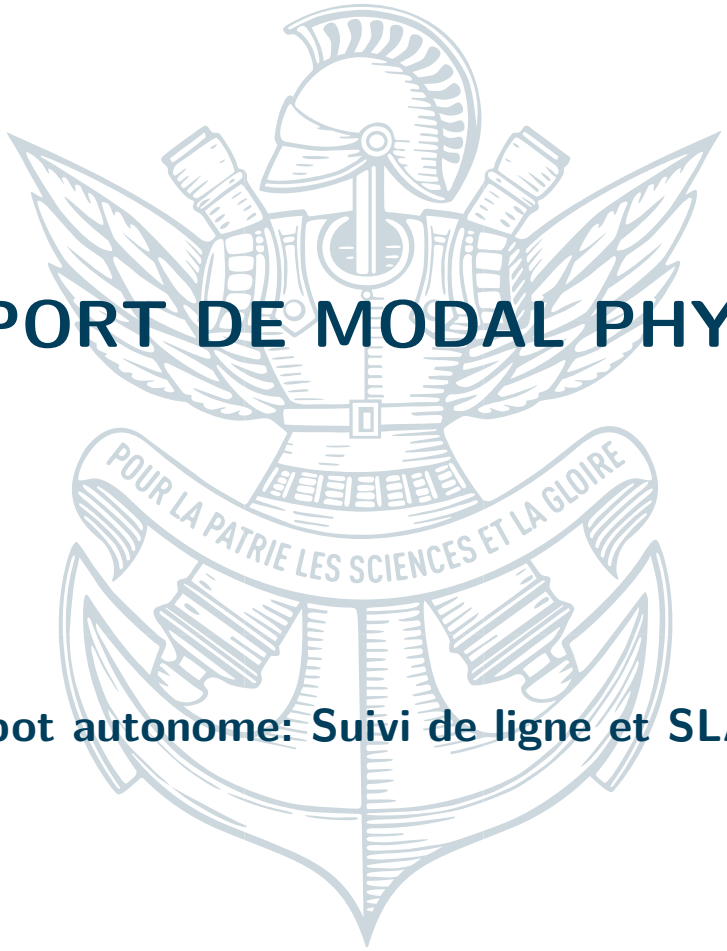


RAPPORT DE MODAL PHY473P

Robot autonome: Suivi de ligne et SLAM

25 mai 2023

Daniel Elmaleh, Aaron Metchinjin

TABLE DES MATIÈRES

INTRODUCTION

La robotique a connu des avancées significatives ces dernières décennies, permettant aux robots de réaliser des tâches de plus en plus complexes et d'interagir de manière autonome avec leur environnement. Deux fonctionnalités cruciales dans ce domaine sont le suivi de ligne et le SLAM (Simultaneous Localization and Mapping).

Le suivi de ligne consiste à guider un robot le long d'une trajectoire spécifiée. l permet aux robots de se déplacer de manière précise le long d'un trajet prédéfini, en s'appuyant sur des capteurs spécifiques pour détecter et suivre une ligne tracée au sol. Cette technologie présente plusieurs avantages majeurs notamment une navigation autonome ce qui ouvre de vastes possibilités dans des domaines tels que la logistique, l'automatisation industrielle et les véhicules autonomes ; la fiabilité car les capteurs de suivi de ligne peuvent compenser les perturbations environnementales et maintenir une trajectoire cohérente ; la simplicité car ne nécessite pas de capteurs sophistiqués ni de calculs intensifs, ce qui le rend accessible à une large gamme d'applications robotiques.

Le SLAM est une technologie essentielle pour les robots autonomes qui doivent cartographier et se localiser dans un environnement inconnu afin de planifier des actions en conséquence. Il combine la perception et la planification, permettant au robot de se déplacer en toute autonomie et de construire une représentation précise de son environnement. Le SLAM présente plusieurs avantages importants notamment l'exploration et cartographie car permet aux robots d'explorer de nouveaux environnements ce qui est particulièrement précieux dans des situations où une connaissance préalable de l'environnement est limitée ou indisponible ; la localisation précise et l'adaptabilité car permet aux robots de s'adapter à des environnements dynamiques et changeants par construction continue de carte.

Le présent rapport présente le projet réalisé en robotique visant à concevoir un robot capable de suivre une ligne rouge dans un premier temps et dans un second temps de réaliser la cartographie d'un milieu tout en s'y déplaçant de telle sorte à éviter des obstacles. Cette deuxième fonctionnalité devra être implémentée en ROS(Robot Operating System). Il se divise en quatre parties : la première présente le cahier de charge et l'échéancier prévisionnel, la deuxième aborde la conception et la réalisation du robot, la troisième fait un bilan en évocant les réalisations, les difficultés rencontrées et les perspectives de manière précise et la dernière aborde la question théorique du SLAM en extérieur.

1
CAHIER DES CHARGES

1.1 Introduction au cahier des charges

Le présent cahier des charges définit les objectifs, les spécifications et les exigences du projet de réalisation d'un robot suiveur de ligne avec des fonctionnalités de slam. Ce projet, réalisé en binôme, vise à concevoir et à développer un système robotique autonome capable de suivre précisément une ligne rouge tracée au sol d'une part, et d'autre part de se déplacer dans un environnement en évitant les obstacles et en le cartographiant.

- Le suivi de ligne représente la fonctionnalité clé du robot, permettant une navigation autonome le long d'un trajet prédéfini. Il doit se faire de manière précise et fiable. Pour atteindre cet objectif, le robot sera équipé de capteurs appropriés qui permettront de détecter la ligne et de maintenir le robot sur sa trajectoire.
- Bien que la réalisation complète du slam ne soit pas une exigence primordiale, nous explorerons les possibilités d'intégrer des fonctionnalités de localisation et de cartographie dans la mesure du possible.

Ce cahier des charges définira donc les spécifications techniques du robot, y compris les composants matériels, les capteurs nécessaires et les codes utilisés. Les contraintes de développement, telles que les ressources disponibles et les délais seront prises en compte tout au long du projet.

Ce cahier des charges servira de référence pour orienter le développement du projet et garantir que les objectifs sont clairement définis. Il constituera un guide essentiel pour l'évaluation de la réussite du robot suiveur de ligne et de slam.

1.2 Spécifications techniques du robot

Voici des spécifications techniques attendues pour notre robot capable de suivre une ligne rouge et de réaliser du SLAM :

1. **Mécanique** :
   - Châssis en plexiglas léger et robuste.
   - trois roues dont deux motrices avec des moteurs à courant continu et une roue folle. Les roues seront en ...
   - Une tour pour fixer le capteur infrarouge.
   - Un servomoteur tournant à vitesse angulaire moyenne $\omega = 1.3$ rad/s et balayant un angle de 180°. Sur lui sera fixée la tour.

2. **Capteurs de suivi de ligne** :
   - Caméra RGB-D USB de résolution 640 pixels/ 480 pixels

3. **Capteurs pour le SLAM** :
   - Capteur infrarouge d'une plage de détection d'au moins 5m, étanche à la poussière et à l'humidité, capable de supporter les températures entre -10°C et 50°C.

4. **Contrôle et traitement des données** :
   - Microcontrôleur puissant (ici Raspberry Pi3) pour le contrôle et le traitement des données.
   - I2C Motor Driver délivrant un courant continu de 1.2A et fonctionnant sous une tension de 3.3V/5V

— Algorithme de suivi de ligne basé sur le seuillage et l'analyse d'image pour maintenir le robot sur la trajectoire de la ligne rouge.
— Algorithme de SLAM pour la localisation et la cartographie simultanées.

5. **Alimentation** :
   — Batterie Li-ion rechargeable de 5 V avec une capacité de 2100 mAh.
   — Autonomie cible : au moins 2 heures de fonctionnement continu.

6. **Fiabilité et robustesse** :
   — Conception mécanique et électronique robuste pour une utilisation prolongée et une résistance aux chocs et vibrations.
   — Mécanismes de correction d'erreur pour compenser les variations de la ligne, les écarts de vitesse des moteurs et les déviations imprévues.

1.3 Exigences et contraintes du projet

1. **Fonctionnalités requises** :
   — Détection et suivi de la ligne rouge
   — Capacité à effectuer des mouvements plus ou moins stables lors du suivi de ligne.
   — Gestion des obstacles rencontrés sur la trajectoire.
   — Cartographie de l'environnement.

2. **Performances attendues** :
   — Précision de positionnement de +/- 2 cm par rapport à son axe central de la ligne rouge.
   — Réactivité du robot pour éviter les obstacles en moins de 0,5 seconde.
   — Précision de localisation pour le slam de +/- 5 cm.

3. **Contraintes environnementales** :
   — Fonctionnement en extérieur avec des variations d'éclairage naturel.
   — Capacité à naviguer sur des surfaces irrégulières, telles que des sols en terre ou en béton.

4. **Contraintes de temps** :
   — Début du projet : 24/02/2023
   — Fin du projet : 19/05/2023

1.4 Livrables attendus

A la fin du projet, il est attendu qu'on délivre :
— Un robot suiveur de ligne fonctionnel et qui de plus puisse faire du SLAM
— Un rapport de projet (la méthodologie utilisée, les résultats obtenus, les difficultés rencontrées)
— Une présentation orale

1.5 Plan de développement, échéancier prévisionnel et répartition des tâches

Le projet se réalisera en 4 principales phases :

Phase 1 : Spécification du matériel et choix des capteurs du 24/02/2023 au 10/03/2023.

Il sera question ici de se familiariser avec le matériel et les capteurs mis à notre disposition, de déterminer leurs fonctions, savoir les mettre au point et les faire fonctionner individuellement puis de réfléchir sur leur utilité pour le projet

De ce qui est de la répartition des tâches, le matériel sera partagé entre les deux personnes formant le binôme pour spécification, étude et réflexion sur l'utilité pour le projet.

Phase 2 : Conception du châssis, intégration du matériel et des capteurs et branchements du 11/03/2023 au 24/03/2023.

Il sera question ici de réaliser la partie hardware du robot.

Lors de cette phase, Aaron se chargera de dessiner le châssis, le portant de la caméra et la tour du capteur infrarouge à la main et Daniel réalisera les dessins sur Fusion360. Les pièces seront imprimées par laser (châssis et la tour) et 3D (portant) au Fablab avec l'aide de Cyril Hasson, Responsable prototypage.

Phase 3 : Développement logiciel pour le suiveur de ligne 25/03/2023 au 22/04/2023.

Il s'agira d'écrire les codes permettant de réaliser le suiveur de ligne rouge à partir de l'image donnée par la caméra et en communiquant les bonnes commandes aux roues. Suivront ensuite les tests et l'ajustement des algorithmes.

Ici, Daniel s'occupera de l'asservissement des roues et Aaron du traitement d'image permettant d'isoler la bande rouge du reste du sol. La recherche d'un moyen de mettre en commun les codes et de communiquer les bonnes commandes aux roues sera faite ensemble de même que les tests.

Phase 4 : Développement logiciel pour le SLAM du 23/04/2023 au 19/05/2023.

Il s'agira d'écrire les codes permettant de réaliser le SLAM. Suivront ensuite les tests et l'ajustement des algorithmes.

Ici, Daniel s'occupera du capteur infrarouge notamment comment transformer les données qu'il renvoie en message LaserScan en Ros ainsi que de la création noeuds et Aaron s'occupera de l'implémentation du SLAM (localisation et carthographie). Les deux réfléchirons à une stratégie de déplacement dans le milieu en évitant les obstacles.

1.6 Méthode de validation

Pour le suiveur de ligne, la validation se fera au laboratoire à partir des lignes rouges présentes au sol.

Pour le SLAM, nous équiperons l'espace libre du laboratoire d'obstacles d'au moins à même hauteur que le capteur infrarouge et nous analyserons le comportement du robot ainsi que la carte générée.

2
CONCEPTION ET RÉALISATION DU ROBOT

Dans cette partie, nous présentons comment la conception et la réalisation du robot se sont déroulées. Nous insisterons sur le choix des composantes, le montage, les méthodes de mise au point du matériel et des capteurs et l'élaboration du code.

2.1 Choix des capteurs et des composants matériels

Après étude de chaque composant matériel et des capteurs mis à notre disposition, il nous a semblé pertinent de retenir :

- Une Raspberry Pi3 : micro contrôleur pour le contrôle et le traitement des données.
- Une carte mémoire de 32Go de capacité.
- Une batterie délivrant 5v de tension et servat d'alimentation pour la Raspberry.
- 3 Roues : 2 servant à la mobilité du robot et connectées à des moteurs et une roue folle servant à équilibrer la structure. Les roues que nous avons utilisées ont 6cm de diamètre.
- Un I2C Motor Driver. C'est un module électronique conçu pour contrôler la vitesse et la direction des moteurs à courant continu à l'aide du protocole de communication I2C (Inter-Integrated Circuit). Il a l'avantage de pouvoir commander deux moteurs à la fois.
- Une caméra RGB-D USB dont l'image sera utilisée et traitée pour le suivi de la ligne.
- Un capteur infrarouge : pour calculer les distances de manière précise.
- Un servomoteur : servant de base pour la tour, sa rotation permettra au capteur infrarouge de balayer un angle de 180°.
- Des jumpers mâles et femelle pour les branchements.
- Une plaque de plexiglas de 6mm d'épaisseur.

Le matériel choisi correspond bien aux spécifications fixées dans le cahier de charges.

2.2 Pièces et schéma de montage

Nous avons opté pour un modèle simple après avoir mesuré approximativement la taille de notre carte Raspberry Pi ainsi que des roues. Nous avons sélectionné le modèle le plus simple qui a été confirmé après une itération de découpe sur la machine laser. Il est important de noter que au moment du dimensionnement du châssis, nous n'avions pas encore la battérie et du coup, n'avons pas bien estimé ses dimensions pour en prendre compte de manière optimale. Nous avons donc dû par la suite réadapter l'emplacement initialement prévu des éléments sur ce châssis. Les deux trous circulaires sur le châssis permettent de passer les câbles électroniques de connexion. L'espace à l'avant du châssis permet de positionner une roue folle pour stabiliser la structure tout en permettant la rotation sur l'axe central du robot.

Nous avons aussi conçu un support incliné pour la caméra et fait un trou aux dimensions de la partie posterieure de la caméra. Ce portant fait un angle de vue vers le sol. Notre inclinaison par rapport à l'horizontal est de 70 degrés et cet élément a été imprimé en 3D.

Par la suite, nous avons imprimé en 3D la tour sur laquelle le détecteur IR est fixé, permettant la connexion avec le moteur servo qui est lui-même installé sur le châssis au centre par ajustement forcé.

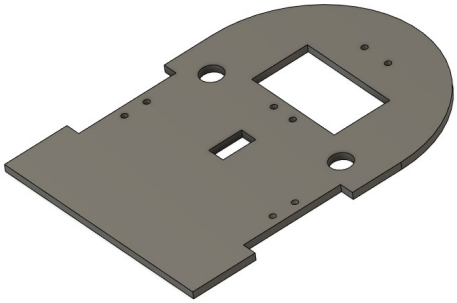


Figure 1 – Châssis 3D

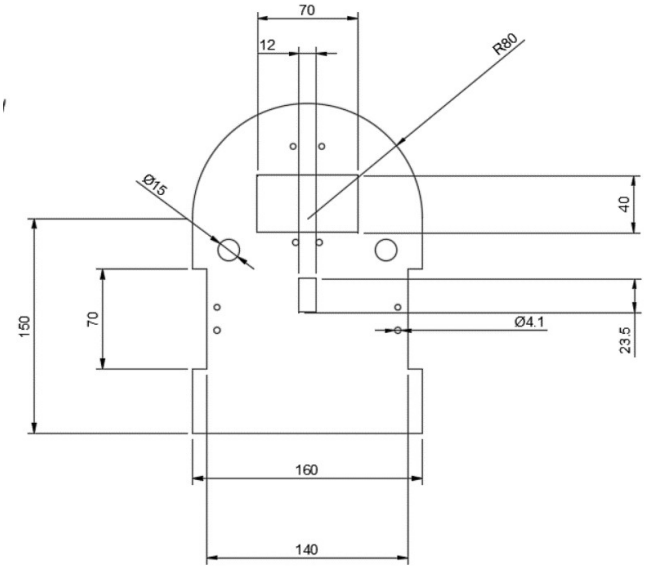


Figure 2 – Dimensionnement du châssis

Figure 3 – Portant de la caméra

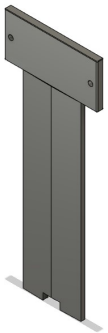


Figure 4 – Tour du capteur infra-rouge

Le schéma de montage est le suivant :

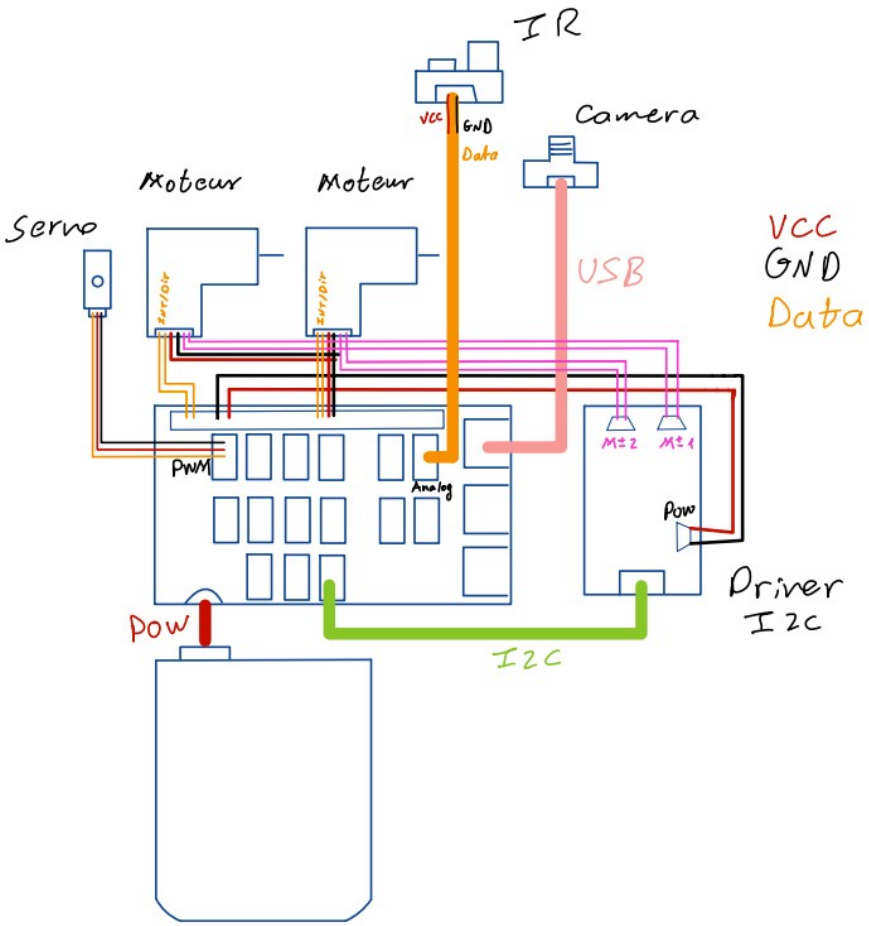


Figure 5 – Schéma de montage

2.3 Programmation du robot pour le suivi de ligne

c'est la partie qui nous a pris le plus de temps et qui correspond à la phase 3 de notre projet. Le code a été réalisé en 3 principales étapes : l'asservissement des roues, le traitement d'image et la mise en commun des deux fonctionnalités.

2.3.1 • Traitement d'image de la caméra

La capture, la lecture et le traitement des flux vidéo en temps réel ont été gérés par la bibliothèque OpenCv.

Tout d'abord, la configuration et la lecture du flux vidéo de la caméra ont été faites puis nous avons converti l'image RGB fournie par la caméra en image HSV.

Ensuite, grace au code pris sur le tutoriel tutorial_threshold_inRange [1][1], on a pu fixer les paramètres H

1. Ceci est un lien vers le tutoriel

(teinte), S (saturation) et V (luminosité) de telle sorte à isoler la couleur rouge. A l'aide de plusieurs documentations, nous avons fixé au départ les paramètres suivants : H entre 0 et 20, S entre 134 et 255 et V entre 50 et 255. Après ajustement avec l'image fournie par la caméra, on a retenu les valeurs H(0, 180), S(100, 255), V(100,255) et on les a fixées dans $lowerBound(0, 100, 100)$ et $upperBound(180, 255, 255)$. Ici, nous étions d'ailleurs surpris de constater que prendre toute la plage de valeur du paramètre H affrait une meilleure qualité du filtrage.

Pour affiner le résultat, nous avons effectué un lissage gaussien de l'image HSV.

```
//convertir l'image en espace hsv
cvtColor(frame, hsv, COLOR_BGR2HSV);

//appliquer le filtre HSV pour filtrer la couleur rouge
inRange(hsv, lowerBound, upperBound,filtered);

//appliquer le lissage gaussien pour lisser l'image
GaussianBlur(filtered, smoothed, Size(5, 5), 0, 0);
```

Figure 6 – Conversion, filtrage et lissage de l'image

Puis, nous avons dessiné les contours [2] de la zone rouge ceci dans le but de récupérer l'abscisse, par rapport au plan de la caméra, du centre de gravité du contour, servant de cible à chaque instant pour le robot. Nous avons veuillé à ce que le programme ne considère que le contour le plus large de l'image et ne s'intéresse pas au petites nuances de rouges qui se trouveraient dans la nature.

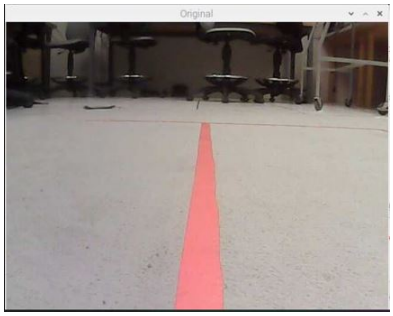


Figure 7 – Image originale

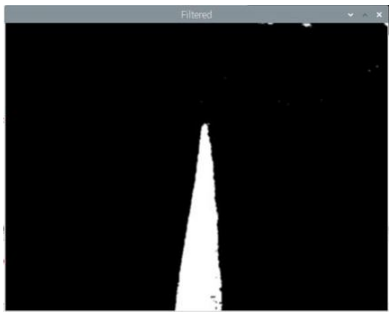


Figure 8 – Image traitée

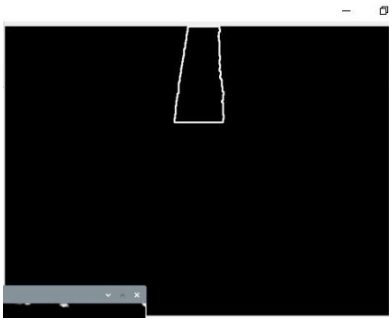


Figure 9 – Image des contours

2.3.2 • Le contrôle des roues

Afin de contrôler les roues de notre robot, nous avons utilisé un microcontrôleur i2C qui fait office de driver pour communiquer avec le moteur et son encodeur. Le moteur dispose de six broches sur son interface : deux pour alimenter le moteur, deux pour alimenter l'encodeur qui calcule la rotation effective du moteur, et deux broches (INT, DIR) permettant à l'encodeur de transmettre ses valeurs à la Raspberry Pi. L'alimentation du moteur autorise également son contrôle par modulation de largeur d'impulsion (PWM). Le rapport moyen, dans le temps, de la durée à l'état haut de l'alimentation par rapport à la période totale du signal envoyé permet au moteur de tourner à une certaine proportion de sa vitesse maximale.

Il était nécessaire en premier lieu de comprendre le protocole de communication i2C (ce qui a été fait avec l'aide de notre enseignant Ludovic), puis de rechercher les adresses qui commandent la rotation des roues notamment l'adresse du bus i2C et les adresses des régistres sur lesquelles commander la vitesse et le sens de rotation de chacun des moteurs. Celles-ci ont pu être trouvées sur le site du constructeur SeedStudio [3][2]. Ainsi, depuis le terminal, les moteurs ont été actionnés.

2. Code en annexe
3. Ceci est un lien

Afin d'écrire un code qui actionne la rotation des moteurs, il a fallu :
— Installer au préalable la bibliothèque WiringPi.
— Configurer le bus I2C ainsi que les différents branchements matériels effectués
— Implémenter le code suivant qui fait tourner la première roue dans le sens direct à vitesse maximale.

```
fd = wiringPiI2CSetup(0x14);
wiringPiI2CWriteReg16(fd, 0x02, 0xff00);
```

Figure 10 – Rotation du moteur

On a pu constater la difficulté que cela présenterait d'utiliser cette syntaxe telle qu'elle dans la suite de notre projet car elle est assez lourde et pas intuitive. Il fallait donc réaliser l'interfaçage des moteurs pour écrire un code plus manipulable pour actionner la rotation des roues, nous avons :
— Ecrit une fonction **per_to_hexa()** [4] qui convertit les entiers entre 0 et 100 en valeurs hexadécimales entre 0 et ff en utilisant une proportionnalité. Le but était de pouvoir contrôler la vitesse de rotation des roues à l'aide d'entiers exprimant le pourcentage de rotation des roues. Ceci simplifie l'interface d'utilisation en convertissant des valeurs comprises entre 0 et 100 en valeurs hexadécimales, interprétables par le registre I2C, pour les envoyer au moteur afin de générer le signal PWM.
— Ecrit une fonction **set_Vitesse_Roues(v1, v2)** [5] qui appelle **wirinPiI2CWriteReg16()** et **per_to_hexa()**
— Des vitesses négatives c'est-à-dire entre -100 et 0 contrôlaient une rotation en sens inverse des roues par modification de l'adresse 0x02 par 0x03.

Naturellement, Il fallait pouvoir récupérer les valeurs réelles (feedback) de vitesses. Pour cela les valeurs de l'encodeur de chaque roue devait être calculées ce qui a été fait en intégrant des interruptions. Les valeurs de l'encodeur représentent deux signaux (collectés sr les broches INT et DIR) corrélés à la vitesse de rotation détectée ; leur combinaison permet de connaître le sens de la rotation et d'améliorer la précision (en prenant à chaque fois la moyenne des deux signaux). La formule permettant de calculer la vitesse de chaque roue est :

```
float speed1 = (count1 - last_count1) * 60.0 / ENCODER_PPR / ((float)(time1 - last_time1) / 1000.0);
speed1 = speed1*100/160;
```

Figure 11 – Calcul de la vitesse de rotation en pourcentage

Où *count* et *last_count* sont les valeurs de l'encodeur pendant un intervalle de temps et $ENCODER\_PPR = 360$. Nous avons eu cette formule d'une source qui nous échappe aujourd'hui mais nous sommes convaincus qu'elle fournit une idée assez précise (à une constante de proportionnalité prêt) de la vitesse du fait de sa dépendance directe en la variation des valeurs de l'encodeur, et inverse en la variation du temps, le reste des calculs étant des multiplications par des constantes.

Cependant, notre but étant d'afficher la vitesse effective en terme de pourcentage, nous avons du pauffiner cette formule. En envoyant une commande de vitesse maximale (0xff), la valeur de vitesse effective donnée par cette formule était entre 159 et 161. En considérant la moyenne, on a compris qu'il fallait multiplier par la formule 100/160 d'où la ligne qui suit. Ceci était dans le but de ramener les vitesses calculées à une échelle de 0 à 100.

Ensuite, se posait la question de l'asservissement de la vitesse des roues. Dans le cadre du modal, notre enseignant Ludovic nous a recommandé de faire un asservissement **PID** [6] : **P** pour "proportional" qui réagit à l'erreur actuelle - la différence entre la valeur désirée et la valeur mesurée, **I** pour "integral" qui réagit à l'erreur

4. Code en annexe
5. Code en annexe
6. Code en annexe

accumulée au fil du temps et **D** pour "dérivé" qui réagit à la vitesse de changement de l'erreur. Ceci permet de réguler et stabiliser le processus dynamique de rotation. Nous voulions avoir un robot qui, lorsqu'il va tout droit, avance à 30-40 de sa vitesse maximale ; ceci représentait donc la vitesse désirée pour tester l'asservissement. Après écriture du code, le paramètre **Kp** a d'abord été fixé jusqu'à obtenir une relativement stable, puis **Kd** afin de diminuer l'erreur constante par rapport à la valeur désirée et enfin **Ki** pour atténuer les oscillations restantes. Au final, nous avons retenu les valeurs suivantes : **Kp = 0.8, Ki = 0.25 et Kd = 0.2**.

Une fois le modèle établi, nous avons obtenu un code qui nous permet de contrôler les roues par le biais d'une fonction de haut niveau control_Mobilite(short vStraight, short vLeft, short vRight) [7] qui appelle PID(short desSpeed) et set_Vitesse_Roues(v1, v2). $vStraight$ est la vitesse à laquelle le robot avancerait tout droit, $vLeft$ à laquelle il va à gauche et $vRight$ à laquelle il va à droite.

2.3.3 • Couplage des deux pour le suivi de ligne

Nous avons reflèchi ici sur une méthode qui permet d'améliorer le traiment d'image et de se servir des informations de l'image (de la position) afin de prendre les décisions adéquats.

Précisément, l'image lissée a été divisée en 3 partie horizontales de part égale :

```
int height_third = smoothed.rows / 4;
Mat part1 = smoothed(Rect(0, 0, smoothed.cols, height_third));
Mat part2 = smoothed(Rect(0, height_third, smoothed.cols, height_third));
Mat part3 = smoothed(Rect(0, height_third * 2, smoothed.cols, height_third));
```

Figure 12 – Division de l'image lissée

Pour chaque partie, le contour rouge a été isolé (par des fonctions telles que $findContours()$ de la bibliothèque openCv) et le centre de gravité de chaque contour a été calculé. La cible du robot a été défini dans une première reflexion (tentative) comme l'abscisse du barycentre des 3 centres de gravité affectés respectivement (de la partie haute de l'image à la partie basse) de coéfficients $\alpha < \beta < \gamma$. Il nous a semblé naturel de prendre γ plus grand pour donner plus de poids au centre de gravité du contour le plus bas (c'est-à-dire le plus proche du robot) pour une meilleure précision du suivi.

Basé sur cet abscisse (la cible), nous avons injecté des valeurs constantes de $vStraight = 30$, $vLeft = +/-10$ ou 0, $vRight = +/-10$ ou 0 ans notre fonction controle_Mobilite(), selon si le centre de gravité est au tiers droite, au tiers gauche ou au milieu de l'écran. Ce critère de décision était pris en charge par une fonction dirDecision(struct Pos2D goal) qui prend en argument la cible. Avec, cette logique, il était difficile d'ajuster les bons coéfficients barycentriques car lors des tests, on observait toujours une perte de la ligne ou un écart assez énorme lors du suivi. On a dû repenser le calcul de la cible mais conservé les décisions à prendre. Ceci sera développer dans la 3.1 partie réalisations et ajustements

2.4 Points d'avancement pour la réalisation du SLAM

2.4.1 • Contrôle du servomoteur et utilisation du capteur infra-rouge

Comme dit plus haut, pour cette fonctionnalité, il fallait savoir transformer le capteur infrarouge qui avalue les distances des obstacles devant le robot grâce à la rotation du servo moteur.

Le servomoteur est un moteur de précision qui permet de positionner des éléments pour des couples de rotations relativement faibles. Sa plage de fonctionnement est un peu plus de 180 degrés. Ce type de moteur est donc parfait pour contrôler la tour située au centre de notre robot, sur laquelle nous avons fixé notre détecteur de

7. Code en annexe

distance à infrarouges (IR). Ce moteur est aussi contrôlé par modulation de largeur d'impulsion (PWM), mais interprété différemment ; ici, la modulation d'impulsion correspond à une position angulaire pour le moteur.

Nous avons d'abord initialisé la communication par I2C avec le moteur a commencé à la position 0. Puis programmé l'algorithme qui produit la rotation entre 0 (côté droit du robot) et 180 (côté gauche du robot). Ce code constitue une boucle infinie qui contient deux boucles for, l'une pour l'aller (0-180) et l'autre pour le retour (180-0). Pour chacune, nous parcourons du PWM minimal jusqu'au maximal (multiplié par 2 pour une meilleure résolution), puis nous envoyons l'impulsion à la fonction softPWM qui transforme, via un logiciel, l'instruction donnée en argument en PWM. Nous avons ajouté un délai à la fin de chaque changement de position pour laisser le temps au moteur d'atteindre la position demandée et permettre au détecteur IR de mesurer une distance. Dans l'implémentation finale, ce code est exécuté en parallèle (thread), pour que les délais de calculs n'affectent pas le reste du code.

Pour atténuer les effets de bord d'instabilité dus au parcours de la plage de valeurs possibles à partir d'une extrémité, nous avons choisi de balayer les 180 degrés à l'intérieur de la plage possible pour le moteur, afin d'éviter d'atteindre l'extrémité du mouvement du moteur. Cette solution simple a permis un parcours sans perturbations aux bords. Les codes sont en annexe.

En ce qui concerne le capteur infra-rouge, après branchement, configuration et lecture des valeurs, le protocole de communication était lui aussi en I2C. Pour lire les valeurs du signal capté, on a du implémenter le code suivant :

```
int dist_volt = wiringPiI2CReadReg16(fd, REG_IR);
```

Figure 13 – Lecture des valeurs données par le capteur infra-rouge

où fd = wiringPiI2CSetup(0x4) et REG_IR=0x20 désigne le type de valeur affiché (ici, une tension). Il était question ensuite de convertir cette valeur en terme de distance. pour ce faire, nous avons utilisé ce code fourni par la documentation de ce capteur :

```
float voltage_to_distance(int voltage) {
    // Use the inverse model of the Sharp 2Y0A02 IR sensor to convert voltage to distance.
    // Check the datasheet for more accurate values.
    float distance = 27.728 * pow((float)voltage / 1000.0, -1.2045);
    //distance = (distance/6.0) + 7.5;
    return distance;
}
```

Figure 14 – Conversion Tension-Distance

Le coefficient 27.728 a été trouvé expérimentantalement en utilisant des mesures à la règle graduée de la distance réelle entre le capteur et un obstacle puis comparaison avec les valeur affichées.

2.4.2 • Création et implémentation du noeud ROS créant le message LaserScan

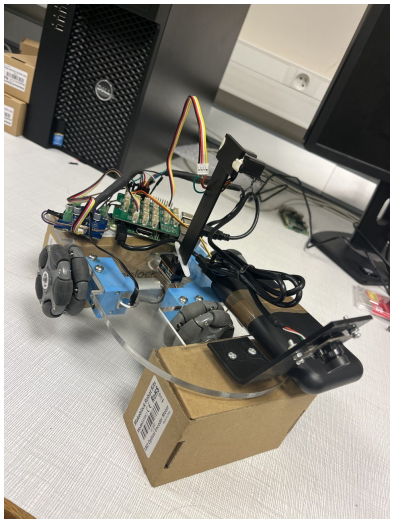
Après avoir pu receuillir les données du capteur infra-rouge en terme de distance et d'angle, il fallait juste créer un noeud ROSS Sensor_IR Node qui publie sur un topic '/scan' un message de type LaserScan. Dès lors que l'on a su les éléments qui rentrent dans ce type de message, il a été facile de le faire en remplissant les champs comme suit :

- header : header.stamp = rospy.Time/now()
- angle_min = 0 (ou -1.57)
- angle_max = 3.14 (ou 1.57)
- angle_increment 0.1745 (soit 10 degré)
- range_min : valeur minimale de distance mesurable par le capteur infrarouge.
- range_max : valeur maximale

— range[] : tableau qui contient les valeurs de distance d'obstacle à chaque angle donné. Dans notre cas (balayage de 180° avec un pas de 10°), il fait une taille de 18. Le code est en annexe.

3
RÉALISATIONS, DIFFICULTÉS RENCONTRÉES ET PERSPECTIVES

3.1 Les réalisations et ajustements



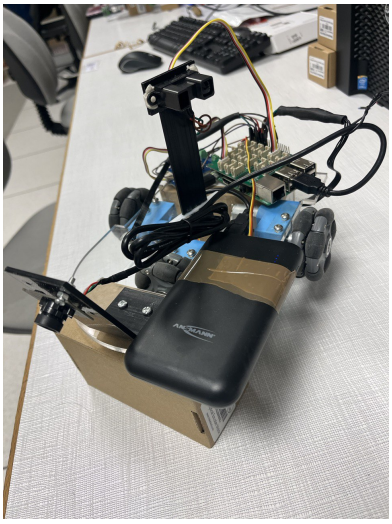


Figure 15 – Photos du Robot

Ci-dessus sont les images de notre robot.

Nous avons pu le doter de la fonctionnalité du suivi de ligne (qui constitue la première partie de notre projet) dans des delais qui n'ont pas suivi l'échéancier prévisionnel. Rappelons que notre objectif fixé sur le cahier de charges était d'avoir un suivi d'une précision de +/- 2cm. Lors des tous prémiers tests (à la $7^{ième}$ séance), nous avons constaté que :

— Pour des vitesses de plus de 40, la réponse du robot aux virages était assez lente et donc pas efficace de telle sorte qu'il perdait la ligne rouge à tous les virages.
— Le suivi de la ligne droite était très instable et donnait une allure de zig-zag assez prononcée.
— On était très loin de la performance attendue.

Pour ajuster le comportement du robot, nous avons décidé de conserver la vitesse d'avancement du robot de 30 et :

— Repensé le traitement d'image L'image étant divisée en 3 parties horizontales, au lieu de s'intéresser au barycentre des centres de gravité de chaque contour affectés de coéfficients, nous nous sommes juste intéressés au centre de gravité du compartiment inférieur ce qui nous paraissait beaucoup plus simple.
— Réduit la vitesse d'une des deux roues de 10 lorsque la décision d'aller à gauche ou à droite était prise.

Avec ces ajustements, le robot perdait difficilement la ligne mais, l'allure d'ensemble était lent et beaucoup plus instable.

Nous avons par la suite demandé au robot de ne s'intéresser qu'au centre de gravité du compartiment médian et remis les commande de vitesse comme avant le précédent ajustement. Là, on observe un déplacement assez stable et une réponse au virage un peu anticicipée ce qui rend le suivi peu précis. Toutefois, le robot perd difficilement la ligne rouge. On a évalué la précision à +/- 6cm.

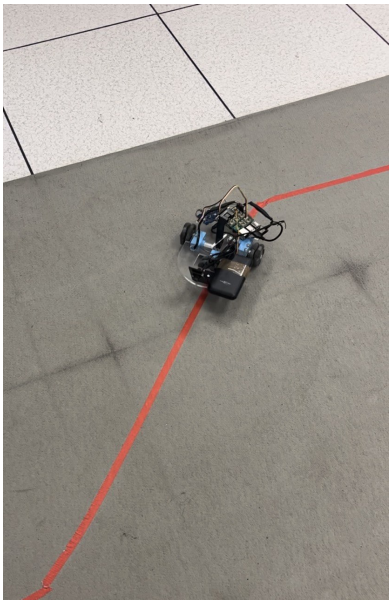


Figure 16 – Suivi de ligne

De ce qui est de la deuxième partie du projet portant sur la réalisation d'un robot qui inspecte un territoire en évitant des obstacles tout en générant la carte du milieu, nous avons pu munir le robot d'un capteur infrarouge et transformer le signal qu'il renvoyait en terme de distance et d'angle des obstacles dans un angle de balayage de 180° devant le robot. Il tourne à la vitesse angulaire $\omega = 1.27$ rad/s et par des mesures effectuées à la règle, la distance obtenue est satisfaisante et a une précision de +/- 0.5cm. Nous avons aussi pu créer un noeud ROS **Sensor_IR Node** qui transforme ces données en un message de type LaserScan et les publie sur le topic '/scan'. Nous avons aussi pu faire tourner ce noeud ROS.

De plus, Nous avons installé sur notre Raspberry Pi avec l'aide du moniteur Ousmane, tous les éléments nécessaires pour réaliser le SLAM, et pour visualiser le modèle du robot sur une interface graphique qui lit les informations provenant des nœuds correspondants. Cela comprend la position et les valeurs du capteur IR envoyées sous forme de messages LaserScan. Malheureusement, les derniers détails d'implémentation et de débogage ne sont pas encore terminés.

3.2 Les difficultés rencontrées

Les difficultés rencontrées lors de la réalisation de ce projet étaient nombreause.

Tout d'abord, nous avons eu des difficultés organisationnelles. Il était difficile de respecter l'échéancier prévisionnel du fait, peut-être d'une gestion pas assez optimale de notre temps, mais aussi du fait qu'un projet aussi audacieux nécéssite plus de temps pour être réalisé.

Ensuite, nous avons eu des difficultés à prendre en main le matériel car la recherche de la bonne documentation était assez laborieuse.

Nous avons aussi rencontré plusieurs problèmes lors de l'installation de programmes et de bibliothèques.

Il n'était pas facile pour nous, d'améliorer la précision du suivi de ligne sur la base de l'architecture du code choisi.

Finalement, Le passage à une autre carte Raspberry Pi avec un système différent et des installations à effectuer nous a empêché de travailler sur l'implémentation de ROS pour notre projet ce qui a également été un point faible pour nous.

3.3 Perspectives

3.3.1 • Pour le suivi de ligne

Perfectionner le suivi de ligne en :

— Nous avons constaté une grande antipation du robot empêchant ainsi un suivi exacte de la ligne rouge. Nous pensons améliorer la qualité du traitement de l'image et du choix de la cible à atteindre. Par exemple, faire en sorte que le robot s'intéresse au différents centres de gravités par échelon : d'abord au centre de gravité du contour du milieu de l'image pour prendre ses décisions. S'il ne voit pas de contour, il s'intéressera à la partie inférieure de l'image.

— Nous avons constaté aussi que la qualité du virage n'est pas adaptée à la position de la cible à atteindre de telle sorte que le robot peut, lors du virage, perdre la ligne rouge. Pour résoudre ce problème, on pourrait faire un asservissement de la direction du robot en calculant les commandes à envoyer sur chacune des roues en fonction de l'erreur entre l'abscisse de la cible et celui du centre de la caméra.

3.3.2 • Pour le SLAM

Pour plusieurs raisons telles que ses bibliothèques et outils, son architecture de communication, son interopérabilité (supporte plusieurs lagages C++, Python, ...), ses outils de visualisation (Rviz), ses packages pour la navigation, on continuera cette implémentation en ROs. Ici, on pourrait continuer la création des noeuds ROS et l'implémentation du SLAM. Sans entrer dans le détail, le graphe de communication ROS sera potentiellement composé des noeuds suivants :

1. **Noeud Odometrie Node** :
   — Responsabilité : calcul de l'odométrie des roues à partir des valeurs de l'encodeur.
   — Publie sur : Topic '/odom' (message 'nav_msgs/Odometry').
2. **Noeud robot_pose_ekf Node** :
   — Responsabilité : Fusion des données d'odométrie avec d'autres capteurs.
   — Abonne à : Topics tels que '/odom' et d'autres capteurs.
   — Publie sur : Topic '/odom' (message 'nav_msgs/Odometry').
3. **Noeud Sensor_IR Node** :
   — Responsabilité : Acquisition des données du capteur Infrarouge.
   — Publie sur : Topic '/scan' (message 'sensor_msgs/LaserScan').
4. **Noeud SLAM Node** :
   — Responsabilité : Gestion du SLAM
   — Abonne à : Topic '/scan' (message 'sensor_msgs/LaserScan'), Topic '/odom' (message 'nav_msgs/Odometry').
   — Publie sur : Topic '/map' (message 'nav_msgs/OccupancyGrid').
5. **Noeud PID Controller Node (Roue Gauche)** :

— Responsabilité : Contrôle en asservissement PID de la roue gauche du robot à partir de la vitesse courante.
— Abonne à : '/odom'
— Publie sur : Topic de commande de puissance du moteur de la roue gauche (par exemple, '/left_wheel_power_cmd').

6. **Noeud PID Controller Node (Roue Droite)** :
   — Responsabilité : Contrôle en asservissement PID de la roue gauche du robot à partir de la vitesse courante.
   — Abonne à : '/odom'
   — Publie sur : Topic de commande de puissance du moteur de la roue gauche (par exemple, '/right_wheel_power_cmd').
7. **Noeud Mobility Controller Node** :
   — Responsabilité : Contrôle de la mobilité du robot, implémentation de la navigation par évitement d'obstacle.
   — Abonne à : '/scan', /left_wheel_power_cmd' et '/right_wheel_power_cmd'.

Dans le noeud **Odometrie Node**, la mise à jour des paramètres odométriques se fera par les formules suivantes :

— **Position** (x, y)
La distance parcourue par chacune des roues : $d1 = \pi D * \Delta N1$ et $d2 = \pi D * \Delta N2$ où D est le diamètre et ΔN la variatious de valeur de l'encodeur pendant Δt.
La distance totale parcourue par le robot : $d = (d1 + d2)/2$
Les déplacements en x et y : $\Delta x = d\cos(\theta)$ et $\Delta y = d\sin(\theta)$. Où θ est l'ancienne orientation du robot.
La nouvelle position : $x = x\_prev + \Delta x$ et $y = y\_prev + \Delta y$.
— **Orientation (θ)**
Variation d'angle : $\Delta\theta = (d2 - d1)/width$. Où $width$ est la distance séparant les deux roues.
Nouvel angle : $\theta = theta\_prev + \Delta\theta$
— **Vitesse linéaire (v)** :
$v = d/\Delta t$. Où Δt est la variation de temps entre deux mesures de l'encodeur.
— **Vitesse angulaire (w)** :
$w = (d2 - d1)/(width * \Delta t)$.

Il convient de noter que le noeud **robot_pose_ekf Node** sera facultatif. Il représente le fait de prendre en compte les informations d'autres capteurs comme le GPS pour mettre à jour les paramètres odométriques avec une meilleure précision. Aussi, le noeud **Mobility Controller Node** utilisera les valeurs de distance des obstacles recueillies sur le topic '/scan' pour implémenter un algorithme d'évitement d'obstacle. Son abonnement aux topics /left_wheel_power_cmd' et '/right_wheel_power_cmd' lui permettra d'obtenir les puissances à envoyer aux roues.

Aussi, au lieu d'écrire nous même l'algorithme de navigation dans le noeud Mobility_Controller, on pourrait utiliser le package **move_base de ROS** est souvent utilisé pour cela. Il implémente un pipeline de planification de chemin qui comprend la planification de la trajectoire globale, la transformation de cette trajectoire en une trajectoire locale qui tient compte des obstacles immédiats, et la génération de commandes de mouvement pour le robot.

4
LA QUESTION THÉORIQUE DU SLAM EN EXTÉRIEUR

4.1 Présentation générale du concept de SLAM

Le SLAM (localisation et cartographie simultanées en français), fait référence à la capacité d'un système autonome, tel qu'un robot ou un véhicule autonome, à se localiser dans un environnement inconnu tout en créant une carte de cet environnement en temps réel. Il peut être divisé en deux étapes principales : la localisation et la cartographie.

Le processus de SLAM repose sur l'utilisation de capteurs tels que des caméras, des lidars (télémètres lasers), des radars ou des capteurs d'inertie (gyroscopes et accéléromètres) pour recueillir des données sur l'environnement et la position du robot. Les algorithmes de SLAM utilisent ensuite ces données pour estimer la position du robot et mettre à jour la carte de l'environnement.

Lorsqu'il est fait en extérieur, il présente des défis amis plusieurs techniques permettent de résoudre ces problèmes et modéliser l'incertitude associée à la position estimée du robot et à la carte créée.

Finalement, le SLAM est utilisé dans de nombreux domaines, notamment la robotique mobile, la réalité augmentée, la réalité virtuelle et la cartographie en intérieur. Il a des applications pratiques telles que la navigation autonome des véhicules, l'exploration de zones dangereuses ou inaccessibles pour les humains, la modélisation 3D d'environnements complexes, la surveillance et bien d'autres.

4.2 Les défis du SLAM en extérieur

Le SLAM en extérieur présente plusieurs défis spécifiques en raison des caractéristiques uniques de l'environnement extérieur. Les plus courants sont :

- **Variations de luminosité** : Les changements de luminosité naturelle en extérieur, tels que les variations dues au soleil, aux nuages, aux ombres et aux réflexions, peuvent affecter la qualité des images capturées par les capteurs visuels. Cela peut rendre la détection et le suivi des caractéristiques de l'environnement plus difficiles, ce qui peut entraîner des erreurs de localisation et de cartographie.
- **Objets en mouvement** : En extérieur, il y a souvent la présence d'objets en mouvement tels que des véhicules, des piétons, des animaux, etc. Ces mouvements peuvent introduire des perturbations et des incohérences dans les données des capteurs, en particulier pour les capteurs visuels.
- **Environnement non structuré** : Contrairement aux environnements intérieurs qui sont souvent plus structurés avec des murs, des portes et des objets fixes, les environnements extérieurs peuvent être plus vastes, ouverts et présentent une diversité d'objets et de surfaces. La présence de grandes zones ouvertes, de paysages naturels, d'arbres, de broussailles, etc., nécessite des techniques spécifiques pour extraire et modéliser les caractéristiques de l'environnement.
- **Dimensionnalité et taille de la cartographie** : En extérieur, la cartographie peut s'étendre sur de plus grandes distances et être plus vaste en comparaison aux environnements intérieurs. Cela peut poser des défis en termes de gestion de la dimensionnalité, du stockage des données et du traitement en temps réel.
- **Conditions météorologiques** : Les conditions météorologiques défavorables telles que la pluie, la neige, le brouillard ou des tempêtes peuvent affecter la qualité des données des capteurs. Les capteurs visuels, en particulier, peuvent être perturbés par des conditions météorologiques imprévisibles, ce qui peut entraîner des erreurs de localisation et de cartographie.

Pour relever ces défis, il est essentiel de choisir des techniques et approches appropriées.

4.3 Approches et techniques utilisées pour le SLAM extérieur

Pour résoudre les défis du SLAM en extérieur, quelques approches et techniques peuvent être envisagées :

- **Utilisation de capteurs adaptés** : Choisir des capteurs robustes et adaptés à l'environnement extérieur, tels que des caméras avec une plage dynamique étendue, des capteurs LiDAR résistant aux conditions météorologiques, des IMU de haute qualité, des récepteurs GPS performants, etc. S'assurer que les capteurs sélectionnés sont capables de fournir des mesures précises et fiables dans des conditions extérieures variées.
- **Prétraitement des données** : Effectuer un prétraitement des données des capteurs pour réduire le bruit, les erreurs et les incohérences. Utiliser des techniques de filtrage, de lissage, de calibration, de compensation pour améliorer la qualité des mesures des capteurs.
- **Techniques de fusion de capteurs** : Intégrer les données provenant de différents capteurs (centrales inertielles, lidar, GPS, caméra,...) pour améliorer la précision et la robustesse du SLAM en extérieur. Utiliser des algorithmes de fusion de capteurs tels que les filtres de Kalman étendus (EKF), les filtres particules ou les approches de fusion basées sur le modèle pour combiner les mesures et obtenir une estimation plus précise de la position et de la cartographie.
- **Adaptation des algorithmes de SLAM** : Adapter les algorithmes de SLAM pour prendre en compte les spécificités de l'environnement extérieur. Par exemple, utiliser des méthodes de détection et de suivi des caractéristiques robustes pour gérer les variations de luminosité, incorporer des modèles de mouvement pour traiter les objets en mouvement, utiliser des techniques de segmentation de scène pour gérer l'environnement non structuré etc.
- **Techniques de gestion de la mémoire** : Pour gérer la dimensionnalité et la taille de la cartographie en extérieur, utiliser des techniques de cartographie incrémentale ou de gestion de la mémoire pour limiter la quantité de données traitées en temps réel. Cela peut inclure la compression de données, la sélection des informations pertinentes et la gestion efficace de la mémoire.

4.4 Méthode de localisation en extérieur : Exemple de la fusion de capteurs

Lorsqu'il s'agit d'améliorer la précision de la pose en utilisant la fusion de capteurs, un algorithme couramment utilisé est le **filtre de Kalman étendu** (Extended Kalman Filter, EKF) [3].

Le filtre de Kalman étendu est une méthode d'estimation récursive qui combine les mesures des capteurs avec un modèle de mouvement pour estimer l'état du système. Il est particulièrement adapté pour les systèmes non linéaires, ce qui en fait un choix populaire pour la fusion de capteurs dans le contexte du SLAM.

L'algorithme du filtre de Kalman étendu suit les étapes suivantes :

- **Prédiction** : Utilise le modèle de mouvement pour estimer l'état futur du système.
- **Correction et mise à jour** : Compare les mesures des capteurs avec les prédictions de l'état pour estimer l'erreur de mesure et corriger l'estimation de l'état puis mettre à jour l'estimation de l'état et sa covariance en utilisant les résultats de la correction.

L'algorithme du filtre de Kalman étendu nécessite une modélisation mathématique du système, y compris les équations de mouvement et les équations de mesure des capteurs. Il utilise des techniques de linéarisation pour traiter les systèmes non linéaires en utilisant des approximations linéaires autour de l'estimation actuelle de l'état.

Concrètement, l'état du robot à l'instant t (par exemple position x_t et vitesse $\dot{x}_t$), représenté par le vecteur $\mathbf{X}_t$ peut être modélisée par l'équation d'évolution :

$$\mathbf{X}_{t+1} = \mathbf{A}\mathbf{X}_t + \mathbf{u}_t$$

où $\mathbf{u}_t$ est le bruit d'évolution supposé gaussien, centré et de matrice de covariance $\mathbf{P}_t$. $\mathbf{u}_t$ représente les perturbations de l'état du robot liées aux imperfections environnementales. Par ailleurs, les instruments de mesure odométrique (encodeurs des roues, ...) sont entachés d'incertitudes sur la mesure de la position et de la vitesse. Cette incertitude est donnée par une matrice $\mathbf{\Sigma}_t$ dont les parties diagonales $\mathbf{\Sigma}_t^{xx}$, $\mathbf{\Sigma}_t^{\dot{x}\dot{x}}$ représentent l'incertitude sur la mesure de la position et de la vitesse respectivement et ceux non diagonaux $\mathbf{\Sigma}_t^{\dot{x}x}$, $\mathbf{\Sigma}_t^{x\dot{x}}$ représentent la correlation entre la mesure du bruit de la position et de la vitesse.

On observe par ailleurs la mesure Y_t de position et de vitesse donnée par d'autres capteurs (par exemple un GPS). Cette mesure évolue suivant le modèle :

$$\mathbf{Y}_t = \mathbf{H}\mathbf{X}_t + \mathbf{v}_t$$

où $\mathbf{v}_t$ est le bruit de mesure supposé gaussien, centré et de matrice de covariance $\mathbf{Q}_t$

A l'étape de la **prédiction**, on trouve que l'état du robot est : $\mathbf{X}_{t+1}^- = \mathbf{A}\mathbf{X}_t$ avec une incertitude de : $\mathbf{\Sigma}_{t+1}^- = \mathbf{P}_t + \mathbf{A}\mathbf{\Sigma}_t\mathbf{A}^\intercal$.

A l'étape de **correction**, on considère la mesure donnée par l'observation $\mathbf{Y}_{t+1}$ (avec un incertitude $\mathbf{S}_{t+1} = \mathbf{Q}_t + \mathbf{H}\mathbf{\Sigma}_{t+1}^-\mathbf{H}^\intercal$) et on compare cette mesure à la prédiction $\mathbf{Y}_{t+1}^- = \mathbf{H}\mathbf{X}_{t+1}^-$ par la différence $\mathbf{\Delta Y_{t+1}} = \mathbf{Y}_{t+1} - \mathbf{H}\mathbf{X}_{t+1}^-$. En définissant le **gain de Kalman** $\mathbf{K}_{t+1} = \mathbf{\Sigma}_{t+1}^-\mathbf{H}\mathbf{S}_{t+1}^{-1}$, l'état du robot devient :

$$\mathbf{X}_{t+1}^+ = \mathbf{X}_{t+1}^- + \mathbf{K}_{t+1}\mathbf{\Delta Y_{t+1}} \text{ avec une incertitude de : } \mathbf{\Sigma}_{t+1}^+ = (I - \mathbf{K}_{t+1}\mathbf{H})\mathbf{\Sigma}_{t+1}^-$$

Le **gain de Kalman** introduit représente le niveau de fiabilité de la mesure prédite ou mesurée par les capteurs. Un gain élevé indique que la mesure des capteurs est très fiable et doit être privilégiée par rapport à la prédiction du modèle. Un gain de Kalman faible, en revanche, indique que la prédiction du modèle est plus fiable et doit être privilégiée.

4.5 Technique de cartographie en extérieur : Exemple de la méthode basée sur les points d'intérêt

Dans cette méthode, **"feature-based SLAM"** [4], des points d'intérêt ou des caractéristiques significatives de l'environnement sont extraits à partir des données sensorielles (comme des images ou des scans LiDAR). Ces points d'intérêt peuvent être des coins, des bords, des textures distinctives ou d'autres structures visuelles ou géométriques qui se démarquent de leur environnement. Les étapes sont :

- **Détection des points d'intérêt** : Les points d'intérêt sont détectés dans les données sensorielles, en utilisant des algorithmes tels que le détecteur de coins Harris (pour la détection des coins en étudiant les fortes variations d'intensité dans toutes les directions), le détecteur de points de Moravec ou le détecteur de caractéristiques SIFT (Scale-Invariant Feature Transform pour des points d'intérêt plus généraux par filtrage multi-échelle et calcul des différences gaussiennes DoG).
- **Extraction des descripteurs** : Une fois que les points d'intérêt ont été détectés, des descripteurs sont calculés pour chacun de ces points. Les descripteurs capturent des informations distinctives sur les caractéristiques locales entourant chaque point d'intérêt (un cercle ou un carré autour du point). Un descripteur est généralement un vecteur numérique qui représente l'histogramme des orientations de la région d'intérêt. Ce vecteur contient des informations sur la distribution des orientations des gradients dans la région, ce qui permet de capturer les caractéristiques locales distinctives
- **Association de caractéristiques** : Lorsque le robot se déplace dans l'environnement, les nouvelles observations de points d'intérêt sont associées aux points d'intérêt précédemment détectés dans la carte. Le point clé ici est la recherche des correspondandances en comparant la similarité des descripteurs par des mesures telles que la distance euclidienne. Cette étape permet de suivre les mouvements du robot et de mettre à jour la carte existante.

— **Estimation de la pose** : En utilisant les correspondances entre les points d'intérêt de la carte et les nouvelles observations, la pose du robot (position et orientation) peut être estimée par des algorithmes d'estimation de pose tels que la méthode des moindres carrés, le RANSAC (Random Sample Consensus). Cela permet au robot de se localiser dans l'environnement.
— **Mise à jour de la carte** : Une fois que la pose est estimée, les nouvelles informations sont utilisées pour mettre à jour la carte existante, en ajoutant de nouveaux points d'intérêt et en ajustant les relations spatiales entre les caractéristiques existantes.

La technique présente plusieurs avantages dans le domaine du SLAM :

— **Réduction de la complexité** : Plutôt que de cartographier l'intégralité de l'environnement, la méthode des points d'intérêt se concentre uniquement sur les caractéristiques significatives et distinctives de l'environnement. Cela permet de réduire la quantité de données à traiter, la complexité algorithmique et les besoins en stockage, ce qui facilite la mise en œuvre du SLAM.
— **Robustesse aux changements d'environnement** : Les points d'intérêt sont généralement choisis pour leur stabilité et leur capacité à être détectés de manière cohérente dans différentes conditions d'éclairage, de perspective et de pose. Cela confère à la méthode des points d'intérêt une certaine robustesse face aux variations d'environnement, aux changements d'éclairage et aux occlusions partielles.
— **Efficacité computationnelle** : L'extraction et la correspondance des points d'intérêt peuvent être réalisées efficacement grâce à des algorithmes bien établis. Étant donné que seuls les points d'intérêt sont considérés, la méthode des points d'intérêt peut offrir une exécution plus rapide que les méthodes de cartographie complète de l'environnement.
— **Information sémantique** : En se concentrant sur les points d'intérêt, la méthode permet de capturer des informations sémantiques sur l'environnement. Les points d'intérêt peuvent être associés à des objets ou des structures spécifiques, ce qui peut faciliter la compréhension de la scène et l'interprétation de la carte générée.
— **Adaptabilité à différents capteurs** : La méthode des points d'intérêt est compatible avec différents types de capteurs, tels que les caméras, les capteurs LiDAR et les capteurs inertiels. Cela permet une certaine flexibilité dans le choix des capteurs en fonction des contraintes et des besoins spécifiques de l'application.

Cependant, il est important de noter que la méthode des points d'intérêt a également certaines limites. Par exemple, elle peut être moins adaptée aux environnements très homogènes ou présentant peu de caractéristiques distinctives. De plus, la précision de la carte dépend de la densité et de la qualité des points d'intérêt détectés.

CONCLUSION

Au cours de ce projet de robotique, nous avons travaillé en étroite collaboration pour concevoir un robot doté de deux fonctionnalités distinctes : le suivi de ligne rouge et la réalisation de la carte d'un milieu en navigant au sein de celui-ci.

L'accomplissement de la première fonctionnalité, le suivi de ligne rouge, a été une étape cruciale dans notre parcours. Grâce à une combinaison d'algorithme de vision par ordinateur, de capteurs et d'actions correctives, notre robot a été en mesure de suivre de manière autonome une ligne rouge préalablement définie. Malgré quelques imprécisions, nous sommes parvenus à développer un système qui réagit. Cela nous a permis de mieux comprendre les concepts fondamentaux de la robotique, tels que la perception, la planification et le contrôle, ainsi que les challenges liés à la navigation autonome.

La deuxième fonctionnalité, le SLAM, constitue une avancée importante dans notre projet, bien que nous n'ayons pu la réaliser pleinement à ce stade. La création du nœud ROS pour le capteur IR nous a permis d'acquérir des données précises sur l'environnement. Cette étape est essentielle pour permettre à notre robot de se localiser avec précision et de prendre des décisions éclairées lors de ses déplacements. Bien que nous n'ayons pas encore intégré pleinement le SLAM dans notre robot, ce travail préliminaire jette les bases solides pour des développements futurs.

En travaillant ensemble sur ce projet, nous avons acquis des compétences techniques précieuses en matière de robotique. Nous avons appris à manipuler des capteurs (tels que les caméras et les capteurs IR, à exploiter des algorithmes de traitement d'image, d'asservissement et à comprendre les principes du SLAM et les défits qu'il présente en extérieur. Nous avons également approfondi notre compréhension du fonctionnement du framework ROS et de son utilisation pour la communication entre les différents composants d'un système robotique.

Ce projet nous a également permis de développer des compétences en travail d'équipe et en gestion de projet. Nous avons dû diviser les tâches de manière efficace, communiquer régulièrement sur l'avancement de nos travaux et prendre des décisions collaboratives pour surmonter les défis rencontrés. Ces compétences seront précieuses pour nos projets futurs, qu'ils soient liés à la robotique ou à d'autres domaines.

RÉFÉRENCES

[1] *https* : *//docs.opencv.org*/4.2.0/*da*/*d*97/*tutorial_threshold_inRange.html*

[2] *https* : *//wiki.seeedstudio.com*/*Grove* − *I*2*C_Motor_Driver* − *TB*6612*FNG*/

[3] Sebastian THRUN, Wolfram BURGARD, Dieter FOX. Probabilistic Robotics. pp. 248-260

[4] Rana Azzam, Tarek Taha, Shoudong Huang and Yahya Zweiri. Feature-based Visual Simultaneous Localization and Mapping : A Survey

ANNEXE

```c
#include <wiringPi.h>
#include <wiringPiI2C.h>

// convertion entier en hexadécimal
unsigned short per_to_hex (unsigned short s)
{
    int d = s*255/100;
    return (unsigned short) d;
}
```

Figure 17 – Fonction Per_To_Hexa()

```c
void set_Vitesse_Roues(short vM1, short vM2)
{
    //vM1= -vM1;
    if(vM1>100) vM1 = 100;
    if(vM1<-100) vM1 = -100;
    if(vM2>100) vM2 = 100;
    if(vM2<-100) vM2 = -100;
    if(vM1>=0) wiringPiI2CWriteReg16(fd, CW_REG, per_to_hex(abs(vM1))<<8);
    if(vM1<0) wiringPiI2CWriteReg16(fd, CCW_REG, per_to_hex(abs(vM1))<<8);
    if(vM2>=0) wiringPiI2CWriteReg16(fd, CW_REG, (per_to_hex(abs(vM2))<<8) | 1);
    if(vM2<0) wiringPiI2CWriteReg16(fd, CCW_REG, (per_to_hex(abs(vM2))<<8) | 1);
}
```

Figure 18 – Fonction Set_Vitesse_Roues()

```c
short PID1(short desSpeed)
{
    static int last_count1 = 0;
    static unsigned long last_time1 = 0;
    unsigned long time1 = millis();
    static float error1 = 0;
    static float last_error1 = 0;
    static float integral1 = 0;
    static float derivative1 = 0;
    static float cmd1 = 0;
    static short scmd1= 0;

    int count1 = encVal1/2;
    float speed1 = (count1 - last_count1) * 60.0 / ENCODER_PPR / ((float)(time1 - last_time1) / 1000.0);
    speed1 = speed1*100/160;

    error1 = desSpeed - speed1;
    printf("Speed1: %lf %lf %i\n", -speed1, error1, -count1);
    integral1 += error1;
    if (integral1 > INTEGRAL_CLAMP) {
        integral1 = INTEGRAL_CLAMP;
    } else if (integral1 < -INTEGRAL_CLAMP) {
        integral1 = -INTEGRAL_CLAMP;
    }
    derivative1 = error1 - last_error1;
    cmd1 = Kp*error1 + Ki*integral1 + Kd*derivative1;
    scmd1 = (short)cmd1;
    last_error1 = error1;
    last_count1 = count1;
    last_time1 = time1;
    return scmd1;
}
```

Figure 19 – Fonction PID()

```c
// fonction de controle des moteurs
void control_Roues(short vM1, short vM2)
{
    //Set des vitesses des roues apres PID
    short scmd1 = PID1(vM1);
    short scmd2 = PID2(vM2);
    set_Vitesse_Roues(scmd1,scmd2);
    delay(SAMPLE_TIME);
    printf("Speed: %i %i %i %i\n", -vM1, vM2, -scmd1, scmd2);
}

// fonction de controle du mouvement
void control_Mobilite(short vStraight, short vLeft, short vRight)
{
    control_Roues(-(vStraight + vLeft), vStraight + vRight);
}
```

Figure 20 – Fonction Controle_Mobilite()

```c
// Pin number based on WiringPi's pin numbering
(https://pinout.xyz/pinout/wiringpi#)
#define SERVO_PIN 26

// Servo motor pulse width constants (in microseconds)
#define SERVO_MIN_PULSE_WIDTH 5
#define SERVO_MAX_PULSE_WIDTH 22
#define SERVO_RANGE 2000

//IR
// Configuration of I2C addresses and registers for IR sensor
#define I2C_ADDRESS_IR 0x4
#define REG_IR 0x20
```

```c
void init_servo()
{
    // Initialize WiringPi
    if (wiringPiSetup() == -1) {
        printf("WiringPi setup failed. Exiting...\n");
        exit(-1);
    }
    pinMode(SERVO_PIN, OUTPUT);
    // Initialize Soft PWM on the servo pin
    if (softPwmCreate(SERVO_PIN, SERVO_MIN_PULSE_WIDTH, 200) != 0) {
        printf("PWM setup failed. Exiting...\n");
        exit(-1);
    }
    delay(1000); // Delay to allow the servo to move to the new position
}

int pulse_width_to_angle(int pulse_width)
{
    return (pulse_width-SERVO_MIN_PULSE_WIDTH) * (180 / (SERVO_MAX_PULSE_WIDTH-
SERVO_MIN_PULSE_WIDTH));
}
```

```
void servo_scan_180()
{
    while (done) {
        // Scan from 0 to 180 degrees and back
        for (pulse_width = SERVO_MIN_PULSE_WIDTH*2; pulse_width <=
SERVO_MAX_PULSE_WIDTH*2; pulse_width++) {
            if(!done){break;}
            softPwmWrite(SERVO_PIN, pulse_width/2);
            angle = pulse_width_to_angle(pulse_width/2);
            printf("angle = %i\n",angle);
            check_distance();
            printf("position of the obstacle
(%i,%i)\n",pos_obstacle.x,pos_obstacle.y);
            delay(100); // Delay to allow the servo to move to the new position
        }

        for (pulse_width = SERVO_MAX_PULSE_WIDTH*2+2; pulse_width >=
SERVO_MIN_PULSE_WIDTH*2+2; pulse_width--) {
            if(!done){break;}
            softPwmWrite(SERVO_PIN, pulse_width/2);
            angle = pulse_width_to_angle(pulse_width/2);
            printf("angle = %i\n",angle);
            check_distance();
            printf("position of the obstacle
(%i,%i)\n",pos_obstacle.x,pos_obstacle.y);
            delay(100); // Delay to allow the servo to move to the new position
        }
    }
}
```

Figure 21 – Fonction Servo_Scan180()

```
findContours(parts[1], contours, hierarchy, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE);

// Search for the largest contour
int partLargestContourIndex = -1;
double partLargestContourArea = 0;
for (size_t i = 0; i < contours.size(); i++){
    double area = contourArea(contours[i]);
    if (area > partLargestContourArea){
        partLargestContourArea = area;
        partLargestContourIndex = i;
    }
}

// If the largest contour in the current part is larger than any previous
if (partLargestContourArea > largestContourArea) {
    largestContourArea = partLargestContourArea;

    // Calculate the center of the largest contour
    Moments moments = cv::moments(contours[partLargestContourIndex]);
    centerX = moments.m10 / moments.m00;
    centerY = moments.m01 / moments.m00;
}
drawContours(contourImage, contours, partLargestContourIndex, Scalar(255, 255, 255), 2, 8, hierarchy, 0, Point());
```

Figure 22 – Détection du plus grand contour dans le deuxième compartiment (part[1]) de limage

```cpp
//------------------------------INCLUDES-------------------------------//
#include <stdio.h>
#include <wiringPi.h>
#include <softPwm.h>
#include <math.h>
#include <signal.h>
#include <wiringPiI2C.h>
#include <stdlib.h>
#include <string.h>

#include <chrono>
#include <functional>
#include <memory>
#include <string>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
#include "sensor_msgs/LaserScan.hpp"

using namespace std::chrono_literals;

//------------------------------CONFIGURATION
    ------------------------------//
// Pin number based on WiringPi's pin numbering (https://pinout.xyz/pinout/
    wiringpi#)
#define SERVO_PIN 26

// Servo motor pulse width constants (in microseconds)
#define SERVO_MIN_PULSE_WIDTH 5
#define SERVO_MAX_PULSE_WIDTH 22
#define SERVO_RANGE 2000

//IR
// Configuration of I2C addresses and registers for IR sensor
#define I2C_ADDRESS_IR 0x4
#define REG_IR 0x20

// Constants for Sharp 2Y0A02
#define LOWER_THRESHOLD  500
#define UPPER_THRESHOLD  2350
#define CRITICAL_DISTANCE 15

struct Pos2D
{
    int x;
    int y;
};

//------------------------------VARIABLES GLOBALES
    ------------------------------//
int angle;
int pulse_width;
```

```
50 int done = 1;
51
52 //IR
53 int fd;
54 bool critical = false;
55 float distance;
56 struct Pos2D pos_obstacle;
57
58 //------------------------------EN TETE DES FONCTIONS
      -------------------------------//
59 void stopcode(void);
60 int angle_to_pulse_width(int angle);
61 void init_servo();
62 void servo_scan_180(float[]& ranges);
63 struct Pos2D convert_angle_distance_to_2D_position(int angle, float distance
     );
64
65 //IR
66 void config_I2C(int I2C_ADDR);
67 float voltage_to_distance(int voltage);
68 void check_critical_distance(float distance);
69 void check_distance();
70
71
72 class IR_servo : public rclcpp::Node
73 {
74   public:
75     IR_servo()
76     : Node("IR_servo"), count_(0)
77     {
78         publisher_ = this->create_publisher<sensor_msgs::LaserScan>("scan",
              10);
79         timer_ = this->create_wall_timer(
80         500ms, std::bind(&IR_servo::timer_callback, this));
81     }
82
83   private:
84     void timer_callback()
85     {
86         auto message = sensor_msgs::LaserScan();
87         servo_scan_180();
88         message.angle_min = angle_min;
89         message.angle_max = angle_max;
90         message.angle_increment = angle_increment;
91         message.time_increment = time_increment;
92         message.scan_time = scan_time;
93         message.range_min = range_min;
94         message.range_max = range_max;
95         message.ranges = ranges;
96         RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.
              c_str());
97         publisher_->publish(message);
```

```
98      }
99      rclcpp::TimerBase::SharedPtr timer_;
100     rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
101     float angle_min = 0;
102     float angle_max = 180;
103     float angle_increment = 10;
104     float time_increment = 41;//ms
105     float scan_time = 2470;//ms
106     float range_min = 500;// to adjust and convert to meters
107     float range_max = 2350;// to adjust and convert to meters
108     float[] ranges[18];
109     size_t count_;
110 };
111
112 int main(int argc, char * argv[])
113 {
114     signal(SIGINT, (__sighandler_t)stopcode);
115     config_I2C(I2C_ADDRESS_IR);
116     init_servo();
117     rclcpp::init(argc, argv);
118     rclcpp::spin(std::make_shared<IR_servo>());
119     rclcpp::shutdown();
120     return 0;
121 }
122
123 //------------------------------FONCTIONS------------------------------//
124 void stopcode(void)
125 {
126     done = 0;
127     printf("The program ended\n");
128 }
129
130 // fonction de configuration du bus I2C
131 void config_I2C(int I2C_ADDR)
132 {
133     //configuration initiale
134     wiringPiSetup();
135     if (wiringPiSetup() == -1) {
136         printf("WiringPi setup failed. Exiting...\n");
137         exit(-1);
138         }
139     fd = wiringPiI2CSetup(I2C_ADDR);
140     if (fd == -1) {
141       printf("Erreur lors de l'ouverture du bus I2C.\n");
142       exit(-1);
143     }
144 }
145
146 void init_servo()
147 {
148     // Initialize WiringPi
149     if (wiringPiSetup() == -1) {
```

```
150         printf("WiringPi setup failed. Exiting...\n");
151         exit(-1);
152         }
153     pinMode(SERVO_PIN, OUTPUT);
154     // Initialize Soft PWM on the servo pin
155     if (softPwmCreate(SERVO_PIN, SERVO_MIN_PULSE_WIDTH, 200) != 0) {
156         printf("PWM setup failed. Exiting...\n");
157         exit(-1);
158     }
159     delay(1000); // Delay to allow the servo to move to the new position
160 }
161
162 int pulse_width_to_angle(int pulse_width)
163 {
164     return (pulse_width-SERVO_MIN_PULSE_WIDTH) * (180 / (
        SERVO_MAX_PULSE_WIDTH-SERVO_MIN_PULSE_WIDTH));
165 }
166
167 struct Pos2D convert_angle_distance_to_2D_position(int angle, float distance
    )
168 {
169     struct Pos2D position;
170
171     // Convert angle to radians
172     float radians = (float)angle * (M_PI / 180.0f);
173
174     // Calculate x and y position
175     position.x = (int)(distance * cosf(radians));
176     position.y = (int)(distance * sinf(radians));
177
178     return position;
179 }
180
181 float voltage_to_distance(int voltage) {
182     // Use the inverse model of the Sharp 2Y0A02 IR sensor to convert
        voltage to distance.
183     // Check the datasheet for more accurate values.
184     float distance = 27.728 * pow((float)voltage / 1000.0, -1.2045);
185     //distance = (distance/6.0) + 7.5;
186     return distance;
187 }
188
189 void check_critical_distance(float distance) {
190     if (distance <= CRITICAL_DISTANCE) {
191         printf("Warning! Object too close (%.2f cm).\n", distance);
192         critical = true;
193     }
194 }
195
196 void check_distance()
197 {
198     pos_obstacle = {0, 0};
```

```
199     int dist_volt = wiringPiI2CReadReg16(fd, REG_IR);
200     printf("Valeur distance IR [mV] : %d\n", dist_volt);
201 
202     if (dist_volt >= LOWER_THRESHOLD && dist_volt <= UPPER_THRESHOLD) {
203         distance = voltage_to_distance(dist_volt);
204         printf("Distance: %.2f cm\n", distance);
205         pos_obstacle = convert_angle_distance_to_2D_position(angle, distance
                );
206 
207         check_critical_distance(distance);
208     } else {
209         printf("Voltage out of bounds. Ignored.\n");
210         pos_obstacle = {0, 0};
211     }
212 }
213 
214 void servo_scan_180(float[]& ranges)
215 {
216         while (done) {
217         float time = millis();
218         float dt;
219                 // Scan from 0 to 180 degrees and back
220                 for (pulse_width = SERVO_MIN_PULSE_WIDTH*2; pulse_width <=
                        SERVO_MAX_PULSE_WIDTH*2; pulse_width++) {
221             float time1 = millis();
222             float dt1;
223                         if(!done){break;}
224                         softPwmWrite(SERVO_PIN, pulse_width/2);
225                         angle = pulse_width_to_angle(pulse_width/2);
226                         printf("angle = %i\n",angle);
227             check_distance();
228             printf("position of the obstacle (%i,%i)\n",pos_obstacle.x,
                    pos_obstacle.y);
229                         delay(100); // Delay to allow the servo to move to
                            the new position
230             dt1 = millis() - time1;
231             printf("time1 = %lf\n",dt1);
232             ranges[(angle/10)-1] = distance;
233                 }
234 
235                 for (pulse_width = SERVO_MAX_PULSE_WIDTH*2+2; pulse_width >=
                        SERVO_MIN_PULSE_WIDTH*2+2; pulse_width--) {
236                         if(!done){break;}
237                         softPwmWrite(SERVO_PIN, pulse_width/2);
238                         angle = pulse_width_to_angle(pulse_width/2);
239                         printf("angle = %i\n",angle);
240             check_distance();
241             printf("position of the obstacle (%i,%i)\n",pos_obstacle.x,
                    pos_obstacle.y);
242                         delay(100); // Delay to allow the servo to move to
                            the new position
243                 }
```

```
244             dt = millis() - time;
245             printf("time = %lf\n",dt);
246         }
247 }
```