UNIVERSITÉ LAVAL
Faculté des sciences et de génie
Département d'informatique et de génie logiciel

Pr. GIGUÈRE Philippe

Cours : GLO-7021

Introduction à la robotique mobile

RAPPORT FINAL

Ce projet a pour objectif d'explorer un environnement en drone à l'aide d'un capteur kinect.

Session Hiver

05-05-2022

Equipe : **N°05**

Présenté par : ROBLES ZAMBRANO Devi Jesus, ROUANE Mouaad et EL MASSAUDI Nassim

1 Introduction

Dans ce projet, nous nous sommes concentrés sur l'exploration et la cartographie autonomes d'environnements 3D à l'aide d'un drone. Nous avons commencé par expérimenter quelques concepts d'exploration de base. Notre objectif a été d'explorer et de cartographier de manière autonome un environnement 3D à l'aide d'un seul drone en utilisant soit des données RVB-D, soit des images optiques en combinaison avec la photogrammétrie. Nous avons considéré pour cela que les espaces fermés, c'est-à-dire que l'objectif du drone a été de cartographier la pièce fournie (par exemple une salle de classe) jusqu'à ce qu'un certain seuil d'incertitude soit atteint. Ce processus implique non seulement l'exécution du SLAM pendant que le drone se déplace dans l'environnement, mais aussi un problème d'exploration.

2 Objectifs du projet

Cette section décrit brièvement les objectifs pour chaque partie du projet, et les points sur lesquelles nous prévoyons de travailler si le temps le permet. Nous limitons la portée de chaque tâche, comme décrit dans les sections non-objectifs, afin de nous concentrer sur la résolution des problèmes essentiels et de ne pas perdre de temps avec des tâches secondaires.

a. Exploration d'un environnement 3D inconnu

Objectifs

- Mise en place des contrôles et capteurs pour un Quadricoptère (drone) dans un environnement simulé.
- Mise en place de la localisation et de la création de la carte de l'environnement 3D.
- Mise en place de la navigation dans l'environnement de manière autonome.

3 Implémentation du SLAM par drone

a. Initialisation du drone ainsi que de l'environnement.

Cette partie du projet a principalement demandé l'utilisation de Gazebo pour les simulations et ROS pour toutes les fonctions de navigation, SLAM, et autres fonctions nécessaires à l'exploration autonome. L'ensemble du code source, des scripts, des fichiers de lancement et des fichiers de configuration peuvent être trouvés dans le dossier en pièce jointe à ce rapport.

Nous nous sommes principalement appuyés sur les simulations Gazebo, tant pour l'environnement de test (intérieur) que pour le drone lui-même. Nous avons utilisé le package **hector_quadrator** pour émuler le drone, à savoir un drone quadrirotor auquel est attaché une Kinect, qui pointe vers l'avant. La caméra Kinect fournit les données de profondeur sous forme de nuage de points 3D. La figure 1 montre le modèle de drone et un exemple de nuage de profondeur tel que reçu par la Kinect, visualisé dans Rviz.

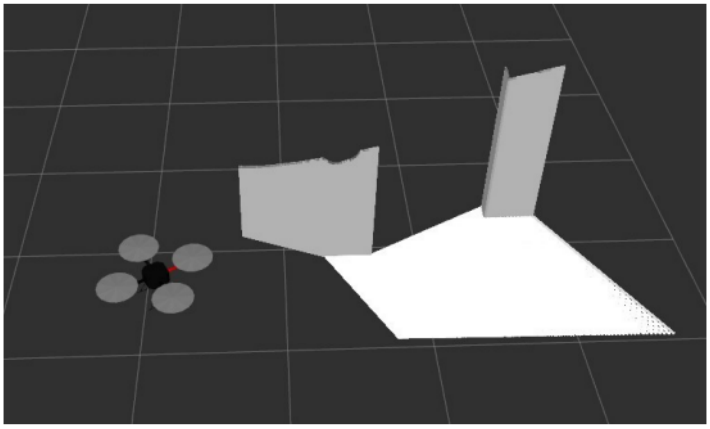

FIGURE 1

b. La cartographie à l'aide de Octomap

La première chose à noter est que, puisque nous travaillons dans un environnement simulé, nous avons une connaissance parfaite de la transformation réelle du drone par rapport au cadre réel. Nous avons donc utilisé cette information pour localiser le robot. Pour la cartographie, nous avons produit une grille d'occupation 3D en utilisant OctoMap. OctoMap utilise les données du nuage de profondeur pour produire une grille d'occupation de résolution arbitraire, en stockant efficacement le résultat final à l'aide d'une représentation octree (arbre avec noeud à 8 branches). Intuitivement, l'utilisation de résolutions inférieures pour la grille d'occupation d'OctoMap permet d'améliorer considérablement les performances. Dans notre configuration, nous avons fini par utiliser la résolution de 15 centimètres par cellule. Cette résolution permet de générer rapidement la grille d'occupation pendant que le drone vole. La **Figure 2** montre un exemple de grille d'occupation produite en faisant voler le drone et en scannant les environs avec Kinect.

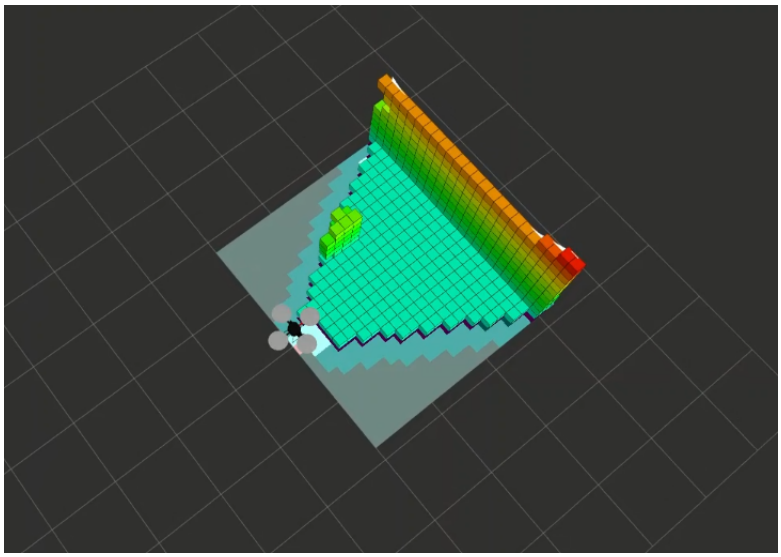

FIGURE 2 – Grille d'occupation 3D générer par OctoMap.

Nous avons décidé d'utiliser une carte 2D à des fins de navigation afin de simplifier les choses. Pour gagner du temps sur la génération de la carte 2D, nous projetons simplement la grille d'occupation 3D jusqu'au plan du sol. Nous spécifions un certain seuil de hauteur pour la grille d'occupation, en dessous duquel toutes les cellules occupées sont ignorées. La spécification d'un seuil de hauteur nous permet d'intégrer le fait que le drone est en vol stationnaire dans notre logique de navigation - puisque notre drone maintient une altitude de 1m au-dessus du sol, nous pouvons ignorer tous les obstacles qui apparaissent en dessous de cette altitude, même si notre grille d'occupation les contient. Cette approche est assez naturelle l'exploration intérieure. Dans les environnements intérieurs, les murs couvrent toute la hauteur de la pièce et sont toujours perpendiculaires au sol, ce qui les rend distincts sur la carte 2D projetée, tandis que le seuil de hauteur limite nous permet de survoler des objets tels que des tables et des chaises.

c. Execution et spécificité

Dans notre référentiel, le principal fichier de lancement ROS est **DroneSlam.launch**, et les paramètres des différents composants de la pile de navigation **move_base** se trouvent dans le répertoire **param/**. L'une des choses les plus importantes à noter ici est que notre drone est en vol stationnaire à une altitude de 1 mètre, ce qui a largement affecté la configuration de notre navigation. Par exemple, dans **costmap_common_params.yaml**, nous avons utilisé une carte voxel pour notre couche d'obstacles, avec les paramètres suivants :

```
obstacle_layer:
    # ...
    origin_z: 0.8
    z_resolution: 0.4
    z_voxels: 1
    # ...
    observation_sources: scan
    scan:
        data_type: LaserScan
        topic: scan
        marking: true
        clearing: true
        inf_is_valid: true
        min_obstacle_height: 0.8
        max_obstacle_height: 1.2
```

d. Explication du code de la méthode d'exploration

Aperçu de haut niveau.

Notre logique d'exploration est basée sur une boucle de contrôle qui fonctionne à la fréquence de 10 Hz.
Elle fonctionne à l'aide d'états :
- l'état **initial** étant le décollage, à partir duquel le drone passe en **FullRadialScan** pour analyser son environnement immédiat et l'enregistrer dans une grille d'occupation 2D/3D (dans notre cas 3D puis une projection 2D se fait).
- l'état **FindGoal** pendant lequel aucun mouvement ne se produit, mais le script analyse les grilles d'occupation générées jusqu'à présent et choisit un objectif pour le robot.
- l'état **TravelToGoal**, au cours duquel le drone suit les commandes de la pile de navigation (le cas échéant). Une fois que le robot atteint le but, il effectue à nouveau un balayage radial complet. À ce stade, si aucun autre objectif ne peut être trouvé, le drone termine l'exploration et entre dans l'état **Land**.

L'idée de base de notre script d'exploration est que nous essayons d'explorer complètement la projection 2D de la grille d'occupation 3D générée par **OctoMap**. Pour ce faire, le drone volera autour de lui en utilisant la caméra Kinect pour scanner non seulement le sol, mais aussi les murs (et parfois les plafonds), tout cela étant capté par octomap_server et incorporé dans la grille d'occupation 3D.
En conséquence, même si notre logique d'exploration utilise une simple projection 2D de la grille d'occupation 3D réelle, le résultat final est toujours un modèle 3D décent de l'environnement. Nous essayons également de prendre des décisions intelligentes quant à l'objectif à atteindre, en mettant certains objectifs sur liste noire si nécessaire pour éviter les boucles infinies. La philosophie principale est que notre script devrait être capable de gérer automatiquement toutes les erreurs et tous les problèmes d'exécution, et de ne s'arrêter que lorsqu'il est certain qu'aucune exploration utile supplémentaire ne peut être effectuée.

Maintien de l'altitude de vol stationnaire.

Notre script maintient une altitude de vol stationnaire constante de 1m. Nous exposons une méthode appelée request_velocity() à tous les composants du script d'exploration. Grâce à cette méthode, un composant peut spécifier une vitesse linéaire ou angulaire particulière qu'il doit exécuter à la fin de l'itération actuelle de la boucle de contrôle. Enfin, à la fin de l'itération, le gestionnaire de mouvement combine (de manière destructive) toutes les vélocités demandées en un seul message, puis ajoute à ce message une vélocité linéaire Z appropriée nécessaire pour maintenir l'altitude, et le poste enfin dans /cmd_vel. Étant donné qu'au décollage et à l'atterrissage, notre logique fournit des vitesses linéaires Z personnalisées, le gestionnaire de mouvement ne maintient pas l'altitude en vol stationnaire lorsque ces états sont actifs.

Balayage radial complet.

Cette opération est assez explicite, le drone tourne sur lui-même pour s'assurer que la caméra Kinect peut scanner tout son environnement immédiat. octomap_server utilise ces données pour produire une grille d'occupation 3D en temps réel qui est également projetée au niveau du sol pour produire une grille d'occupation en 2D. Il est très important de choisir la bonne vitesse angulaire - si la rotation est trop rapide, OctoMap ne pourra pas suivre, ce qui laissera de grands espaces vides. Si la rotation est trop rapide, OctoMap ne pourra pas suivre, ce qui laissera de grands espaces dans la grille d'occupation. Nous avons constaté qu'une vitesse angulaire de 0,4 radians par seconde est la plus rapide que l'on puisse atteindre tout en générant une grille d'occupation décente. En l'état actuel des choses, le balayage radial complet est quelque peu inefficace : il scanne souvent des zones dans lesquelles il n'y a aucune incertitude (par exemple, toutes les cellules dans un secteur de 180 degrés derrière le drone sont connues comme étant des zones de sécurité mais le

balayage radial les couvrira quand même).

Recherche d'un objectif approprié.
Notre logique de recherche d'un objectif approprié intègre plusieurs optimisations pour la vitesse d'exploration, la sécurité du drone et le calcul. Tout le code pertinent peut être vu dans drone_data.py (uniquement pour le script d'exploration - à ne pas confondre avec les planificateurs move_base). Notez que nous utilisons une grille d'occupation 2D à faible résolution dans notre script d'exploration, ce qui rend certains algorithmes de recherche sur grille plus efficaces.
Nous commençons le processus de recherche d'objectifs en identifiant les frontières d'exploration. Notez que nous utilisons en fait la carte des coûts globaux pour trouver les frontières, et non la grille d'occupation publiée dans le sujet /map. L'avantage d'utiliser la carte des coûts globaux est que nous pouvons utiliser la inflation_layer que nous avons spécifiée plus tôt - cette inflation_layer ajoute une zone de danger autour de tous les obstacles connus, ce qui permet de s'assurer que nous ne choisissons pas une frontière qui est directement à côté d'un mur (connu).
Nous définissons une frontière comme une cellule de la carte des coûts qui a au moins trois voisins libres et au moins trois voisins inconnus (les voisins diagonaux sont également comptés). Toutes les autres cellules doivent être soit libres, soit inconnues. Nous n'autorisons pas les cellules non nulles de la carte des coûts en tant que voisines, car cela pourrait nous faire choisir un objectif trop élevé et car cela pourrait nous amener à choisir un objectif trop proche d'un obstacle.

À ce stade, nous disposons d'une liste de frontières potentielles. Ensuite, nous filtrons toutes les frontières indésirables en utilisant plusieurs méthodes heuristiques. Tout d'abord, nous supprimons toutes les frontières qui se trouvent dans les régions de la liste noire (nous y reviendrons). Cela nous permet d'éviter les boucles infinies où nous pourrions choisir un objectif, la pile de navigation (navigation stack) le rejetterait, puis nous le choisirions à nouveau. Après cela, nous nous assurons qu'il y a un écart d'au moins 1 mètre entre toutes les frontières potentielles, ce qui est réalisé en éliminant les frontières qui ont été rejetées en éliminant les frontières qui sont trop proches les unes des autres.
Nous avons maintenant une liste de frontières potentielles éparses, qui ne se trouvent pas toutes dans les régions de la liste noire. Nous calculons le gain d'information attendu (EIG) pour chaque frontière potentielle. Puisque nous effectuons toujours un balayage radial complet une fois que nous arrivons à un objectif, pour calculer le GIE, nous considérons les valeurs d'occupation des cellules autour de l'objectif dans un rayon approximativement égal à la portée du traitement OctoMap, qui est effectivement d'environ 3 mètres.

Enfin, nous choisissons la frontière avec le meilleur IGE et essayons de trouver un endroit sûr pour observer la frontière. Pour ce faire, nous considérons 8 cellules dans un carré de 3 (cellules) de côté autour de la frontière, puis nous évaluons la sécurité de chacune des 8 cellules. Enfin, nous choisissons la cellule avec la meilleure note de sécurité et la définissons comme notre prochain objectif. Si la frontière que nous avons choisie ne peut être observée depuis aucun point sûr, nous nous déplaçons simplement vers la frontière avec le deuxième meilleur GIE et ainsi de suite. Notez que la logique de recherche de but peut épuiser toutes ses options et renvoyer 'None comme but. À ce stade, le script d'exploration suppose que nous ne pouvons plus effectuer d'exploration utile, le drone atterrit donc et le script d'exploration se termine.

Déplacement vers le but.
Les commandes de vélocité pour se rendre au but sont assez simples. Une fois que le drone a publié un objectif dans la pile de navigation et qu'il est passé à l'état TravelToGoal, notre script commence à écouter les commandes de vitesse publiées dans le sujet /planned_cmd_vel. Chaque commande de vélocité, c'est-à-dire les messages, sont prétraitées selon la logique décrite précédemment et publiée dans /cmd_vel. Comme mentionné précédemment, cette partie est assez simple. La principale caractéristique de notre logique lorsqu'il s'agit de se déplacer vers un objectif est l'utilisation de toutes les capacités de ROS actionlib. actionlib est un cadre qui permet à différents nœuds ROS de lancer et d'exécuter des actions complexes. Plus important encore, il est possible d'interroger l'état des actions émises par le biais d'actionlib - cela signifie que pendant que notre drone se déplace vers un objectif, nous pouvons interroger en permanence la pile de navigation pour obtenir l'état de notre objectif. Lorsque la pile de navigation ne peut pas planifier une trajectoire vers le but (par exemple, il n'y a pas de chemin sûr en raison d'obstacles), elle abandonne le but que nous avons spécifié. Notre logique d'exploration note quand un objectif a été abandonné, et ajoute l'objectif à une liste noire d'objectifs. En conséquence, toutes les frontières potentielles qui se trouvent dans un rayon de 0,6 m autour de l'objectif sur liste noire seront ignorées dans l'exploration future. Bien sûr, le fait de blacklister une région comme cela est un geste audacieux qui peut parfois nuire à l'efficacité (et à la complétude) de l'exploration. Nous supposons que la configuration que nous avons utilisée pour la pile de navigation n'abandonne que les objectifs qui

sont vraiment inaccessibles.

Finalement, nous avons malheureusement rencontré plusieurs problèmes avec l'implémentation du SLAM avec le drone. Le principal problème rencontré a été l'utilisation de **hector_quadrator**, le package de simulation développé par l'université de Darmstadt. En effet, une fois le drone importé, et la kinect configurée, nous n'avons pas obtenu sur Octomap les résultats attendus. Ce problème est dû au fait que **hector_quadrator** n'est plus mis à jour et ne fonctionne qu'avec la version ROS Kinetic (version datant de 2018). Nous avons donc entrepris d'adapter le package afin qu'il puisse fonctionner avec ROS Noetic (version plus récente) puisque lancer le drone sur l'ancienne version ROS Kinetic n'était pas possible en raison de la dépendance avec énormément d'autres packages dont on avait plus accès, en plus d'avoir la contrainte de devoir utilisé ubuntu 16.04 (Xenial). Pour l'adaptation du package **hector_quadrator**, nous nous sommes principalement basé sur l'implémentation réalisé dans la vidéo suivante [Con21] et le répertoire github suivant [Rob20]. Nous avons donc finalement réussi à implémenter une solution ASLAM lais celle-ci n'est pas vraiment concluante en raison d'un problème de detection de mur par notre kinect dont la cause nous dépasse. En effet, nous pouvions voir sur Octomap (cf. **Figure 4**) que plusieurs murs inexistants étaient considérés. Vous pouvez regarder cette illustration que nous avons réalisé **ici**. Après plusieurs jours à essayer de résoudre ce problème, nous avons malheureusement abouti à aucune solution.

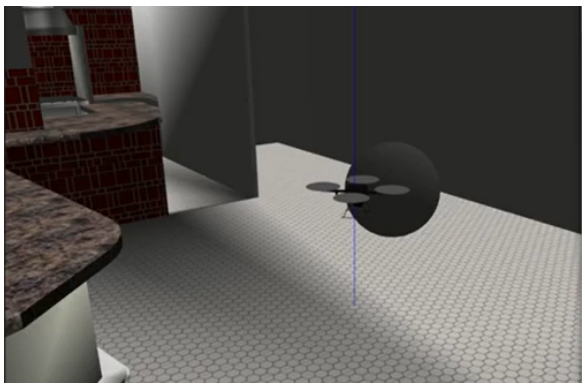

FIGURE 3 – Le hector quadrator en mode exploration autonome (Taking-off)

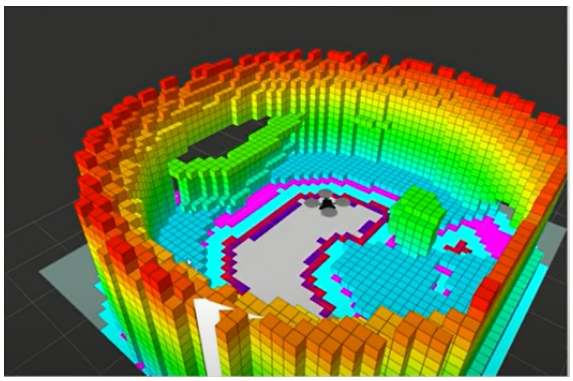

FIGURE 4 – Construction de la map de l'environnement par le drone (Rviz)

Nous ne nous sommes toutefois pas arrêtés là, nous avons décidé de changer notre approche en utilisant cette fois-ci un robot à conduite différentielle avec une kinect, et nous n'avons, forte heureusement, pas été confronté à ce problème.

4 SLAM par robot à conduite différentielle

Notre objectif ici est de réaliser un SLAM avec un véhicule à conduite différentielle à l'aide d'une kinect.

a. Configuration

Nous travaillons uniquement sur un environnement fermé pour réaliser notre SLAM. Nous avons ainsi sélectionné un environnement, issu de package *AWS RoboMaker Small House World ROS*, afin d'y réaliser notre exploration, localisation et navigation avec notre robot à conduite différentielle (cf. **Figure 5)**

FIGURE 5 – Environnement pour l'exploration de notre robot

Dans un premier temps nous avons créer un fichier **urdf** pour notre robot dont les spécificités sont : un châssis, une roue pivotante à l'avant et 2 roues arrières. Une fois notre robot bien défini, nous avons ajouté le plugin Gazebo **Differential Drive** pour notre robot. Ce plugin permet entre autre d'obtenir un contrôleur de base pour notre robot à conduite différentielle. Il reste alors à initialiser Rviz pour pouvoir voir en détail ce que notre robot percoit. Rviz est un outil de visualisation pour ROS. Rviz s'occupe de nous montrer ce que le robot perçoit pendant que Gazebo nous montre l'état réel de notre map.

La prochaine étape consiste à ajouter une capteur kinect et les plugins Gazebo associé à notre capteur. Contrairement à notre précédente implémentation avec le drone, nous n'avons pas rencontré de problème ici.

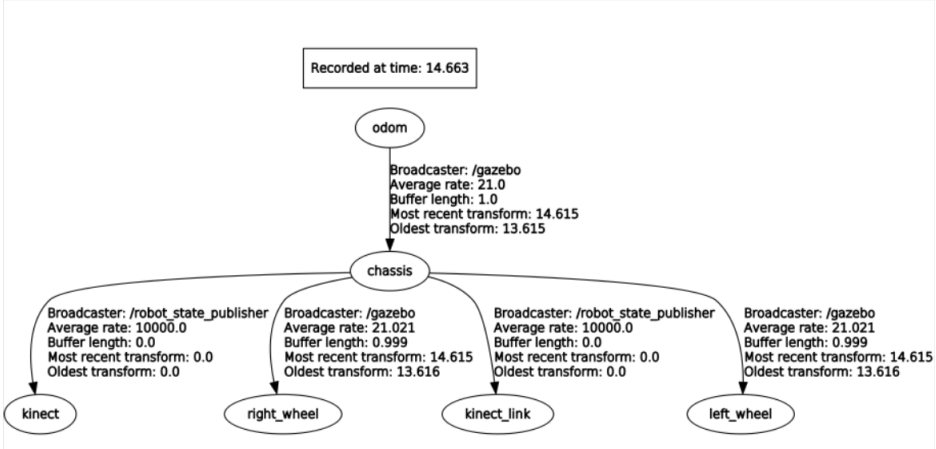

FIGURE 6

La **Figure 6**, ci-dessous, illustre les liens entre les différents tfs. Les tfs font parti d'un package qui permet en autre à l'utilisateur de garder la trace de plusieurs frames de coordonnées dans le temps.

b. Implémentation du Mapping

Concernant le Mapping, on crée un fichier de lancement **gmapping.launch** pour lancer **slam_gmapping**. Le **gmapping** est un package qui contient un wrapper ROS pour Gmapping d'OpenSlam. Le package gmapping fournit un SLAM basé sur les lasers, sous la forme d'un nœud ROS appelé **slam_gmapping**. Grâce à ce noeud, on peut créer une grille d'occupation 2D (ou 3D OctoMap) à partir de données laser et de pose collectées par notre robot.
Les cartes ainsi obtenus suite à la phase d'exploration sont illustrées dans la **Figure 7** et **Figure 8**.

Une fois le Mapping réalisé, la carte est sauvegardée dans **/catkin_ws/src/navros_pkg/maps/ directory**, pour pouvoir être utilisée aux étapes suivantes (localisation et navigation).

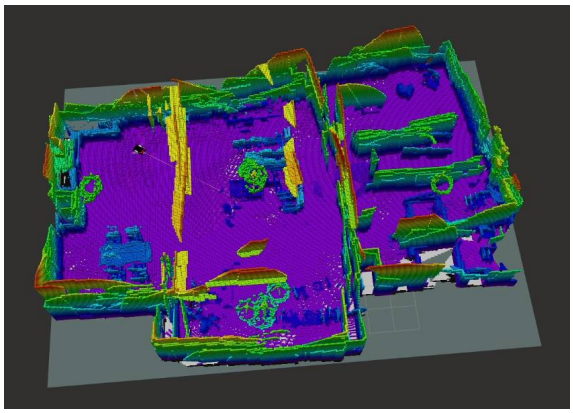

FIGURE 7 – Grille d'occupation 3D (Octomap)

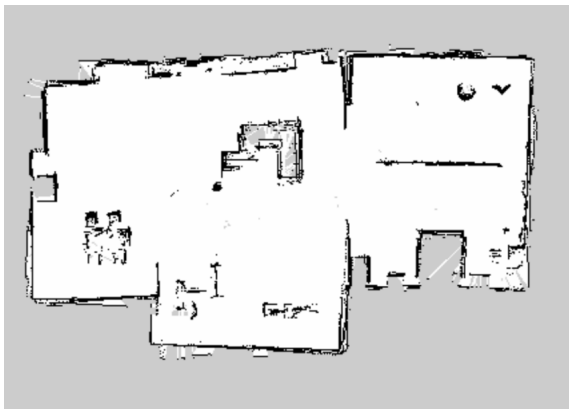

FIGURE 8 – Grille d'occupation 2D

Nous avons réalisé une vidéo en vitesse x10 pour illustrer l'évolution du mapping, ici.

c. Implémentation de la localisation et map navigation

Pour la localisation, nous avons tout d'abord créer un fichier **amcl.launch** pour lancer le noeud **amcl**. **amcl** est un système de localisation probabiliste utilisé pour la localisation 2D. Il met en œuvre l'approche de localisation adaptative Monte Carlo, qui utilise un filtre particulaire pour suivre la pose d'un robot par rapport à une carte connue. Un point à relever est que le noeud **slam_gmapping** doit être supprimé avant de lancer le noeud **amcl**.

On obtient d'excellent résultat sur la localisation et la navigation dans l'espace cartographié. Comme on peut le voir dans la **Figure 7**, on observe une zone d'incertitude en rouge relativement compacte, et une cartographie de l'environnement plutôt cohérente. Nous avons réalisé une vidéo de simulation pour illustrer les résultats obtenus, ici.

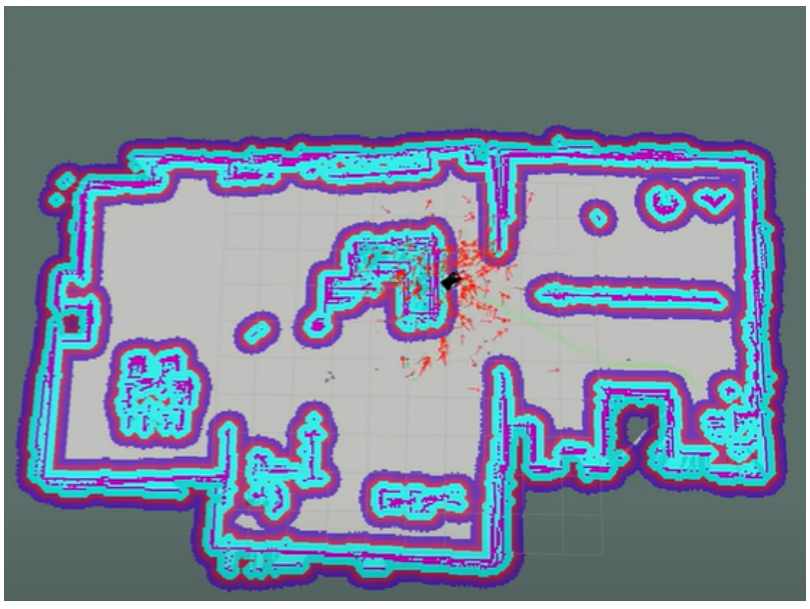

FIGURE 9 – Estimé de la position (en rouge) du robot à conduite différentielle

On observe que lorsque l'on atteint des espaces plus étroit, notre incertitude augmente drastiquement. On relève une plus grande entropie sur l'estimé de notre position (extrait de la vidéo, **ici**, à 0 :20s).

5 Conclusion et piste d'amélioration possible

L'approche SLAM a été réalisé avec l'intervention humaine dans la partie exploration et construction de la carte, et concernant la partie navigation et localisation, on obtient globalement de bons résultats sauf lorsque les passages sont plus étroits ; on observe alors qu'il est beaucoup plus difficile d'avoir un estimé de la position précis.

Aprés l'analyse de plusieurs simulations, nous avons constaté que les cartes obtenues à partir des nuages de points 3D ne sont pas détaillés que les images de projection 2D en termes de densité et ne fournissent pas toujours des caractéristiques suffisantes pour la mise en correspondance. Par exemple, dans les endroits où il y a peu d'obstacles, il est difficile d'aligner les nuages de points, ce qui peut entraîner une perte de la localisation du robot. En outre, la mise en correspondance des nuages de points nécessite généralement une puissance de traitement élevée, il est donc nécessaire d'optimiser les processus pour améliorer la vitesse. En raison de ces défis, la localisation de notre robot peut être améliorer par la fusion d'autres résultats de mesure tels que le système mondial de navigation par satellite et les données IMU.

Une piste d'amélioration possible de notre projet consisterait donc à reprendre le travail initial et de pouvoir implémenter un SLAM à l'aide d'une politique d'exploration optimale (une approche DQN serait intéressant) qui s'apparenterait à une autre approche ASLAM [LLA21]. La possibilité d'adopter une approche en essaim est également un défi intéressant à relever.

6 Livrables et configuration de la simulation

a. Robot à conduite différentielle

Dans notre répertoire **Github)**, vous retrouverez l'intégralité de l'implémentation ROS que nous avons réalisé avec les explications détaillés pour l'éxecution de celui-ci. La version 20.04 d'Ubuntu ainsi que ROS Noetic sont nécessaires à l'éxecution de notre simulation. Le code est également fourni au format zip sous le nom de **catkin_ws**.

b. Drone hector_quadrator

Pour l'implémentation du drone que l'on a faite, il est possible de l'executer en suivant le fichier **README.md** présent dans le dossier **slam**.

7 Références

[Con21] The Construct. [ROS Q&A] 210 - How to Install hector_quadrotor on ROS Noetic. https://www.youtube.com/watch?v=qJ9v1fITeVk, January 2021.

[LLA21] Iker Lluvia, Elena Lazkano, and Ander Ansuategi. Active mapping and robot exploration : A survey. *Sensors*, 21(7) :2445, 2021.

[Rob20] Open Robotics. Note on deprecations. https://github.com/osrf/gazebo/blob/gazebo11/Migration.md, November 2020.