte), mais également grâce à des fonctions de dessin (ligne, rectangle, cercle), qui permettent de définir certaines cellules comme libres ou occupées.

4.4.4 move.py

Ce module définit la classe Move qui gère les ensembles de mouvements à être exécutés par le robot et le simulateur. Des mouvements sont ajoutés sous forme de "points de contrôle", qui sont ensuite utilisés pour définir les commandes à envoyer aux moteurs (voir 3.3). Des méthodes permettant le génération de mouvements "Stop-Turn-Go" (3.6.1), "Smooth turn" (3.6.2) ou "Bezier" (3.6.3).

Les différentes méthodes implémentées permettent de générer des commandes pour se déplacer en ligne droite, tourner sur place, tourner avec un rayon arbitraire, etc.

4.4.5 main.py

Le module principal est responsable de créer l'interface graphique avec matplotlib [5] (voir 2.1), de construire la carte de l'environnement, de simuler le robot et d'envoyer les commandes nécessaires à celui-ci.

5 Résultats

On peut voir ici les différents tests et résultats acquis au fur et à mesure des itérations.

Plusieurs observations peuvent être notées : premièrement, un décalage additif de l'angle lié aux erreurs d'approximation de l'odométrie a tendance à causer une déformation progressive de la carte. Dans une arène relativement petite comme celle utilisée ici (70 cm x 70 cm) l'erreur reste faible, mais elle augmente plus le chemin parcouru est long.

Deuxièmement, étant donné que le TOF est orienté légèrement vers le haut, le robot a tendance à manquer des obstacles lorsqu'il est à l'autre bout de l'arène. On peut en voir un exemple à la Fig. 9b, où les deux stormtroopers que l'on peut voir en photo à la Fig. 2 sont trop petits pour être discernée de loin.

Finalement, il faut noter que les résultats de la carte reconstruite sont tout à fait à la hauteur de nos attentes sachant que le seul contrôle de position se fait par odométrie, sans réajustement ultérieur. Ce serait une base prometteuse pour aller plus loin dans le contrôle et implémenter les algorithmes de SLAM (Simultaneous Localization And Mapping) [6] [7] [8].

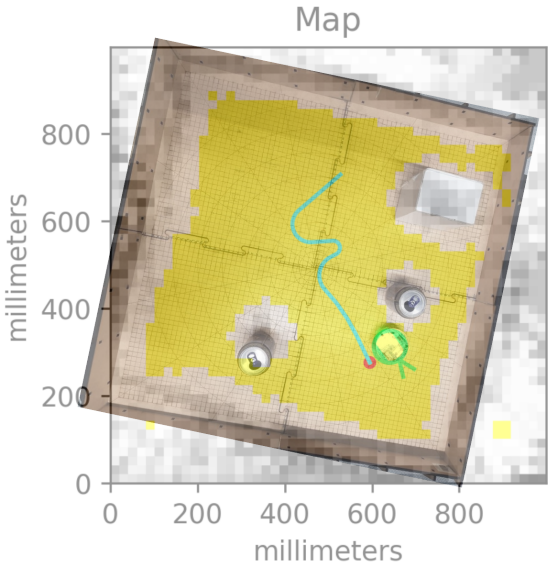
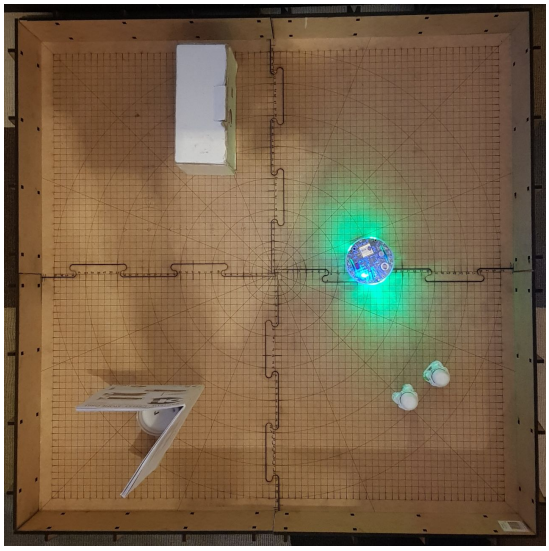
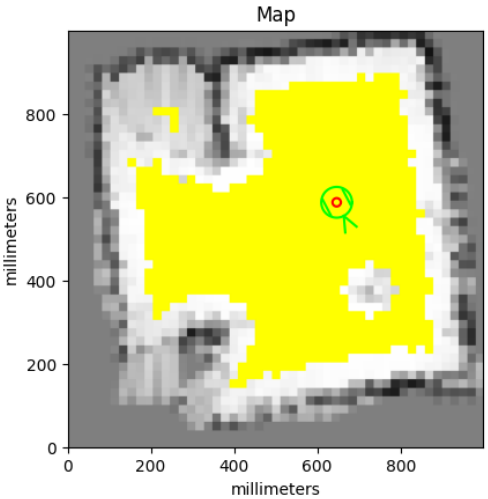


FIGURE 8 – *Vue en superposition de l'arène réelle et de la carte reconstruite*



(a) Arène réelle



(b) Carte reconstruite

FIGURE 9 – *Reconstruction de la carte de l'arène*

6 Conclusion

Pour conclure, il est important de noter que le but premier de ce projet était de réussir à utiliser la carte construite afin de se repérer et d'ajuster la position supposée du robot (algorithme de SLAM, contrôle en boucle fermée), par exemple en effectuant une mesure de corrélation entre deux scans.

Cependant, la seule implémentation de la construction de la carte en boucle ouverte et de l'odométrie s'est révélée plus compliquée que prévu, ce qui nous a contraint d'abandonner le contrôle en boucle fermée, par manque de temps.

Plusieurs pistes pourraient être explorées pour continuer le développement de ce projet : la mise en place d'un vrai algorithme de SLAM avec feedback de la carte pour ajuster la position, mise à jour en temps réel de la position du robot sur l'interface (envoi périodique de `!POS` (voir 3.3.4)), utilisation de l'image acquise par la caméra et de la librairie OpenCV déjà utilisée afin de faire de la reconnaissance d'objets, etc.

Références

[1] *Python OpenCV*. URL : https://pypi.org/project/opencv-python/. (accédé : 2022-05-15) (cf. p. 8).

[2] *A* search algorithm*. URL : https://en.wikipedia.org/wiki/A*_search_algorithm. (accédé : 2022-05-15) (cf. p. 9).

[3] *python-pathfinding*. URL : https://opensourcelibs.com/lib/python-pathfinding. (accédé : 2022-05-15) (cf. p. 9).

[4] *Bezier curve*. URL : https://en.wikipedia.org/wiki/B%5C%C3%5C%A9zier_curve. (accédé : 2022-05-15) (cf. p. 9).

[5] *Matplotlib*. URL : https://matplotlib.org/. (accédé : 2022-05-15) (cf. p. 13).

[6] *Simultaneous localization and mapping*. URL : https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping. (accédé : 2022-05-15) (cf. p. 13).

[7] *What Is SLAM ?* URL : https://ch.mathworks.com/de/discovery/slam.html. (accédé : 2022-05-15) (cf. p. 13).

[8] Shan HUANG ; Hong-Zhong HUANG ; Qi ZENG ; Peng HUANG. « A Robust 2D Lidar SLAM Method in Complex Environment ». In : *PHOTONIC SENSORS* 12.4 (2022). URL : https://link.springer.com/content/pdf/10.1007/s13320-022-0657-6.pdf. (accédé : 2022-05-15) (cf. p. 13).

[9] *Monocular Visual Simultaneous Localization And Mapping*. URL : https://ch.mathworks.com/help/vision/ug/monocular-visual-simultaneous-localization-and-mapping.html#MonocularVisualSimultaneousLocalizationAndMappingExample-3. (accédé : 2022-05-15).

[10] *ORB-SLAM2*. URL : https://pythonrepo.com/repo/raulmur-ORB_SLAM2-python-deep-learning. (accédé : 2022-05-15).

[11] *pyslam*. URL : https://github.com/luigifreda/pyslam. (accédé : 2022-05-15).

[12] *IEPF Line Extraction Algorithm*. URL : https://github.com/ekorudiawan/IEPF-Line-Extraction. (accédé : 2022-05-15).

[13] Daniel ESTLER. « Path Planning and Optimization on SLAM-Based Maps ». In : (2016). (accédé : 2022-05-15, url : https ://ipvs.informatik.uni-stuttgart.de/mlr/papers/16-estler-BSc.pdf).

[14] *Visibility graph*. URL : https://en.wikipedia.org/wiki/Visibility_graph. (accédé : 2022-05-15).

[15] Haarika KONERU. *Visibility graphs*. URL : http://www.cs.kent.edu/~dragan/ST-Spring2016/visibility%5C%20graphs.pdf. (accédé : 2022-05-15).

[16] *Visibility graphs*. URL : http://www.science.smith.edu/~istreinu/Teaching/Courses/274/Spring98/Projects/Philip/fp/visibility.htm. (accédé : 2022-05-15).

[17] *Visibility Graph Path Planning*. URL : https://www.cs.columbia.edu/~allen/F19/NOTES/lozanogrown.pdf. (accédé : 2022-05-15).

[18] *Odométrie*. URL : https://fr.wikipedia.org/wiki/Odom%5C%C3%5C%A9trie. (accédé : 2022-05-15).

Table des figures

Liste des tableaux