Polytech'Nice-Sophia – Spécialité robotique
1645 Route des Lucioles – Parc de Sophia Antipolis
06410 BIOT

Compte-Rendu De Projet R&D :
AUTOFLOW

Robotique et systèmes autonomes – Année 2024/2025

Jimmy Vu – jimmy.vu@etu.unice.fr
Younousse Soufiani – younousse.soufiani@etu.unice.fr

Table des matières

Introduction 3
I. Modélisation et prototypage 3
1. Conception 3
2. Electronique 3
II. Localisation de l'opérateur 4
III. Algorithme de suivi sans obstacle 5
IV. Contrôle des moteurs 7
V. Flux de données avec ROS2 C++ 8
VI. Simulation 3D avec ROS 9
1. Modèle URDF 9
2. Gazebo 9
3. RViz 10
VII. Navigation 10
1. LIDAR 11
2. Mapping 11
3. Algorithme de navigation 12
VIII. Problèmes rencontrés 13
1. Librairies pour contrôler les GPIOS 13
2. Temps de compilation 13
3. Mesures et précision des capteurs de distance pour la localisation 13
5. Simulation 14
Conclusion : 14

Introduction

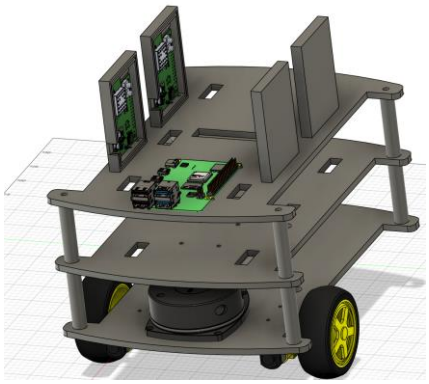
Les robots suiveurs autonomes connaissent un essor dans la logistique grâce à leur capacité à transporter des charges lourdes qu'une personne normale ne pourrait pas porter. Ils permettent notamment d'éviter les blessures musculaires et les lésions permanentes au niveau du dos dues au stress quotidien.

Le but du projet est de développer un petit robot autonome différentiel suiveur avec ROS2 en C++. Nous utiliserons le principe de trilatération pour pouvoir localiser l'opérateur (i.e. la personne que le robot doit suivre) et les outils de simulation proposés par ROS2 pour faciliter le développement du robot. Durant le projet, nous porterons un regard important sur la maintenabilité du robot.

I. Modélisation et prototypage

1. Conception

Dans le cadre du projet, nous avons donc eu à réaliser un châssis. Pour la réalisation de ce châssis, nous avons pris exemple sur les châssis disponibles pour les kits Arduino. Ce type de châssis permettant un accès simplifié à l'électronique embarquée sur le robot. Nous avons donc choisi de construire notre robot sur trois étages. Nous aurions alors sur le premier étage le lidar, la batterie et le nécessaire pour l'alimentation. Le second étage portera le MPU6050 mais servira surtout pour le passage des fils pour la connections des différents éléments.



2. Electronique

Pour ce qui est de l'électronique, voici une liste des différents éléments que nous utilisons pour ce projet :

Robot :

Composant	Tension nécessaire (V)	Courant nécessaire (mA)
DWM1001-DEV (x4)	3.6 - 5.5	500 (2000)
RPLIDAR A1	5	230
MPU6050	2.375-3.46	<20
Moteur Jaune (x2)	6	350 (700)
Raspberry pi 4	5	3000

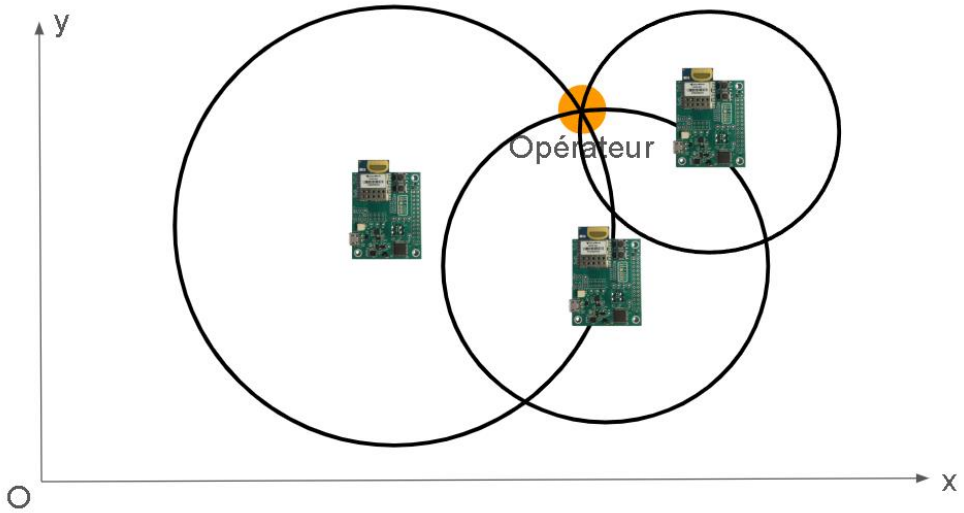
Opérateur :

Composant	Tension nécessaire (V)	Courant nécessaire (mA)
DWM1001-DEV	3.6 - 5.5	500
Raspberry pi pico W	3.3	50

II. Localisation de l'opérateur

Pour que le robot puisse suivre l'opérateur, il faut tout d'abord qu'il le localise. Pour se faire, le robot est équipé de 3 balises DWM1001-dev tandis que l'opérateur tient sur lui une quatrième balise. Les balises peuvent avoir deux modes de fonctionnement : le mode "anchor" et le mode "tag". Celles situées sur le robot sont en mode anchor et celle tenue par l'opérateur est en mode tag.

La localisation de l'opérateur repose sur le principe de trilatération :



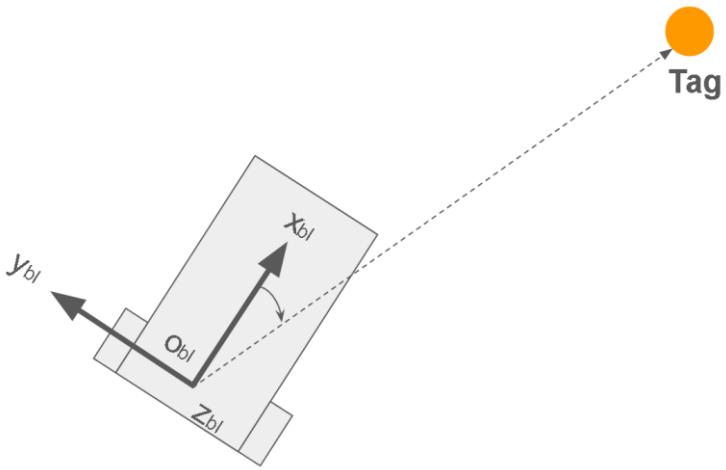
Chaque anchor connaît seulement sa distance par rapport au tag, d'où un cercle qui a pour centre un anchor et pour rayon la distance entre l'anchor et le tag. L'intersection de trois

cercles donne un seul point et c'est ce point qui représente la position de l'opérateur par rapport au robot.

Il existe une application téléphone qui se nomme "DRTLS" et qui permet de modifier le mode de fonctionnement des balises, de voir lesquelles sont actives et d'observer en direct la position du tag évoluer dans un repère (x,y) classique.

III. Algorithme de suivi sans obstacle

Supposons qu'il n'y ait pas d'obstacle entre le robot et l'opérateur. La situation peut simplement être illustrée par ce schéma :



Ici, "bl" signifie "base_link" et est la terminologie utilisée dans beaucoup de projets ROS. C'est le repère ayant pour origine le milieu de l'axe qui relie les deux roues et l'axe x qui pointe vers la direction de translation du robot.

A partir de ce schéma, nous avons un algorithme simple qui permet au robot de suivre la personne :

1. Obtenir les coordonnées de l'opérateur sur le robot
2. Orienter le robot vers l'opérateur
3. Avancer en ligne droite vers l'opérateur jusqu'à une distance acceptable

Nous allons expliquer chaque étape de l'algorithme.

1. <u>Obtenir les coordonnées de l'opérateur sur le robot</u>

Pour envoyer la position de la personne sur le robot, nous avons connecté en UART la balise de l'opérateur sur une Raspberry Pico W. Cette petite carte est responsable du décodage des bits que la balise envoie afin de retrouver une position (x,y) en mètres. Lorsque le décodage est terminé, la carte va envoyer les données de position via WIFI vers la Raspberry PI 4 qui se trouve sur notre robot. Ci-dessous un schéma illustratif du parcours des données :

Grâce à cela, nous pouvons désormais savoir où est l'opérateur par rapport au robot.

2. Orienter le robot vers l'opérateur

Il faut maintenant faire tourner le robot en direction de l'opérateur. Pour se faire, nous faisons tourner les roues de notre robot de telle sorte à avoir une rotation mais pas de translation. Pour y arriver, nous utilisons les formules suivantes :

$$V = \omega \cdot R = \frac{v_R + v_L}{2}$$

$$\omega = (v_R - v_L)/b$$

Où V est la vitesse de translation [m/s] sur l'axe x, b est la distance [m] entre les deux roues du robot, R est le rayon des roues [m], oméga est la vitesse angulaire [rad/s] du robot autour de l'axe z, V_R et V_L sont les vitesses linéaires [m/s] des roues droite et gauche.

On voit qu'il est possible d'avoir une rotation sans avoir de translation. En effet, il suffit de poser $V_R = -V_L$: le numérateur de la première équation sera 0 tandis que le numérateur de la deuxième équation ne le sera pas. Réciproquement, on peut avoir une translation sans rotation : cette fois il suffit de poser $V_R = V_L$ pour que le numérateur de la deuxième équation soit 0 tandis que le numérateur de la première équation ne sera pas nul.

Pour mesurer la rotation du robot et la comparer à ce qui est demandé, nous avons le capteur MPU-6050 qui mesure la vitesse angulaire. En intégrant les mesures par rapport au temps, nous pouvons également reconstituer l'angle du robot par rapport à un repère de référence. Le capteur est connecté en I2C à la Raspberry PI 4. Normalement, les estimations d'angle devraient diverger au fur et à mesure que le temps passe à cause des erreurs d'intégration qui s'accumulent, mais nous n'avons pas encore rencontré ce problème pendant les tests.

3. Avancer en ligne droite le robot vers l'opérateur jusqu'à une distance acceptable

Après avoir orienté le robot vers l'opérateur, il suffit d'avancer en ligne droite jusqu'à une distance limite pour ne pas heurter la personne. Pour se faire, nous devons mettre la même vitesse de rotation sur les moteurs du robot afin d'avoir une rotation nulle. Lorsque le robot est trop près du tag, il s'arrête et attend que l'opérateur s'éloigne pour le suivre.

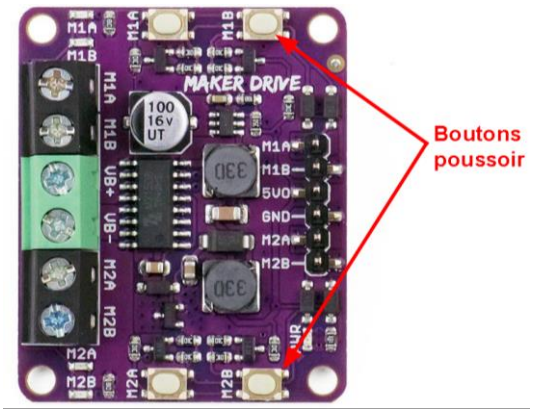
IV. Contrôle des moteurs

Les moteurs utilisés sont montés avec un encodeur qui permet d'estimer la vitesse angulaire et la vitesse angulaire du robot en comptant le nombre de révolutions. Lorsque l'encodeur tourne, l'état du GPIO auquel il est connecté passe de l'état bas à l'état haut. L'encodeur fait 1 tour toutes les 8 impulsions à l'état haut. Dans la documentation, il est nommé "PPR" pour "Pulses per révolutions" et dans notre cas, le PPR vaut 8.
Ensuite, l'encodeur est relié à un motoréducteur de rapport 120:1 (aussi appelé "gear ratio").
Pour reconstruire la vitesse linéaire du robot, on utilise cette formule :

$$V = 2\pi R \frac{N_{pulsions}}{P \cdot GearRatio}$$

V est la vitesse linéaire [m/s], R est le rayon de la roue [m], P est le PPR qui, $N_{pulsions}$ est le nombre d'états haut par seconde que l'on doit compter.
Nous pouvons alors estimer la vitesse de chaque roue grâce aux encodeurs.

Nous disposons d'un driver Cytron Maker Drive pour facilement tester les moteurs : sans aucun code nécessaire, ce driver nous permet de savoir si les moteurs sont bien connectés ou s'ils sont dysfonctionnels. Il suffit d'appuyer sur les boutons poussoirs pour les faire tourner dans un sens ou dans l'autre.



Afin de faire tourner les moteurs à la bonne vitesse, nous devons leur donner en entrée le bon rapport cyclique (i.e. le bon PWM).
Le rapport cyclique alpha est un nombre qui permet de régler le PWM codé sur 8 bits non signés c'est-à-dire de 0 à 255. La vitesse de rotation du moteur est proportionnelle au rapport cyclique. Pour que le contrôle du robot soit le plus simple possible, le mieux serait de demander une vitesse angulaire et une vitesse linéaire puis de déduire le rapport cyclique à appliquer aux moteurs. Nous pouvons alors déduire les formules suivantes grâce aux formules précédentes et à la l'hypothèse de proportionnalité :

$$\alpha_{left} = \frac{V - \omega \cdot b}{2 \cdot V_{max}}$$

$$\alpha_{right} = \frac{V + \omega \cdot b}{2 \cdot V_{max}}$$

Nous avons alors alpha pour chaque moteur en fonction de données que nous connaissons : V est la vitesse linéaire que l'on veut, oméga est la vitesse de rotation que l'on veut, V_{max} est fixé et b est la distance entre les roues du robot.

V. Flux de données avec ROS2 C++

Nous utilisons la version Jazzy ROS2 C++ pour gérer les données venant des capteurs. Nous pouvons facilement encapsuler nos capteurs et nos actionneurs grâce à des ROS nodes (noeuds) et des topics. Un topic est un "endroit" où des données de même type sont envoyées par une ou plusieurs nodes pour être lues par une autre node. Chaque topic a son propre nom.
Un node peut être :

- Un publisher : il publie/envoie des données vers un topic et ne devrait pas faire d'opérations complexes. Sa seule utilité est d'envoyer des données.
- Un subscriber : il reçoit les données d'un topic auquel il est abonné et peut exécuter des instructions complexes pour donner du sens aux données reçues.
- Les deux à la fois : publisher et subscriber

Prenons un exemple. Nous avons développé les nodes suivants :

- Robot_controller : le node principal qui est un publisher et un subscriber. Ce node est responsable du contrôle du robot
- Motor_node : un subscriber abonné au topic "cmd_vel" (pour "command velocity")

Robot_controller peut publier sur le topic cmd_vel la valeur de vitesse linéaire et de vitesse angulaire voulue. Motor_node, qui est abonné à cmd_vel, va recevoir ces valeurs et changer en conséquence la vitesse de rotation des moteurs.

Nous avons créé d'autres nodes, notamment le node "DWM1001_reader" pour récupérer les coordonnées de l'opérateur via WIFI et les donner à Robot_controller qui agira en conséquence.

Les nodes sont isolables : nous pouvons les tester individuellement pour voir s'il y a un problème. Pour avoir le "comportement normal" du robot, il suffit de lancer tous les nodes en même temps grâce à un fichier "launch.py" qui se trouve dans le GitHub.

VI. Simulation 3D avec ROS

ROS2 nous permet aussi de créer un modèle virtuel de notre et de simuler un environnement ce qui nous permet d'avoir une première idée de comment notre robot réel pour réagir en réalité.

1. Modèle URDF

Pour cela, tout d'abord nous avons besoin d'un modèle 3D de notre robot. Pour générer notre robot dans ROS2 il nous faut une description de celui-ci sous la forme d'un fichier URDF. Fusion 360 ou encore SolidWorks nous permettent de facilement générer un fichier URDF depuis nos modèles 3D. Pour éviter un modèle complexe qui demanderait à l'ordinateur d'importants calculs, nous avons choisi de recréer nous-même le robot dans un fichier URDF. Pour cette version nous l'avons créé avec des formes plus simples.

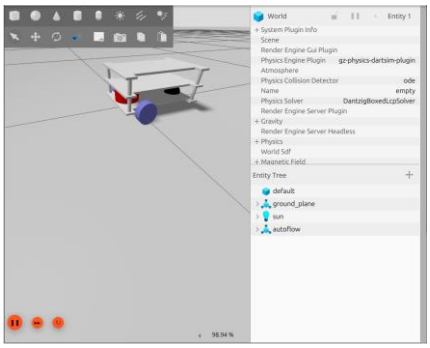
Un fichier URDF (Unified Robot Description Format) permet de décrire un robot de manière standardisée et compréhensible par ROS et les simulateurs comme Gazebo. Il permet de :

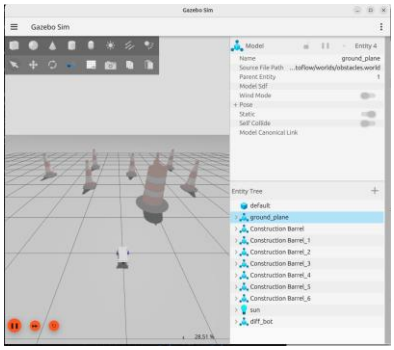
- Modéliser la structure physique du robot : Définir les parties du robot (liens) et leurs connexions (joints), incluant les propriétés physiques (masse, inertie, etc.).
- Représenter visuellement le robot : Décrire son apparence pour des outils comme RViz où Gazebo.
- Gérer les collisions : Spécifier des formes simplifiées pour détecter les collisions.
- Intégrer des capteurs et des actuateurs : Ajouter des LIDARs, caméras, ou moteurs pour une utilisation en simulation.
- Préparer l'utilisation dans ROS : Fournir les bases pour publier les transformations TF, simuler les capteurs, et connecter des plugins dans Gazebo.

2. Gazebo

Utiliser Gazebo permet de créer un environnement virtuel où il est possible de configurer de nombreux paramètres pour rendre la simulation aussi proche que possible de la réalité. Cela inclut la physique (gravité, frottements), les interactions avec le robot (collisions, capteurs), et les caractéristiques du monde (terrains, obstacles, objets dynamiques). Une telle simulation offre une première évaluation des comportements du robot, permettant d'identifier et d'anticiper différents problèmes potentiels avant une mise en œuvre dans le monde réel. Cela réduit les risques, accélère le développement et optimise les coûts liés aux tests physiques.

Gazebo utilise la description du robot que nous avons créé avant et si l'on veut la description du monde où se déroulera la simulation.

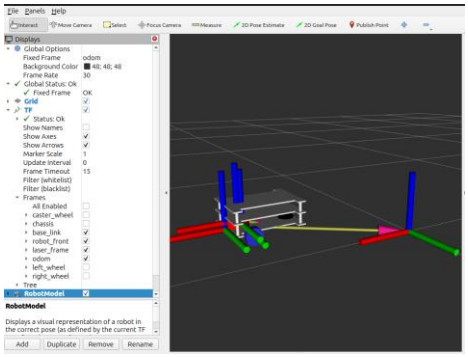


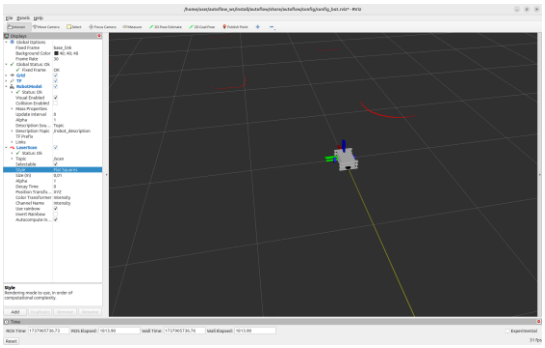


Avec Gazebo nous pouvons alors simuler nos capteurs dans cet environnement virtuel, les données simulées sont alors publiées on peut donc les récupérer pour les visualiser avec RViz.

3. RViz

RViz, nous permet donc de visualiser le robot en utilisant la description URDF ainsi que les données capteurs sur lesquelles il est abonné. Nous l'utilisons dans un premier temps pour visualiser les données renvoyées par le lidar que nous simulons dans Gazebo.





Ici les nuages de points relevés par le lidar simulé sont visibles sous la forme de points rouges apparaissant dans le monde RViz.

Dans ce cas les données récupérées sont des données simulées. Il nous sera possible de remplacer le lidar simulé dans Gazebo par notre Lidar réel. Pour cela il nous faudra juste publier les données du lidar dans le topic "/scan" à la place des données simulées et faire s'abonner RViz à ce topic.

VII. Navigation

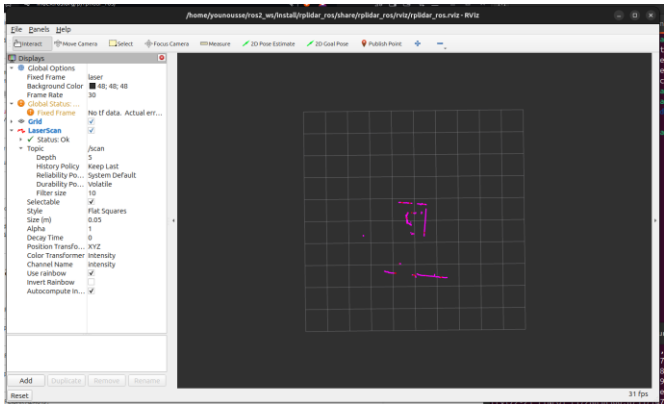
Notre robot va aussi avoir besoin d'un système de navigation pour se déplacer vers l'opérateur. Pour cela rien de plus simple, se diriger en ligne droite vers l'opérateur.
Cependant, le chemin entre l'opérateur et le robot ne sera pas toujours sans obstacle, le robot devra alors s'adapter à cette situation. Il aura alors besoin d'un moyen de détecter l'obstacle,

et si possible d'enregistrer les différents obstacles rencontrés en chemin et enfin d'un algorithme de navigation.

1. LIDAR

Commençons par la localisation des obstacles. Pour cela nous utilisons un lidar, plus précisément le Rplidar A1. Celui-nous permet de récupérer un nuage de point 2D, nous permettant alors de récupérer la distance qui nous sépare des obstacles tout autour de celui-ci.

L'implémentation avec ROS2 est assez simple, Rplidar nous donne accès à un package permettant d'utiliser ces différents lidars avec ROS. On peut alors visualiser les données relevées par celui-ci.

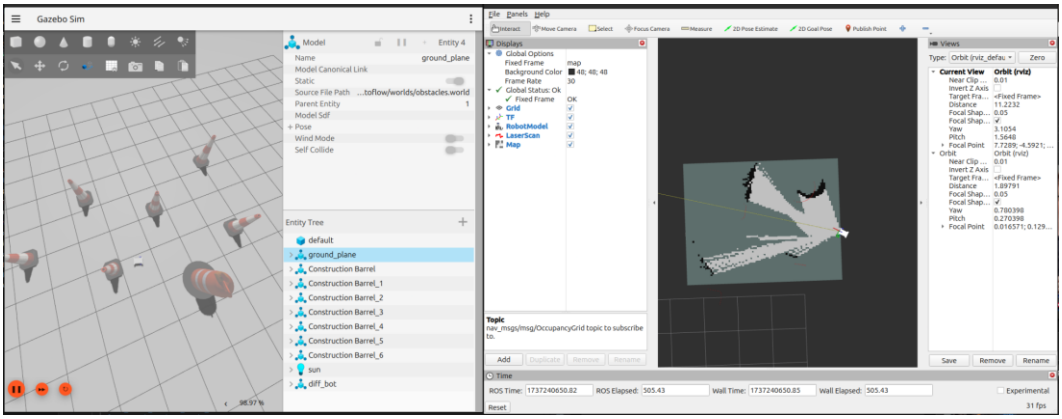


Ici, les points violets représentent les obstacles détectés.
Dans la réalité, le robot ayant comme objectif de suivre un opérateur, nous limiterons sa visibilité. Il devra voir seulement devant lui, avec un champ de vision de 120°. Réduire ainsi son champ de vision nous permet ainsi de réduire les données à traiter de deux tiers (120° au lieu de 360°).

2. Mapping

Maintenant que nous avons la détection d'obstacle, il nous faut les retenir pour prévoir la trajectoire.

Pour cela ROS2 nous laisse à disposition un outil "ros-slam toolbox" nous permettant de faire du slam (localisation et cartographie simultanée). Nous nous en servons donc pour créer une carte de l'environnement en temps réel en utilisant ici le lidar, mais aussi pour localiser le robot dans l'environnement. On peut aussi sauvegarder les cartes crées pour les réutiliser dans le futur.



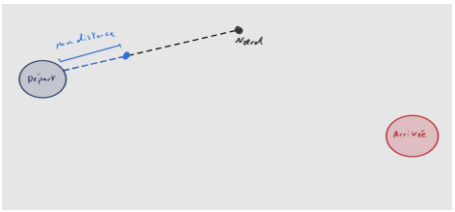
On peut maintenant utiliser les infos relevées dans cette carte avec un algorithme de navigation pour en éviter les obstacles.

3. Algorithme de navigation

Une fois arrivé face à l'obstacle, plusieurs options nous sont disponibles pour l'éviter. L'algorithme qui nous intéresse est le RRT*, une évolution du RRT (Rapidly-exploring Random Tree).

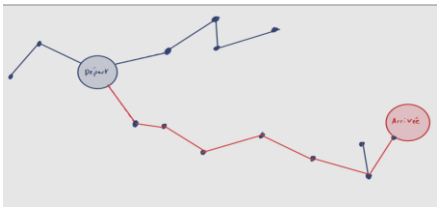
Pour expliquer son fonctionnement, il nous faut d'abord expliquer le fonctionnement du RRT.

Étape 1)

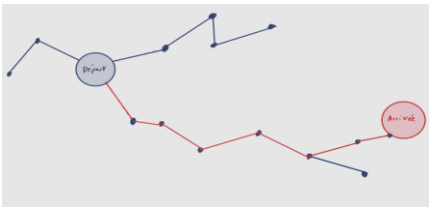


Étape 2)

Étape 3)



Étape 4)



Étape 1 :
Génération d'un nœud au hasard dans la carte, on cherche ensuite à lier ce nœud au point le plus proche (l'arrivée n'est pas prise en compte pour trouver le point le plus proche).
Si un obstacle se trouve entre les deux nœuds, alors on génère un nouveau nœud.
Nous limitons la distance entre deux nœuds, le nouveau nœud sera placé sur le nœud généré s'il est dans la distance fixée, sinon à la distance maximale entre le nœud généré et le nœud le plus proche.

Étape 2 :
On répète l'opération, dans ce cas le nœud le plus proche est toujours le point de départ.

Étape 3 :
Après plusieurs répétitions nous atteindrons l'arrivée, c'est ici que s'arrête l'algorithme RRT.

Le RRT* ajoute l'étape 4 :
Une fois à l'arrivée on continue de générer des nœuds au hasard, seulement cette fois-ci au lieu de simplement lier les nœuds créés au nœud le plus proche, on vérifie si en liant le nœud généré à d'autres nœuds dans un rayon que nous définissons il serait possible d'avoir un chemin plus optimal vers l'arrivée.

VIII. Problèmes rencontrés

1. Librairies pour contrôler les GPIOS

Pour contrôler les GPIOs de la Raspberry Pi, nous avons eu recours à des librairies spécialement faites pour cet usage. Cependant, il a été difficile d'en trouver une qui est fonctionnelle. La librairie WiringPi n'est plus maintenue et son site officiel n'est plus en service alors que beaucoup de réponses sur les forums la recommandaient. La librairie pigpio semblait être une bonne option car elle est aussi très présente dans les forums, mais le majeur problème est l'obligation d'utiliser "sudo" pour lancer le code du robot, ce qui est très gênant. Nous avons finalement trouvé la librairie "pigpiod_if2.h" qui fonctionne presque sans contrainte. La seule obligation est de lancer une seule fois "sudo pigpiod" dans le terminal avant de lancer le code, sinon le robot ne bougera pas. On peut le faire automatiquement en éditant le fichier ~/.bashrc.

2. Temps de compilation

Un des points faibles de ROS2 C++ est le temps de compilation nécessaire même pour une simple modification dans le code. La durée dépend du fichier compilé, mais peut rapidement monter à 2 minutes par compilation, ce qui fait perdre énormément de temps. On ne peut pas réduire la durée de compilation mais on peut essayer de ne pas l'allonger. Pour ce faire, une solution est de mettre nos variables de code dans un fichier de configuration qui n'est pas un .h ou un .cpp. Nous avons choisi le format .yaml qui est très lisible et facile à modifier.

3. Mesures et précision des capteurs de distance pour la localisation

Le majeur problème que l'on a rencontré est la précision des mesures de capteurs de distance utilisés pour localiser l'opérateur. Nous arrivons à recevoir des valeurs mais il arrive plus de valeurs aberrantes que de valeurs correctes. Monsieur Caminada nous a dit que c'est sans doute dû à la distance entre les anchors qui est trop proche. Les signaux arrivent au même moment et le tag n'a pas le temps de différencier lequel des signaux correspond à quel capteur. Une solution possible est d'agrandir le robot afin d'avoir plus d'espace entre les

capteurs ou alors de complètement enlever les capteurs du robot et de les mettre dans les coins d'une pièce.

4. Qualité de vie

Pour ajouter un peu de confort dans le développement du robot, nous avons mis en place quelques options de qualité de vie. Par exemple, nous avons l'habitude d'arrêter les programmes lancés dans le terminal avec CTRL-C. Nous avons implémenté un code qui permet d'arrêter les moteurs grâce à ce raccourci, sans quoi nous devons débrancher le moteur ou la Raspberry Pi, puis les rebrancher ensemble ce qui fait perdre beaucoup de temps.
ROS2 nous permet aussi d'écrire de personnaliser facilement nos messages d'erreurs, de warning et d'informations avec différentes couleurs, ce qui est très utile pour voir clairement ce qu'il se passe à travers un très grand nombre de messages défilant à grande vitesse.
De plus, les explications détaillées des formules utilisées se trouvent directement dans le code, écrits en commentaire. Cela permet d'éviter le phénomène de "formule magique" ou de "nombre magique" : les prochaines personnes auront une bien plus grande facilité à travailler sur le robot.

5. Simulation

Du côté simulation, la plupart des problèmes venaient du manque de documentation du fait que la distribution Jazzy est assez nouvelle. Les différentes sources que nous utilisions venaient de versions antérieures où seulement certaines fonctionnalités ont été modifiées. Un exemple est la version de Gazebo : ROS2 Jazzy utilise la version Harmonic qui diffère de la version utilisée sur ROS2 Humble (Gazebo Fortress). Les différents tutoriels et documentation n'étaient donc pas toujours adaptés ce qui a beaucoup ralenti le travail sur la simulation.

Conclusion :

Nous avons conçu un robot qui est théoriquement capable de suivre une personne. Cependant, les mesures du capteur de position sont trop instables pour le moment, ce qui ne permet pas un suivi parfait. Lorsque nous réussirons à avoir des mesures correctes, le robot devrait normalement fonctionner. Quant aux autres capteurs, ils sont opérationnels.
Par rapport à la simulation, le modèle actuel du robot est réutilisable et modifiable, il pourra être modifié pour l'ajout de capteurs comme le DWM1001-dev. On pourrait ainsi simuler la détection de l'opérateur.
Pour ce qui est de la navigation, il reste des problèmes à régler au niveau du mapping, ensuite nous pourrons implémenter l'algorithme de navigation.