BERTRAND TURCK

FISE 2021

Spécialité Robotique

Rapport de projet de fin d'études

NAVIGATION AUTONOME D'UN ROBOT MOBILE EN PLEIN CHAMP

8 Mars - 30 Septembre 2021

ENCADRANTS :

ARNAUD LELEVÉ

MINH TU PHAM

RICHARD MOREAU

LABORATOIRE AMPÈRE

20 avenue Albert Einstein

69100 VILLEURBANNE

TUTEUR ÉCOLE :

LUC JAULIN

ENSTA BRETAGNE

2 rue François Verny

29200 BREST

Table des matières

Résumé

Le monde de l'agriculture est en pleine évolution. En effet, la demande de ressources augmente de plus en plus avec l'accroissement de la population. Aussi, il faut en même temps limiter ou encore arrêter l'utilisation de produits phytosanitaires pour préserver la biodiversité. Les produits phytosanitaires augmente le rendement des plantations, et ont été largement utilisés parce que le gain est bien plus important que le coût. Cependant, une alternative existe : la mécanisation, c'est-à-dire l'utilisation de systèmes mécaniques pour agir directement sur les parasites. L'utilisation d'humains peut être envisagée, mais la main d'œuvre est difficile à trouver à bas prix pour ce travail pénible [1]. Avec les avancées technologiques de ces dernières années, en particulier concernant la puissance de calcul, les algorithmes de *machine learning* et la capacité des batteries, il est maintenant possible d'envisager l'utilisation de la robotique autonome en agriculture. En effet, des réseaux de neurones bien entrainés sont capables de discerner parasites et autres pestes afin d'appliquer un traitement local adapté. La miniaturisation des systèmes permet aussi d'embarquer des systèmes toujours plus puissants dans des espaces restreints.

Le projet Greenshield vise à réduire l'utilisation de pesticides en développant un module robotisé embarqué sur un robot autonome pour combattre les pestes de cultures (invertébrés, maladies, mauvaises herbes). Le robot se déplacera automatiquement dans les cultures pour repérer les pestes, et les neutralisera avec un laser. Ce projet est mené par plusieurs laboratoires français et industrialisé par la startup Green Shield.

Le laboratoire Ampère est chargé du développement du robot mobile [2]. L'objectif de ce stage est de développer un système de navigation capable de se déplacer au-dessus de jeunes plants, en ne connaissant pas l'environnement et en adaptant la vitesse en fonction des besoins de reconnaissance et de destruction.

Pour que le robot soit capable de suivre les rangées de plants et de faire la transition entre les rangées, nous avons intégré les capteurs nécessaires à l'estimation de la position du robot dans son environnement. Une simulation a été créée afin de tester les algorithmes dans un environnement contrôlé. Un système de cartographie du champ en temps réel a été développé en parallèle d'un algorithme de détection des rangées afin de trouver un chemin à parcourir. Une implémentation avec le *middleware* ROS facilite l'échange de données entre les différentes parties du programme et les différents modules du robot. Aussi, une communication avec le système de destruction laser permet d'adapter la vitesse du robot en fonction des traitements à réaliser sur les pestes.

Abstract

The agricultural world is changing. In fact, the demand for resources is increasing with population's growth. In the meantime, it is necessary to limit or stop using phytosanitary products to preserve biodiversity. Phytosanitary products increase the yield of plantations and have been widely used because the gain is much more important than the cost. However, there is an alternative : mechanization, which is the use of mechanical systems to act directly on the pests. The use of humans can be considered, but labor is difficult to find at low cost for this tedious work [1]. With the technological achievement of the last few years, especially regarding computing power, machine learning algorithms and battery capacity, use of autonomous robotics in agriculture is now possible. Indeed, well-trained neural networks can be able to discern pests and other plagues in order to apply alocal treatment. The miniaturization of systems also allows to embed more and more powerful systems in small spaces.

The Greenshield project aims to reduce the use of pesticides by developing a robotic module embedded in an autonomous robot to fight crop pests (invertebrates, diseases, weeds). The robot will automatically move in the crops to locate the pests and neutralize them with a laser. This project is led by several French laboratories and industrialized by the startup Green Shield.

The Ampère laboratory is in charge of the development of the mobile robot [2]. The objective of this internship is to develop a navigation system able to move over young plants, not knowing the environment and adapting the speed according to the needs of recognition and destruction.

In order for the robot to be able to follow the rows of plants and to make the transition between rows, we integrated new sensors needed to estimate the position of the robot in its environment. A simulation was created to test the algorithms in a controlled environment. A real-time field mapping system was developed, in parallel with a row detection algorithm to find a path to follow. An implementation with the ROS middleware facilitate the data exchange between the different parts of the program and the different modules of the robot. Also, a communication with the laser destruction system allows to adapt the speed of the robot according to the treatments to be carried out on the plagues.

Remerciements

Avant de commencer ce rapport, je tiens à remercier toutes les personnes qui ont contribué de près ou de loin à mon stage et qui m'ont aidé lors de la rédaction de ce rapport.

En premier lieu, je tiens à remercier Mr Arnaud Lelevé, Mr Minh Tu Pham ainsi que Mr Richard Moreau, mes tuteurs de projets, pour m'avoir proposé ce stage et pour m'avoir apporté de précieux conseils pour mener à bien ma partie du projet.

Je tiens particulièrement à remercier Vivien Novales pour sa confiance et ses idées, qui m'ont permis d'avancer plus facilement sur mon travail. Je remercie aussi Duc Toan Nguyen et Javier Díaz Sempere pour leur présence et l'aide qu'ils m'ont apporté.

Je remercie également Xavier Durant pour sa gentillesse et le temps qu'il nous a consacré pour construire le carter du robot.

Enfin, je tiens à remercier tous les membres du laboratoire Ampère de l'INSA de Lyon pour leur accueil et leur bonne humeur.

1 Contexte du stage

Le stage s'inscrit dans le cadre du projet Greenshield [3] , un projet de recherche commun entre les laboratoires Ampère, BF2I, INL et FEMTO-st et la start-up Green Shield [4]. L'objectif est de développer un robot mobile pour combattre les pestes, en utilisant l'analyse spectrale pour la détection et un module laser pour la destruction. Chaque laboratoire travaille en parallèle sur un aspect précis du problème : détection des pucerons, destruction des pucerons et intégration des solutions dans un robot mobile. C'est pour contribuer à cette dernière tâche que ce stage a été proposé.

Le stage se déroule au sein du laboratoire Ampère, à l'INSA Lyon, situé sur le campus de la Doua à Villeurbanne (69100, France). Notre équipe est composée d'un ingénieur et de 3 stagiaires. Nous avons la charge d'intégrer toute la chaîne de destruction des pucerons de façon automatique dans un robot mobile. L'objectif est de pouvoir faire des tests sur une rangée de plantes infectées par des pucerons, et ainsi valider la chaîne de destruction. Une présentation aura lieu en septembre pour présenter les résultats du projet.

1.1 Fonctionnement du robot

Lors de son déplacement, le robot va se placer au-dessus d'une rangée de plantes, avec 2 roues de chaque côté pour que les plants passent dans le robot (voir figure 1.1).

Fig. 1.1 – Le robot se place au-dessus d'une rangée et les plantes passent dans le robot

Une caméra est placée à l'intérieur du robot, de façon à voir une plante à la fois. Un réseau de neurones récupère le flux vidéo, et détecte les pucerons. Lorsqu'il y a des pucerons à neutraliser, le robot s'arrête et la phase de destruction commence. Un système de miroirs permet de diriger le faisceau laser sur les pucerons . L'objectif est de les rendre infertiles afin de limiter leur propagation

dans le champ. Le laser de puissance utilisé n'émet pas dans le visible, il est donc couplé à un pointeur laser vert pour repérer l'endroit où le laser de puissance vise.

L'intérieur du robot est un environnement clos, pour faciliter la gestion de l'éclairage. En effet, il faut un éclairage constant pour aider à la reconnaissance d'images. Aussi, le faisceau laser est asservi en repérant le pointeur laser visible sur le flux vidéo, il est donc nécessaire d'éteindre l'éclairage interne et d'être dans le noir pour le détecter.

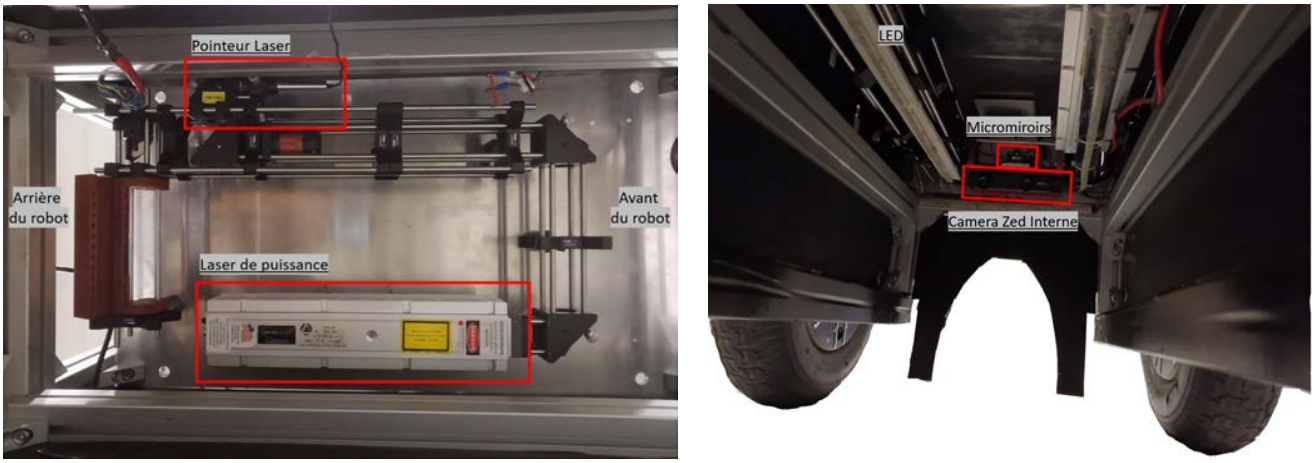


Fig. 1.2 – Système optique, avec le pointeur laser et le laser de puissance (vue de dessous)

Le système optique permet de diriger le faisceau laser sur le système de micromiroirs fixé sur la pièce bordeaux à l'arrière du robot. Après les réglages nécessaires, les faisceaux du laser de puissance et du pointeur laser deviennent confondus.

L'éclairage interne du robot est réalisé grâce à des bandeaux de LED fixés sur le système optique.

1.2 Cahier des charges

Afin de définir correctement les objectifs du projet, un cahier des charges a été créé. Le projet étant une preuve de concept, certains objectifs ne seront peut-être pas atteints. Le cahier des charges fournis un ordre de grandeur des résultats attendus pour un système viable.

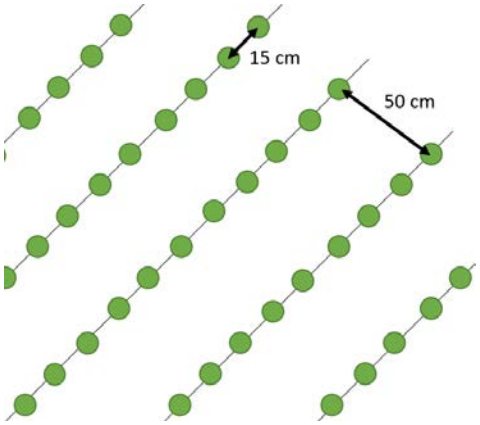


Fig. 1.3 – Dimensions d'un champ type (distance entre les plantes et les rangées)

Spécifications liées aux performances du robot	
1	Le robot doit pouvoir traiter 10 mètres de plants par minute
2	Le robot doit neutraliser 60% des pucerons en 1 passage
3	Le robot doit pouvoir fonctionner pendant 3 heures avec une charge de batterie
Spécifications liées à la structure et l'ergonomie du robot	
4	Les dimensions du robot doivent lui permettre de se déplacer dans un champ aux dimensions définies (figure 1.3)
5	La masse du robot ne doit pas excéder 100 kg afin de ne pas compacter la terre
6	Le robot doit être adapté à l'environnement extérieur (étanche à la pluie et à la poussière, gestion de la chaleur, ...)
7	Le robot doit être démontable et réparable par 2 opérateurs pour des opérations de maintenance
8	Le robot ne doit pas compromettre la sécurité des personnes et de l'environnement
9	La cartérisation doit être suffisante pour assurer la sécurité des personnes et de l'environnement vis-à-vis du laser de puissance

TABLE 1.1 – Cahier des charges

1.3 Travaux préliminaires

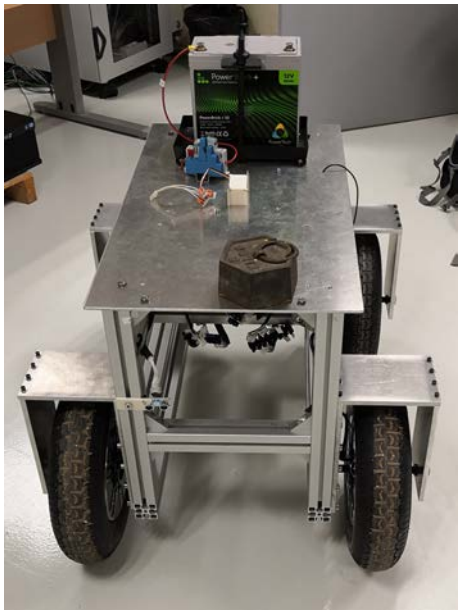


Fig. 1.4 – Le robot au début du stage

La structure mécanique du robot a été conçue par 2 stagiaires du département Génie Mécanique de L'INSA Lyon. Le robot est équipé de 4 roues motrices commandées par 2 cartes de commande. Le robot est de type char : aucune roue n'est directionnelle et le robot change de cap en utilisant un différentiel de commande sur les roues droites et gauches. Ces stagiaires ont développé un code qui permet de contrôler le robot à distance en utilisant la technologie Bluetooth, ainsi que d'arrêter le robot si des obstacles sont détectés par des capteurs ultra-sons. La partie de code relative au contrôle en vitesse pour un moteur est fournie comme exemple par le

constructeur des cartes de commandes et a été modifiée par leur soin. Un travail pour que le robot suive une ligne en utilisant une IMU a aussi été réalisé, mais les résultats n'étaient pas concluants d'après leur rapport.

Une des 2 cartes de contrôle des moteurs ne fonctionnait pas et l'installation électrique du robot était rudimentaire, nous n'avons donc jamais testé le robot avec le code fourni par les stagiaires précédents.

1.4 Répartitions des tâches

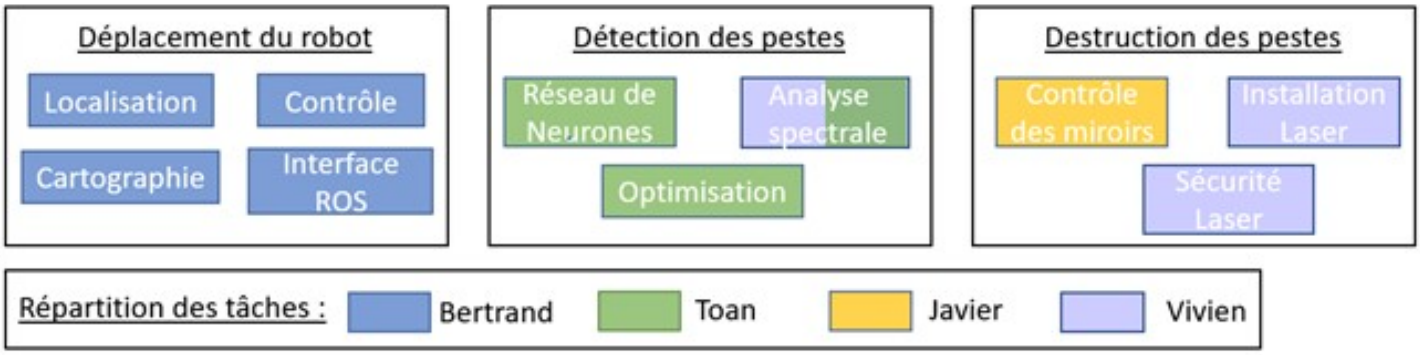


Fig. 1.5 – Répartition des tâches au sein du laboratoire Ampère

Chaque membre de l'équipe s'occupe d'une partie précise du développement.

Javier s'occupe de l'asservissement des micromiroirs afin de diriger un laser pour tuer les pucerons. Un asservissement visuel permet de détecter l'emplacement du pointeur laser et de diriger le faisceau vers l'emplacement des pucerons.

Toan est chargé de mettre en place un réseau de neurones capable de détecter les pucerons et les plants. Après la construction et l'annotation d'une base de données comprenant un millier d'images, il a entrainé plusieurs algorithmes de *machine learning* et a choisi celui qui donne les meilleurs résultats, à savoir Yolo-Darknet. Ses algorithmes optimisés traitent une vingtaine d'images par seconde, et détectent environ 70% des pucerons.

Notre ingénieur, Vivien Novales, a travaillé avec Toan pour tester différentes caméras, avec spectre visible et non visible, ainsi que différents éclairages (Visible, IR, UV) pour aider à la reconnaissance d'image. Vivien est en permanence sur place pour nous superviser et garder une vision globale du projet. Il a aussi à charge l'installation et l'utilisation du laser de puissance, et toutes les mesures de sécurité qui vont avec. Il supervise aussi l'intégration des parties de chacun sur le robot, afin de ne pas se perturber et de réussir à tout installer dans le robot.

Des membres des laboratoires partenaires ont aussi contribué au développement du robot.

Virginie du laboratoire partenaire BF2I nous fournit régulièrement en plants de fèves afin de tester et valider notre système au fur et à mesure du développement. Elle travaille sur la reconnaissance des pucerons par analyse spectrale. En utilisant une caméra hyper-spectrale, elle cherche une longueur d'onde optimale pour repérer les pucerons. Cette longueur d'onde correspondrait à un pic d'absorption ou de réflexion de l'onde par le puceron. Cependant, les pucerons et les feuilles sur lesquelles ils vivent ont globalement le même spectre d'absorption, ce qui complique la détection.

Robin, ingénieur au laboratoire INL, nous a conseillé sur l'installation, la sécurité et l'utilisation du laser de puissance. Il travaille sur les techniques de visés du laser, et la construction de systèmes optiques. Il est en outre chargé de trouver la bonne puissance que le laser doit fournir pour neutraliser les pucerons, sans détruire l'environnement autour.

2 État de l'art

Pour mener à bien sa mission, le robot doit pouvoir appréhender son environnement afin de se déplacer correctement au-dessus d'une ligne de plantes. Pour cela, le système doit être capable de repérer les jeunes plants, qui sont relativement petits et espacés. Des systèmes existent dans le commerce, et sont installés directement sur des tracteurs. Par exemple, le système auto pilot de chez Class utilise deux palpeurs mécaniques qui viennent détecter la position de la machine par rapport à 2 rangées de plantes [5]. Cependant le système est breveté et est utilisé sur des maïs. Dans notre cas, les palpeurs viendraient plier les jeunes plants, ce n'est donc pas une solution envisageable.

Une autre solution consiste à utiliser des capteurs inductifs pour repérer et suivre des fils en métal enfouis dans le sol [6]. Ce système est utilisé dans des serres pour suivre des tuyaux métalliques. L'utilisation dans un champ engendrerait des coûts élevés pour l'installation des fils, en plus des problèmes pour labourer le champ s'il y a des fils dans la terre.

L'utilisation d'un LiDAR pour repérer les plantes peut aussi être envisagée [7]. Des essais ont été réalisé sur des rangés d'arbres en installant un LiDAR 2D sur le toit d'un tracteur. Dans notre cas, les plants étant petits, ils seront difficilement repérables. Aussi, notre robot étant plus haut que les plants, nous ne pouvons pas mettre le LiDAR sur le toit. Il faudrait alors le mettre au niveau des plantes, en perdant le champ de vision à 360. Cette solution n'a donc pas été testée.

La solution retenue est l'utilisation d'une caméra visible, système utilisé dans de nombreux cas [8]. Une caméra installée à l'avant de robot détecte les plants et des algorithmes en déduisent le chemin à suivre. La détection des plants est traditionnellement réalisée par segmentation de couleurs et reconnaissance de formes. Etant donné qui nous avons à disposition des cartes de calculs, nous utiliserons un algorithme de machine learning pour détecter les plantes.

3 Solution envisagée

Pour réaliser sa mission, notre robot doit réussir à appréhender son environnement et se positionner. Une fusion des capteurs (IMU, GPS et odométrie des roues) par filtre de Kalman fournit une estimation de la position du robot dans son environnement.

Un algorithme de *machine learning* développé par Toan est capable de fournir une estimation de la position des plantes grâce à une caméra installée à l'avant du robot. Cependant, le champ de vision du robot est obstrué par une casquette de protection, il ne repère donc que les plants situés à partir 30 cm de l'avant du robot (donc plus de 70 cm du centre du robot). Aussi, cet algorithme de *machine learning*, bien que très performant, est sujet à des erreurs. Il y a des faux positifs qu'il faut filtrer et parfois des plants ne sont pas repérés d'une image à l'autre. L'algorithme renvoie une liste de rectangle englobant (*bounding box*), centré sur chaque plante vue, et la caméra stéréo fournit une carte de profondeur (*depth map*). La distance entre la caméra et chaque plante est calculée en moyennant la valeur de profondeur de la *bounding box*, ce qui n'est pas trés précis. Un changement de repère est ensuite appliqué pour estimer une position dans le repère du robot, ce qui augmente encore les erreurs.

Connaissant les limites de la reconnaissance d'image, il faut trouver un filtrage robuste qui garde en mémoire la position des plants, tout en oubliant les plantes si elles ne sont pas vues depuis un certain temps. Ce filtre doit aussi associer deux plantes proches comme étant la même.

Ces contraintes motivent donc la création d'une carte globale de l'environnement, où le champ est échantillonné en une carte de probabilité de la présence d'une plante. Ainsi des lignes de fortes probabilités de présence de plante représenteront les rangées que le robot doit suivre.

De plus, la carte positionne les rangées de plantes les unes par rapport aux autres. La recherche de la trajectoire du robot en est donc facilitée.

4 Architecture du robot

Pour mettre en œuvre la solution envisagée, il nous faut repenser le matériel installé sur le robot. Jusqu'à maintenant sont installés une IMU bon marché, un module Bluetooth, des capteurs ultra-sons et une carte de commande pour les moteurs. Les travaux effectués précédemment sur l'IMU ont montré qu'elle n'était pas assez performante, il en faut donc une nouvelle. Lors de la prise en main de la carte Microchip, qui contrôle les moteurs, il est vite apparu que la carte ne supporterait pas les calculs nécessaires au déroulement de la mission. Un ordinateur central chargé de prendre les décisions sera installé, et les cartes de contrôle seront seulement utilisées pour asservir les moteurs en vitesse. Le module Bluetooth et les capteurs ultra-sons qui étaient interfacés directement sur la carte Microchip vont être supprimés.

4.1 Architecture matérielle

Le robot embarque beaucoup de matériel nécessaire au bon déroulement de la mission.

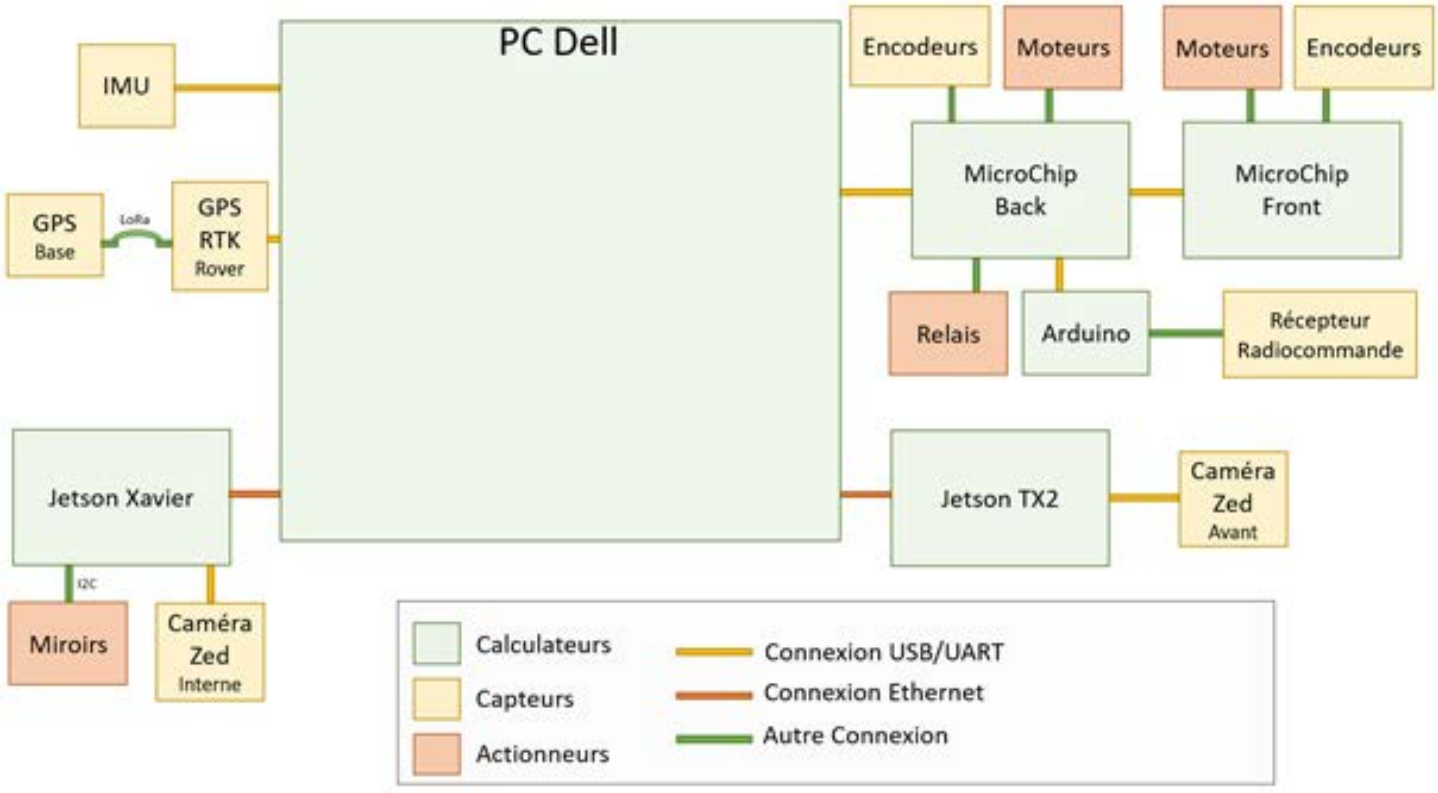


Fig. 4.1 – Schéma de l'architecture matérielle du robot

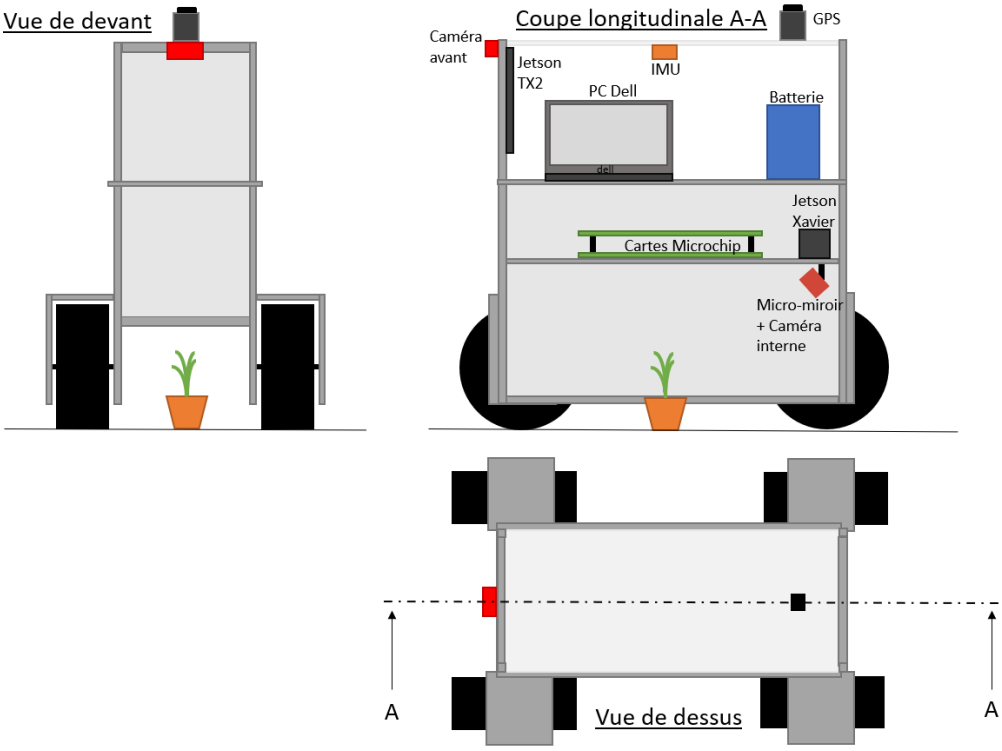


Fig. 4.2 – Différentes vues du robot

Tout d'abord, le matériel de calcul et de prise de décision a été installé sur le robot (en vert sur la figure 4.1, entourés en rouge la figure 4.3).

— PC Dell : ordinateur en charge de prendre les décisions. Fonctionnant sous Ubuntu 18.04, il embarque le *middleware* ROS afin de faciliter la prise d'informations et la communication avec les autres parties du système.

— Jetson Xavier : ordinateur chargé de repérer les pucerons grâce à un algorithme de *machine learning*. Les cartes Jetson, fabriquées par NVidia, sont équipées d'une carte graphique pour réaliser le calcul des réseaux de neurones. Cet ordinateur est aussi connecté aux miroirs qui permettent de positionner le rayon LASER pour neutraliser les pucerons.

— Jetson TX2 : ordinateur chargé de traiter les images venant de la caméra avant. Il permet de trouver la position des plantes par rapport au robot, en utilisant un réseau de neurones préalablement entrainé.

— Cartes Microchip : contrôleur pour les moteurs du robot. Une carte s'occupe des 2 moteurs avants et une autre des 2 moteurs arrières.

— Arduino : microcontrôleur qui récupère le signal PWM de la radiocommande et le renvoie par liaison série à la carte Microchip arrière.

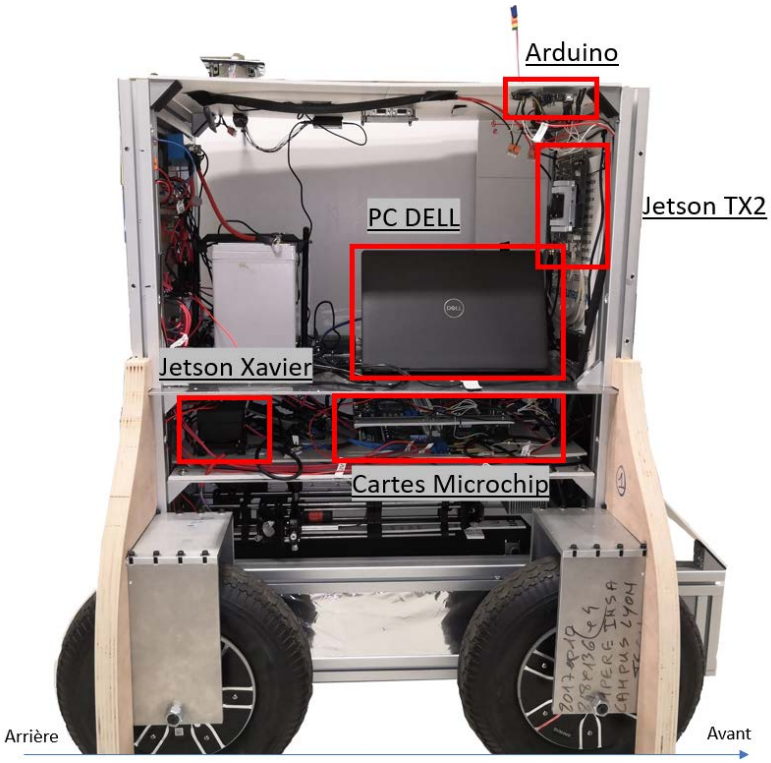


Fig. 4.3 – Les calculateurs installés sur le robot

Le robot est équipé de nombreux capteurs afin d'estimer sa position et d'appréhender son environnement (en rouge sur la figure 4.4) :

— IMU : centrale inertielle, qui estime le cap du robot, ainsi que accélérations. Le modèle embarqué est une MTI 630 fabriquée par XSens. L'IMU embarque un module AHRS (*Attitude and Heading Reference System*), qui filtre les données en sorties du capteur pour donner un cap directement exploitable.

— GPS : GNSS qui permet de positionner notre robot dans champ. Le système est équipé de 2 GPS : un premier GPS installé de manière fixe pendant toute la campagne de traitement transmet des données de correction au second GPS, afin qu'il puisse appliquer des corrections RTK. L'estimation de position du robot est alors plus précise (de l'ordre du centimètre d'après la documentation). Le GPS utilisé est un Reach RS+ pour le GPS fixe et un Reach M+ pour le GPS équipé dans le robot. Ce système est commercialisé par Emlid.

— Caméra Zed Avant : caméra stéréo qui repère les plants. La vision stéréo permet d'estimer une profondeur dans l'image, et d'en déduire une estimation de la position des plants dans le repère caméra. Cette caméra est située en hauteur et à l'avant pour voir les plants en face du robot et définir les objectifs de déplacement du robot.

— Caméra Zed Interne : caméra stéréo qui repère les pucerons sur les plantes. Elle est installée à l'intérieur du robot afin d'avoir un éclairage constant qui la protège des perturbations lumineuses externes.

— Capteurs à effet Hall : chaque roue est équipée de 3 capteurs à effet Hall qui permettent de commander les roues et d'estimer leur vitesse.

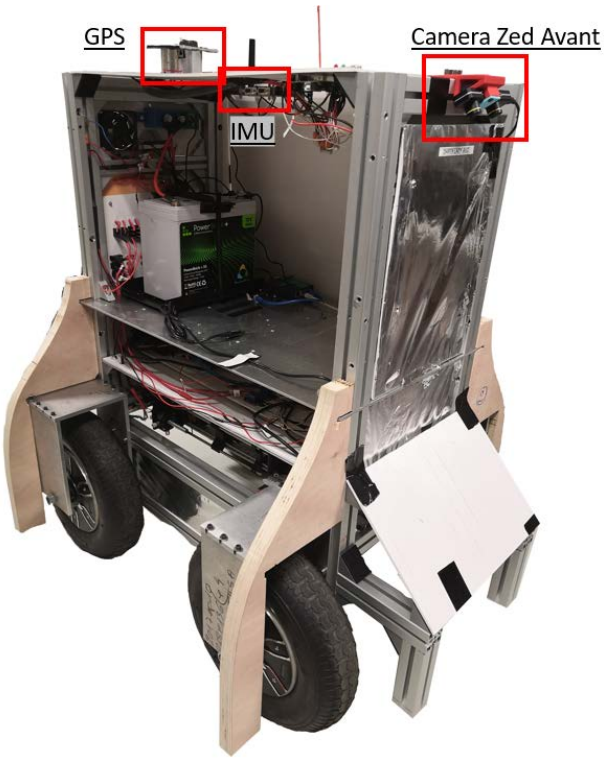


Fig. 4.4 – Les capteurs installés sur le robot

L'antenne du GNSS est installée sur le toit du robot afin de faciliter la connexion avec les satellites. L'IMU est installée au centre du robot afin de limiter les accélérations lors des changements de directions. Elle est installée le plus loin possible des moteurs, source de champs magnétiques donc de perturbation. Cependant, l'IMU reste relativement proche de la batterie, ce qui n'est pas forcément optimal.

Pour finir, le robot est équipé d'actionneurs pour interagir avec son environnement :

— Moteurs : le robot est équipé de 4 moteurs *brushless* à courant continue (BLDC), alimentés en 12V et développant chacun une puissance de 250W. Les moteurs et les capteurs effet Hall sont inclus dans chaque roue (le centre de la roue est fixe et le pneu tourne autour).

— Relais : les relais permettent d'allumer ou d'éteindre l'éclairage LED à l'intérieur du robot. Un relais contrôle aussi les ventilateurs de refroidissement du robot.

— Miroirs : un jeu de micromiroir dirige le faisceau laser pour neutraliser les pucerons.

4.2 Restructuration de la partie électrique du robot

A notre arrivée, l'alimentation du robot était rudimentaire. Une batterie alimentait la carte Microchip, qui alimentait à son tour l'IMU et les capteurs ultra-sons. Aussi, les éléments électriques (disjoncteur, relais, porte fusible) étaient dimensionnés pour un réseau électrique 230 V alternatif, alors que la batterie est en 12V continue.

En vue de l'installation des calculateurs et autres éléments, l'installation électrique a été retravaillée. Vivien a aidé à vérifier les schémas électriques, à mettre en place l'installation et ensuite à la tester.

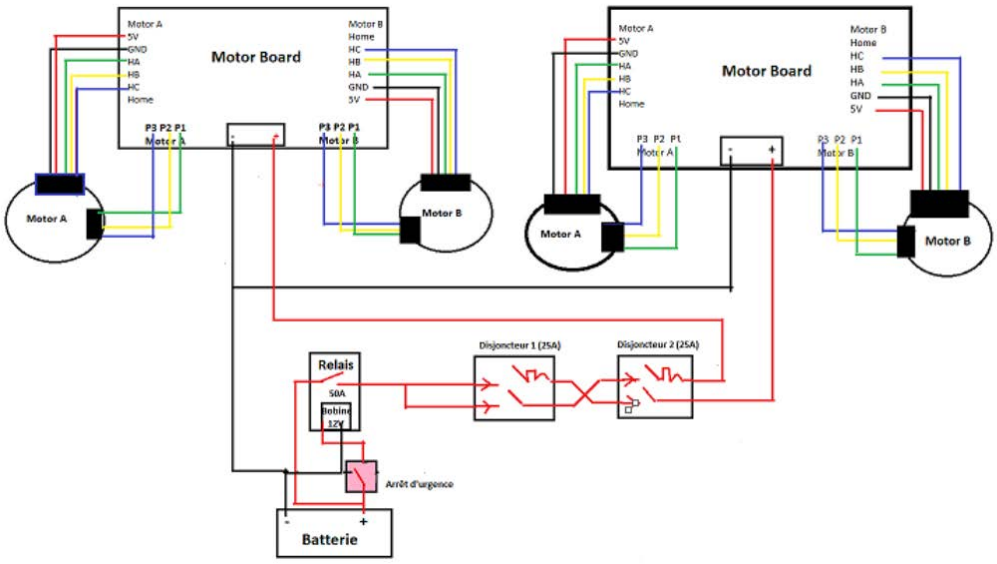


Fig. 4.5 – Schéma électrique initial (crédit image : Anthony Heyer et Hoang Xuyen)

Tout d'abord, la figure 4.5 montre que l'arrêt d'urgence coupe toute l'alimentation du robot (puissance et calcul). Aussi, la mission ne doit pas se finir en cas d'arrêt d'urgence mais seulement se mettre en pause. Il faut séparer la partie de puissance (moteur et laser) de la partie commande (calculateurs, capteurs, ...). Pour cela, deux boutons d'arrêt d'urgence ont été installés, et actionnent un relais de puissance qui isole la partie de puissance. Les deux arrêts d'urgence sont installés en série, donc si l'un des 2 est actionné, le circuit s'ouvre et le relais coupe l'alimentation de la partie puissance. Aussi, les cartes de commande moteurs Microchip supportent l'alimentation séparée de la carte de commande et de la carte d'alimentation des moteurs, elles ne s'éteindront pas en cas de coupure.

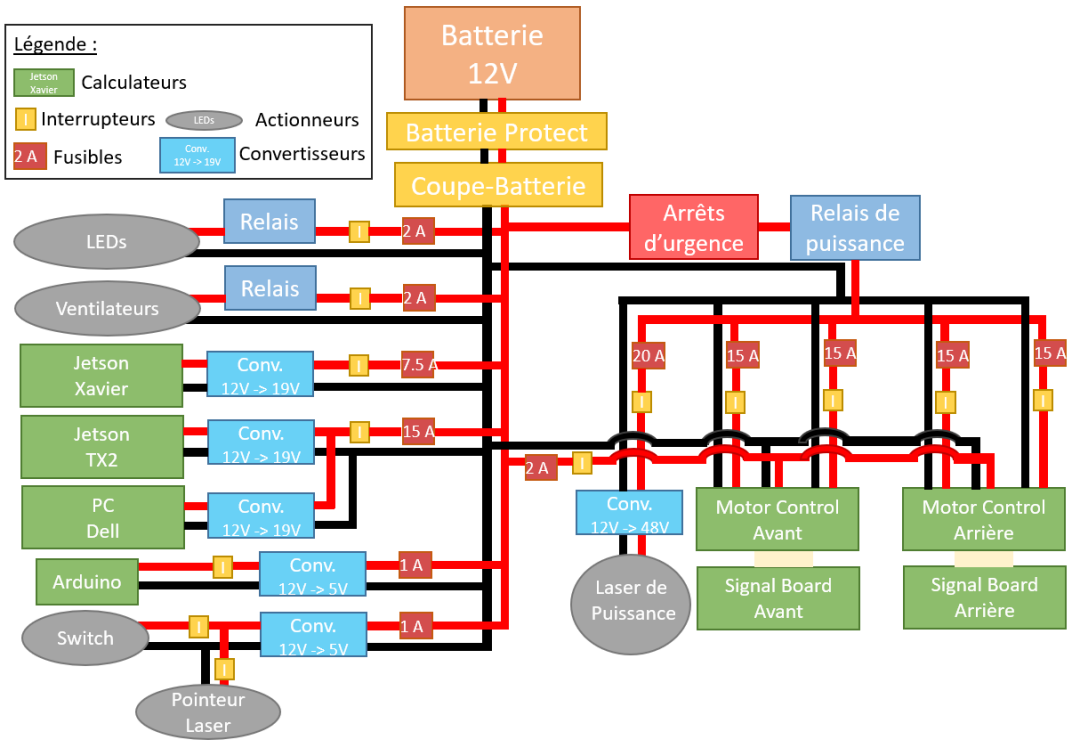


Fig. 4.6 – Schéma électrique du robot

Ensuite, en prévision de l'installation des calculateurs nécessitant des tensions variées, il a fallu penser l'alimentation pour avoir des tensions adaptées à tout le matériel. Des convertisseurs 12V vers 19V ont été installés pour alimenter les 2 cartes Jetson et l'ordinateur de contrôle . Deux convertisseurs 12V vers 5V délivrent la tension nécessaire à l'alimentation du commutateur Ethernet (*Switch*), de la carte Arduino, des relais et du pointeur laser. Un convertisseur 12V vers 48V alimente le laser de puissance. Le GPS et l'IMU sont alimentés par le port USB de l'ordinateur Dell.

Un coupe-batterie permet de couper toute l'alimentation du robot lorsque ce dernier n'est pas utilisé, lors du stockage par exemple.

Un panneau de contrôle (voir la figure 4.7) a été créé afin de pouvoir déconnecter chaque élément indépendamment. Des fusibles ont été installés pour les sécuriser. Un module Batterie Protect est aussi installé pour couper l'énergie si la tension de la batterie est trop faible et la protéger des courts-circuits.

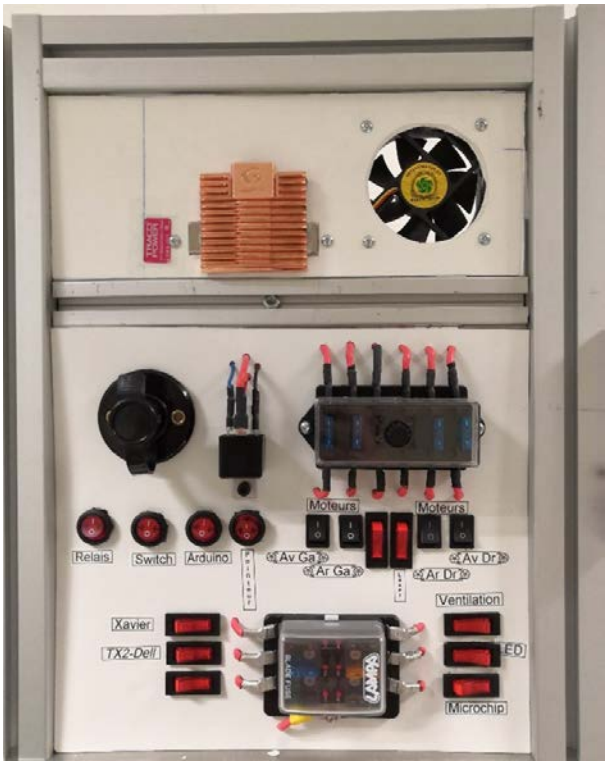


Fig. 4.7 – Panneau de contrôle du robot

4.3 Radiocommande

Un module Bluetooth était installé sur le robot, cependant l'interface était inutilisable en pratique. Il fallait se connecter avec un ordinateur et envoyer des caractères dans un terminal. Chaque caractère correspond à une action (avancer, reculer, tourner à droite, ...) que le robot peut réaliser. Ce n'est donc pas aisé pour l'utilisateur de contrôler le robot.

Fig. 4.8 – La radiocommande du robot et les commandes associées

Pour faciliter le déplacement du robot, une radiocommande a donc été installée. L'utilisateur peut alors déplacer le robot jusqu'à sa position initiale, on encore le stocker plus facilement. Le modèle de radiocommande retenu est le T8S de Radiolink. Outre son faible prix, cette radiocommande à la forme d'une manette est facile à prendre en main et n'a pas de fonction superflue. Elle permet de transmettre jusqu'à 8 signaux PWM (*Pulse Width Modulation*). La radiocommande étant programmable, les boutons ont été adaptés pour l'utilisation du robot :

— Le bouton *Dead Man's Switch* doit être enfoncé pour que les commandes de la radiocommande soient envoyées. C'est une sécurité pour vérifier qu'il y a bien un utilisateur qui la commande.

— Un curseur permet de choisir la vitesse maximale du robot, et génère des mouvements plus précis.

— Le passage de la marche avant à la marche arrière se fait grâce à un interrupteur à 3 positions.

La radiocommande communique avec un récepteur qui envoie des signaux PWM (voir annexe A.3 pour les détails). Ces signaux sont ensuite récupérés par une carte Arduino et renvoyés par liaison série à la carte Microchip arrière.

Lorsque le *Dead Man's Switch* est enclenché, la carte de commande des moteurs exécute en priorité les commandes venant de la radiocommande. Si une mission autonome est en cours, la prise de décisions automatiques continue mais les commandes en vitesse correspondantes ne sont plus exécutées par les moteurs. L'utilisateur peut donc arrêter le robot à distance en appuyant sur le *Dead Man's Switch*, et s'il relâche le bouton, le robot reprend la mission de là où il est. Il est important de préciser que c'est un arrêt logiciel, il est toujours plus sûr de couper l'alimentation de puissance avec l'arrêt d'urgence en cas de problèmes.

5 Architecture logicielle du robot

5.1 Architecture ROS

Afin de gérer la communication entre les différents éléments du robot, ainsi que pour faciliter l'implémentation de la prise de décisions pour mener à bien la mission, le *middleware* ROS sera utilisé [9]. ROS permet de gérer des canaux de communications (*topics*) sur lesquels les programmes viennent lire ou écrire des données. Les types de données sont spécifiés par des standards fixés, ce qui permet de réutiliser facilement des programmes développés par la communauté. De plus, ROS propose des outils de sauvegarde des données ainsi que des outils de visualisation très utiles pour faire des essais.

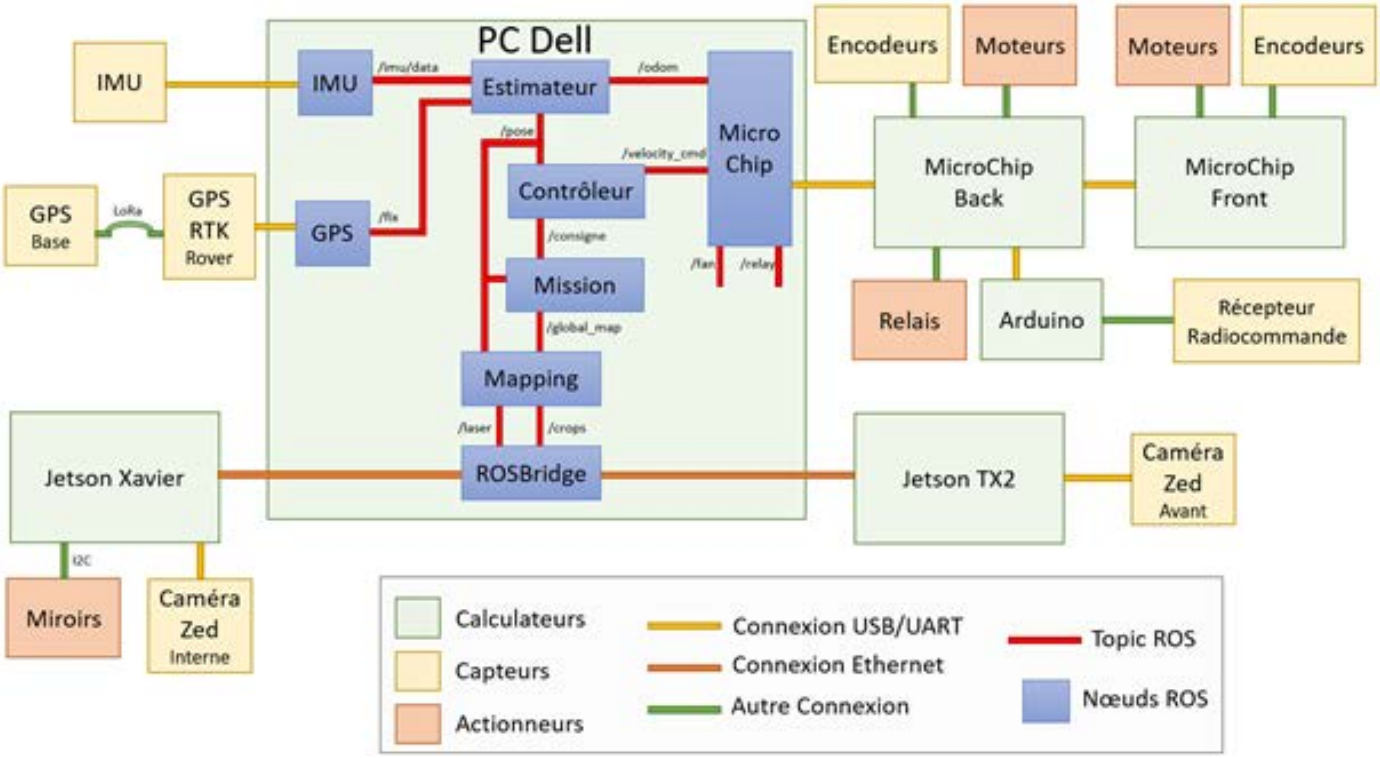


Fig. 5.1 – Architecture logiciel du robot, et matériel embarqué

— IMU : nœud d'interface entre l'IMU et ROS. Publie sur le topic $/imu/data$ un message de type $sensor\_msgs/Imu$. Ce nœud est fourni par le fabriquant [10].

— GPS : nœud d'interface entre le GPS et ROS. Publie sur le topic $/fix$ un message de type $sensor\_msgs/NavSatFix$. Ce nœud est fourni par la communauté [11].

— TX2 : nœud d'interface avec la Jetson TX2. Envoie la liste des coordonnées des plantes vues par la caméra avant. Publie sur le topic $/crops$ un message de type $geometry\_msgs/PoseArray$.

— Xavier : nœud d'interface avec la Jetson Xavier. Envoie un booléen pour savoir si le robot doit ralentir car il y a des plantes à traiter. Publie sur le topic */laser* un message de type *std_msgs/Bool* où False signifie qu'il n'y a pas de pucerons et True qu'il faut s'arrêter.

— Estimateur : Nœud qui estime la position du robot en fusionnant les données IMU, GPS et d'odométrie. La position sera donnée dans un référentiel ENU (East, North, Up) donc l'axe des x vers l'est, y vers le nord et z vers le haut. L'angle zéro de référence sera donc plein est. La position initiale du robot définira le 0 de la position. Publie sur le topic */pose* un message de type *geometry_msgs/Pose*. Ce nœud est fourni par la communauté [12].

— Mapping : Nœud de création d'une carte de l'environnement. Prend en entrée la position du robot et la position des plantes vues par la caméra avant et publie le topic */map* une carte de l'environnement sous un format *nav_msgs/OccupancyGrid*.

— Mission : Nœud qui s'occupe de gérer les différents états du robot et les transitions correspondantes. Il prend en entrée la position estimée du robot, l'état du LASER et la carte créée par le nœud Mapping. En fonction des données acquises, ce nœud déduit le cap et la vitesse que le robot doit suivre. Publie sur le topic */consigne* un message de type $std\_msgs/Float32MultiArray$ où le premier élément correspond au cap à suivre et le second la vitesse. Ce nœud publie aussi le topic */state* un entier représentant l'état du robot (0 : IDLE, 1 : arrêt, 2 : marche).

— Contrôleur : Nœud de contrôle du robot, gère la commande en vitesse à envoyer aux moteurs. Prend en entrée le topic */consigne* et calcule la commande en vitesse à envoyer aux moteurs droits et gauches pour réaliser celui-ci. Publie sur le topic */velocity_cmd* un message de type $std\_msgs/Float32MultiArray$ où le premier élément de l'*array* sera la commande droite et le deuxième la commande gauche.

— Microchip : Nœud d'interface avec la carte Microchip. Il écoute le topic */velocity_cmd* pour envoyer les commandes correspondantes aux moteurs ainsi que les topics */fan* et */relay* (de type *std_msgs/Bool*). Publie sur les topics */odom_microchip* une estimation de l'odométrie du robot et sur */state_motors* un entier représentant l'état des moteurs (0 : IDLE, 1 : Manuel, 2 : Automatique). Il publie la vitesse cible droite et gauche dans un message de type $std\_msgs/Float32MultiArray$ où le premier élément de l'*array* sera la vitesse cible droite et le deuxième la vitesse cible gauche.

5.2 Odométrie et mesure de vitesse

La méthode pour mesurer la vitesse d'un moteur utilisée jusque-là, proposée par le fabriquant de la carte Microchip, n'est pas utilisable pour notre application. En effet, cette méthode consiste à calculer la moyenne des périodes entre les fronts montants des 3 capteurs à effets Hall, et à en déduire la vitesse de la roue. Cependant, lorsque les roues ne tournent pas vite, cette méthode ne fonctionne pas. En effet, la limitation des blocs Matlab Simulink ne permet pas à ces périodes d'être plus grandes que 200 ms. Ainsi, lorsque la vitesse du robot est plus faible que 0.2 m/s, cette méthode considère que le robot ne bouge pas. Or, Il faut que le robot aille lentement pour que la détection et la neutralisation opère.

Les capteurs à effet Hall contenus dans les roues nous permettent de mesurer un nombre d'impulsions, correspondant au nombre de fronts montants en sortie des capteurs. Ensuite, en connaissant le rapport de réduction du moteur et le rayon des roues, et en faisant une hypothèse de roulement sans glissement, le déplacement linéaire est calculé. Cependant, il n'y a pas d'indication sur le sens de ce déplacement.

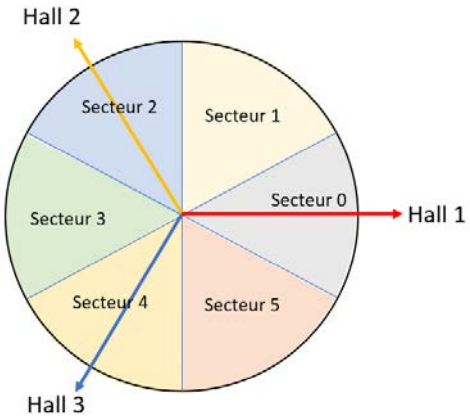


Fig. 5.2 – Secteurs et capteurs à effet Hall

Pour mesurer la distance parcourue, les données des 3 capteurs à effet Hall sont converties en un secteur angulaire, numéroté de 0 à 5 (voir figure 5.2). Ensuite, en soustrayant le secteur angulaire précédent à notre secteur, nous trouvons le nombre algébrique de secteurs parcourus par la roue, et donc le déplacement avec un sens cohérent.

La discontinuité lors du passage entre les secteurs 0 et 5 (ou inversement) est gérée en appliquant un modulo. Cependant, comme la différence de secteur peut être négative, la fonction modulo disponible avec Simulink ne donne pas le résultat attendu. La solution mise en place consiste à additionner +/- 6 en fonction du signe de la différence des secteurs, si la valeur absolue de la différence est égale à 5 (figure 5.3).

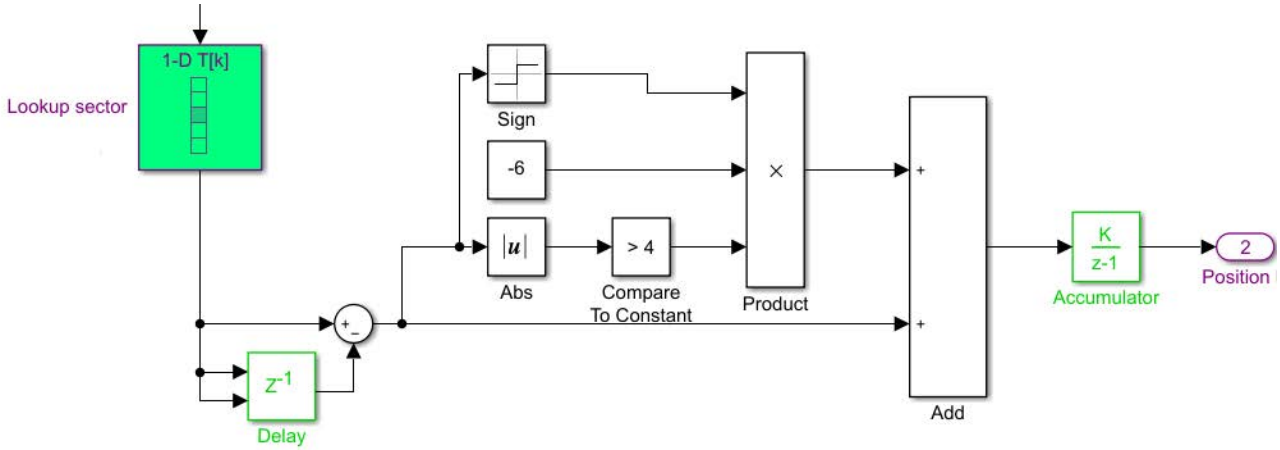


Fig. 5.3 – Blocks Simulink pour compter le nombre de secteurs parcourues

Chaque roue comporte 8 pairs d'aimants permanents, et non 1 seule comme sur le schéma. Tout se passe alors comme si nous avions 2*8*3=48 secteurs dans un tour, soit une rotation de 7° par tick. Nous avons aussi un rapport de réduction de 1/30, ce qui donne 1440 ticks par tour de roue (ce qui a été vérifié expérimentalement). Pour connaitre la vitesse maximale mesurable,

nous savons qu'une roue peut parcourir au maximum 5 secteurs angulaires entre 2 mesures, et la fréquence de mesure est de 1000 Hz (période = 1/1000 s). Ce qui donne donc 7°/30*5 secteurs * 1000 (1/période) = 22 rad/s

Avec un rayon de roue = 0.20m, le robot atteint une vitesse linéaire maximale de 4.3 m/s ou 15 km/h, ce qui est largement suffisant pour notre application.

En comptant les nombres d'impulsions sur les capteurs des roues droite et gauche, une estimation du déplacement du robot est calculée. Le *package* navigation de ROS fournit un exemple d'implémentation d'odométrie [13].

$$\text{Nous avons : } \begin{cases} dist_{wheel} = 0.57 \text{ (distance entre les roues droites et gauches)} \\ tick_{tour} = 240 \text{ (nombre d'impulsions par tour de roue)} \\ size_{wheel} = 0.2 \text{ (rayon d'une roue)} \end{cases}$$

$$\text{On en déduit la variable } tick_{metre} = \frac{tick_{tour} * 3}{\pi * size_{wheel}}$$

Soit $tick_t^{right}$ et $tick_t^{left}$ le nombre d'impulsions à droite et à gauche, à l'instant t

Soit [X_t,Y_t,θ_t] l'état estimé du robot à l'instant t

$$d_{right} = \frac{(tick_t^{right} - tick_{t-1}^{right})}{tick_{metre}}$$

$$d_{left} = \frac{(tick_t^{left} - tick_{t-1}^{left})}{tick_{metre}}$$

$$dxy_{avg} = \frac{(d_{right} + d_{left})}{2}$$

$$d_{th} = \frac{(d_{right} - d_{left})}{size_{wheel}}$$

$$d_x = \cos(d_{th}) * dxy_{avg}$$

$$d_y = -\sin(d_{th}) * dxy_{avg}$$

$$X_t = X_{t-1} + \cos(\theta_{t-1}) * d_x - \sin(\theta_{t-1}) * d_y$$

$$Y_t = Y_{t-1} + \sin(\theta_{t-1}) * d_x + \cos(\theta_{t-1}) * d_y$$

$$\theta_t = \theta_{t-1} + d_{th}$$

Cependant, le robot a 4 roues et non 2 comme le voudrait le modèle, et la condition de roulement sans glissement n'est jamais respectée. Ainsi lorsque la rotation est trop brusque, l'odométrie ne fournit plus un résultat correct car le centre de rotation n'est pas au centre des roues. Nous observons en violet sur la figure 5.4 que l'erreur s'accumule très vite.

Aussi, comme le robot est équipé d'une IMU qui fournit le cap du robot, nous avons décidé d'utiliser la mesure de cette dernière à la place du cap calculé par l'odométrie. La figure 5.4 nous montre la trajectoire fournie par l'odométrie seule (en violet) et l'odométrie couplée au cap IMU (en rouge).

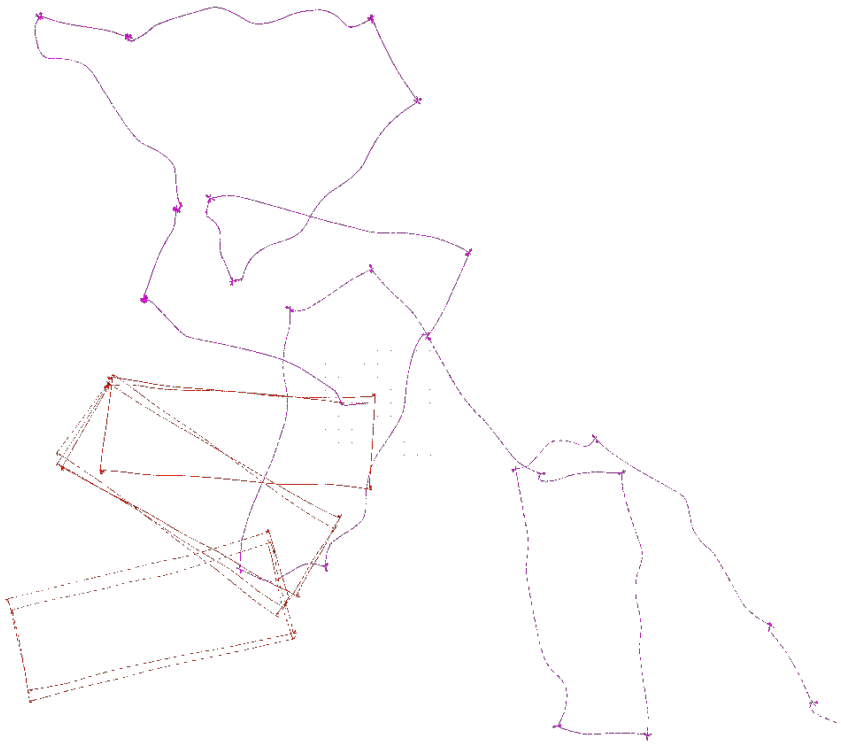


Fig. 5.4 – Estimation de l'odométrie du robot. En violet en utilisant seulement les capteurs des roues et en rouge en combinaison avec l'IMU

Pour l'expérience, le robot a parcouru 6 fois le même rectangle. Nous voyons que le couplage avec l'IMU donne un résultat beaucoup plus cohérent. La dérive subite de la trajectoire rouge est due à une mauvaise calibration de l'IMU, le problème est résolu par la suite.

Le calcul d'odométrie peut encore être amélioré en utilisant la méthode UMBmark [14] qui permet de trouver les erreurs systématiques dues aux imperfections du robot (comme une différence de diamètre des roues par exemple), et de compenser ces erreurs dans le calcul pour être plus précis. Cette méthode n'a pas été mise en place par manque de temps.

5.3 Estimation de la position du robot

Pour estimer la position du robot, nous utilisons le *package* ROS *robot_localization* [12], qui implémente une interface entre des topics ROS et un filtre de Kalman étendu (EKF). Les données fournies sont les données de l'IMU, l'odométrie venant des roues et la position estimée par le GPS.

Le *package* intègre un nœud qui converti le *topic* du GPS en un *topic* d'odométrie utilisable par l'estimateur.

Le *package robot_localization* permet de changer de nombreux paramètres afin de régler le filtre de Kalman au mieux vis-à-vis de notre utilisation. Ainsi, le filtre est configuré pour fonctionner à 2 dimensions. La position du robot est fournie par 2 sources indépendantes, le GPS et la carte Microchip. Aussi, l'estimation de position venant de la carte Microchip est définie en mode différentielle. Le filtre soustrait alors la valeur donnée à l'étape d'avant à la nouvelle valeur, ce qui permet de limiter la dérive.

Pour ajuster les covariances des capteurs et les paramètres du filtre de Kalman, nous avons utilisé la méthode qui consiste à parcourir le même rectangle plusieurs fois de suite [15], en

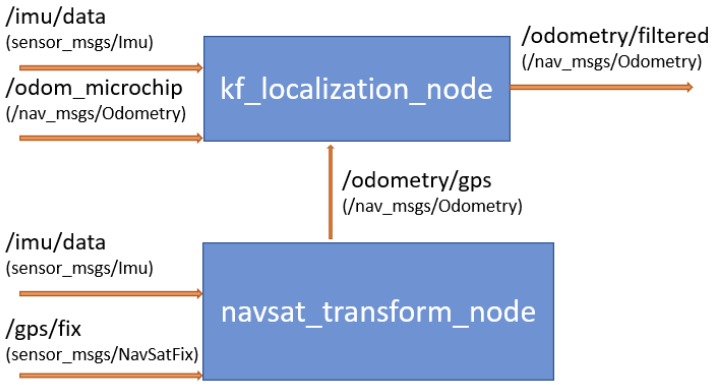


Fig. 5.5 – Architecture ROS de l'estimateur d'état

sauvegardant les données avec l'outil bag de ROS [16]. Les données sont ensuite rejouées plusieurs fois pour ajuster les paramètres et trouver la combinaison qui donne le meilleur résultat.

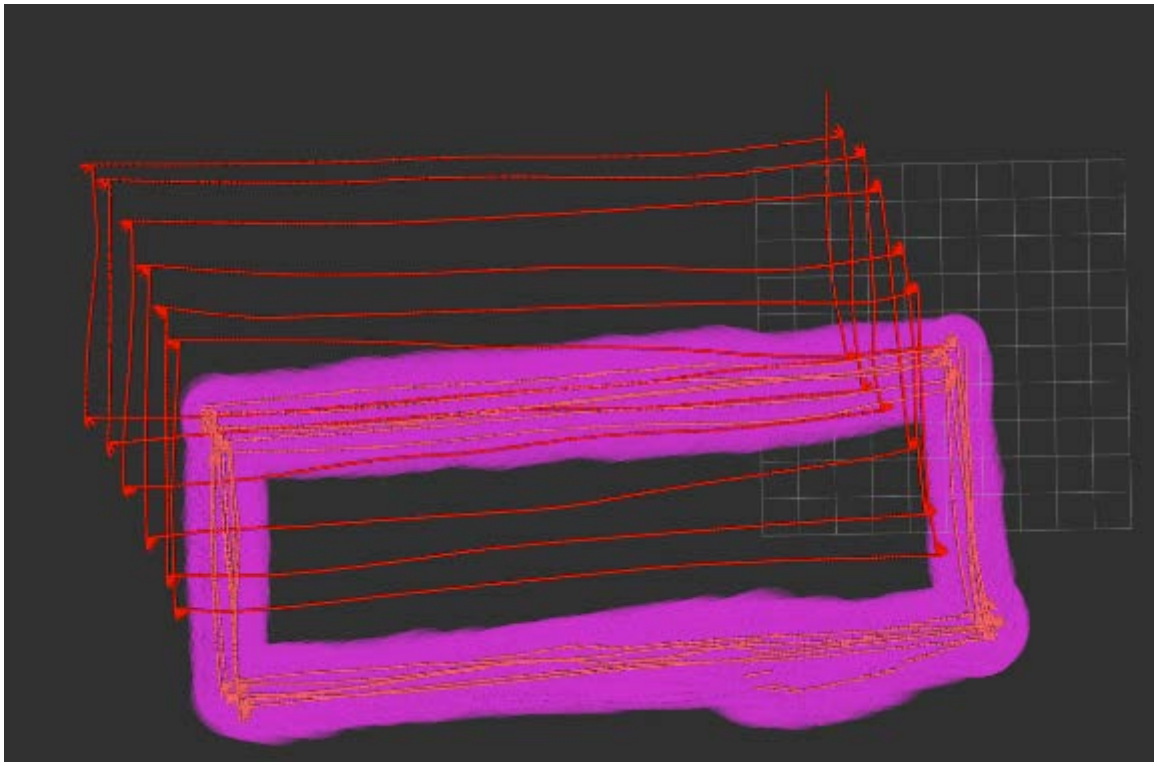


Fig. 5.6 – Trajectoire du robot. En rouge, odométrie des roues et en orange la sortie du filtre de Kalman

La figure 5.6 montre que la sortie du filtre de Kalman (trace orange), ne dérive plus. On notera les sauts de la position filtrée en bas à droite. Un arbre se trouve à cet endroit et perturbe le GPS (à gauche sur la figure 5.7).

Nous avons essayé d'utiliser l'odométrie visuelle venant de la caméra avant, mais les résultats n'étaient pas encourageants.

Fig. 5.7 – Essai pour le filtre de Kalman, avec un arbre qui perturbe la mesure, et la base RTK à droite

5.4 Communication entre les modules du robot

Pour des raisons de compatibilité des versions d'OpenCV utilisées par les caméras et la version utilisée par ROS, ce dernier n'a pas été installé sur les cartes Jetson. Aussi, le *middleware* ROS ne peut pas être utilisé pour transmettre les informations entre l'ordinateur central et les 2 cartes Jetson.

Le *package* ROSBridge [17] propose de faire une interface entre un server WebSocket et ROS. L'utilisation première de ce package est l'interfaçage entre ROS et une page internet utilisant javascript. Ce package fonctionne en créant un server TCP, qui reçoit et envoie des requêtes dans un format JSON, et interagit avec ROS en conséquence. Un programme n'utilisant pas ROS peut alors envoyer des requêtes sur le serveur afin de publier sur un *topic*, ou encore de demander au serveur de renvoyer les données publiées sur un autre *topic*.

Dans notre cas nous incorporons, dans le code python des réseaux de neurones, une partie de code pour se connecter sur le serveur hébergé sur l'ordinateur central. Nous pouvons ainsi publier les *topics* relatifs à la position des plantes (Jetson TX2 et caméra avant) ainsi que la présence de pucerons et donc la nécessité de stopper le robot (Jetson Xavier et caméra interne).

La partie de code utilisée est inspirée d'un exemple de code hébergé sur Github [18] pour la partie relative au protocole de communication. Nous avons réalisé une intégration pour que le code s'exécute en parallèle du réseau de neurones et qu'il ne soit pas bloquant. Aussi, une gestion de la plupart des exceptions a été prévue. Ainsi, lors d'une déconnexion, le programme renvoie automatiquement les commandes pour s'inscrire sur les *topics* ROS. Un système de *buffer* permet de garder en mémoire les messages qui n'aurait pas été envoyés pour les renvoyer dès que la connexion est à nouveau établie.

Pour pouvoir communiquer avec ROS, il faut d'abord signaler que le programme va publier sur un topic. Par exemple, la requête à envoyer par le *topic* relatif aux positions des plantes sera la suivante :

msg = {'op' : 'advertise' , 'topic' : '/crops', 'type' : 'std_msgs/Float32MultiArray'}

Où *op* est le type d'action à réaliser, *topic* le *topic* qui sera créer et *type* le type de donné qui sera envoyé.

Ensuite, pour envoyer les positions des plantes, la requête sera :

```
msg = {'op' : 'publish',
        'topic' : '/crops',
                  'msg' : { 'data' : l_crops_xy,
                            'layout' : { 'dim' : [
                {'label' :'x', 'size' :int(len(l_crops_xy)/2), 'stride' :int(len(l_crops_xy))},
                {'label' :'y', 'size' :int(len(l_crops_xy)/2), 'stride' :int(len(l_crops_xy)/2)}],
                            'data_offset' :0
                                      }
                           }
        }
```

Où *l_crops_xy* correspond à une liste résultant de la concaténation de la liste des positions des plantes selon x et de la liste des positions selon y. La structure du message correspond à la structure du message *Float32MultiArray*, comme précisé dans la documentation de ROS [19].

6 Réalisation d'une mission de navigation

6.1 Simulation

Pour faciliter le développement des programmes du robot, il a été décidé de créer une simulation dans lequel le robot peut évoluer. Une simulation python a donc été développée et utilise la bibliothèque ViBes pour l'affichage [20]. Le déplacement du robot est purement cinématique et aucune collision n'est programmée. La simulation prend en entrée une commande en vitesse pour les roues droites et gauches, et le robot se déplace en conséquence. La simulation publie une odométrie bruitée du robot ainsi que la position des plantes qui se trouvent devant le robot. L'utilisation de ROS permet de tester tous les nœuds en simulation sans avoir à modifier les codes.

Sur la simulation, l'utilisateur choisi les paramètres du champ (longueur, largeur, espacement entre les rangées et espacement entre les plants, orientation du champ). Les lignes de plants sont caractérisées par un polynôme du second degré, dont il faut aussi donner les coefficients.

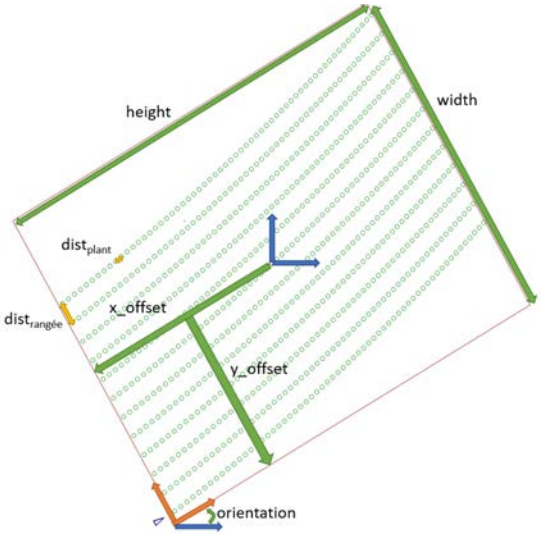


Fig. 6.1 – Paramètres d'un champ simulé

La première étape de la simulation est la création d'une liste de toutes les plantes du champ. Le premier problème rencontré est d'espacer correctement les plantes sur une même rangée. Réaliser un espacement uniforme sur l'axe des ordonnées ne convient pas car pour des grandes valeurs de x et/ou des grands coefficients de polynôme, la distance entre 2 plantes successives serait bien trop grande (en rouge sur la figure 6.2)).

Les calculs se font dans un repère local (repère orange sur la figure 6.1), un changement de repère sera réalisé par la suite.

Nous travaillons sur une rangée qui suit le polynôme d'équation $y = a * x^2 + b * x + c$.

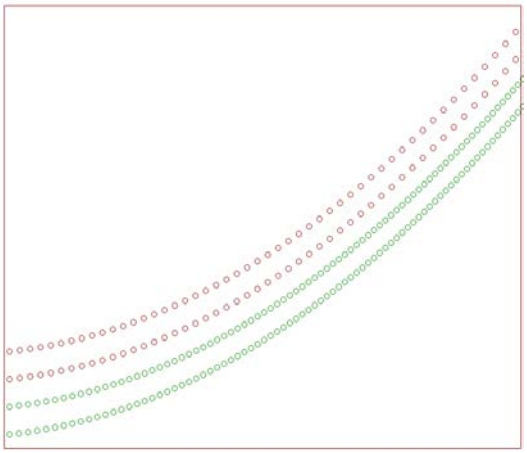


Fig. 6.2 – Création des lignes de plantes, avec une méthode naïve en rouge et une méthode vectorielle en vert

Nous définissons une suite qui à l'ordonnée d'un plant $P_n[x_n, y_n]$ donne une approximation de l'ordonnée du plant suivant $P_{n+1}[x_{n+1}, y_{n+1}]$ (figure 6.3). Tout d'abord, nous pouvons approximer le polynôme par sa pente en x_n, ce qui nous donne l'équation (1). Le théorème de Pythagore nous permet de trouver l'équation (2). Nous pouvons maintenant en déduire la valeur de d_x.

$$\begin{cases} \frac{d_x}{d_y} = 2 * a * x_n + b \ (1) \\ dist^2_{plant} = d_x^2 + d_y^2 \ (2) \end{cases} \Rightarrow dist^2_{plant} = d_x^2 + d_x^2 * (2 * a * x_n + b)^2$$

$$\Leftrightarrow d_x^2 = \frac{dist^2_{plant}}{1 + (2 * a * x_n + b)^2}$$

$$\Leftrightarrow d_x = \frac{dist_{plant}}{\sqrt{1 + (2 * a * x_n + b)^2}}$$

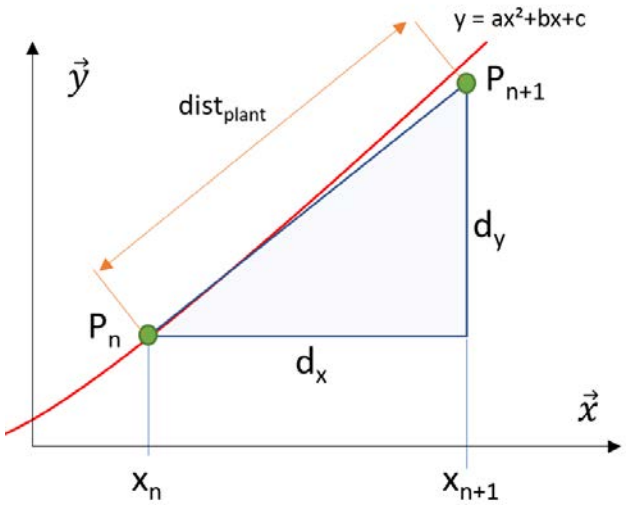


Fig. 6.3 – Schéma du calcul de la suite des ordonnées des plantes

Nous avons une rangée de plants dans un repère local, qu'il faut passer dans le repère global. Le changement de repère est réalisé en appliquant une translation puis une rotation :

Soit $[x_p, y_p]$ les coordonnées d'une plante dans le repère locale

Les coordonnées [X,Y] de la plante dans le repère globale sont données par :

$$\begin{cases} X = \cos orientation * (x_p + x_{offset} - \sin orientation * (y_p + y_{offset} \\ Y = \sin orientation * (x_p + x_{offset} + \cos orientation * (y_p + y_{offset} \end{cases}$$

A chaque tour de boucle de simulation, la première étape consiste à mettre à jour l'état du robot. La consigne de vitesse venant de ROS est utilisée pour appliquer les formules d'odométries.

Soit $u[v_{right}, v_{left}]$ la consigne en vitesse du robot

Soit $[X_t, Y_t, \theta_t]$ l'état estimé du robot à l'instant t

On a toujours $size_{wheel} = 0.57$

$$d_{right} = v_{right} * dt$$

$$d_{left} = v_{left} * dt$$

$$dxy_{avg} = \frac{d_{right} + d_{left}}{2}$$

$$d_\theta = \frac{d_{right} - d_{left}}{size_{wheel}}$$

$$d_x = \cos d_\theta * dxy_{avg}$$

$$d_y = -\sin d_\theta * dxy_{avg}$$

$$X_t = X_{t-1} + \cos \theta_{t-1} * d_x - \sin \theta_{t-1} * d_y$$

$$Y_t = Y_{t-1} + \sin \theta_{t-1} * d_x + \cos \theta_{t-1} * d_y$$

$$\theta_t = \theta_{t-1} + d_\theta$$

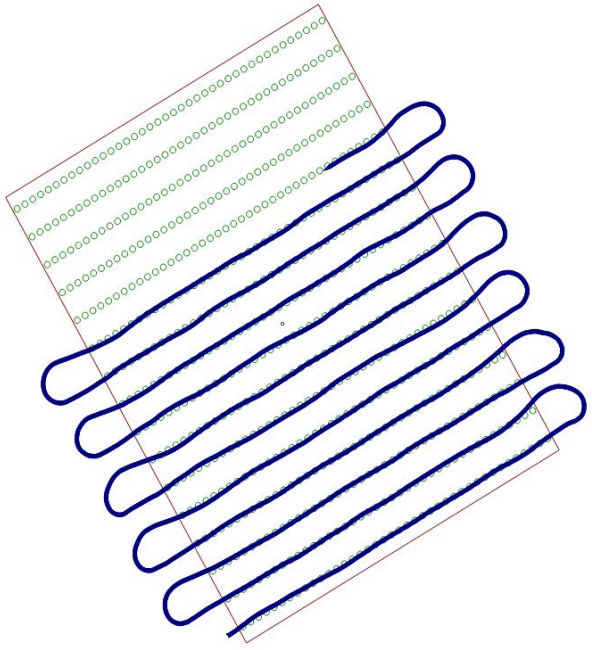


Fig. 6.4 – Trajectoire simulée du robot pour un suivi de rangées rectilignes

Maintenant, nous cherchons toutes les plantes qui sont dans le champ de vision du robot. Pour cela, un changement de repère est appliqué sur la position des plantes pour les avoir dans le repère du robot et, et ainsi sélectionner les plants qui sont dans un rectangle devant le robot (en rouge sur la figure 6.5).

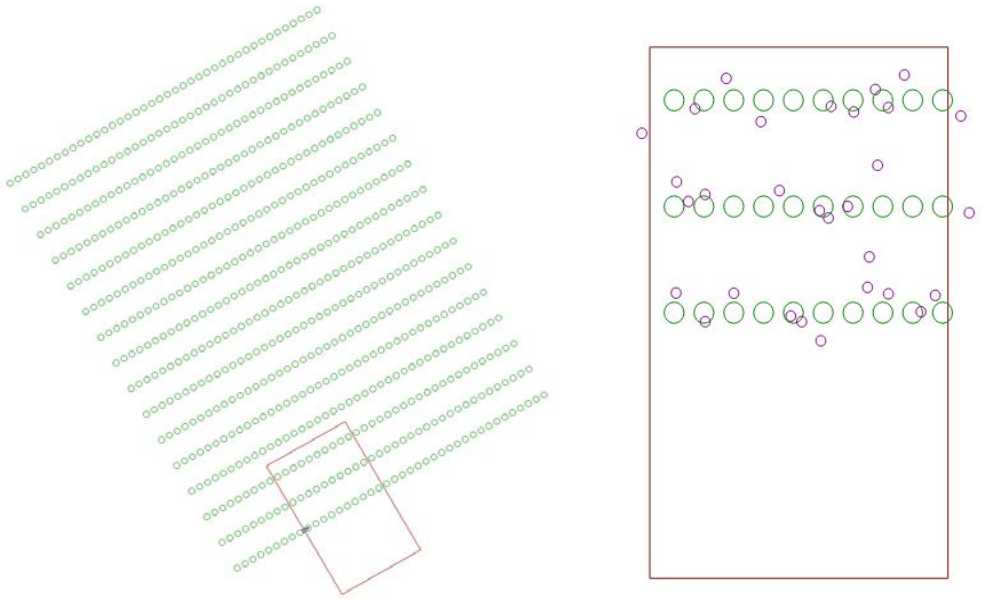


Fig. 6.5 – Plantes vues par le robot, avec positions bruitées en violet, et champ de vision en rouge

A la fin de chaque étape de la simulation, nous pouvons publier sur les topics ROS l'odométrie du robot et les coordonnées des plants vues par le robot. Pour simuler les erreurs d'un environnement réel, nous rajoutons un bruit gaussien sur les valeurs.

6.2 Construction d'une carte de l'environnement

Tout d'abord, le *middleware* ROS nous permet d'utiliser des outils utilisés par la communauté. Ainsi, nous allons définir notre matrice de la même façon qu'une *occupancy grid* [21], qui est utilisée pour représenter une carte d'occupation lors de l'utilisation de LiDAR [22]. Dans notre cas, la carte ne représentera pas des obstacles mais des points d'intérêts. Chaque élément de la matrice prendra donc la valeur -1 si son statut est inconnu, 0 s'il n'y a pas de plants ou une valeur entre 1 et 100 correspondants à une probabilité de présence. Une valeur de 100 correspond donc à la certitude d'avoir un plant à un endroit donné.

La taille de la matrice est définie par l'utilisateur. Il fournit la taille de la carte ainsi que des *offsets*, pour définir l'espace dans lequel se déroulera la mission. Une résolution est aussi fixée par l'utilisateur. Elle correspond à la taille du maillage. Plus la résolution est petite, meilleurs seront les résultats. Cependant, les calculs nécessaires seront beaucoup plus longs car la matrice sera plus grande. L'utilisateur fixe aussi une orientation du champ, afin de faire coller le plus possible la carte à la réalité du terrain et réduire la taille de la matrice nécessaire.

Nous définissons maintenant 3 systèmes de coordonnées qui sont utilisés dans le code :

— Système global : l'origine est la position du robot au démarrage. Orienté avec le standard ENU (East, North, Up), donc une orientation de 0 degré correspond à un robot dirigé vers l'est. L'unité est le mètre.

— Système local : système lié à la carte, l'origine est définie par rapport au système global par l'utilisateur avec les variables *x_offset* et *y_offset*. Les limites sont définies par les variables *height* et *width* (voir figure 6.6). Ce système est orienté par l'utilisateur de façon à avoir les lignes dans la longueur de la carte et ainsi optimiser la taille de la carte.

— Système carte : système lié au tableau représentant la carte. Le système est donc discret, avec une résolution définie par l'utilisateur. L'origine est définie comme le système local, mais l'unité n'est plus le mètre mais un nombre entier.

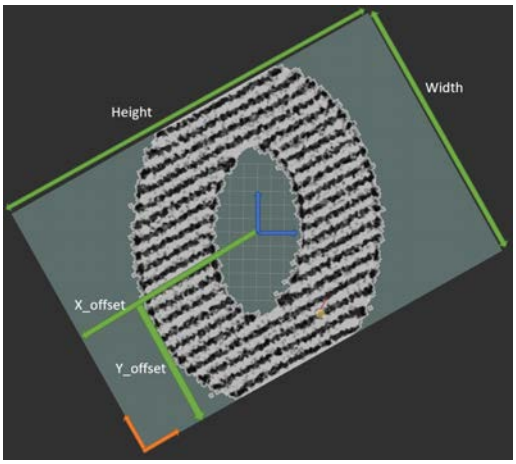


Fig. 6.6 – Les différents paramètres qui définissent une carte

Lors de la réception de la position, un changement de repère est effectué, pour passer du repère de la caméra au repère de la carte. L'utilisation des coordonnées homogènes permet de simplifier les calculs. L'ajout d'un noyau gaussien centré sur la position de chaque plante fait apparaitre les rangées de forte probabilité de présence des plantes. Les paramètres de ce noyau sont tels que les éléments de la matrice situé à moins de 15 cm du centre du noyau sont modifiés. Cette valeur arbitraire a été choisie pour prendre un compte les erreurs en positionnement des plantes par la caméra et le positionnement du robot. La probabilité de présence autour de la position de chaque plante est donc augmentée, et après quelques répétitions nous avons une bonne estimation des lignes de plants.

Afin d'oublier les plantes mal placées et de diminuer les erreurs, nous retirons une valeur fixe à tous les pixels dans le champ de vision du robot. Le champ de vision du robot est déterminé expérimentalement en plaçant 4 plantes dans les positions extrêmes (voir figure 6.7).

Fig. 6.7 – Champ de vision du robot. Les plots jaunes sont à la limite de la détection de la caméra

Pour les essais en laboratoire, nous utilisons des plots de couleurs à la place des plantes car sinon il nous faudrait commander beaucoup de plantes trop régulièrement. Le réseau de neurones a été réentraîné pour détecter les plots, de la même manière qu'il détecte les plantes.

Nous pouvons ensuite mesurer les coordonnées des plots dans la réalité, qui donneront les coordonnées des sommets d'un polygone traduisant les limites du champ de vision du robot. Ainsi, nous retranchons une valeur à tous les éléments de la matrice dont les coordonnées se trouvent dans ce polygone.

Le polygone est défini par une liste des coordonnées des sommets, données dans le sens trigonométrique.

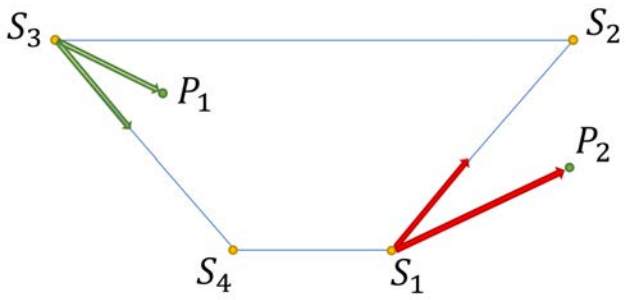


Fig. 6.8 – Polygone représentant le champ de vision. Le point P2 n'est pas dans le polygone car le produit vectoriel est positif

La condition d'appartenance du point de coordonnée $P(x, y)$ dans le polygone défini par les n sommets $S_i(x_i, y_i)$ se trouve en appliquant l'algorithme 1 qui calcul n produit vectoriel, et qui regarde le signe de chaque produit vectoriel. S'il est négatif, le point est à gauche du vecteur. L'appartenance se définie alors comme étant à gauche de tous les côtés (dans notre cas où les sommets sont donnés dans le sens trigonométrique, et polygone convexe).

Algorithm 1 Détermine l'appartenance du point P dans le polygone défini par les n sommets $S_i(x_i, y_i)$

$i \leftarrow 0$
while $i < n$ **do**
  %Définissions des vecteurs $\overrightarrow{S_iP}$ et $\overrightarrow{S_iS_{i+1}}$:
  $vp_i^x \leftarrow x - x_i$
  $vp_i^y \leftarrow y - y_i$
  $v_i^x \leftarrow x_{i+1} - x_i$
  $v_i^y \leftarrow y_{i+1} - y_i$
  %Calcul du produit vectoriel des 2 vecteurs :
  $pv \leftarrow vp_i^x * v_i^y - v_i^x * vp_i^y$
  if pv > 0 **then**
    return FALSE
  end if
  $i \leftarrow i + 1$
end while
return TRUE

En appliquant l'algorithme 1 sur tous les points de la matrice, nous savons à quels éléments nous devons retrancher la valeur. Nous en profitons aussi pour passer la valeur des éléments

négatifs à 0, pour traduire les endroits où le robot est passé. Le champ de vision ainsi que les plots vus sur la figure 6.8 sont visible sur la figure 6.9.

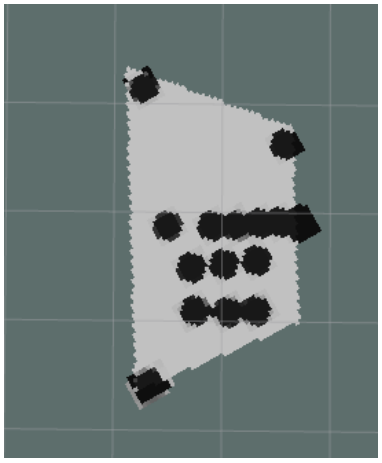


Fig. 6.9 – Champ de vision du robot réel

L'utilisation de la bibliothèque Numpy permet de réduire grandement les temps de calculs (environ 50ms pour une matrice représentant un champ de 10m par 5m avec une précision de 4cm), mais l'algorithme reste relativement lourd pour une plus grosse matrice. Pour optimiser les calculs sur la carte ainsi que pour diminuer la taille de celle-ci pour l'envoie du *topic* ROS, nous pouvons imaginer le découpage de la carte en sous-carte. Cette technique est souvent utilisée dans le développement de jeux vidéo pour ne charger que les *chunks* (morceaux de cartes) proche du joueur. Nous pourrions alors travailler sur 9 sous-cartes, centrées sur la position du robot. La taille des sous-cartes serait à définir (3x3 mètre par exemple). Cela permettrait aussi de sauvegarder une carte pour une utilisation ultérieure. La bibliothèque python Numpy permet de gérer facilement la lecture et l'écriture de matrices dans des fichiers CSV. En connaissant le mouvement du robot, nous pourrions prévoir quels morceaux de cartes il faudrait bientôt charger et utiliser du parallélisme pour aller lire les fichiers correspondants, afin d'avoir les données disponibles au bon moment.

Pour améliorer la création de la carte, un calcul pour avoir une taille de noyau gaussien dynamique pourrait aussi être réalisé. Lors du changement de repère de la position des plants, pour passer du repère de la caméra au repère global, des erreurs apparaissent à cause de l'erreur de positionnement du robot. Aussi, l'erreur en angle aura un plus fort impact pour les plantes éloignées que pour les plantes proches. Nous sommes donc plus précis pour des plantes poches, nous pourrions donc utiliser un noyau gaussien plus ou moins grand en fonction de la distance à la caméra. La taille du noyau pourrait être calculée en utilisant le calcul par intervalle, outil mathématique très utile pour propager les erreurs.

6.3 Recherche des lignes de plants courbés

Afin de rechercher les lignes de plantes sur notre carte, nous voulons utiliser du traitement d'image OpenCV [23]. Ainsi, notre tableau est converti en une image binaire en utilisant la technique du *threshold*. Nous définissons un seuil arbitraire et toutes les valeurs du tableau plus grandes que ce seuil correspondront à un pixel de valeur 1, sinon ce sera un pixel de valeur 0

(figure 6.10).

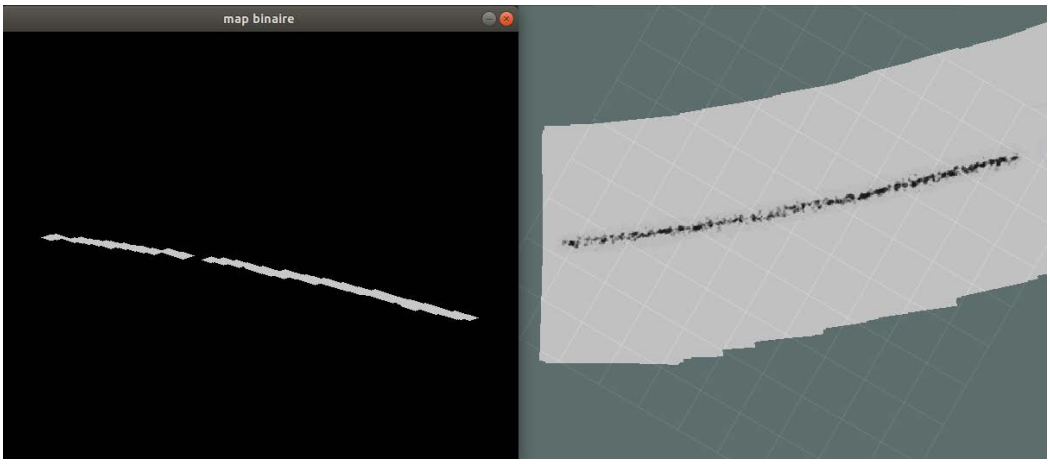


Fig. 6.10 – Image binarisée à gauche, avec la carte correspondante à droite. Les images OpenCV sont définies avec l'axe y positif vers le bas

Une succession de dilatation et d'érosion permet de filtrer les points isolés et de regrouper les points proches. La fonction *findContours* d'OpenCV renvoie une liste de contours entourant les différents nuages de points présents dans l'image. Pour chaque contour trouvé, nous calculons les coefficients du polynôme du second degré qui passent le mieux par le nuage des points. Pour cela, nous utilisons la méthode des moindres carrés. Comme la fonction *findContours* renvoie les coordonnées des points formant le contour du nuage de point, nous dessinons sur une image temporaire les contours avec la fonction openCV *drawContours*, en précisant dans les paramètres de remplir la figure. Tous les points non nuls de cette image sont donc des points du nuage de point, qui sont ensuite utilisés pour calculer les coefficients. Il est à noter que les dernières version d'OpenCV utilisent des tableaux Numpy pour stocker les images, nous pouvons donc utiliser facilement des fonctions de ces 2 bibliothèques sans modification du type de données. Pour chaque contour, nous avons alors les coefficients du polynôme et un rectangle englobant (*bounding box*).

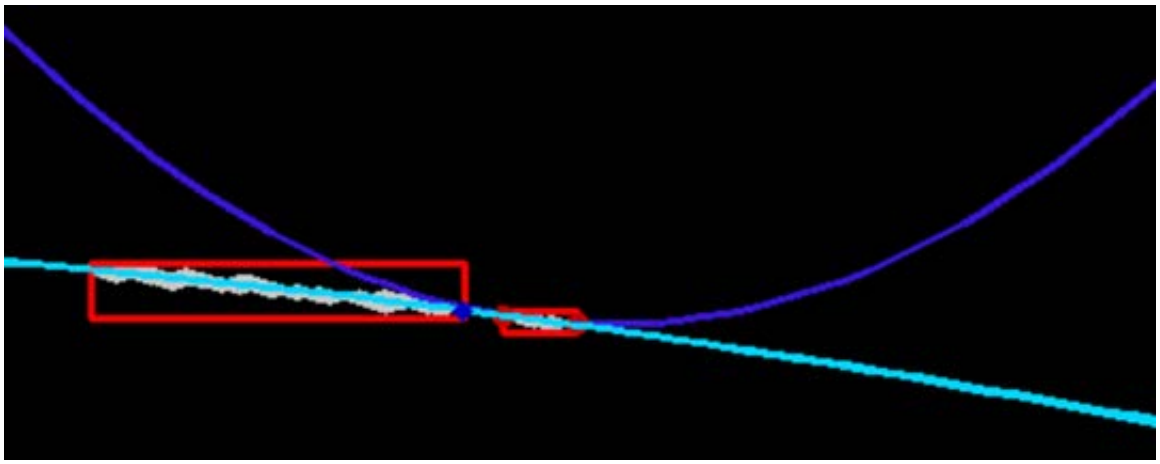


Fig. 6.11 – Approximation des polynômes qui passent dans les nuages de points, et bounding box des nuages de points

Un filtrage est appliqué pour combiner 2 nuages de points si le polynôme approximant de l'un passe suffisamment proche de l'autre nuage de points. Pour cela, nous prenons 2 abscisses extrêmes du second nuage de point. Nous choisissons les points à gauche et à droite, dont l'abscisse est

donnée par sa *bounding box* (en rouge sur la figure 6.11), pour le nuage de point à gauche, couleur cyan). Pour ces 2 points, nous calculons l'ordonnée correspondant aux 2 polynômes approximant. Si la différence entre ces 2 ordonnées est suffisamment petite, alors le premier polynôme passe par le second nuage de points, donc les nuages de points sont à combiner.

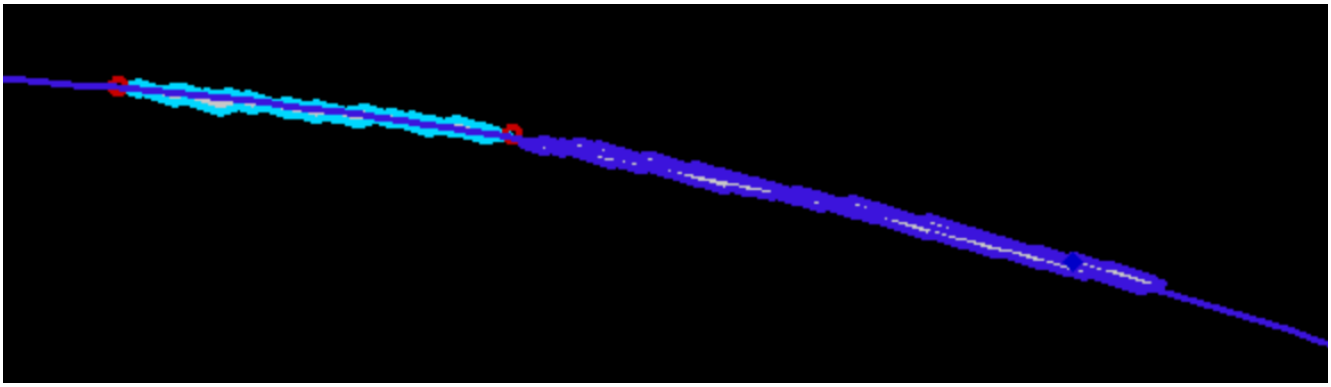


Fig. 6.12 – Combinaison de 2 nuages de points

Pour combiner les nuages de points, nous créons une image temporaire sur laquelle nous affichons les 2 nuages de points, pour ensuite appliquer une nouvelle estimation par les moindres carrés (figure 6.12).

Un second filtrage est ensuite appliqué pour enlever les polynômes aberrants, c'est-à-dire ceux dont les coefficients sont les plus éloignés des autres. Pour se faire, nous calculons une moyenne des coefficients, pondérée par la taille du nuage de point. Ainsi, les longs nuages de points, qui représentent mieux la réalité, auront plus de poids que les courts. La moyenne se fait pour les coefficients de degré 1 et 2 du polynôme, le coefficient de degré 0 représentant l'ordonnée à l'origine, il est différent pour chaque courbe (figure 6.13).

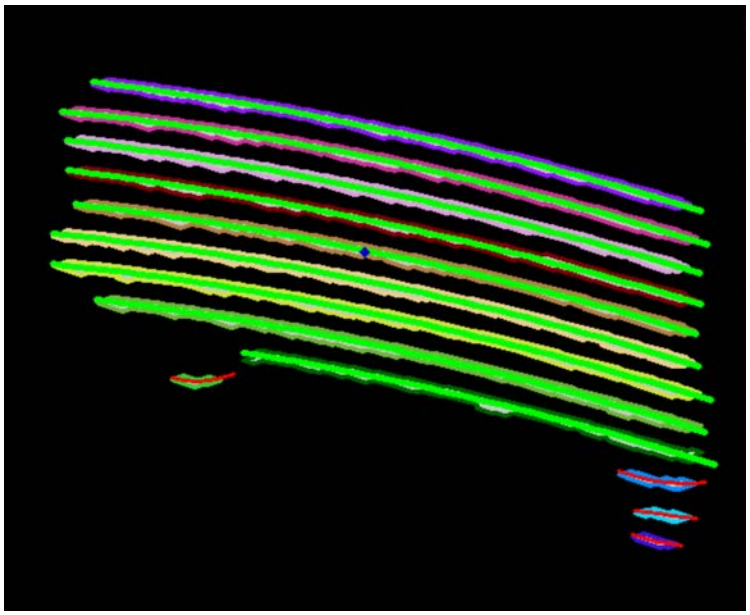


Fig. 6.13 – Filtrage par moyenne pondérée. En vert, les courbes acceptées, en rouge les courbes rejetées

Pour aider au traitement d'image, nous affichons sur l'image binaire initiale les polynômes qui ont été acceptés à l'itération précédente.

Nous avons maintenant une liste de courbe que le robot peut potentiellement suivre, il faut encore trouver quelle courbe le robot doit suivre et trouver la commande que le robot doit appliquer

pour suivre une courbe donnée.

6.4 Définition des objectifs

Pour faciliter la construction d'une mission, nous définissons 3 objectifs élémentaires que le robot peut réaliser. Nous avons donc défini les objectifs suivi de ligne, suivi de cercle et suivi de polynôme du second degré.

Pour faciliter l'implémentation et la clarté du code, nous utilisons de la programmation orientée objet. Nous pouvons ainsi tirer parti des héritages et du polymorphisme afin de créer des fonctions qui seront surchargées par les classes filles. Nous avons donc défini une classe mère Objectif et les 3 classes filles Line, Circle et Curve correspondant aux 3 objectifs élémentaires.

Les fonctions surchargées sont :

— La fonction *getCommand*, qui prend en argument la position estimée du robot et qui retourne la commande en cap et vitesse que le robot doit prendre pour réaliser l'objectif.

— La fonction *isFinished*, qui peut prendre en argument la position du robot et qui dans ce cas vérifie si l'objectif est réalisé. Si l'objectif a été réalisé, la fonction retourne le booléen True.

— La fonction *getType* qui renvoie un entier correspondant au type d'objectif (1 pour ligne et 2 pour cercle, 3 pour courbe). Cette fonction est utilisée pour l'affichage des objectifs.

Le robot étant contrôlé en cap, la commande est calculée en utilisant la technique des champs de potentiel. Cette méthode consiste à modéliser des forces attractives ou répulsives et de calculer le gradient en un point pour trouver le cap à prendre [24]. Connaissant des forces élémentaires (points attractifs, lignes attractives et cercle dans notre cas), Il faut les combiner et convertir les objets mathématiques avec nos données.

Pour chaque objectif, nous allons maintenant définir les paramètres nécessaires pour les caractériser, ainsi que le calcul effectué par le contrôleur en cap et la condition de fin. On notera $[x_{rob}, y_{rob}, \theta_{rob}]$ l'état du robot.

6.4.1 Suivi de ligne

Paramètres

Pour le suivi de ligne, les paramètres à fournir (voir figure 6.14) sont les coordonnées du point d'arrivé $[x_{start}, y_{start}]$ et les coordonnées du point final $[x_{end}, y_{end}]$. Il faut aussi donner une distance de validation $dist_{valid}$.

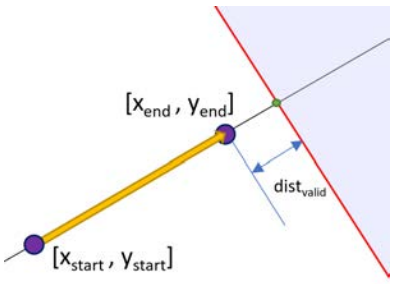


Fig. 6.14 – Paramètres pour définir un objectif suivi de ligne

Commande en cap

La commande en cap est calculée en combinant 2 forces élémentaires [24]. Nous combinerons ainsi une force pour une ligne attractive et une force de point attractif, vers un point situé après le point de fin de ligne. Ces 2 forces sont multipliées par un coefficient compris entre 0 et 1, proportionnel à la distance entre le robot et la ligne, pour qu'un robot éloigné se rapproche fortement de la ligne et qu'un robot proche se contente de suivre la ligne. Ce coefficient sera nul si le robot est plus éloigné de la ligne qu'une distance donnée et vaudra 1 si le robot est sur la ligne. Le calcul de la distance entre un point et une droite se fait en calculant l'air d'un triangle de 2 manières différentes [25].

$$vect_x \leftarrow x_{end} - x_{start}$$

$$vect_y \leftarrow y_{end} - y_{start}$$

$$norm \leftarrow \sqrt{vect_x^2 + vect_y^2}$$

$$P_x \leftarrow x_{robot} - x_{end}$$

$$P_y \leftarrow y_{robot} - y_{end}$$

$$N_x \leftarrow -\frac{vect_y}{norm}$$

$$N_y \leftarrow \frac{vect_x}{norm}$$

$$W1_{ligne} \leftarrow -N_x * (N_x * P_x - N_y * P_y)$$

$$W2_{ligne} \leftarrow -N_y * (N_x * P_x - N_y * P_y)$$

$$x_{waypoint} \leftarrow \frac{vect_x}{norm} * 2 * dist_{valid} + x_{end}$$

$$y_{waypoint} \leftarrow \frac{vect_y}{norm} * 2 * dist_{valid} + y_{end}$$

$$Px_w \leftarrow x_{robot} - x_{waypoint}$$

$$Py_w \leftarrow y_{robot} - y_{waypoint}$$

$$norm_w \leftarrow \sqrt{Px_w^2 + Py_w^2}$$

$$W1_{waypoint} \leftarrow \frac{Px_w}{norm_w}$$

$$W2_{waypoint} \leftarrow \frac{Py_w}{norm_w}$$

$$dist \leftarrow \frac{vect_x * (y_{end} - y_{robot}) - vect_y * (x_{end} - x_{robot})}{norm}$$

$$coeff \leftarrow \frac{min(max(d - abs(dist), 0), d))}{d}$$

$$W_1 \leftarrow (1 - coeff) * W1_{ligne} + coeff * W1_{waypoint}$$

$$W_2 \leftarrow (1 - coeff) * W2_{ligne} + coeff * W2_{waypoint}$$

$$cap \leftarrow arctan2(W_2, W_1)$$

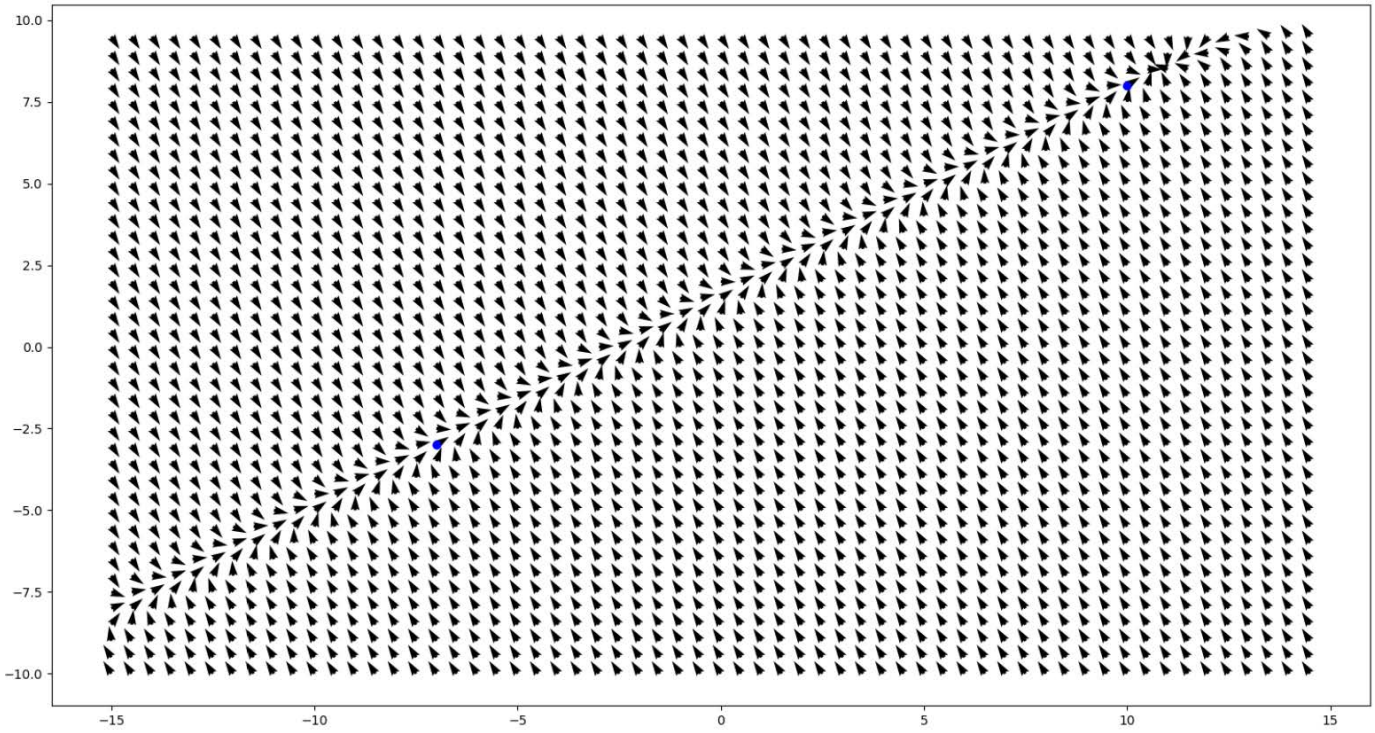


Fig. 6.15 – Représentation du champ de potentiel pour un suivi de ligne

Condition de fin

Le robot a fini la ligne lorsque qu'il dépasse le point final de la distance $dist_{valid}$. Nous utilisons la technique des demi-plans pour plus de robustesse. Ainsi nous définissons un demi-plan dans lequel nous considérons que le robot a réussi son objectif (en bleu sur la figure 6.14). Ce demi-plan est défini par un vecteur qui est normal à la ligne, et qui passe par le point de fin. Le robot est dans ce demi-plan si la distance algébrique entre ce vecteur et la position du robot est plus grande que le paramètre $dist_{valid}$. La localisation du robot n'étant pas parfaite, un compteur est incrémenté à chaque fois que le robot se trouve dans le demi-plan et l'objectif est considéré réalisé si ce compteur est plus grand que 5.

$$vect_x \leftarrow x_{end} - x_{start}$$
$$vect_y \leftarrow y_{end} - y_{start}$$
$$vect_{xnormal} \leftarrow -vect_y$$
$$vect_{ynormal} \leftarrow vect_x$$
$$disy \leftarrow \frac{vect_{xnormal} * (y_{end} - y_{robot}) - vect_{ynormal} * (x_{end} - x_{robot})}{\sqrt{vect_x^2 + vect_y^2}}$$
$$fini \leftarrow dist > dist_{valid}$$

6.4.2 Suivi de Cercle

Paramètres

Pour le suivi de cercle, il faut donner les coordonnées du centre du cercle $[x_c, y_c]$. Il faut aussi fournir le rayon du cercle $[r]$ ainsi que le sens de parcours $[s]$, qui prend la valeur 1 pour un suivi

de cercle dans le sens trigonométrique et -1 pour le sens horaire. Nous donnerons aussi un cap de sortie $[cap_{sortie}]$pour la condition de fin d'objectif.

Commande en cap

La commande pour le suivi de cercle est directement donnée [24].

$$p_1 \leftarrow \frac{x_{robot} - x_c}{r}$$

$$p_2 \leftarrow \frac{y_{robot} - y_c}{r * sens}$$

$$W_1 \leftarrow \frac{-p_1^3 - p_1 * p_2^2 + p_1 - p_2}{}$$

$$W_2 \leftarrow \frac{-p_2^3 - p_2 * p_1^2 + p_1 + p_2}{}$$

$$cap \leftarrow arctan2(W_2, W_1)$$

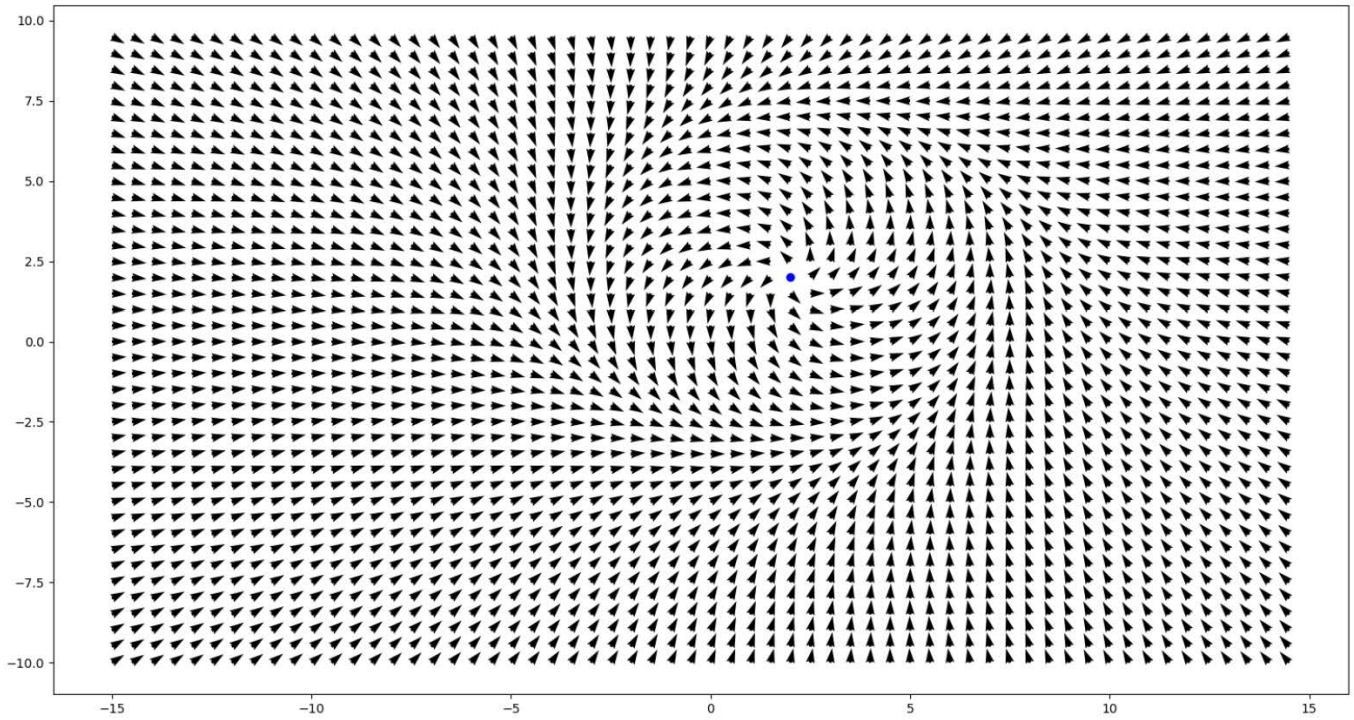


Fig. 6.16 – Représentation du champ de potentiel pour un suivi de cercle

Condition de fin

La condition de fin pour le suivi de cercle est une condition angulaire, nous comparons le cap du robot et le cap de sortie cap_{sortie} donné en paramètre.

$$fini \leftarrow abs(sawtooth(theta_{robot} - cap_{sortie})) < 0.1$$

$$\text{avec } sawtooth(x) \leftarrow 2 * arctan(tan(\frac{x}{2}))$$

6.4.3 Suivi de Courbe

Paramètres

Pour le suivi de courbe, il faut donner en paramètres l'abscisse de gauche [x_{start}] et l'abscisse de droite [x_{end}] correspondant aux limites de la courbe, ainsi qu'un paramètre [$sens$] qui vaut 1 si on va dans le sens des abscisses positif et -1 sinon. Il faut en outre donner les coefficients du polynôme [a, b, c] ainsi que l'orientation [o] du repère de la courbe par rapport au repère global. Il faut aussi donner une distance de validation $dist_{valid}$.

Le paramètre d'orientation permet de gérer les changements de repère lors de la réception de la position du robot pour le calcul de la commande et de la condition de fin. En effet, les coefficients du polynôme sont trouvés dans un repère local, il est plus facile de passer les coordonnées du robot dans ce repère que de trouver les coefficients du polynôme dans le repère global.

Commande en cap

Pour calculer la commande en cap pour le suivi de courbe, nous voulons réutiliser des formules de force simple. Pour cela, nous allons approximer le polynôme par une droite tangente à ce dernier, tel que le point tangent soit le plus proche du robot.

Pour commencer, il nous faut définir la distance entre un point et un polynôme du second degré.

$$\text{Soit } P : [x_p, y_p] \text{ un point du plan}$$

$$\text{Soit } C : [y = ax^2 + b * x + c] \text{ un polynome du second degré}$$

$$x_m \text{minimise la distance entre le point P et le polynome C}$$

$$\Leftrightarrow \forall x \in \mathbb{R}, dist(P, [x_m, a * x_m^2 + b * x_m + c]) \leq dist(P, [x, a * x^2 + b * x + c])$$

$$\text{avec dist la norme euclidienne}$$

En pratique, l'abscisse x_m est calculé en utilisant un algorithme de type descente de gradient, qui trouve un minimum local, suffisant pour notre application. L'algorithme 2 ne nécessite pas de calculer la dérivée de la distance et s'arrête lorsque l'erreur sur l'abscisse x_m est plus petite qu'un *epsilon* donné. L'utilisateur fourni une première approximation x_{start} de x_m afin de donner un point de départ à l'algorithme. Le calcul de distance se fait en utilisant le théorème de Pythagore.

Algorithm 2 Calcul de l'abscisse x_m qui minimise la distance entre le point $P[x_p, y_p]$ et le polynome de coefficients [a, b, c]

Require: $x_p, y_p, a, b, c, x_{start}, eps$
  $pas \leftarrow 64 * eps$
  $sens \leftarrow 1$
  $x_0 \leftarrow x_{start}$
  $y_0 \leftarrow a * x_0^2 + b * x_0 + c$
  $dist_0 \leftarrow dist([x_p, y_p], [x_0, y_0])$
  while $pas > eps$ **do**
    $x_{next} \leftarrow x_0 + sens * pas$
    $y_{next} \leftarrow a * x_{next}^2 + b * x_{next} + c$
    $dist_{next} \leftarrow dist([x_p, y_p], [x_{next}, y_{next}])$
    if $dist_{next} < dist_0$ **then**
      $x_0 \leftarrow x_{next}$
      $dist_0 \leftarrow dist_{next}$
    else
      $sens \leftarrow -sens$
      $x_{next} \leftarrow x_0 + sens * pas$
      $y_{next} \leftarrow a * x_{next}^2 + b * x_{next} + c$
      $dist_{next} \leftarrow dist([x_p, y_p], [x_{next}, y_{next}])$
      if $dist_{next} < dist_0$ **then**
        $x_0 \leftarrow x_{next}$
      else
        $pas \leftarrow pas/2$
      end if
    end if
  end while
  return $(x_0, dist_0)$

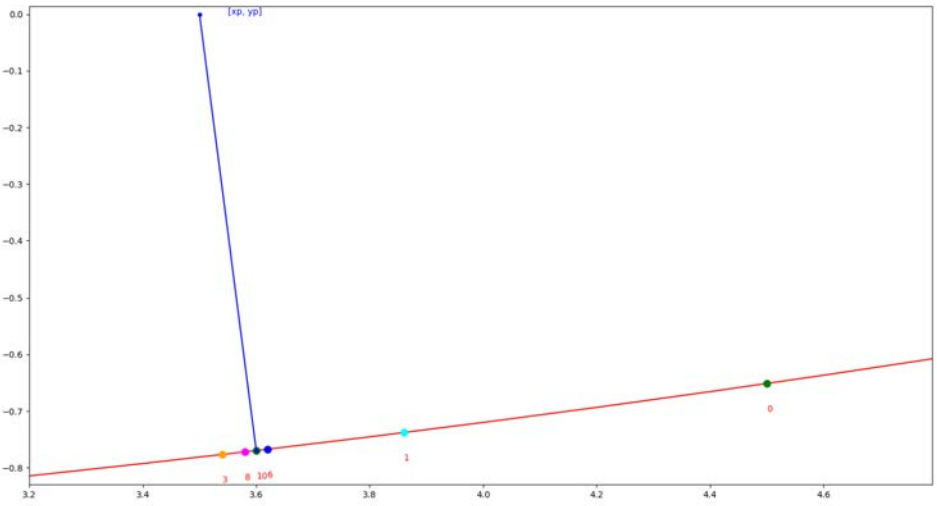


Fig. 6.17 – Itérations successives de l'algorithme de recherche de minimum

Sur la figure 6.17, nous observons les itérations successives. Le nombre en rouge correspond au numéro de l'itération, donc l'algorithme 2 commence à l'itération 0 en vert à droite jusqu'à l'itération 10. Lorsque le pas est divisé par 2, une itération est réalisée sans modifier l'abscisse. L'abscisse oscille autour du résultat final jusqu'à être suffisamment proche.

Nous pouvons maintenant trouver le point du polynôme qui minimise la distance à la position du robot. La pente de la tangente est calculée en évaluant la dérivée du polynôme en l'abscisse correspondante. Les formules de suivi de ligne sont désormais applicables. La combinaison d'une ligne attractive et d'un champ constant donne un bon résultat, et permet d'orienter facilement le suivi de courbe.

Les détails du calcul de la commande en cap sont donnés ci-dessous :

$$x_c, dist \leftarrow recherche\_min(x_{robot}, y_{robot}, a, b, c, x_{robot}, 0.01)$$

$$y_c \leftarrow a * x_c^2 + b * x_c + c$$

$$pente \leftarrow 2 * a * x_c + b$$

$$vect_x \leftarrow 1$$

$$vect_y \leftarrow pente$$

$$norm \leftarrow \sqrt{vect_x^2 + vect_y^2}$$

$$x_d \leftarrow x_c + vect_x$$

$$y_d \leftarrow y_c + vect_y$$

$$P_x \leftarrow x_{robot} - x_d$$

$$P_y \leftarrow y_{robot} - y_d$$

$$N_x \leftarrow -\frac{vect_y}{norm}$$

$$N_y \leftarrow \frac{vect_x}{norm}$$

$$W1_{ligne} \leftarrow -N_x * (N_x * P_x - N_y * P_y)$$
$$W2_{ligne} \leftarrow -N_y * (N_x * P_x - N_y * P_y)$$

$$W1_{constant} \leftarrow \frac{vect_x}{norm_w}$$
$$W2_{constant} \leftarrow \frac{vect_y}{norm_w}$$

$$d \leftarrow 1$$
$$coeff \leftarrow \frac{min(max(d - abs(dist), 0), d))}{d}$$
$$W_1 \leftarrow (1 - coeff) * W1_{ligne} + coeff * W1_{constant}$$
$$W_2 \leftarrow (1 - coeff) * W2_{ligne} + coeff * W2_{constant}$$
$$cap \leftarrow arctan2(W_2, W_1)$$

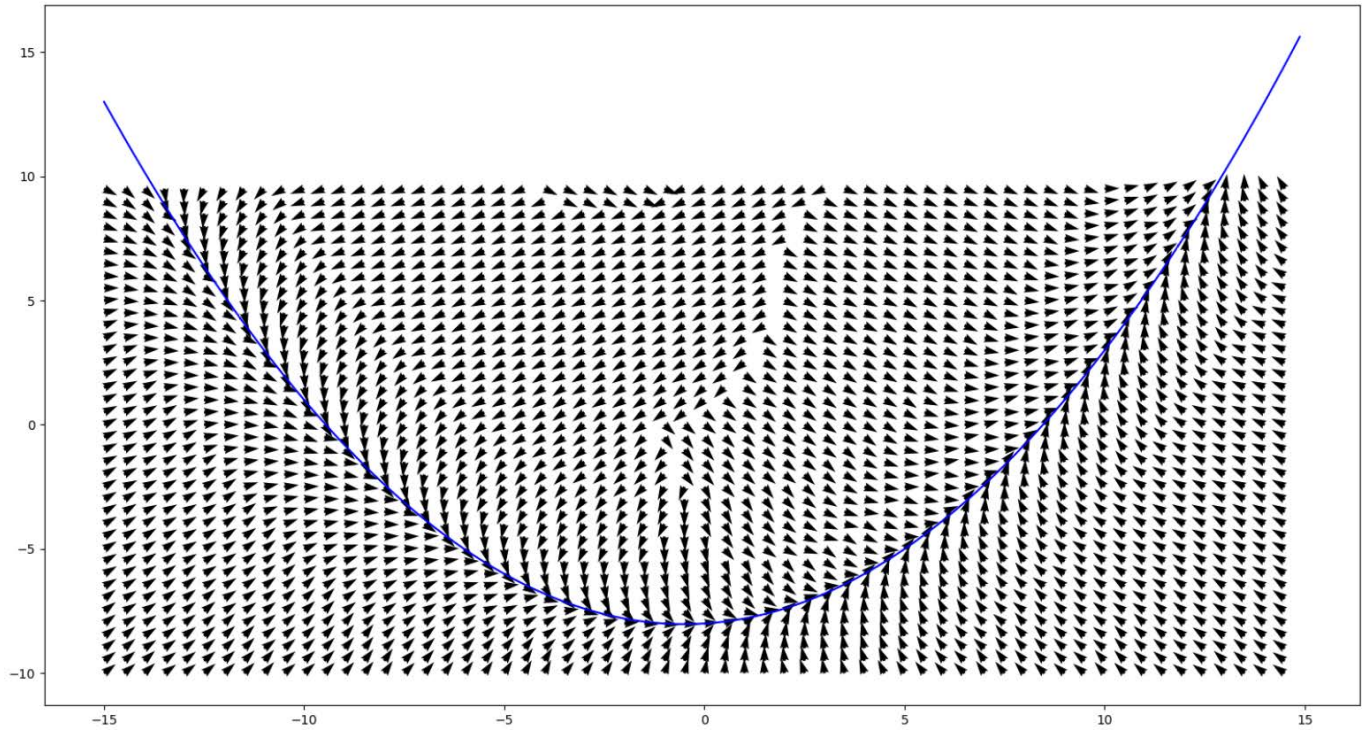


Fig. 6.18 – Champ de potentiel pour un suivi de courbe, avec signe positif

Condition de fin

De la même manière que pour le suivi de ligne, nous considérons que le robot a fini la ligne lorsque qu'il dépasse le point final de la distance $dist_{valid}$. Le point final a pour abscisse x_{end} ou x_{start} en fonction du sens de parcours de la courbe. Nous utilisons encore la technique des demi-plans pour plus de robustesse. Ce demi-plan est défini par un vecteur qui est normal à la courbe au point final. Le robot est dans ce demi-plan si la distance algébrique entre le vecteur normal et la position du robot est plus grande que le paramètre $dist_{valid}$. Pour le cas où le sens de parcours de la courbe est positif :

$$vect_x \leftarrow 1$$

$$vect_y \leftarrow 2 * a * x_{end} + b$$

$$vect_{xnormal} \leftarrow -vect_y$$

$$vect_{ynormal} \leftarrow vect_x$$

$$disy \leftarrow \frac{vect_{xnormal} * (y_{end} - y_{robot}) - vect_{ynormal} * (x_{end} - x_{robot})}{\sqrt{vect_x^2 + vect_y^2}}$$

$$fini \leftarrow dist > dist_{valid}$$

6.5 Recherche d'objectifs

Maintenant que nous avons défini nos objectifs, il faut les instancier avec les paramètres des courbes repérer sur la carte. A chaque mise à jour de la carte, nous avons une liste de courbe vue et l'estimation de la position du robot dans celle-ci.

Tout d'abord, il faut initialiser notre mission. Parmi les courbes vues, nous prenons la plus proche du robot. Comme la carte prend un peu de temps à se stabiliser au début, nous ne choisirons la première courbe que lorsqu'un critère de stabilité sera vérifié. Une liste des 10 derniers coefficients de degré 1 et 2 du polynôme calculés lors du second filtrage est gardée en mémoire. Nous considèrerons la carte stable si les écarts-types sont inférieurs respectivement à 0.02 pour le coefficient de degré 2 et 0.2 pour le coefficient de degré 1.

Au cours de la mission, il faut mettre à jour l'objectif courbe en cours en même temps que la carte. En effet, au début le robot ne voit qu'un bout de la rangée du fait du champ de vision de la caméra. Ainsi, plus le robot avance, plus la courbe vue devient grande. Il faut alors modifier les abscisses extrêmes et les coefficients du polynôme. Pour se faire, nous parcourons tous les objectifs qui ne sont pas finis et comparons la distance avec chaque courbe vue. Le critère de distance entre 2 courbes est le même que dans la partie de recherche de courbe, c'est-à-dire que nous évaluons les 2 polynômes en 2 abscisses extrêmes et nous calculons les distances entre les 2 points. Si les courbes sont proches, l'objectif est mis à jour. Pour cela, les 2 polynômes sont évalués sur 30 points régulièrement réparti sur leur longueur, et la formule des moindres carrés est utilisé pour trouver les nouveaux coefficients. Les valeurs extrêmes de l'objectif mis à jour sont trouvées en comparant les extrêmes de la courbe et de l'objectif.

Lorsqu'un objectif courbe est terminé, nous recherchons l'objectif suivant. Pour cela, nous choisissons la courbe la plus proche du robot, en ne prenant pas en compte les courbes auxquelles on peut associer un objectif déjà réalisé. Si pendant 10 mises à jour de suite aucune nouvelle courbe n'est trouvée, alors la mission est terminée et le robot s'arrête.

Pour faciliter la transition entre les rangées, les étapes sont explicitées au robot.

6.6 Stratégie de demi-tour

La stratégie de demi-tour est contrainte par la structure du robot. En effet, ce dernier ne peut pas tourner sur lui-même, il faut donc prendre en compte un rayon de braquage, qui correspond

au plus petit cercle que le robot peut parcourir. La stratégie envisagée consiste à tourner d'un quart de tour dans la direction de la prochaine courbe, de reculer et de faire un second quart de tour pour se placer en face de la prochaine courbe (voir figure 6.19).

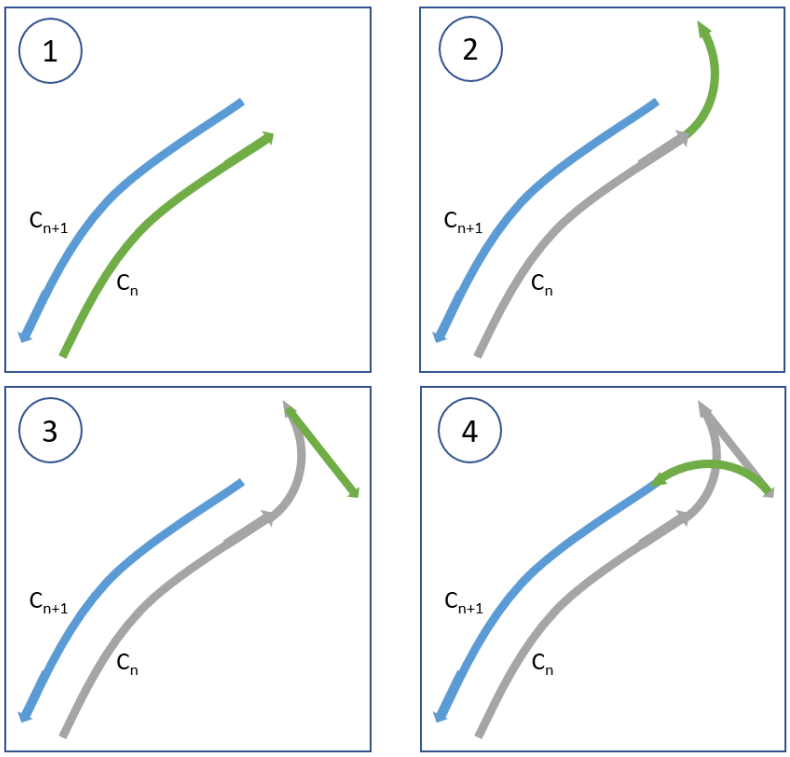


Fig. 6.19 – Stratégie de demi-tour

Les paramètres pour instancier les objectifs intermédiaires sont trouvées en faisant de la géométrie vectorielle.

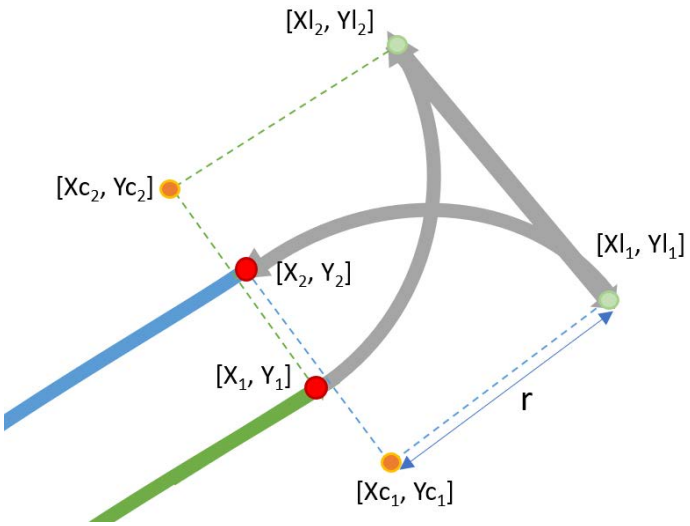


Fig. 6.20 – Paramètres nécessaires pour définir les objectifs pour le demi-tour

On pose $(a_1, b_1, c_1) et (a_2, b_2, c_2)$ les coefficients des 2 polynomes, et r le rayon de braquage

Les abscisses X_1 et X_2 sont données, on en déduit Y_1 et Y_2

.1) On définit les vecteurs colinéaires aux courbes :

$$vect^1_{col}x = 1$$

$$vect^1_{col}y = 2 * a_1 + X_1 + b_1$$

$$vect^2_{col}x = 1$$

$$vect^2_{col}y = 2 * a_2 + X_2 + b_2$$

2) On définit une variable sens, pour savoir si le premier quart de cercle va vers la droite ou la gauche :

$$sens = sign(vect^1_{col}x * (Y_2 - Y_1) - vect^1_{col}y * (X_2 - X_1))$$

3) On en déduit les vecteurs normaux :

$$vect^1_{nor}x = sens * (-vect^1_{col}y)$$

$$vect^1_{nor}y = sens * vect^1_{col}x$$

$$vect^2_{nor}x = sens * (-vect^2_{col}y)$$

$$vect^2_{nor}y = sens * vect^2_{col}x$$

4) On a alors :

$$Xc_1 = X_1 + r * \frac{vect^1_{nor}x}{\sqrt{vect^1_{nor}x^2 + vect^1_{nor}y^2}}$$

$$Yc_1 = Y_1 + r * \frac{vect^1_{nor}y}{\sqrt{vect^1_{nor}x^2 + vect^1_{nor}y^2}}$$

$$Xc_2 = X_2 - r * \frac{vect^2_{nor}x}{\sqrt{vect^2_{nor}x^2 + vect^2_{nor}y^2}}$$

$$Yc_2 = Y_2 - r * \frac{vect^2_{nor}y}{\sqrt{vect^2_{nor}x^2 + vect^2_{nor}y^2}}$$

$$Xl_1 = X_2 - r * \frac{vect^2_{nor}x}{\sqrt{vect^2_{nor}x^2 + vect^2_{nor}y^2}} + r * \frac{vect^2_{col}x}{\sqrt{vect^2_{col}x^2 + vect^2_{col}y^2}}$$

$$Yl_1 = Y_2 - r * \frac{vect^2_{nor}y}{\sqrt{vect^2_{nor}x^2 + vect^2_{nor}y^2}} + r * \frac{vect^2_{col}y}{\sqrt{vect^2_{col}x^2 + vect^2_{col}y^2}}$$

$$Xl_2 = X_1 + r * \frac{vect^1_{nor}x}{\sqrt{vect^1_{nor}x^2 + vect^1_{nor}y^2}} + r * \frac{vect^1_{nor}x}{\sqrt{vect^1_{nor}x^2 + vect^1_{nor}y^2}}$$

$$Yl_2 = Y_1 + r * \frac{vect^1_{nor}y}{\sqrt{vect^1_{nor}x^2 + vect^1_{nor}y^2}} + r * \frac{vect^1_{nor}y}{\sqrt{vect^1_{nor}x^2 + vect^1_{nor}y^2}}$$

5) Pour définir les objectifs suivi de cercle, il nous faut fournir un cap de fin :

$$cap^1_{sortie} = arctan2(vect^1_{nor}y, vect^1_{nor}x)$$
$$cap^2_{sortie} = arctan2(vect^2_{nor}y, vect^2_{nor}x) + \pi$$

6) Nous pouvons maintenant définir nos 3 objectifs :

$$circle_1 = Circle(Xc_1, Yc_1, r, sens, cap^1_{sortie})$$
$$line = LineBackward(Xl_2, Yl_2, Xl_1, Yl_1)$$
$$circle_2 = Circle(Xc_2, Yc_2, r, sens, cap^2_{sortie})$$

Un nouvel objectif suivi de ligne en marche arrière a été créé. La seule différence avec la ligne en marche avant est que la vitesse est négative. Afin de garder le robot dans le bon sens, il est aussi nécessaire de rajouter π radian au cap consigne fourni au contrôleur.

6.7 Déplacement du robot

Le robot à maintenant une liste d'objectifs à parcourir. À chaque itération, la commande en cap est calculée de façon à suivre l'objectif courant. Aussi, nous vérifions si la condition de fin est vérifiée. Nous passons à l'objectif suivant le cas échéant.

La commande en cap est envoyée au contrôleur qui en déduit une commande différentielle à envoyer aux roues gauches et droites. Le contrôleur fonctionne en utilisant la différence entre le cap mesuré et le cap consigne. Afin de prévenir la discontinuité lorsque l'angle est proche de 0, nous utilisons la fonction sawtooth [24].

On définir la fonction sawtooth :

$$sawtooth : x \to 2 * atan(tan(\frac{x}{2}))$$

Le contrôleur est maintenant défini comme suit :

Soit cap_{sortie} le cap que le robot doit suivre et cap_{robot} le cap du robot estimé par l'IMU

On cherche vit_{droite} et vit_{gauche} les vitesses à donner aux moteurs droits et gauches

On donne aussi le paramètre vitesse, qui correspond à la vitesse linéaire du robot

$$var_{vitesse} = \frac{abs(vitesse)}{3}$$
$$diff = var_{vitesse} * coeff * sawtooth(capsortie - cap_{robot})$$
$$vit_{droite} = vitesse + diff$$
$$vit_{gauche} = vitesse - diff$$

Le coefficient *coeff* doit être modifié pour contraindre le robot à ne pas tourner sur de trop petits cercles. Il est défini à 1 par défaut, des expérimentations sont nécessaire pour le fixer.

Les essais en simulation (figure 6.21) montrent que le robot est capable de réaliser une mission simulée jusqu'au bout. Le développement logiciel est donc concluant, mais il faut encore tester le robot dans la réalité.

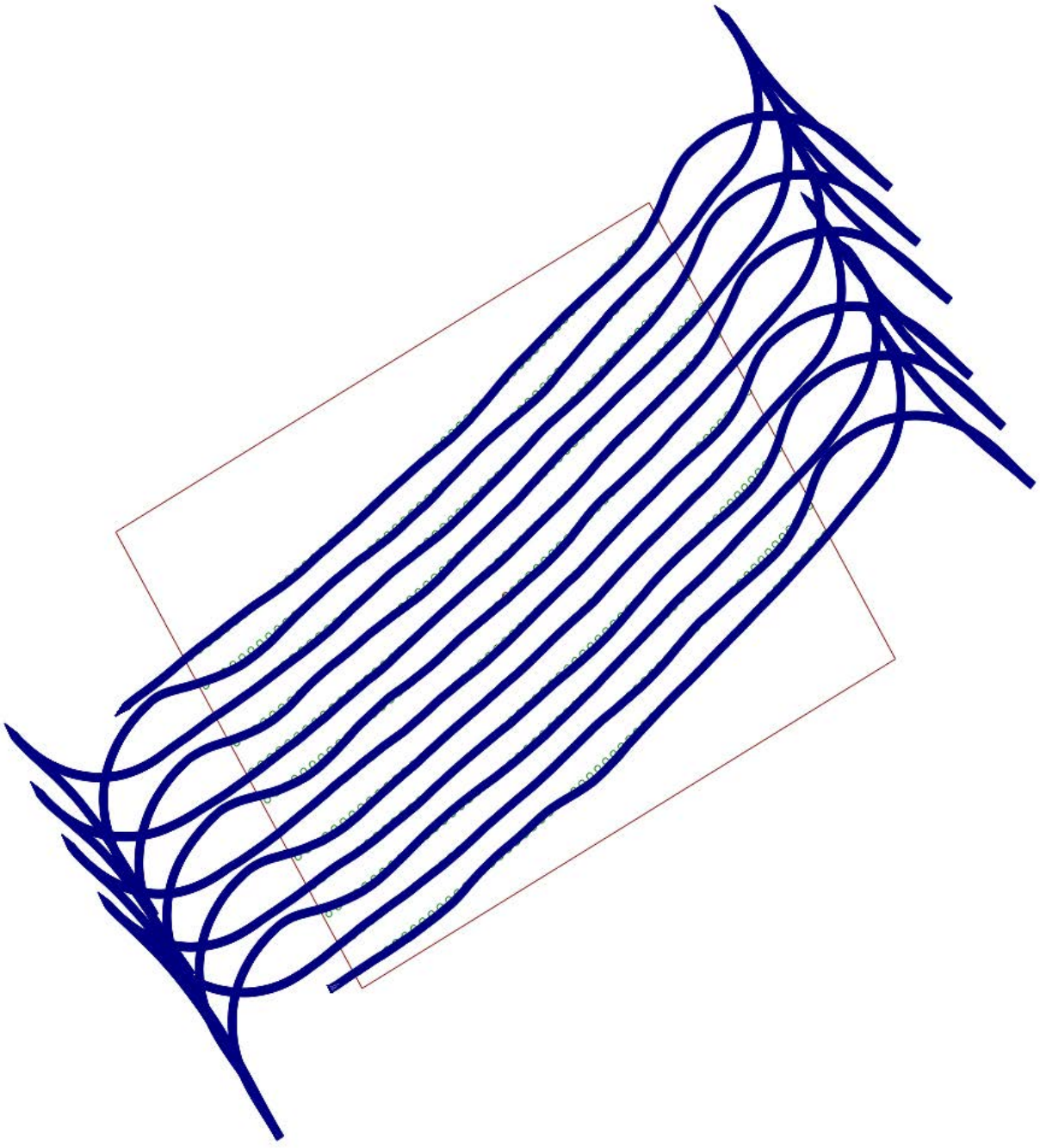


Fig. 6.21 – Réalisation du suivi des lignes de plants, en simulation

Expérimentation

Nous avons maintenant toute la chaîne de déplacement du robot, les essais réels peuvent commencer. Les essais de déplacement se font avec des plots colorés, pour éviter d'utiliser trop de plantes.

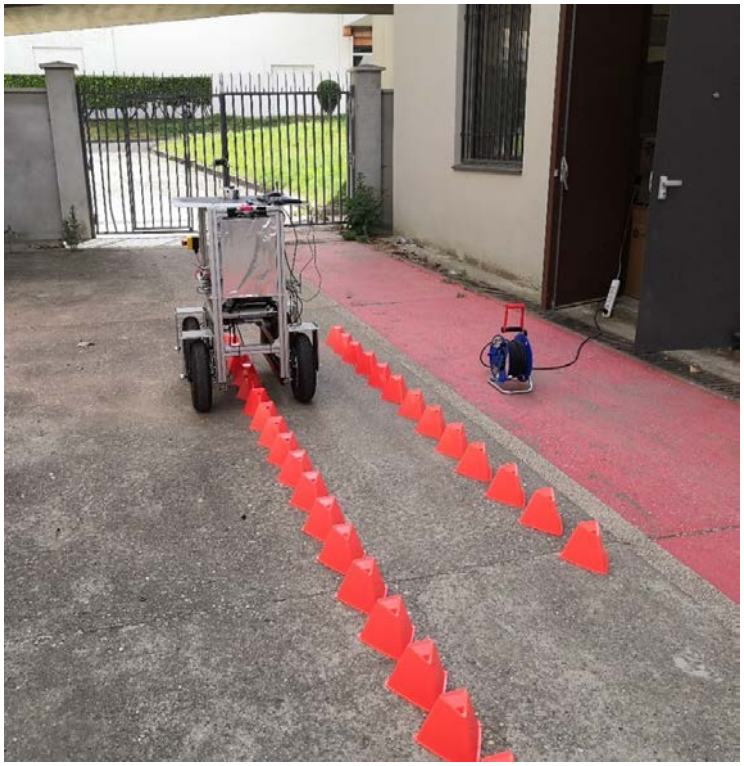


Fig. 6.22 – Essai de déplacement du robot en mode autonome

Les essais ont été concluants pour le suivi d'une rangée de plots. Le robot se place au-dessus des plots et s'arrête bien à la fin. La vitesse du robot est réglée sur 20 cm par seconde et le système réagit bien. Le programme est modifié pour ne pas faire la transition avec la rangée suivante. En effet, les essais du robot en mode manuel avec la radiocommande ont montré que le robot est difficilement contrôlable en virage à cause de son architecture. Le centre de gravité du robot est haut, et les frottements entre les roues et le goudron sont importants, donc le robot devient instable lors des virages.

Par manque de temps, il n'y a pas eu plus d'expérimentations de réalisées. Des expérimentations auront lieu courant septembre afin de déterminer le rayon de braquage du robot, ainsi que son autonomie.

Conclusion

Le développement du système de navigation est concluant pour des essais en laboratoire. Il reste encore à tester le robot dans des situations réelles, dans un champ avec des plantes.

A cause de retard de livraison, le laser de puissance n'a été installé que récemment sur le robot et aucun essai de la chaîne de destruction complète n'a encore été réalisé. Nous espérons pouvoir faire les essais en septembre pour conclure sur les performances du robot vis-à-vis du cahier des charges. Aussi, le **critère 1** du cahier des charges sur la vitesse du robot a été défini en pensant que le robot ralentirait lors de la neutralisation des pestes. Cependant, comme les algorithmes de détection des pucerons nécessitent de la lumière et que l'algorithme de ciblage du laser nécessite d'être dans le noir, le robot doit s'arrêter pendant le traitement. La vitesse de traitement s'en retrouve donc grandement affectée.

Pour l'instant, il n'y a aucune interface avec l'utilisateur. Pour démarrer une mission, il faut allumer les 2 cartes Jetson, brancher un écran avec souris et clavier et lancer les programmes à la main. Cette solution temporaire n'est pas pratique. A termes, il faudra pouvoir allumer les cartes Jetson à distance, soit lors de la mise sous tension, soit avec un *Wake-on-LAN*. Cette dernière solution permet de lancer un appareil à distance en envoyant une séquence définie sur son port Ethernet. Il faudra ensuite configurer les programmes pour qu'ils se lancent au démarrage des cartes. A la fin de la mission, il faut aussi trouver une solution pour éteindre les programmes et les cartes proprement avant la mise hors tension. Un *topic/service* ROS spécifique peut servir d'interface pour demander l'arrêt des machines. Afin de monitorer la mission à distance, il peut être envisageable de développer une interface graphique. Etant donné que le nœud ROSBridge est déjà utilisé, une interface web avec un *backend* javascript qui communique avec ROS peut être développé facilement. Le PC principal est déjà configuré comme borne wifi, nous pouvons donc facilement nous connecter avec celui-ci.

Pour ce qui est de l'étanchéité du robot à la pluie et à la poussière (**critère 6** du cahier des charges), un menuisier nous aide à installer un carter en bois autour du robot. Le carter sera équipé de portes pour faciliter l'accès aux éléments essentiels du robot (batterie, ordinateur principal, panneau de commande). Des ventilateurs ont été installés pour refroidir l'intérieur du robot, et en particulier les convertisseurs de tension. La structure avec les profilés métalliques rend le robot démontable, mais difficile à remonter. Pour ce qui est de la réparabilité du robot (**critère 7**), les pièces sont disponibles à l'achat sur internet. Cependant, les moteur-roues installés ne sont plus disponibles à la vente, donc il peut y avoir un problème d'approvisionnement si une roue devient défectueuse (ce qui nous est arrivé).

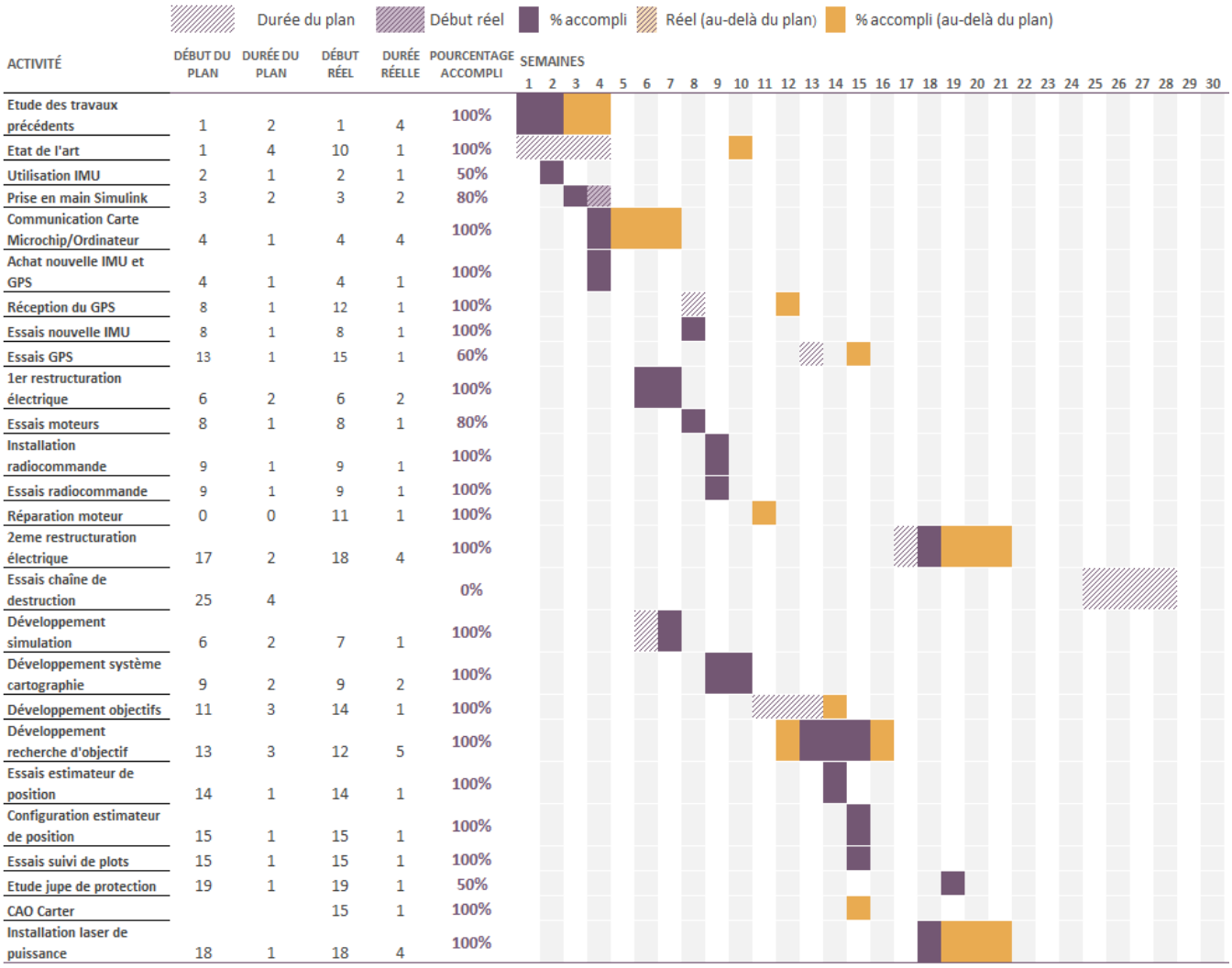
La sécurité des personnes est assurée avec les boutons d'arrêts d'urgence, et la radiocommande qui peut reprendre le contrôle du robot à tout moment (**critère 8**). En prévision des essais du

laser de puissance, nous avons commandé une jupe de protection qui résiste au faisceau du laser de puissance. Nous allons donc l'installer pour que le faisceau ne quitte jamais l'intérieur du robot (**critère 9**).

Ce projet fut très intéressant et enrichissant. Il m'a permis d'utiliser mes compétences acquises lors de ma formation et de contribuer au développement du robot. La liberté et la confiance données par les membres du laboratoire m'ont permis de m'épanouir et de mener à bien ma partie du projet. Je regrette un peu les délais et retard pris dans les commandes qui ont impacté le projet, mais personne n'en est responsable. Les expérimentations continueront en septembre, en espérant que les résultats soient concluants.

A Annexe

A.1 Diagramme de Gantt



ACTIVITÉ	DÉBUT DU PLAN	DURÉE DU PLAN	DÉBUT RÉEL	DURÉE RÉELLE	POURCENTAGE ACCOMPLI
Etude des travaux précédents	1	2	1	4	100%
Etat de l'art	1	4	10	1	100%
Utilisation IMU	2	1	2	1	50%
Prise en main Simulink	3	2	3	2	80%
Communication Carte Microchip/Ordinateur	4	1	4	4	100%
Achat nouvelle IMU et GPS	4	1	4	1	100%
Réception du GPS	8	1	12	1	100%
Essais nouvelle IMU	8	1	8	1	100%
Essais GPS	13	1	15	1	60%
1er restructuration électrique	6	2	6	2	100%
Essais moteurs	8	1	8	1	80%
Installation radiocommande	9	1	9	1	100%
Essais radiocommande	9	1	9	1	100%
Réparation moteur	0	0	11	1	100%
2eme restructuration électrique	17	2	18	4	100%
Essais chaîne de destruction	25	4			0%
Développement simulation	6	2	7	1	100%
Développement système cartographie	9	2	9	2	100%
Développement objectifs	11	3	14	1	100%
Développement recherche d'objectif	13	3	12	5	100%
Essais estimateur de position	14	1	14	1	100%
Configuration estimateur de position	15	1	15	1	100%
Essais suivi de plots	15	1	15	1	100%
Etude jupe de protection	19	1	19	1	50%
CAO Carter			15	1	100%
Installation laser de puissance	18	1	18	4	100%

Fig. A.1 – Diagramme de Gantt

Les grandes étapes du projet sont récapitulées dans le diagramme de Gantt (figure A.1). La semaine 1 correspond à la première semaine de stage (du lundi 8 mars au vendredi 12 mars 2021). Les semaines 22, 23, 24 correspondent à la fermeture du laboratoire pour les congés d'été.

La restructuration de la partie électrique du robot a eu lieu en même temps que le développement de la partie logiciel car le matériel est arrivé au fur et mesure. Les problèmes de livraison ont été récurrents tout le long du projet.

Le mois de juillet a été consacré à la finition de la gestion de l'alimentation et à l'intégration des parties de chacun sur le robot. Nous avons donc installé le laser de puissance ainsi que les micromiroirs.

Le mois de septembre sera consacré aux derniers essais pour vérifier que le robot correspond au cahier des charges. La dernière semaine (semaine 29) sera consacrée à l'écriture de documents techniques pour expliquer la mise en œuvre du robot aux personnes qui travailleront dessus plus tard.

A.2 Protocole de communication

Pour les connexions UART ou USB, il est nécessaire de définir quelles informations sont envoyées et comment elles sont codées. La vitesse consigne correspond à la vitesse calculée par le contrôleur ROS et la vitesse cible est la vitesse que les moteurs ont en consigne. La radiocommande étant prioritaire, la vitesse cible et la vitesse consigne peuvent être différentes. La variable *state_robot* est fournie par le contrôleur (0 : IDLE, 1 : arrêt, 2 : marche) et la variable *state_motors* est fournie par la carte Microchip (0 : IDLE, 1 : ROS, : radiocommande).

A.2.1 Protocole entre Ordinateur -> Microchip :

— Header : 1 octet : 255

— Vitesse droite consigne : 4 octets

— Vitesse gauche consigne : 4 octets

— Autres consignes : 1 octet

7	6	5	4	3	2	1	0
1	0	0	0	boolean : consigne ventila-teurs	boolean : consigne Leds	state_robot (MSB)	state_robot (LSB)

A.2.2 Protocole entre Microchip Back -> Ordinateur :

— Header : 1 octet : 255

— Vitesse droite cible : 4 octets

— Vitesse gauche cible : 4 octets

— Nombre d'impulsions droit : 4 octets

— Nombre d'impulsions gauche : 4 octets

— Autres consignes : 1 octet

7	6	5	4	3	2	1	0
1	0	0	0	boolean : état des ventila-teurs	boolean : état des Leds	state_motors (MSB)	state_motors (LSB)

A.2.3 Protocole entre Arduino -> Microchip Back :

— Header : 1 octet : 255

— Commande Forward-Backward : 1 octet : commande de vitesse entre 0 et 255, où 127 est la vitesse nulle et 255 vitesse à fond en avant

— Commande Left-Right : 1 octet : commande de vitesse entre 0 et 255, où 127 est une avance en ligne droite, et 255 tourner à droite

— Commande Vitesse max : 1 octet : commande de vitesse max 0 et 255, où 0 est la vitesse nulle et 255 la vitesse max

— Autres consignes : 1 octet

7	6	5	4	3	2	1	0
1	0	0	0	boolean : consigne ventila-teurs	boolean : consigne Leds	sens : 0 si marche ar-rière et 1 si marche avant	mode_manu

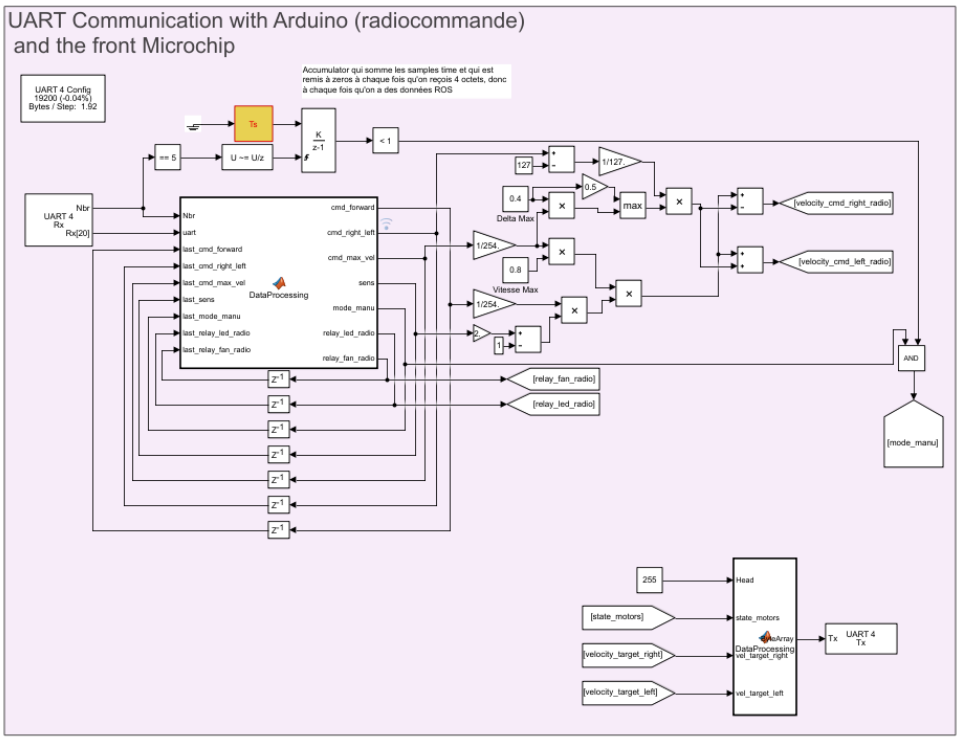


Fig. A.2 – Programme Simulink pour Microchip, pour le traitement des données venant de la radio-commande

A.2.4 Protocole entre Microchip Back -> Microchip Front :

— Header : 1 octet : 255

— Vitesse droite cible : 4 octets

— Vitesse gauche cible : 4 octets

— Autres informations : 1 octet

7	6	5	4	3	2	1	0
1	0	0	0	0	0	state_motors (MSB)	state_motors (LSB)

A.3 Programmation de la radiocommande

Liste des sorties du récepteur et fonction correspondante :

— CH1 : Direction : -100 à gauche et 100 à droite (Arduino pin 3)

— CH2 : non utilisé (Elevator) (Arduino pin 4)

— CH3 : Vitesse : -100 en bas et 100 en haut (Arduino pin 6)

— CH4 : non utilisé : Throttle (Arduino pin 7)

— CH5 : -100 pour LED, 0 pour rien et 100 pour ventilateur (Arduino pin 8)

— CH6 : Dead Man's switch : -100 si non enfoncé, 100 si enfoncé (Arduino pin 9)

— CH7 : sens Forward/Backward : -100 pour forward, 0 arrêt et 100 backward (Arduino pin 11)

— CH8 : Vitesse max : -100 si tout à gauche et 100 à droite (Arduino pin 13)

A.4 Commande des moteurs

Le robot est équipé de 4 moteurs *brushless* à courant continue (BLDC), alimentés en 12V et développant chacun une puissance de 250W. Les moteurs sont commandés par 2 cartes de la marque Microchip, et programmés avec Matlab-Simulink. La bibliothèque MPlab for Simulink, fournie par le fabricant des cartes, donne des blocs pour interagir avec les entrées-sorties des cartes. Un code de contrôle pour 1 moteur est fourni par le fabriquant de la carte, mais sans explication sur son fonctionnement. Ce code a été repris par les stagiaires précédents qui l'ont adapté pour 2 moteurs.

D'une manière générale, les moteurs sans balais (Brushless) ont des aimants permanents sur le rotor et des bobines sur le stator. Lorsque l'on fait passer un courant dans les bobines, les aimants du rotor s'alignent avec les champs magnétiques générés par les bobines et l'axe du moteur bouge. Si l'on synchronise comme il faut le mouvement du rotor et l'alimentation des bobines, on peut mettre en rotation l'axe du moteur.

Les moteurs *brushless* sont disponibles avec ou sans capteurs de position. Les moteurs *brushless* sans capteurs (moteurs *sensorless*) peuvent être commandés par des ESC (*Electronic Speed*

Hall A, B, C	Position	Secteur	Phase A	Phase B	Phase C
4 (1, 0, 0)	(-30°, 30°)	1	NC	+Vm	GND
6 (1, 1, 0)	(30°, 90°)	2	GND	+Vm	NC
2 (0, 1, 0)	(90°, 150°)	3	GND	NC	+Vm
3 (0, 1, 1)	(150°, 210°)	4	NC	GND	+Vm
1 (0, 0, 1)	(210°, 270°)	5	+Vm	GND	NC
5 (1, 0, 1)	(270°, 330°)	6	+Vm	NC	GND

TABLE A.1 – Séquence de commutation des phases

Controller) mais fournissent une vitesse arbitraire du moteur. Dans notre cas, nous avons des moteurs avec capteurs (moteurs *sensored*), nous pouvons donc les contrôler plus précisément en vitesse.

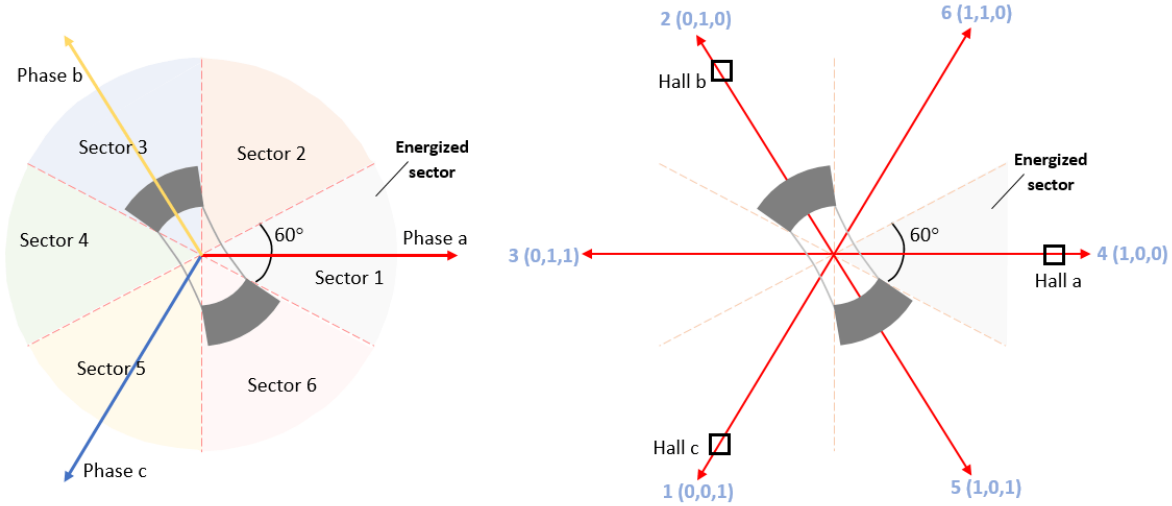


Fig. A.3 – Notre moteur est équipé de 3 capteurs à effet Hall, ce qui nous permet de distinguer 6 secteurs angulaires différents. (image : mathworks.com)

Les 3 capteurs à effets Hall qui équipent nos moteurs nous permettent de distinguer 6 secteurs angulaires (figure A.3). Aussi, à chaque secteur correspond une séquence de commutation des phases, récapitulée dans le tableau A.1. Ainsi, connaissant la position du rotor, il nous faut appliquer la séquence correspondante et le moteur tournera. C'est ce qu'on appelle la méthode de commutation 6-steps.

Nous avons maintenant un moteur auto-entraîné, c'est-à-dire qu'il va prendre de la vitesse jusqu'à arriver à un équilibre, mais nous ne contrôlons toujours pas la vitesse de rotation. Le contrôle de vitesse se fait un modifiant la valeur de la tension aux bornes de la bobine (+Vm dans le figure A.3). En faisant de la modulation de largeur d'impulsion (PWM) sur la phase, nous sommes capables de réguler la vitesse du moteur. Le rapport cyclique de notre signal PWM est obtenu par un contrôleur PID qui travaille sur la différence entre la vitesse mesurée et la vitesse consigne.

Lors de nos essais, une roue a commencé à avoir un comportement inexplicable. Après quelques essais, il s'est avéré qu'un des capteurs à effet Hall ne fonctionnait plus. La réparation du moteur

a été l'occasion de voir plus en détail comment sont construits les moteurs. Sur la figure A.4, nous observons bien le rotor au centre, avec 8 paires d'aimants permanents, ainsi que les bobines du stator. Les 3 capteurs à effets Hall sont en dessous du circuit imprimé vert, recouvert de résine époxy noire.

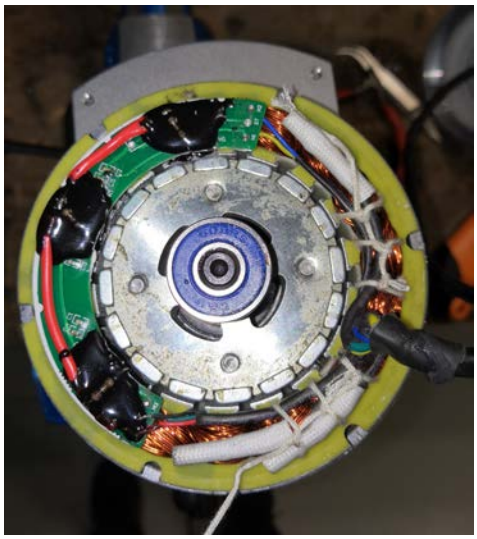


Fig. A.4 – L'intérieur d'un de nos moteurs

A.5 Recherche des lignes de plants rectiligne

Dans un premier temps, nous avons travaillé avec des rangées de plants rectilignes. Un travail avait donc été fait pour repérer les rangées et les filtrer. Les traitements sont donc expliqués dans ce paragraphe, mais ne sont plus utilisés.

Afin de rechercher les lignes de plantes sur notre carte, nous voulons utiliser du traitement d'image OpenCV. Ainsi, notre tableau est converti en une image binaire en utilisant la technique du *threshold*. Nous allons définir un seuil arbitraire et toutes les valeurs du tableau plus grande que ce seuil correspondront à un pixel de valeur 1, sinon ce sera un pixel de valeur 0. Avec notre image binaire, nous pouvons appliquer une transformée de Hough, qui nous donne une liste de lignes visibles sur l'image. Les lignes fournies sont définies par 2 points définissant les limites de notre segment.

La transformée de Hough fourni un nombre important de lignes, dont la plupart ne nous intéressent pas. Il faut donc d'abord filtrer ces lignes. Pour chaque ligne, nous calculons une direction et une longueur. Nous filtrons donc les lignes qui ne sont pas horizontales ou qui sont trop petites.

La prochaine étape consiste à regrouper les lignes proches afin de trouver les lignes passant le mieux par les rangées de plants. Pour cela, nous allons créer des paquets de lignes proches pour ensuite trouver une droite qui passe au mieux. Ainsi, pour toutes les lignes trouvées par Hough et filtrées, nous allons les ajouter au premier paquet le plus proche. Si aucun paquet assez proche n'est trouvé, nous allons créer un nouveau paquet. Les lignes étant horizontales, la distance entre 2 lignes est calculée comme la différence entre la moyenne des coordonnées verticales de chaque ligne (figure A.5).

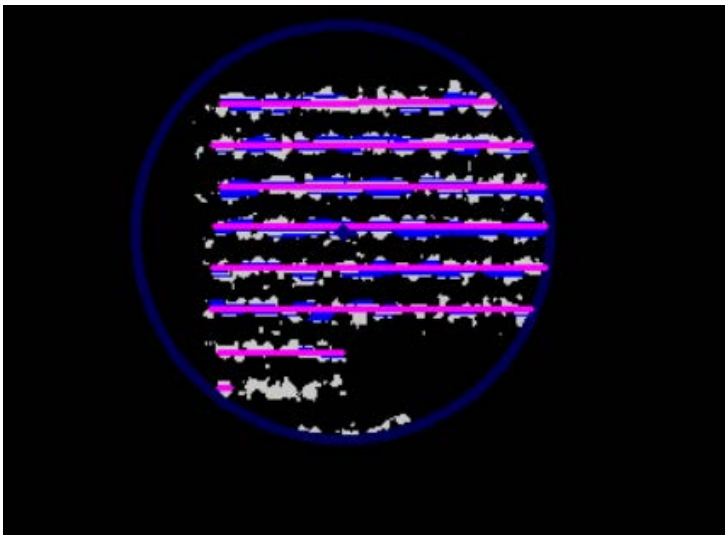


Fig. A.5 – Filtrage des lignes et regroupement

A la fin, un filtrage des paquets est réalisé. En effet, pour chaque paquet, la ligne de référence est la première ligne ajoutée au paquet, pour éviter de recalculer une moyenne à chaque ajout de ligne. Aussi, des erreurs apparaissent car nous cherchons la première ligne assez proche, qui n'est pas forcément la plus proche. Ainsi, il est nécessaire de vérifier que 2 paquets ne sont pas trop proche, et si c'est le cas de les combiner. On supprime aussi à ce moment-là tous les paquets qui ne contiennent pas assez de lignes.

Nous pouvons maintenant calculer la ligne qui passe au mieux dans les paquets. Le calcul est simplifié car les lignes sont horizontales. Aussi, la transformée de Hough d'OpenCV nous donne toujours les lignes dans le même sens, c'est-à-dire que tous les premiers points définissant les lignes sont à gauche des lignes, et les second à droite. Pour calculer les coordonnées de la droite moyenne, nous appliquons ces formules qui permettent de trouver la plus grande ligne passant au milieu.

Soit (x_1, y_1) et (x_2, y_2) les coordonnées des points extrèmes de la droite moyenne

Soit $X^1_{hough}, Y^1_{hough}, X^2_{hough}, Y^2_{hough}$ les listes des coordonnées des droites contenues dans un paquet

Alors, les coordonnées de la droite moyennes sont données par :

$$\begin{cases} x_1 = min(X^1_{hough}) \\ y_1 = mean(Y^1_{hough}) \\ x_2 = max(X^2_{hough}) \\ y_2 = mean(Y^2_{hough}) \end{cases}$$

Les coordonnées étant trouvées dans le repère de la carte, un changement de repère est nécessaire pour exploiter ces droites.

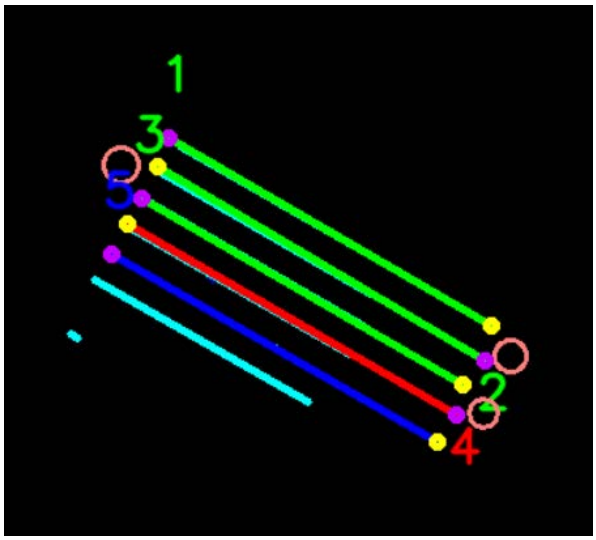


Fig. A.6 – Mission de suivi de ligne

Bibliographie

[1] R. Lenain, N. Tricot et M. Berducat, « La Robotique Agricole : L'essor de Nouveaux Outils Pour l'agroécologie, » *Sciences Eaux & Territoires*, n° 29, p. 64-67, 2019. DOI : 10.14758/SET-REVUE.2019.3.14. adresse : https://hal.archives-ouvertes.fr/hal-02372629 (visité le 18/08/2021).

[2] *Le robot laser Green Shield veut détrôner les pesticides chimiques*, juill. 2021. adresse : https://www.insa-lyon.fr/fr/actualites/robot-laser-green-shield-veut-detroner-pesticides-chimiques (visité le 18/08/2021).

[3] *GREENSHIELD 2017-2021 Contrôle Robotisé sans Pesticides Des Pestes de l'Agriculture*. adresse : https://anr-greenshield.insa-lyon.eu/fr (visité le 18/08/2021).

[4] *Home - Greenshield*. adresse : https://greenshield.fr/ (visité le 18/08/2021).

[5] *Post-équipement en systèmes de guidage par GPS*. adresse : https://www.claas.fr/produits/easy/post-equipement-en-systemes-de-guidage-par-GPS/systeme-de-guidage-automatique (visité le 18/08/2021).

[6] R. R. Shamshiri, C. Weltzien, I. A. Hameed, I. J. Yule, T. E. Grift, S. K. Balasundram, L. Pitonakova, D. Ahmad et G. Chowdhary, « Research and development in agricultural robotics : A perspective of digital farming, » *International Journal of Agricultural and Biological Engineering*, t. 11, n° 4, p. 1-14, août 2018, Number : 4, ISSN : 1934-6352. DOI : 10.25165/ijabe.v11i4.4278. adresse : https://ijabe.org/index.php/ijabe/article/view/4278 (visité le 18/08/2021).

[7] O. Barawid, A. Mizushima, K. Ishii et N. Noguchi, « Development of an Autonomous Navigation System using a Two-dimensional Laser Scanner in an Orchard Applicatio, » *Biosystems engineering.*, t. 96, n° 2, p. 139-149, fév. 2007, ISSN : 1537-5129. DOI : 10.1016/j.biosystemseng.2006.10.012. adresse : https://doi.org/10.1016/j.biosystemseng.2006.10.012 (visité le 18/08/2021).

[8] S. Fountas, N. Mylonas, I. Malounas, E. Rodias, C. Santos et E. Pekkeriet, « Agricultural Robotics for Field Operations, » *Sensors*, t. 20, p. 2672, mai 2020. DOI : 10.3390/s20092672.

[9] *ROS.org Powering the world's robots*. adresse : https://www.ros.org/ (visité le 18/08/2021).

[10] *Xsens_mti_driver - ROS Wiki*. adresse : https://wiki.ros.org/xsens_mti_driver (visité le 18/08/2021).

[11] *Nmea_navsat_driver - ROS Wiki*. adresse : https://wiki.ros.org/nmea_navsat_driver (visité le 18/08/2021).

[12] *Robot_localization - ROS Wiki*. adresse : https://wiki.ros.org/robot_localization (visité le 18/08/2021).

[13] *Setting Up Odometry — Navigation 2 1.0.0 Documentation*. adresse : https://navigation.ros.org/setup_guides/odom/setup_odom.html (visité le 18/08/2021).

[14] J. Borenstein et L. Feng, « UMBmark : A Benchmark Test for Measuring Odometry Errors in Mobile Robots, » in *Mobile Robots X*, t. 2591, International Society for Optics and Photonics, déc. 1995, p. 113-124. doi : 10.1117/12.228968. adresse : https://www.spiedigitallibrary.org/conference-proceedings-of-spie/2591/0000/UMBmark--a-benchmark-test-for-measuring-odometry-errors-in/10.1117/12.228968.short (visité le 18/08/2021).

[15] methylDragon, *Sensor Fusion in ROS*, original-date : 2018-08-03T03 :58 :04Z, août 2021. adresse : https://github.com/methylDragon/ros-sensor-fusion-tutorial (visité le 18/08/2021).

[16] *Rosbag - ROS Wiki*. adresse : https://wiki.ros.org/rosbag (visité le 18/08/2021).

[17] *Rosbridge_suite - ROS Wiki*. adresse : https://wiki.ros.org/rosbridge_suite (visité le 18/08/2021).

[18] *Rosbridge-websocket-examples/python at master · nickvaras/rosbridge-websocket-examples*. adresse : https://github.com/nickvaras/rosbridge-websocket-examples (visité le 18/08/2021).

[19] *Std_msgs/Float32MultiArray Documentation*. adresse : https://docs.ros.org/en/melodic/api/std_msgs/html/msg/Float32MultiArray.html (visité le 18/08/2021).

[20] V. Drevelle et J. Nicola, « VIBes : A Visualizer for Intervals and Boxes, » *Mathematics in Computer Science*, t. 8, n° 3-4, p. 563-572, sept. 2014, issn : 1661-8270, 1661-8289. doi : 10.1007/s11786-014-0202-0. adresse : http://link.springer.com/10.1007/s11786-014-0202-0 (visité le 18/08/2021).

[21] *Nav_msgs/OccupancyGrid Documentation*. adresse : https://docs.ros.org/en/noetic/api/nav_msgs/html/msg/OccupancyGrid.html (visité le 18/08/2021).

[22] H. Moravec et A. Elfes, « High Resolution Maps from Wide Angle Sonar, » *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, 1985. doi : 10.1109/ROBOT.1985.1087316.

[23] *Home opencv*. adresse : https://opencv.org/ (visité le 18/08/2021).

[24] L. Jaulin, *Mobile Robotics*, ISTE, éd., sér. Mobile Robotics. nov. 2015. adresse : https://hal.archives-ouvertes.fr/hal-01236489 (visité le 18/08/2021).

[25] *Distance d'un point à une droite*, Page Version ID : 182829197, mai 2021. adresse : https://fr.wikipedia.org/w/index.php?title=Distance_d%27un_point_%C3%A0_une_droite&oldid=182829197 (visité le 18/08/2021).