Étudiants :
Quentin CUNHA - Groupe A1

Professeur référent :
Yves Guinand

université PARIS-SACLAY
IUT DE CACHAN
Département GEii 2

# Rapport de projet

## Le LIDAR

# Remerciements :

Je tenais à remercier toute l'équipe enseignante sans qui ce projet n'aurait pu voir le jour.

Et en particulier, M. Guinand qui a su transmettre parfaitement ses connaissances et ses compétences durant tout le semestre. J'ai acquis de nouveaux savoirs en réalisant un démonstrateur de projet base mobile, de la programmation d'application en python et dans le traitement de donnée lié à un lidar.

Je souhaite aussi adresser mes remerciements à Mme Docarmo qui m'a accompagné et conseillée dans la rédaction de ce rapport.

Et enfin, je remercie M. Ardillier pour l'aide apportée à mon projet sur la conception de ma carte électronique.

# Table des matières

## Introduction :

La robotique est une plateforme fascinante pour explorer les frontières de l'innovation et de la technologie. Dans le cadre de la 31e édition des Rencontres de Robotique, notre équipe le CRAC (Club de Robotique des Amis de la Coupe) s'est engagée à préparer la voie vers l'infini en se concentrant sur notre destination première, Mars. Toutefois, une question vitale se pose : comment garantir la subsistance de notre équipage lors de cette mission interplanétaire ? C'est là que nos robots entrent en scène, avec pour objectif essentielle de préparer le terrain et de constituer des réserves alimentaires.

Notre tâche se décline en plusieurs volets indépendants, chacun représentant un défi unique pour nos robots. Du rempotage des plantes à l'orientation des panneaux solaires, en passant par la pollinisation et le rechargement des batteries, chaque action joue un rôle capital dans la réussite de notre expédition.

## Accès au Code Source sur GitHub

Le code source du projet est disponible sur GitHub, offrant une transparence totale et la possibilité pour d'autres étudiants de reproduire mon travail. Suivez les étapes ci-dessous pour accéder au code source :

- GitHub Repository: https://github.com/quent1-lab/CRAC-2024

### Clonage du Répertoire :

Pour obtenir une copie locale du code, utilisez la commande Git suivante dans votre terminal :

- Commande bash : « git clone https://github.com/quent1-lab/CRAC-2024 »

### Structure du Projet :

Voici une brève explication de la structure des répertoires :

- [Lidar] : Dossier comprenant les programmes pythons utile au fonctionnement du lidar
- [Lidarobot] : Dossier comprenant le projet « platformio » du démonstrateur
- [CAN] : Dossier comprenant les programmes correspondant à la communication CAN

### Exécution du Code :

Pour exécuter le code, assurez-vous de suivre les instructions spécifiques du fichier *README.md* situé à la racine du répertoire. De plus, une image de l'OS Raspberry Pi est présente avec les bonnes versions des logiciels utilisés.

Conseil : n'hésitez pas à ouvrir des issues sur GitHub en cas de questions ou de problèmes rencontrés lors de la reproduction du projet.

## Présentation de la Coupe :

La 31e édition des rencontres de robotique se déroule dans un environnement complexe représentant une serre autonome sur Mars. L'aire de jeu (*figure n°2*), un plan rectangulaire est parsemé de divers éléments tels que des plantes, des panneaux solaires, des pots en fer, des jardinières, des coccinelles robotiques et des zones de départ.

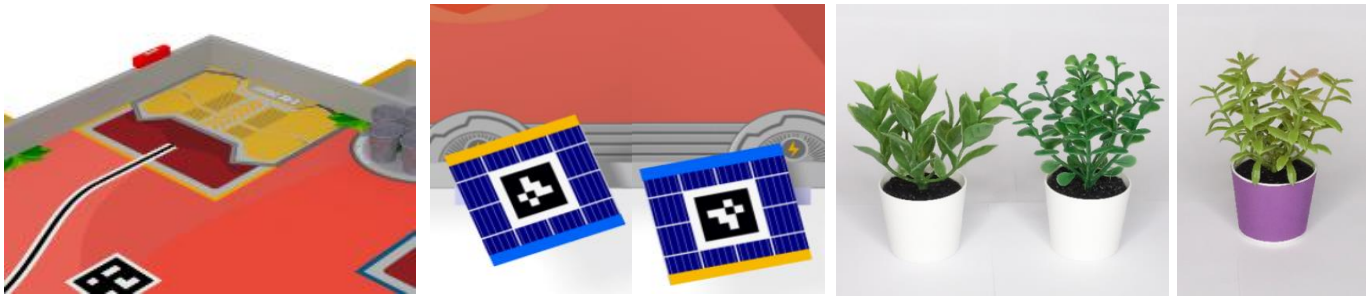

*Figure 1 : divers éléments présents sur le terrain*

## Missions principales

- **Rempoter les plantes et les mettre en culture :** les robots doivent transférer les plantes nouvellement arrivées de la Terre dans des pots adaptés et les placer dans des environnements de culture spécifiques pour améliorer leur productivité. Différents types de plantes ont des exigences environnementales spécifiques.

- **Orienter les panneaux solaires :** pour assurer un approvisionnement énergétique optimal, les robots doivent orienter les panneaux solaires vers le côté de la table associé à leur équipe.

- **Assurer la pollinisation des plantes :** les robots doivent relâcher des coccinelles (Petits Actionneurs Mobiles Indépendants - PAMI) pour polliniser les plantes. Les coccinelles doivent atteindre des plantes ou des pots spécifiques.

- **Retourner se recharger les batteries :** après avoir accompli leurs tâches, les robots doivent retourner dans leurs aires de recharge respectives pour recharger leurs batteries. Cependant, ils ne peuvent pas utiliser les zones de départ comme points de recharge.

- **Anticiper le futur rendement de la récolte :** les équipes doivent estimer le nombre de points qu'elles pensent réaliser pendant le match. Cette estimation peut être affichée de manière statique avant le match ou de manière dynamique pendant le match. Un bonus est accordé en fonction de la précision de l'estimation.

## Aire de jeu

L'aire de jeu (*figure n°2*) est un plan rectangulaire de 3 000 mm par 2 000 mm avec des bordures de 70 mm de haut et 22 mm d'épaisseur.

*Figure 2 : vue générale de l'aire de jeu*

## Les points

Chaque mission accomplie rapporte des points, mais des contraintes spécifiques doivent être respectées pour que les actions soient valides. Les points estimés en fin de match peuvent également contribuer à un bonus, l'objectif est de déterminer le nombre de points récoltés sur une partie automatiquement et de vérifier si ce score est juste.

## Les robots

Chaque équipe peut homologuer un robot principal et, éventuellement, un ou plusieurs PAMIs (Petits Actionneurs Mobiles Indépendants), avec des contraintes dimensionnelles spécifiques. Pour garantir une distinction claire entre les équipes, les robots et les PAMIs doivent être reconnaissables depuis le public.

Les dimensions des robots respectent des règles strictes (*figure n°3*), avec un périmètre maximal de 1200 mm au départ et de 1300 mm lorsqu'ils sont totalement déployés au cours du match. De plus, la hauteur du robot et des objets manipulés ne doit pas dépasser 350 mm pendant le match, à l'exception du bouton d'arrêt d'urgence, autorisé à atteindre 375 mm.

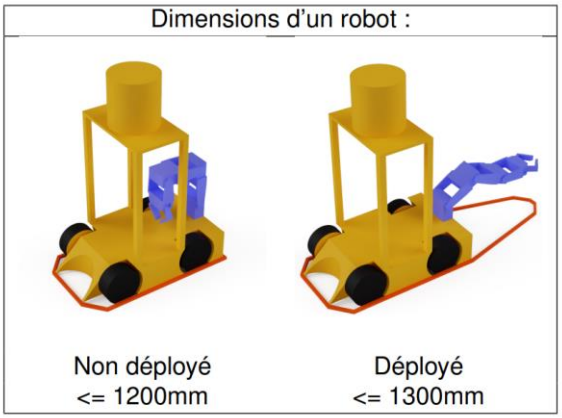


*Figure 3 : dimensions d'un robot*

# I. Le LIDAR

## 1. Etude de faisabilité

### a. La problématique

La difficulté du projet réside dans la détection précise et efficace d'un robot adverse au sein de l'aire de jeu. La capacité à localiser avec précision ces robots constitue un élément déterminant pour la réussite des différentes missions assignées à notre équipe. Sans une détection adéquate, notre robot risque de ne pas réagir de manière appropriée aux mouvements des adversaires, compromettant ainsi la réalisation des objectifs.

Le lidar sur lequel l'étude est portée (*figure n°4*) a pour référence : « RPLidar S1 ».


*Figure 4 : illustration RPLidar S1*

### b. Les solutions envisageables

Dans la phase initiale du projet, j'ai envisagé deux solutions principales pour le traitement des données du lidar : l'utilisation d'une ESP32 ou l'utilisation d'une Raspberry Pi.

- **Solution 1 : ESP32**

L'ESP32, est un microcontrôleur « puissant », et est une option à considérer pour traiter les données du lidar. Sa taille compacte et son faible coût en font un choix attractif. Cependant, sa capacité de traitement de données limité et sa capacité de stockage restreinte m'ont alarmé quant à sa capacité à gérer les tâches complexes associées au traitement du lidar et à la communication des informations en temps réel.

- **Solution 2 : Raspberry Pi**

La Raspberry Pi étant que micro-ordinateur, offre une puissance de calcul plus importante, une capacité de stockage étendue, et la possibilité d'utiliser des langages de programmation plus avancés. Cela le positionne comme une option plus robuste pour traiter les données en temps réel, avec la flexibilité nécessaire pour évoluer en fonction des exigences du projet.

### c. L'avantage de la Raspberry PI

La Raspberry Pi a été sélectionnée comme la solution privilégiée pour plusieurs raisons :

- **Puissance de Calcul**

La Raspberry Pi offre une puissance de calcul significativement supérieure à l'ESP32, permettant un traitement plus rapide et plus efficace des données du lidar. Cela s'avère crucial dans un environnement en temps réel où des décisions rapides et précises sont nécessaires.

- **Flexibilité et Langages de Programmation**

La Raspberry Pi offre une flexibilité en termes de langages de programmation. Cela permettra à l'équipe d'implémenter des algorithmes de traitement de données plus avancés en utilisant des langages tels que Python ou le langage C, offrant ainsi une plus grande souplesse dans le développement du logiciel de contrôle du robot.

- **Évolutivité**

La nature évolutive de la Raspberry Pi offre la possibilité d'ajouter des fonctionnalités supplémentaires à l'avenir sans compromettre les performances. Cela permettra d'adapter le robot à des missions plus complexes ou à des environnements changeants.

En conclusion, la Raspberry Pi s'avère être la solution idéale pour le traitement des données du lidar dans le cadre du projet de robotique. Sa puissance, sa capacité de stockage et sa flexibilité vont permettent d'optimiser le fonctionnement du robot, offrant ainsi une réponse plus efficace aux défis rencontrés sur le terrain de la coupe.

### d. Le langage de programmation

Le choix du langage de programmation revêt une importance capitale dans le traitement des données du lidar. Python a été sélectionné en raison de ses caractéristiques intrinsèques, particulièrement adaptées aux exigences du projet.

Python se distingue par sa syntaxe claire et sa simplicité, rendant le code facile à lire et à comprendre. Cette qualité est cruciale dans un projet complexe, où la collaboration entre membres de l'équipe peut grandement bénéficier d'une compréhension aisée du code. De plus, il est facile de déployer un programme python sur tous les OS existants.

Python offre un écosystème de bibliothèques riche et varié, notamment des bibliothèques spécialisées dans le traitement de données scientifiques et la manipulation de données spatiales. La communauté Python est dynamique et engagée, offrant un support continu ainsi que des ressources abondantes.

Python facilite la mise en œuvre du multitâche grâce à des modules tels que "multiprocessing ". Cette caractéristique permet à plusieurs tâches d'être exécutées simultanément, comme la navigation en fonction des données du lidar, et la communication avec d'autres systèmes.

La flexibilité de Python se révèle également dans l'intégration de communication sans fil. La gestion du Wifi, du Bluetooth et du bus CAN est simplifiée, permettant une communication fluide entre le robot et d'autres dispositifs.

### e. Le fonctionnement d'un lidar

Le lidar, acronyme de "Light Detection and Ranging," est un dispositif de télédétection qui utilise des faisceaux laser pour mesurer la distance entre l'émetteur-récepteur du lidar et une surface cible (*figure n°5*). Son principe de fonctionnement repose sur l'émission et la réception de pulsation lumineuse de type laser vers l'environnement, suivie de la mesure du temps écoulé entre l'émission et la réception du signal réfléchi. La vitesse de la lumière permet de calculer la distance avec une précision remarquable.

Le dispositif effectue un balayage rotatif, créant ainsi une image en deux dimensions de la topographie environnante. Les points capturés représentent les coordonnées spatiales et les distances dans le plan horizontal, fournissant une représentation précise des obstacles et de la structure à proximité.

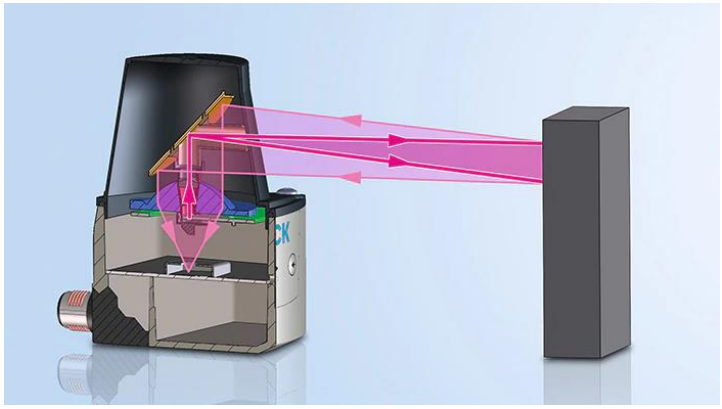

*Figure 5 : illustration du fonctionnement d'un capteur lidar*

En robotique, le lidar est souvent utilisé pour la perception de l'environnement, la navigation autonome, et la détection d'obstacles. La technologie lidar est indispensable dans divers domaines, allant des applications industrielles à la conduite autonome, offrant une perception précise et en temps réel de l'espace environnant.

### f. Le schéma synoptique et le logigramme

La conception du système repose sur la clarté des différentes tâches à accomplir. Le schéma synoptique (*figure n°6 et annexe n°*6) permet de comprendre la structure du projet ainsi que son fonctionnement.

Le lidar, monté sur la tête du robot, émet des faisceaux laser qui interagissent avec l'environnement, générant ainsi des données brutes. Ces données sont transmises au Raspberry Pi, où le programme de traitement s'engage. Une fois les données traitées, le Raspberry Pi envoie des ordres aux autres systèmes du robot via un bus CAN.

Le logigramme (*figure n°7 et annexe n°*7) détaille ensuite le flux logique du programme. Après l'initialisation de la connexion, le moteur du lidar est activé, déclenchant le processus de balayage. Le traitement des données intervient par la suite tout en s'adaptant et prenant en compte les composantes externes.

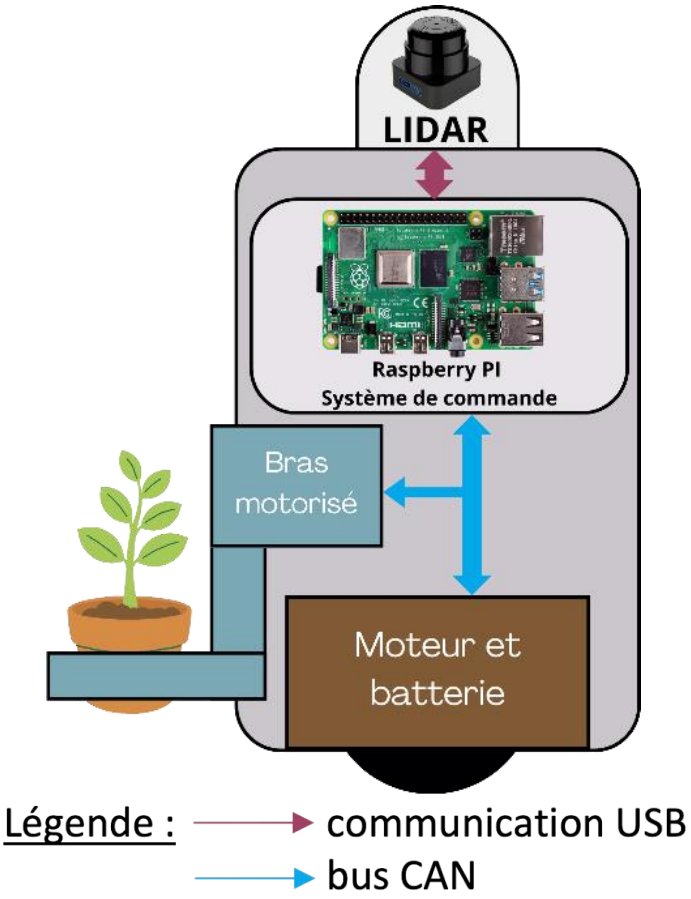


*Figure 6 : schéma synoptique du projet lidar*

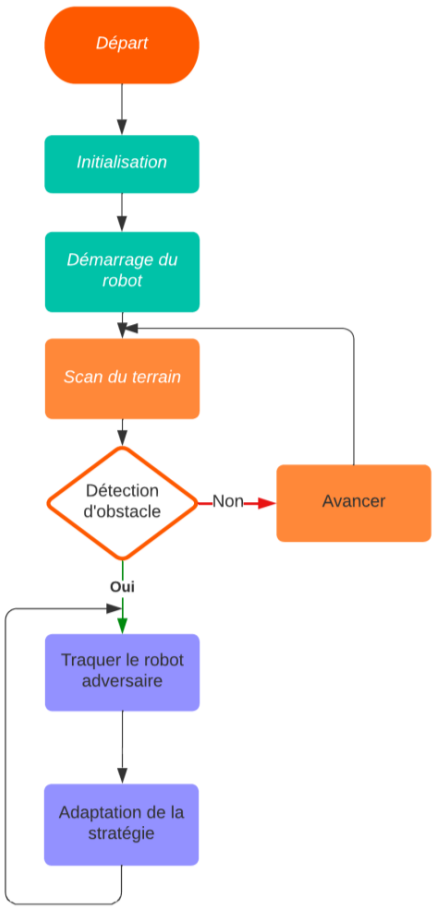


*Figure 7 : logigramme représentant le fonctionnement du projet lidar*

### g. Le planning

Le projet lidar a été lancé en septembre avec une vision des étapes à franchir, reflétée dans un diagramme de Gantt (*figure n°8 et annexe n°8*). Chaque phase planifiée, a contribué à la réalisation du lidar, des premières étapes de recherche et de conception jusqu'à la phase de développement et de tests. Les jalons ont été respectés, et le projet atteindra sa conclusion début janvier.

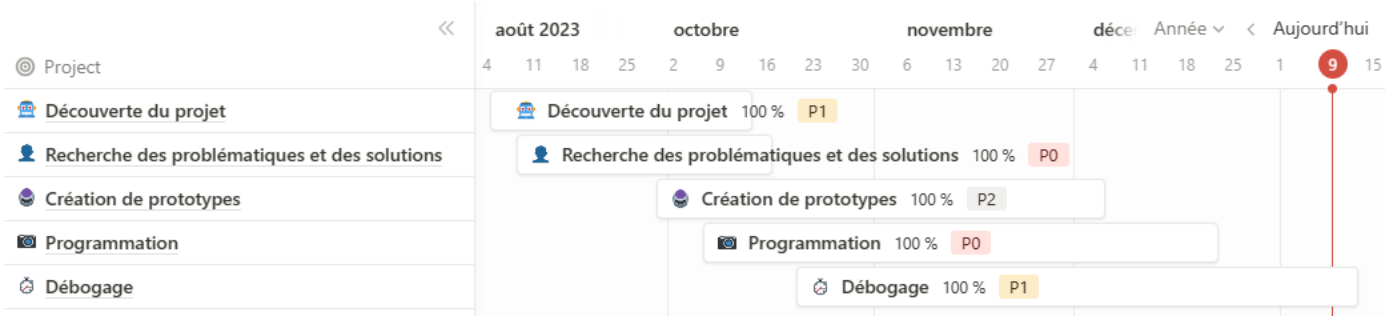


*Figure 8 : diagramme de Gantt du projet Lidar*

## 2. Le traitement des données

Le développement du logiciel pour l'intégration sur le robot se réalise sur deux supports. Dans un premier temps, l'élaboration d'un programme fiable en phase de débogage sur un ordinateur. Puis l'intégration du système sur le robot grâce à la Raspberry Pi.

### a. La bibliothèque compatible avec le lidar

Pour le traitement des données du lidar, j'ai exploité la bibliothèque "rplidar" qui propose une solution simple pour interagir avec les capteurs de télémétrie RPLidar offrant une communication efficace et précise grâce à ses fonctions intégrées. Cependant, quelques adaptations ont été nécessaires en raison de son origine conçue pour des versions plus récentes que celle du matériel sur le projet.

- **Vitesse de communication**

La connexion se fait via une communication série, avec deux options de vitesse de communication : 115 200 ou 256 000 bauds (*figure n°9*). Lors de l'initialisation de la connexion, la valeur par défaut de la bibliothèque est de 115 200. Or après plusieurs tests, il s'est avéré que cette vitesse n'était pas compatible avec l'installation sans pouvoir en déterminer clairement la cause. Il a fallu modifier la vitesse de communication à 256 000 (*figure n°10 et annexe n°10*).


*Figure 9 : image de l'adaptateur du lidar montrant les vitesses en bauds*

```
86  class RPLidar(object):
87      '''Class for communicating with RPLidar rangefinder scanners'''
88
89      _serial_port = None  #: serial port connection
90      port = ''  #: Serial port name, e.g. /dev/ttyUSB0
91      timeout = 1  #: Serial port timeout
92      motor = False  #: Is motor running?
93      baudrate = 256000  #: Baudrate for serial port
94
95      def __init__(self, port, baudrate=256000, timeout=1, logger=None):
```

*Figure 10 : lignes de code de l'initialisation de la classe RPLidar dans rplidar.py*

- **Démarrage du moteur**

Des erreurs sont survenues lors de l'exécution d'un programme de test, indiquant un problème de déchiffrement de la trame « Descriptor length mismatch ». Ces problèmes étaient liés à la fonction de démarrage du moteur, initialement configurée pour des modèles de lidar plus récents.

Pour résoudre ce problème, la commande « SCAN_BYTE » (*figure n°11*) doit être envoyée au lidar. Cette commande permet de démarrer le moteur, ainsi que la mesure de l'environnement.

```
40  SCAN_BYTE = b'\x20'
```

*Figure 11 : ligne de code corresponde à l'ordre de lancement du scan*

Cette solution généré une nouvelle anomalie, le programme cherchant à récupérer trop rapidement les données du lidar afin de les traitées sans que le moteur ait atteint sa vitesse nominale, aucunes données n'étaient envoyées et le programme s'arrêter brusquement. Pour cela, il fallut simplement ajouter une temporisation de 2 secondes (*figure n°12)* avant de traiter les données.

```
144  def start_motor(self):
145      '''Starts sensor motor'''
146      self.logger.info('Starting motor')
147      cmd = SCAN_BYTE
148      self._send_cmd(cmd)
149      time.sleep(2)
```

*Figure 12 : lignes de code de la fonction de lancement du scan*

- **Appel de la fonction « start_motor »**

Cette fonction ne fait pas que démarrer le moteur, mais lance aussi le scan, il a fallu définir le bon endroit pour l'appeler. Dans le cas où cette fonction serait appelée juste après la connexion à l'ordinateur, des erreurs vont survenir. Par exemple, si la fonction permettant de récupérer l'état de santé du lidar est appelée. Car, en utilisant la commande « SCAN_BYTE », le lidar renvoie directement les données brutes des mesures et n'attend pas de recevoir d'autres ordres que de s'arrêter.

Le meilleur endroit où appeler cette fonction, est dans la fonction « iter_measurments », pendant l'initialisation, juste avant la boucle principale.

Une fois tous ces points modifiés, toutes les conditions sont réunies pour que les données du lidar soient récupérées et traitées.

Toutefois, il est possible que le lidar ne s'arrête pas correctement en fin de programme, ou qu'une anomalie apparaisse lors de la connexion série entre le lidar et l'ordinateur. Ces erreurs devront être détectées dans le programme principal par l'utilisation de la méthode python « try » et de la classe « RPLidarException » (*figure n°13)* de la bibliothèque « rplidar » qui hérite de la méthode python « Exception ». Bien pris en compte dans le logiciel, cette fonction permettra de lever les exceptions liées à l'usage du lidar sans interrompre le fonctionnement.

```
63  class RPLidarException(Exception):
64      '''Basic exception class for RPLidar'''
```

*Figure 13 : lignes de code de la classe RPLidarException*

### b. Programme de test du lidar

Afin de vérifier le bon fonctionnement des modifications apportées, le lidar a été testé avec un programme basique. Ce dernier crée le lien entre le lidar et l'ordinateur, puis lance le scan de la pièce en affichant les points mesurés. Ce programme d'exemple « TestLidar.py » permet de comprendre la façon dont les données d'angle et de distance sont récupérées pour être traitées. Ce programme fonctionne aussi bien sur Linux que sur Windows, il suffit d'indiquer le bon port USB correspondant au lidar (*figure n°14)*. Une autre méthode de connexion permet plus facilement de se connecter via le numéro de série du port du

lidar. Pour cette méthode, il faut s'assurer qu'aucun autre appareil ne soit branché à l'ordinateur (Bluetooth compris) avant de l'exécuter, et enfin de noter le numéro de série affiché sur le terminal.

```
14  #Récupère le numéro de série du port disponible
15  serial_number = [port.serial_number for port in serial.tools.list_ports.comports()][0]
16  print("Numéro de série :", serial_number) #Affiche le numéro de série du port disponible
17  #PORT_NAME = [port.name for port in serial.tools.list_ports.comports() if serial_number in port.serial_number][0]
18
19  PORT_NAME = 'COM5'  # À modifier en fonction du port utilisé ; sur linux : /dev/ttyUSBx (x = 0, 1, 2, ...)
```

*Figure 14 : lignes de code correspondant au choix du port du lidar*

Ensuite, on retrouve dans le programme la boucle de récupération des mesures (*figure n°15)*. La distance mesurée est remplie dans un tableau « scan_data » de 360 cases, chaque case correspondant à l'angle de mesure. La fonction « min([359, floor(angle)]) », permet de ne jamais dépasser la taille du tableau.

Enfin, le tableau de mesure est donné en argument à la fonction « process_data » (*annexe n°27)*, qui permet d'afficher les points mesurés grâce à une bibliothèque graphique.

```
41  for scan in lidar.iter_scans():
42      for (_, angle, distance) in scan:
43          scan_data[min([359, floor(angle)])] = distance
44      process_data(scan_data)
```

*Figure 15 : lignes de code de la boucle de récupération des données du lidar*

En observant les images ci-dessous, on remarque qu'avec une simple installation du lidar dans une pièce, on obtient 360 points représentant la pièce scannée avec les meubles (*figure n°16 et figure n°17)*.

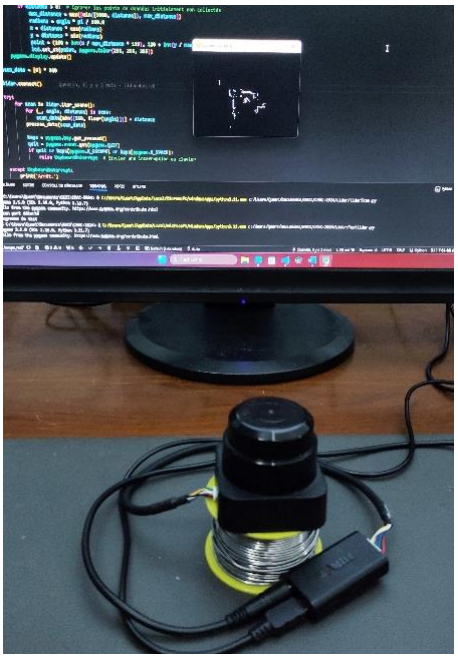

*Figure 16 : premier test du lidar avec affichage*


*Figure 17 : capture d'écran de la fenêtre d'affichage des mesures du lidar*

### c. Programme de traitement des données du lidar

Une fois l'acquisition des points obtenue, le traitement des données fiabilise la localisation des obstacles dans toutes les circonstances. Que ça soit dans la robustesse du programme, dans son intégration sur Raspberry pi ou encore dans la pertinence des données traitées.

Lorsque le robot circule sur l'aire de jeu, le lidar se déplace avec lui, et donc les points mesurés se « déplace » aussi. Pour cela, avant d'exploiter les données, il faut transformer chaque point mesuré afin de compenser le déplacement du robot pour qu'un point immobile sur l'aire de jeu garde les mêmes coordonnées par exemple.

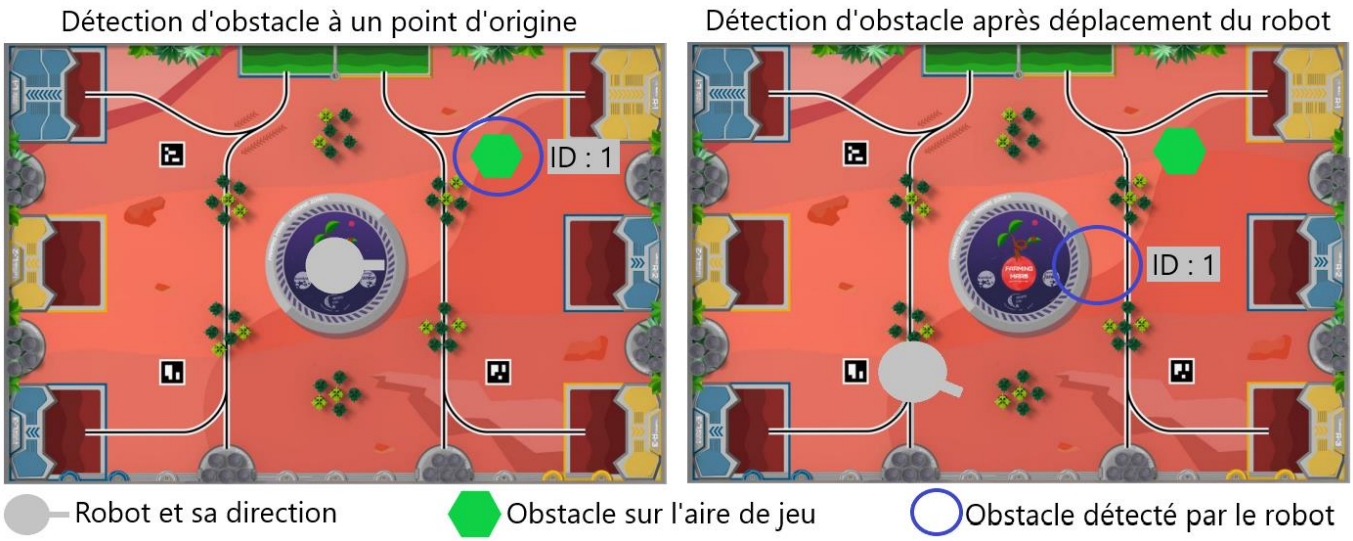


*Figure 18 : illustration de détection obstacle sans transformation des coordonnées*

Sur l'illustration ci-dessus (*figure n°18)*, on remarque dans le cas où le robot se serait déplacé, les coordonnées de l'objets détecté ne sont pas les mêmes que l'objets réel qui lui est resté immobile.

Pour résoudre ce problème, tous les points mesurés vont donc être repositionnés en fonction de l'emplacement du robot sur l'aire jeu (*figure n°19)*. Cette étape consiste à calculer une position (x, y) d'un point grâce à sa distance et son angle en fonction des coordonnées (x, y, thêta) du robot.

```
484  new_angle = angle - self.ROBOT_ANGLE
485  new_angle %= 360
486  if new_angle < 0:
487      new_angle += 360
488
489  if distance != 0:
490      x = distance * math.cos(math.radians(new_angle)) + self.ROBOT.x
491      y = distance * math.sin(math.radians(new_angle)) + self.ROBOT.y
```

*Figure 19 : lignes de code représentant le calcul des nouvelles coordonnées d'un point de mesure*

Dans ce code, le nouvel angle calculé est ramené dans l'intervalle [0, 360] afin de calculer les coordonnées du point.

### d. Unité et saturation

Concernant l'unité de mesure, le programme est configuré pour fonctionner en millimètres (mm), assurant ainsi une précision fine des mesures. Cette unité a été sélectionnée en tenant compte de la résolution du lidar.

En termes de saturation des points de mesure, la conception du programme prend en compte la taille du terrain comme limite (*figure n°20)*. La saturation est étroitement liée à la zone d'opérations du robot, garantissant que seuls les objets situés à l'intérieur du périmètre défini par le terrain sont pris en compte. Cette approche est essentielle pour éviter que le robot ne détecte des objets situés à l'extérieur du champ opérationnel, minimisant ainsi les risques d'interférences inutiles. De plus, la saturation est définie avec une marge de sécurité tenant compte de la taille du robot. Ainsi, le programme optimise la détection d'obstacles, assurant une réactivité adéquate tout en limitant les fausses alertes.


*Figure 20 : illustration de la saturation de détection d'obstacle*

### e. Exploitation des données traitées

Chaque objet détecté est représenté par une instance dans la classe « Objet », caractérisée par son identifiant unique, ses coordonnées « (x, y) », sa taille et des informations sur son mouvement.

La fonction « detect_object » (*annexe n°28*) parcourt les scans afin d'identifier les objets présents dans l'environnement. Elle prend en compte la proximité des points détectés, exclut ceux déjà associés à des objets connus, et calcule les coordonnées moyennes pondérées des points autour de chaque objet. De plus, elle détermine la taille des objets en fonction des écarts entre les points.

La gestion des objets est effectuée dans une liste, permettant ainsi le suivi dynamique d'objets au fil du temps. Chaque objet est mis à jour avec sa nouvelle position, taille, direction et vitesse lors de chaque itération. Cette approche offre une représentation en temps réel des objets détectés (*figure n°21)*.

En complément des fonctionnalités de suivi d'objets, le programme intègre une anticipation des trajectoires des robots sur le terrain. La méthode « trajectoires_anticipation » (*annexe n°29*) permet de simuler les déplacements futurs du robot actuel et du robot adverse, ainsi que de générer une trajectoire d'évitement en cas de collision potentielle.

La simulation utilise les positions et vitesses actuelles des robots pour anticiper leurs déplacements sur une durée spécifiée (duree_anticipation). À chaque pas de temps (pas_temps), les nouvelles positions sont calculées, et les trajectoires des robots sont mises à jour.

Si la distance entre les robots devient inférieure à une valeur critique (distance_securite), indiquant un risque de collision, le programme propose une trajectoire d'évitement.


*Figure 21 : représentation de l'exploitation des données*

Actuellement, toutes les fonctions d'affichage sont intégrées dans le programme de traitement des données afin de faciliter le développement du projet. À terme, le fonctionnement des programmes se feront en multitâches et sur différents appareils. Cela permettra d'augmenter la capacité de calcul sur la Raspberry pi en déportant la supervision.

### f. Programme de simulation

La fonction « programme_simulation » joue un rôle essentiel dans le développement et le test du système de détection et de suivi d'objets du robot. Grâce à la fonction « valeur_de_test » des valeurs simulé du lidar permettent de le remplacer en attribuant des distances aléatoires à des angles spécifiques. Ces

valeurs servent de données d'entrée pour évaluer la performance du système sans la nécessité d'un environnement physique.

Le programme de simulation reproduit quasiment à l'identique le fonctionnement réel. En utilisant la classe « ComESP32 », il peut également communiquer avec un microcontrôleur ESP32, afin de tester la récupération des données du démonstrateur. L'interface graphique (*figure n°22)* est la même qu'en usage classique et affiche le robot, les objets détectés, ainsi que les trajectoires anticipées du robot en fonction des objets environnants.

L'interaction clavier permet de déplacer le robot virtuel, ce qui facilite le test de l'algorithme de détection et d'évitement d'obstacles dans divers scénarios. Cette simulation en temps réel offre un moyen pratique de vérifier la robustesse du système, d'ajuster les paramètres, et d'optimiser les algorithmes avant le déploiement sur le robot physique.


*Figure 22 : représentation du programme de simulation*

### g. L'interface graphique

L'emploi de PyGame pour l'interface graphique a simplifié le développement du logiciel. PyGame, est une bibliothèque de développement de jeux en Python, permettant la création d'une interface utilisateur interactive et intuitive. Les fonctionnalités graphiques de PyGame ont permis de l'intégrer plus facilement au projet.

# II. La Raspberry pi

La Raspberry Pi constitue un élément essentiel du projet, nécessitant une configuration minutieuse et une interface graphique adaptée à ses différentes phases d'utilisation.

## 1. La configuration

La configuration de la Raspberry Pi est une étape fondamentale pour assurer son bon fonctionnement. L'idéal est d'installer le système d'exploitation optimisé pour la Raspberry Pi. La procédure standard inclut le téléchargement de l'image, son installation sur une carte micro-SD, et l'initialisation du système. Cette étape est simplifiée par l'usage du logiciel « Raspberry Pi Imager ». L'OS recommandé est « Raspberry Pi OS (64-bit). Mais il est possible de choisir la version serveur, afin de retirer l'interface graphique et ainsi d'économiser les ressources de la machine.

Il est également crucial de configurer les paramètres réseau d'avance dans les options avancées du logiciel, assurant un accès à distance par défaut. De plus, une fois l'installation terminé, il est conseillé d'activer dans les paramètres de l'OS les options suivantes : serial ; wire ; VNC.

## 2. L'interface graphique

Durant la phase de développement, une interface graphique fonctionnelle facilite la programmation, le débogage, et la surveillance en temps réel. Cependant, une fois le programme développé et intégré au robot, l'interface graphique sur la Raspberry Pi devient superflue. Dans cette configuration, il est judicieux de déporter l'interface graphique sur un autre appareil, permettant une gestion à distance plus efficace, tout en libérant les ressources de la Raspberry Pi pour des tâches plus spécifiques à la robotique. Ce déplacement stratégique améliore les performances globales du système.

## 3. L'extension CAN

Étant donné que l'ensemble des systèmes du robot communique via un bus CAN, il était nécessaire d'ajouter une extension CAN à la Raspberry Pi. Le module « RS485 CAN HAT » (*figure n°23)* permet d'ajouter une communication CAN à la Raspberry Pi. Un code d'exemple est présent dans le projet Github. De plus, une petite carte électronique (*figure n°24 et figure n°25)* a été ajoutée afin de rendre compatible la connectique avec le système existant.


*Figure 23 : module RS485 CAN HAT*

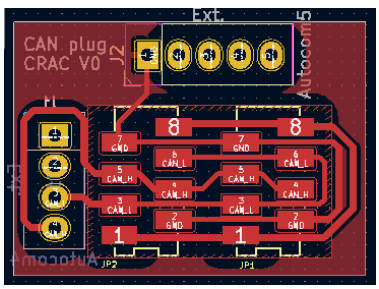


*Figure 24 : PCB de l'adaptateur du bus CAN*

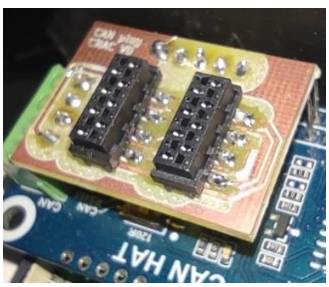


*Figure 25 : schéma électrique de l'adaptateur du bus CAN*

# III. Le démonstrateur

## 1. Le rôle du démonstrateur

Le démonstrateur (*figure n°*26) occupe une place importante dans le développement et la validation des fonctions du lidar, notamment en lien avec les mécanismes d'odométrie de la base roulante. Sa conception vise à mettre en œuvre et à tester les algorithmes intégrés au programme au logiciel du lidar. Grâce à cette approche, le démonstrateur permet de valider l'efficacité et la précision des méthodes utilisées, contribuant ainsi à l'optimisation des performances globales du système.

La base roulante a été choisie sur une base Pololu, avec des motoréducteurs encodeurs. Ce sont des moteurs continus ayant une réduction de 1/100 et un encodeur intégré. Ce matériel a été choisi après divers tests et recherche, afin de correspondre au mieux aux besoins.

De plus, les fichiers 3D modélisés pour le démonstrateur sont mis à disposition dans le projet GitHub. Le programme C++ associé au démonstrateur est également accessible sur la plateforme GitHub, fournissant des bibliothèques complètes, écrites et testées pour l'utilisation des moteurs et des encodeurs.

Cependant, des améliorations peuvent être apportées, notamment en fabriquant une nouvelle carte électronique prenant mieux en compte les demandes énergétiques de la base roulante et de la Raspberry Pi, ainsi que dans l'ergonomie globale.

*Figure 26 : démonstrateur du projet lidar*

# IV. Conclusion

Le projet arrivant à son terme, tous les jalons attribués ont été validé. Certains points ont également été poussé au-delà des tâches prédéfinies afin de rendre un produit fonctionnel et optimisé. Le robot est désormais capable de se déplacer sur l'aire de jeu sans risquer de percuter un autre robot. De plus, le projet à permis d'apporter une base au développement du multitâche sur la Raspberry Pi.

En conclusion de ce projet, qui a représenté un investissement significatif sur une période de plusieurs mois, je retire une richesse d'enseignements issus des multiples défis et expériences rencontrés tout au long du projet. La gestion globale du projet, la recherche active de solutions et la résolution de problèmes variés ont constitué une plateforme d'apprentissage exceptionnelle.

Ce parcours m'a permis d'améliorer un éventail de connaissances telles que :

- la collaboration efficace en équipe ;
- la conception de cartes électroniques ;
- une maîtrise accrue des langages Arduino/C++ et Python ;
- l'affinement de compétences dans l'utilisation des logiciels Fusion360 et SolidWorks.

# Table des figures

# Annexes

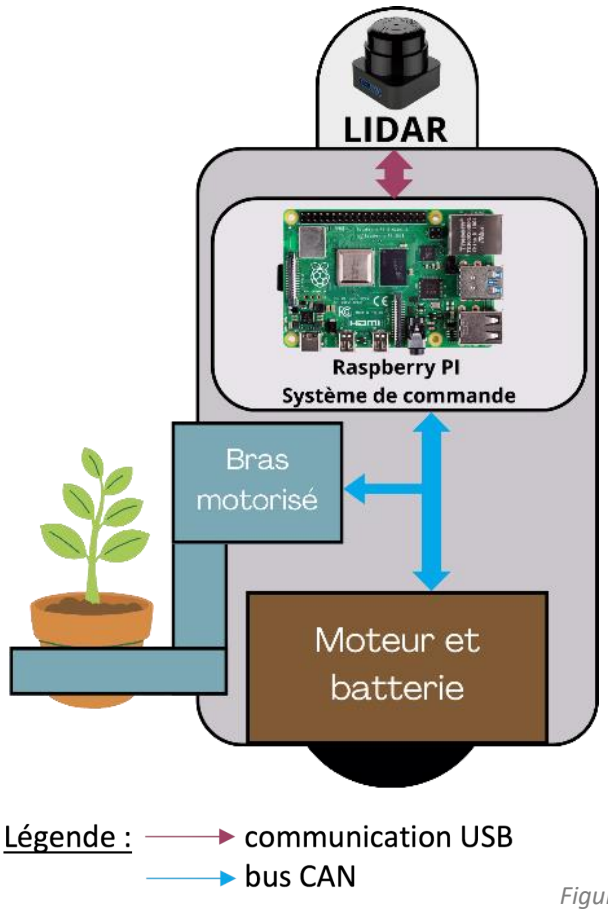


*Figure 6 : schéma synoptique du projet lidar*

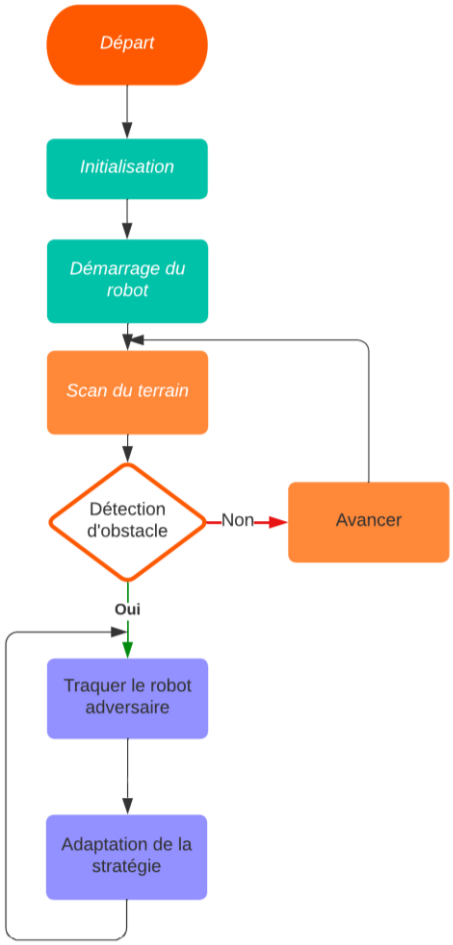


*Figure 7 : logigramme représentant le fonctionnement du projet lidar*

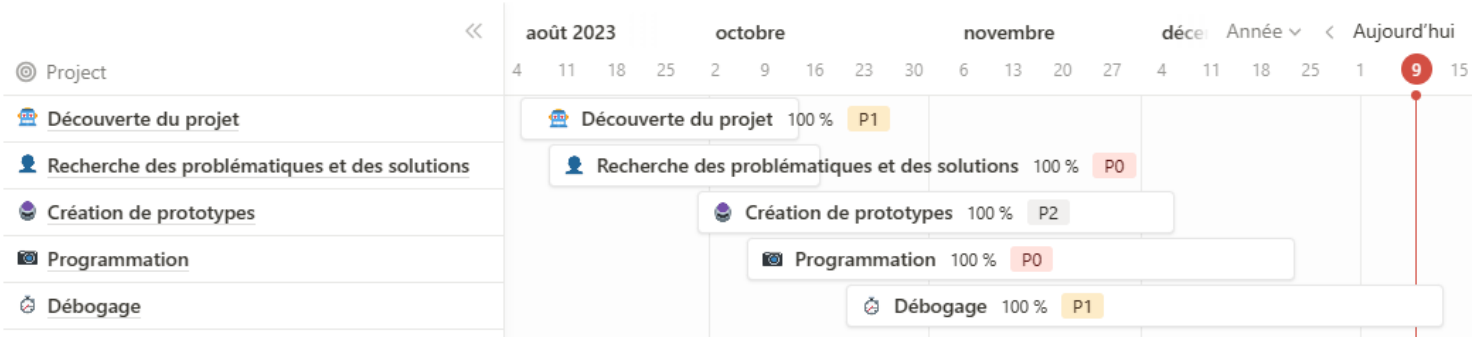


*Figure 8 : diagramme de Gantt du projet Lidar*

```
86  class RPLidar(object):
87      '''Class for communicating with RPLidar rangefinder scanners'''
88
89      _serial_port = None  #: serial port connection
90      port = ''  #: Serial port name, e.g. /dev/ttyUSB0
91      timeout = 1  #: Serial port timeout
92      motor = False  #: Is motor running?
93      baudrate = 256000  #: Baudrate for serial port
94
95      def __init__(self, port, baudrate=256000, timeout=1, logger=None):
```

*Figure 10 : lignes de code de l'initialisation de la classe RPLidar dans rplidar.py*

```
28  def process_data(data):
29      global max_distance
30      lcd.fill((0, 0, 0))
31      for angle in range(360):
32          distance = data[angle]
33          if distance > 0:  # Ignorer les points de données initialement non collectés
34              max_distance = max([min([5000, distance]), max_distance])
35              radians = angle * pi / 180.0
36              x = distance * cos(radians)
37              y = distance * sin(radians)
38              point = (160 + int(x / max_distance * 119), 120 + int(y / max_distance * 119))
39              lcd.set_at(point, pygame.Color(255, 255, 255))
40      pygame.display.update()
```

*Figure 27 : lignes de code de la fonction process_data*

```python
def detect_object(self, scan, max_iteration=2):
    iteration = 0
    while True:
        # Liste des points associés aux objets déjà trouvés
        points_objets_trouves = []
        for k in range(iteration):
            if k < len(self.objets):
                points_objets_trouves += self.objets[k].points

        # Sélectionne le point le plus proche du robot en excluant les points des objets déjà trouvés
        points_non_objets = [point for point in scan if point not in points_objets_trouves]
        if not points_non_objets:
            # Aucun point trouvé en dehors des objets, retourner None
            return None

        # Sélectionne le point le plus proche du robot
        point_proche = min(points_non_objets, key=lambda x: x[2])
        distance_objet = point_proche[2]
        angle_objet = point_proche[3]
        points_autour_objet = []

        # Sélectionne les points autour de l'objet en fonction des coordonnées (x, y) des points
        points_autour_objet = self.get_points_in_zone(points_non_objets, distance_objet, angle_objet)

        if not points_autour_objet or len(points_autour_objet) < 3:
            # Aucun point autour de l'objet ou pas assez de points, retourner None
            return None

        # Calcul des coordonnées moyennes pondérées des points autour de l'objet
        x = sum([point[0] for point in points_autour_objet]) / len(points_autour_objet)
        y = sum([point[1] for point in points_autour_objet]) / len(points_autour_objet)

        iteration += 1
        if iteration > max_iteration:
            return None

        # Calcul de la taille de l'objet
        x_min = min(points_autour_objet, key=lambda x: x[0])
        x_max = max(points_autour_objet, key=lambda x: x[0])
        y_min = min(points_autour_objet, key=lambda x: x[1])
        y_max = max(points_autour_objet, key=lambda x: x[1])
        taille = math.sqrt((x_max[0] - x_min[0])**2 + (y_max[1] - y_min[1])**2)

        # Seuil de détection d'un objet en mm
        SEUIL = 100  # en mm (distance que peut parcourir le robot entre deux scans)

        id_objet_existant = self.trouver_id_objet_existants(x, y, SEUIL)

        if id_objet_existant != None:
            # Si l'objet est déjà suivi, mettre à jour ses coordonnées
            self.objets[id_objet_existant - 1].update_position(x, y)
            self.objets[id_objet_existant - 1].taille = taille
            self.objets[id_objet_existant - 1].points = points_autour_objet
        else:
            # Si l'objet n'est pas déjà suivi, créer un nouvel objet
            self.id_compteur += 1
            nouvel_objet = Objet(self.id_compteur, x, y, taille)
            nouvel_objet.points = points_autour_objet
            self.objets.append(nouvel_objet)
```

*Figure 28 : ligne de code de la fonction detect_object*

```python
def trajectoires_anticipation(self, robot_actuel, robot_adverse, duree_anticipation=1.0, pas_temps=0.1, distance_securite=50):
    """
    Dessine les futures trajectoires des robots et la trajectoire d'évitement anticipée.

    :param robot_actuel: Objet représentant le robot actuel
    :param robot_adverse: Objet représentant le robot adverse
    :param duree_anticipation: Durée d'anticipation en secondes
    :param pas_temps: Pas de temps pour la simulation en secondes
    :param distance_securite: Distance de sécurité minimale entre les robots
    """
    # Copie des positions actuelles des robots
    x_actuel, y_actuel = robot_actuel.x, robot_actuel.y
    x_adverse, y_adverse = robot_adverse.x, robot_adverse.y

    # Copie des vitesses actuelles des robots
    _, vitesse_actuel = robot_actuel.get_direction_speed()
    _, vitesse_adverse = robot_adverse.get_direction_speed()

    # Liste pour stocker les points des trajectoires
    trajectoire_actuel = [(x_actuel, y_actuel)]
    trajectoire_adverse = [(x_adverse, y_adverse)]
    trajectoire_evitement = []

    # Simulation de mouvement pour anticiper la trajectoire future des robots
    for temps in range(int(duree_anticipation / pas_temps)):
        # Calcul des nouvelles positions des robots
        new_x_R, new_y_R = robot_actuel.calculate_dx_dy(robot_actuel.direction, vitesse_actuel, pas_temps)
        new_x_A, new_y_A = robot_adverse.calculate_dx_dy(robot_adverse.direction, vitesse_adverse, pas_temps)

        new_x_R += trajectoire_actuel[-1][0]
        new_y_R += trajectoire_actuel[-1][1]
        new_x_A += trajectoire_adverse[-1][0]
        new_y_A += trajectoire_adverse[-1][1]

        # Ajout des points aux trajectoires
        trajectoire_actuel.append((new_x_R, new_y_R))
        trajectoire_adverse.append((new_x_A, new_y_A))

        # Calcul de la distance entre les robots
        distance_entre_robots = math.sqrt((new_x_R - new_x_A)**2 + (new_y_R - new_y_A)**2)

        # Vérification de la collision anticipée
        if distance_entre_robots < distance_securite:
            # Proposer un chemin d'évitement
            trajectoire_evitement = [(x_actuel, y_actuel)]
            for temps_evitement in range(int(duree_anticipation / pas_temps)):
                # Choisir une direction d'évitement
                direction_evitement = (robot_actuel.direction + math.pi) % (2 * math.pi)

                # Simulation de mouvement pour l'évitement
                new_x_E, new_y_E = robot_actuel.calculate_dx_dy(direction_evitement, vitesse_actuel, pas_temps)

                new_x_E += trajectoire_evitement[-1][0]
                new_y_E += trajectoire_evitement[-1][1]

                # Ajout des points à la trajectoire d'évitement
                trajectoire_evitement.append((new_x_E, new_y_E))

            break
```

*Figure 29 : ligne de code de la fonction trajectoires_anticipation*