

Projet recherche et développement
de Robotique 2024-2025:

Kumo



Remerciements.....	3
I. Introduction.....	4
II. Choix des nouveaux composants.....	4
A. Choix de la caméra.....	4
B. Choix du contrôleur.....	5
III. Déplacement.....	5
A. Simulation du robot.....	5
B. Cinématique inverse.....	6
C. Filtre de Kalman.....	7
D. Déplacement tripode.....	9
E. Control du robot.....	9
IV. Machine Vision.....	10
A. Base des programmes.....	10
B. Détection d'obstacles.....	10
C. Interpolation et Régression linéaire.....	12
D. Segmentation en plans.....	13
V. Schéma électrique du projet.....	14
VI. Coût du projet.....	14
VII. Bibliographie.....	14
VIII. Conclusion.....	15



Remerciements

Nous voulons remercier tous nos enseignant pendant ses 3 année d'étude pour leurs temps accordé et leurs compétences transmises et plus particulièrement :

- Guillaume DUCARD
- Pascal MASSON
- François JACOB

Nous voulons également remercier les “encadrants” du fablab qui nous ont aidé à réaliser au mieux nos projets :

- Frédéric JUAN
- Xavier LEBRETON
- Khadija AIT-HASSOU
- Lucas INACIO

Nous voulons enfin remercier les étudiants qui nous ont aidé lors de ce projet : Jimmy VU, Nino MULAC et Tristan LARGUIER.



I. Introduction

Nous sommes en Master2, ce rapport représente notre projet de fin d'étude **KUMO**. Ce projet est la continuité d'un projet développé en troisième année, sur lequel nous voulions effectuer des améliorations.

Le robot **KUMO** est un hexapode dont les pattes sont faites en plastique PETG et le corps en aluminium. Chacune des 6 pattes comportent 3 servomoteurs permettant une liberté des positions possibles.

L'objectif de notre projet est de réaliser le déplacement de ce robot araignée pouvant s'adapter à son environnement. Pour ce faire, nous utilisons ROS2 pour simuler le robot et relier l'algorithme de déplacement avec l'algorithme pour repérer les obstacles utilisant une caméra.

Nous allons dans un premier temps parler des choix des nouveaux composants, puis du déplacement et enfin comment il se repère dans son environnement.

II. Choix des nouveaux composants

A. Choix de la caméra

Afin de choisir le bon modèle de caméra, nous devions connaître les caractéristiques minimum nécessaire qu'il nous fallait pour notre projet. Tout d'abord, cela devait être une caméra de profondeur afin d'assurer la détection d'obstacles et éventuellement de pouvoir faire du SLAM.

Ensuite, nous avons déterminé qu'il nous fallait un angle de vue vertical d'au moins $40\pm5^\circ$ et horizontal d'au moins 70° . Pour trouver ces angles, nous nous sommes basés sur les positions et inclinaisons possibles de la caméra sur le robot. Nous sommes donc partis du principe que la caméra sera soit positionnée horizontalement à l'avant du robot soit légèrement inclinée (environ 5 à 10°) vers le bas.

Cependant, pour choisir une caméra, nous devons nous baser sur son champ de vision diagonal. Pour le calculer, il s'agit juste de faire un théorème de Pythagore :

$$\sqrt{40^2 + 70^2} = \sqrt{1600 + 4900} = \sqrt{6500} \approx 80.62$$

Nous pouvons donc conclure qu'il nous faut une caméra avec un champ de vision diagonale d'au moins 85° (en prenant une marge).

De plus, il serait souhaitable de prendre une caméra facile à implémenter au niveau informatique sur le robot. C'est-à-dire une caméra ayant déjà des bibliothèques python/paquet ROS2 de développer afin d'accélérer le projet.

Notre choix s'est donc porté sur la caméra intel realsense D435. Ce modèle de caméra a un champ de vision diagonal de plus de 90° et un paquet ROS2 fait par le fabricant.



B. Choix du contrôleur

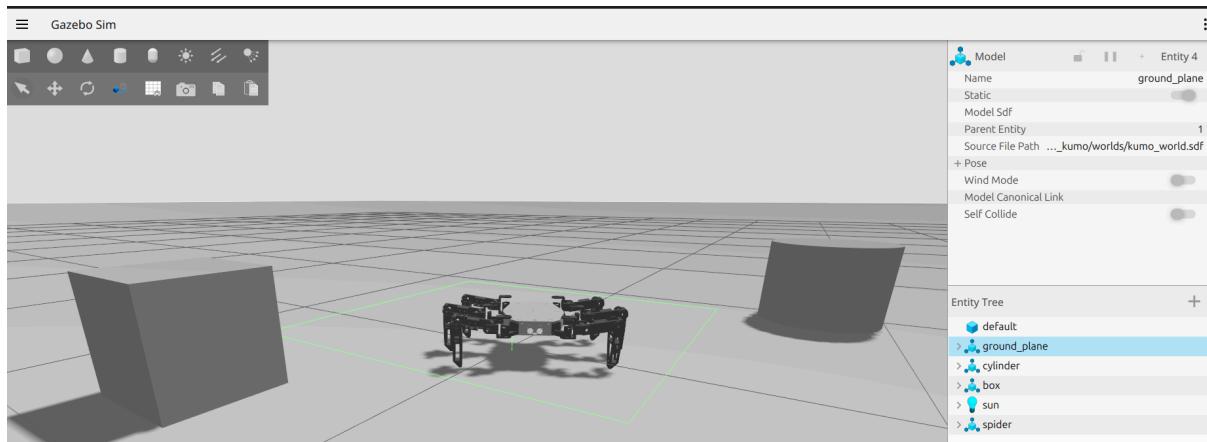
Ensuite, nous avons décidé de changer le microcontrôleur du robot. Notre premier choix s'était porté sur la carte Nvidia Jetson Nano fournie par l'école depuis notre 3ème année. Pour utiliser cette carte avec ROS2, nous avons dû mettre à jour le système de la carte (passer de Ubuntu 18.04 à 20.04). Ensuite, nous avons essayé d'installer les drivers de la caméra sur la carte. Le kernel de la Jetson n'étant pas compatible avec eux, nous avons tenté de le mettre à jour. Cependant, nous avons découvert que les kernels compatibles n'étaient pas disponibles pour les cartes Nvidia Jetson Nano.

Pour résoudre ce problème, nous avons d'abord pensé à acheter le modèle de carte compatible, la carte Nvidia Jetson Xavier. Mais, cette carte coûtait trop chère par rapport au budget accordé à notre projet. Nous avons donc choisi de gérer la caméra et ses données par ordinateur, quitte à devoir laisser un ordinateur connecté via un câble au robot lors du fonctionnement. Nous avons aussi eu des difficultés pour l'installation des drivers de la caméra sur nos ordinateurs, ce qui a encore plus retardé la partie programmation à l'aide de la caméra.

III. Déplacement

Dans cette partie, nous allons voir la simulation du robot, comment il se déplace, avec quels calculs et comment il peut s'adapter à son terrain.

A. Simulation du robot



Simulation sur Gazebo

Nous voulons simuler le déplacement de l'araignée grâce au logiciel de simulation Gazebo Harmonic sur ROS2 Jazzy. Pour ce faire, nous avons dû représenter la description du modèle du robot à l'aide d'un fichier au format URDF pour qu'il puisse être utilisé lors de la simulation. Ce fichier contient toutes les informations de chaque pièce du robot avec leur inertie, leur masse, les joints qui relient chaque pièce, les limites de vitesse, de position des joints de révolution. Il contient aussi le programme pour interagir avec les contrôleurs ROS2, permettant ainsi de contrôler Gazebo par ROS2 à l'aide de topics grâce à des messages de la position angulaire de chaque joint de liaison révolution du robot à chaque instant.



Le topic utilisé est : `/forward_position_controller/commands` et le message est un : `Float64MultiArray`

La description du robot est située dans le package `description_kumo`. C'est là où il y a l'URDF et les meshes du robot et le fichier pour contrôler le robot avec gazebo.

Nous avons eu beaucoup de problèmes pour réussir à faire apparaître le robot fonctionnel sur gazebo et aussi pour réussir à communiquer entre Gazebo et ROS. En effet, Gazebo harmonique et ROS2 Jazzy sont toutes les deux des nouvelles versions avec peu de documentations. Nous avons eu des problèmes de dépendances et nous avons eu du mal et passé beaucoup de temps à bien créer notre fichier launch pour lancer la simulation.

Dans le package `simulation_kumo`, nous avons le monde créé pour notre simulation ainsi que notre fichier pour lancer la simulation.

Pour lancer la simulation nous utilisons la commande :

`ros2 launch simulation_kumo kumo.launch.py`

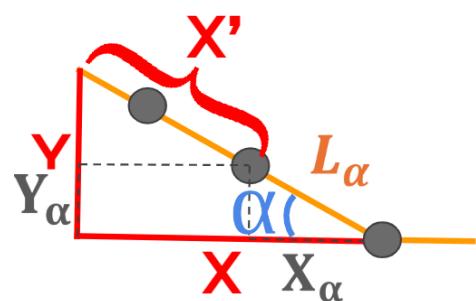
B. Cinématique inverse

Pour simuler le déplacement de l'araignée, nous avons utilisé la cinématique inverse. Pour chaque patte, nous connaissons la position où nous voulons que le bout de la patte soit en X,Y,Z. Nous voulons en retour les angles α , β et γ . Ses angles représentent les entrées demandées par chaque servomoteur.

$$f(X, Y, Z) = [\alpha, \beta, \gamma]$$

Nous commençons par déterminer α représentant l'angle de la liaison pivot sur l'axe Z.

Vue de haut sur le plan X-Y :

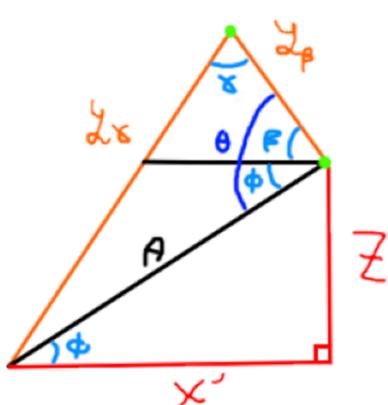


$$\begin{aligned}\alpha &= \tan\left(\frac{Y}{X}\right) \\ X' &= \sqrt{(X - X_\alpha)^2 + (Y - Y_\alpha)^2} \\ X_\alpha &= L_\alpha \cos(\alpha) \\ Y_\alpha &= L_\alpha \sin(\alpha)\end{aligned}$$

Nous calculons X' à l'aide du théorème de pythagore.

Nous déterminons ensuite les angles β et γ représentant les angles des 2 liaisons pivots sur l'axe Y.

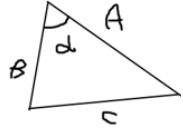
Vue de coté sur le plan X'-Z :



$$\begin{aligned}A &= \sqrt{X'^2 + Z^2} \\ \phi &= \arcsin\left(\frac{Z}{X'}\right) \\ \theta &= \arccos\left(\frac{L_\beta^2 + A^2 - L_\gamma^2}{2L_\beta A}\right) \\ \Rightarrow \beta &= \theta - \phi \\ \gamma &= \arccos\left(\frac{L_\beta^2 + L_\gamma^2 - A^2}{2L_\beta L_\gamma}\right)\end{aligned}$$



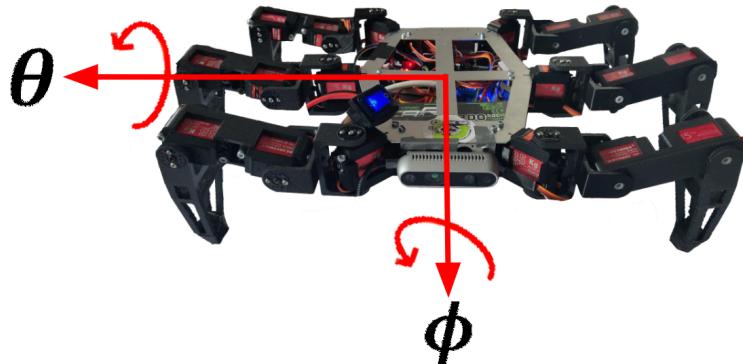
Nous déterminons θ et γ grâce à la loi du cosinus qui dit : $\cos(\alpha) = \frac{A^2 + B^2 - C^2}{2AB}$



Nous pouvons contrôler les angles des servomoteurs pour que chaque patte aille à l'endroit que nous voulons. Nous allons maintenant coordonner tout cela pour créer un déplacement.

C. Filtre de Kalman

Nous voulons estimer l'orientation du robot : le roulis et le tangage pour ensuite le compenser si besoin. Nous voulons que le corps du robot soit toujours droit par rapport à la surface de la terre.



Pour ce faire, nous utilisons une IMU (Inertial Measurement Unit) GY-BNO08X composé d'un accéléromètre et d'un gyromètre. Le lacet n'étant pas nécessaire, nous n'avons pas besoin d'utiliser le magnétomètre.

Nous utilisons un filtre de Kalman étendu pour permettre d'améliorer les estimations des angles. Il permet de compenser les dérives du gyromètre et l'instabilité de l'accéléromètre à court terme.

Le code de ce filtre de Kalman est dans le package controller_pub_kumo, nommé kalman_filter.py

Dans ce fichier, nous récupérons les informations de l'IMU à l'aide d'une carte Arduino et d'une liaison série puis nous utilisons le filtre de kalman pour améliorer les estimations des angles.

Le concept principal d'un filtre de Kalman est de combiner :

- Un modèle mathématique basé sur les données du gyromètre (p, q, r) représentant les vitesses angulaires autour des axes X, Y et Z. Auquel on soustrait le biais. Le vecteur d'état X décrit les angles estimés. Notre modèle est non linéaire car la matrice F dépend du point de fonctionnement et doit être recalculée à chaque itération.



$$\begin{pmatrix} \dot{\theta} \\ \dot{\phi} \end{pmatrix} = \begin{pmatrix} 1 & \tan(\hat{\theta}) \sin(\hat{\phi}) & \tan(\hat{\theta}) \cos(\hat{\phi}) \\ 0 & \cos(\hat{\phi}) & -\sin(\hat{\phi}) \end{pmatrix} \begin{pmatrix} p - \text{biais}_x \\ q - \text{biais}_y \\ r - \text{biais}_z \end{pmatrix}$$

\dot{X} F *Sortie du gyromètre*

- Les mesures de capteurs de l'accéléromètre. Les accélérations sur X, Y et Z permettent, via des calculs trigonométriques, d'estimer les angles de roulis et de tangage.

$$\hat{\theta} = \sin^{-1} \frac{a_x}{g} \text{ et } \hat{\phi} = \tan^{-1} \frac{a_y}{a_z}$$

Cette combinaison se fait à l'aide le la formule suivant :

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(y_k - h(\hat{x}_{k|k-1}))$$

Avec $\hat{x}_{k|k}$: l'estimation actualisée de l'état.

$\hat{x}_{k|k-1}$: la prédiction à partir du modèle.

K_k : le gain de matrice de Kalman calculé de manière récursive

y_k : le vecteur de mesure

$h(\hat{x}_{k|k-1})$: Vecteur de mesure prévu par le modèle

A chaque itération du filtre, les étapes suivantes sont réalisées :

- Mise à jour des paramètre :

- Le gain de Kalman en tenant compte de l'incertitude de l'accéléromètre.
- La matrice F correspondant au modèle au point de fonctionnement.
- Le vecteur d'état avec les nouvelles valeurs des capteurs (l'accéléromètre).
- La matrice d'erreur P du modèle.

- Extrapolation :

- La matrice d'erreur P en prenant en compte de l'erreur du modèle.
- Le vecteur d'état avec les anciennes valeurs du vecteur d'état et la nouvelle estimation.

Grâce à ces itérations, nous avons une estimation précise des angles de roulis et tangage en compensant les problèmes des deux capteurs.

Pour transmettre ces angles, nous utilisons le topic : `/kalman_filter`. Le message transmis est un `Float32MultiArray` comportant les deux angles estimés par le filtre.



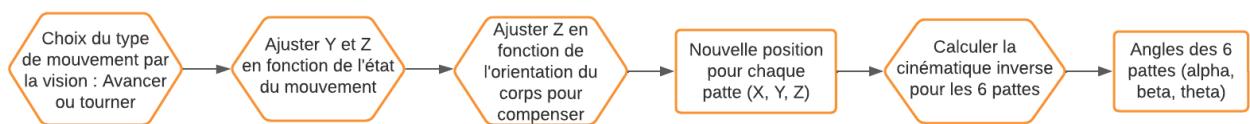
D. Déplacement tripode

La gestion du mouvement de l'araignée permet d'avancer, de reculer de tourner à gauche et de tourner à droite grâce à un déplacement tripode : trois points touchent le sol et sont coordonnées.

Il y a deux types de mouvement qui permettent en fonction de la coordination des 3 pattes de pouvoir soit avancer et reculer ou bien soit tourner à gauche et à droite. Pour savoir si dans un type de mouvement, on fait l'un ou l'autre, ça correspond au moment du mouvement où les pattes touchent le sol.

Nous avons un code python Move.py qui permet de faire la coordination des mouvements de chaque patte lors de la cinématique inverse. Ce code provient du package controller_pub_kumo.

Voici l'algorithme de ce programme :



C'est dans le programme robot_control_sim.py que l'on va relier les informations pour contrôler le mouvement du robot.

Pour ce faire, ce programme va souscrire à 2 topics :

- `/direction` : pour savoir quel type de déplacement faire
- `/kalman_filter` : pour connaître l'orientation du corps du robot

Puis il va utiliser les fonctions du programme Move.py pour récupérer toutes les 50ms la liste de la nouvelle position des servomoteurs comportant 18 angles.

Et enfin, cette liste va être envoyée à travers le topic suivant :

`/forward_position_controller/commands`.

E. Control du robot

Pour pouvoir contrôler la position du robot physique, nous utilisons un contrôleur de servomoteur SSC-32U contrôlé par une carte Arduino contrôlée par l'ordinateur à l'aide de ROS2.

Pour chaque servomoteur, nous avons déterminé un offset à la position angulaire pour être le plus précis possible.

Nous utilisons le script robot_control.py du package controller_pub_kumo pour permettre ce contrôle.

Dans ce code, pour récupérer les informations des 18 angles à envoyer à la carte arduino nous nous souscrivons au topic : `/forward_position_controller/commands`.

A l'aide du port série, nous envoyons à l'Arduino le message suivant :

"pin" P"position" T"temps"



Avec :

- *pin* : numéro de pin
- *position* : position voulue entre [500 ; 2500]
- *temps*: le temps pour aller à cette position en ms: ici 50 car la fréquence est de 50ms.

Les angles récupérés sont en radian entre $[-\frac{\pi}{2} ; \frac{\pi}{2}]$ et nous voulons qu'ils soient entre [500 ; 2500]. Nous faisons donc un remap. Puis à chaque itération, nous soustrayons l'offset aux positions remappées.

Grâce à cela, il est possible de contrôler la position de tous les servomoteurs.

Pour lancer le control du robot, il faut utiliser la ligne de commande suivante :

`ros2 launch controller_pub_kumo control_kumo.launch.py`

Pour lancer la compensation de la position angulaire du robot sans déplacement :

`ros2 launch controller_pub_kumo kalman_control_kumo.launch.py`

IV. Machine Vision

Cette partie est relative à la machine vision, c'est-à-dire ce que va pouvoir voir le robot grâce à la caméra. Nous allons maintenant évoquer l'utilisation de la caméra dans les déplacements du robot, le but étant que le robot évite des obstacles ou se repère dans l'environnement (création d'une carte).

A. Base des programmes

Nous avons tout d'abord créé un paquet ROS2. Ce paquet contiendra tous nos programmes. Chaque programme utilisera la même base. Cette base est constitué de lignes de code permettant dans l'ordre :

- de s'abonner au topic “`/camera/camera/depth/image_rect_raw`” (disponible grâce au paquet ros-realsense2 créé par le fabricant);
- de récupérer les données brutes de la caméra;
- de convertir ces données en un nuage de points puis de créer une image 2D ou de profondeur;
- d'afficher l'image 2D.

L'affichage de l'image 2D est fait uniquement pour pouvoir vérifier le bon fonctionnement du programme avec un œil humain. Le robot n'aura pas besoin de l'afficher s'il a réussi à la créer.

Il faut aussi noter que les programmes suivant ne marcheront que s'il y a une node lancée avec cette commande : `ros2 launch realsense2_camera rs_launch.py enable_rgbd:=true enable_sync:=true align_depth.enable:=true enable_color:=true enable_depth:=true`
Elle lancera la capture des données de profondeur de la caméra.



B. Détection d'obstacles

Ensuite, nous avons commencé à coder. Le premier programme que nous avons créé est un programme de détection d'obstacles. Il calcule d'abord la distance séparant la caméra de chaque point du nuage. Ensuite, il modifie l'image 2D afin qu'elle ne soit constituée uniquement des pixels/points présents dans un intervalle de distance précis que nous devions définir.

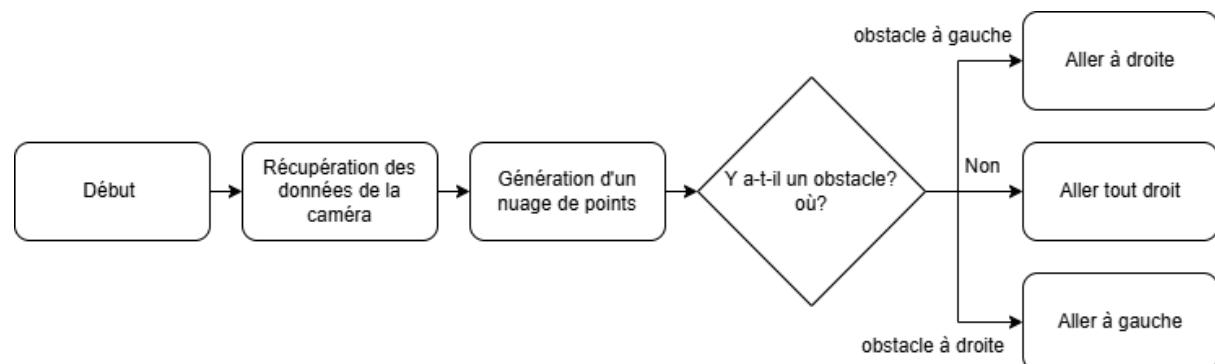
En pratique, nous avons remarqué que la caméra avait du mal à interpréter les données lorsqu'il y avait un obstacle entre 0 et 17 cm. De plus, nous avons décidé que le robot devrait éviter un obstacle à partir du moment où la distance qui les sépare est au maximum 40 cm. L'intervalle de distance choisi est donc [170;400] [mm].

Après avoir modifié l'image, le programme calcule le nombre de pixels à gauche et à droite de l'image. Si ce nombre dépasse un certain niveau, cela signifie qu'un obstacle est présent. Le programme publiera ensuite un message (un `IntArray`) sur un topic ROS2 en fonction d'où se situe l'obstacle :

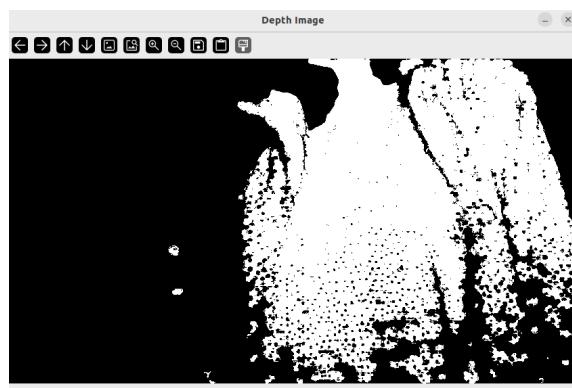
- 1 si l'obstacle est à gauche, indiquant qu'il faut tourner à droite;
- 2 si l'obstacle est à droite, indiquant qu'il faut tourner à gauche;
- 0 s'il n'y a aucun obstacle en vue, indiquant que le robot peut avancer en ligne droite.

Ce topic sera ensuite lu par le paquet ROS2 assurant les déplacements du robot, permettant ainsi d'éviter l'obstacle.

Voici un algorigramme de ce programme :



Et voici un exemple d'image 2D affiché lors de la détection d'un obstacle dans l'intervalle :



Après validation, nous avons constaté que ce programme fonctionnait correctement et que le robot évitait bien les obstacles.

Cependant, cette manière de procéder comporte des inconvénients :

- Il était très coûteux en termes de calculs : nous calculions la distance pour chaque point du nuage, ce qui ne pose pas de problème actuellement puisque nous utilisons un ordinateur. Mais, si dans l'avenir, nous décidions de passer sur une carte Nvidia Xavier ou un microcontrôleur similaire, cela pourrait ralentir le processus en combinaison avec tous les autres programmes qui tourneront en parallèle;
- Il ne nous donnait aucune interprétation de l'environnement : un nuage de point n'a aucune interprétation géométrique, ce qui ne permettra pas au robot de connaître son environnement.

La solution que nous avons trouvé pour ces problème est la suivante : au lieu de gérer des points, nous allons détecter et gérer des plans. Un plan est une entité géométriquement interprétable et compacte par rapport à un nuage de points, ce qui réduira les calculs. Il sera aussi plus simple de faire comprendre au robot qu'un plan vertical est un mur ou un obstacle. De plus, si nous souhaitons utiliser du SLAM, des plans nous permettront de construire des cartes plus précises.

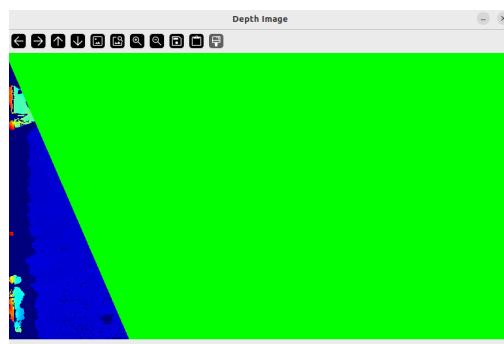
Pour parvenir à trouver des plans dans le nuage de points, nous avons utilisé deux méthodes différentes que nous avons implémentées dans deux nouveaux programmes.

C. Interpolation et Régression linéaire

La première méthode que nous avons utilisée est une combinaison de l'interpolation et de la régression linéaire. Ici, l'interpolation linéaire nous permet de générer une profondeur interpolée d'une distance entre des groupes de pixels. Nous utilisons ensuite ces données avec la régression linéaire afin de leur ajuster un plan, c'est-à-dire de trouver l'équation d'un plan qui correspondrait à un groupe de pixels.

Après plusieurs essais et modifications du programme qui applique cette combinaison, nous avons remarqué les problèmes suivant :

- Notre manière de procéder et les librairies/modules python que nous utilisons ne nous permettent de trouver un seul et unique plan;
- La vérification sur l'image de profondeur est difficile, le plan calculé est très souvent affiché sur l'ensemble de la fenêtre sans nuance de profondeur ou de position comme vous pouvez le voir ci-dessous:



Face à ces problèmes, nous avons décidé de passer à la seconde méthode envisagée : la segmentation en plans.

D. Segmentation en plans

E.

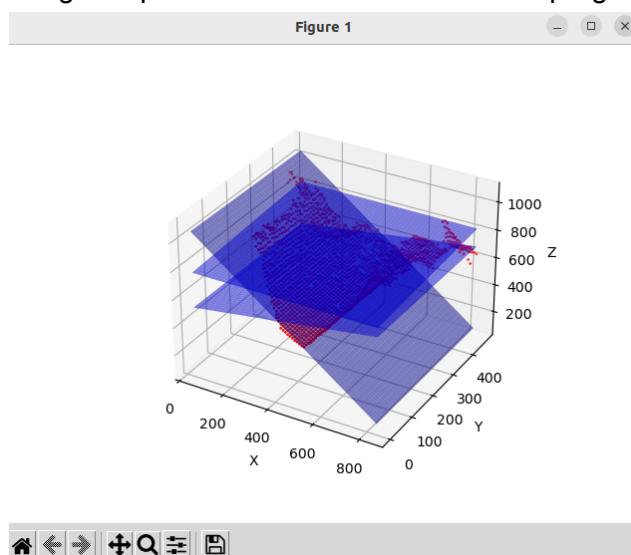
Cette seconde méthode consiste en théorie à nous permettre de trouver plusieurs plans dans notre image de profondeur. Cela est fait à l'aide de l'algorithme RANSAC. RANSAC est un algorithme qui a pour but de trouver un modèle (ici, un ou plusieurs plans) qui s'ajuste à nos données pertinentes (les inliers) tout en ignorant les données qui ne lui correspondent pas (les outliers).

Cet algorithme se déroule de la façon suivante pour trouver des plans à partir de notre nuage de points. D'abord, il sélectionne d'abord 3 points aléatoires. Puis, il calcule les paramètres du plan fait à partir de ces 3 points. Ensuite, il vérifie quels points du nuage sont proches de ce plan (les inliers) et répète ces étapes pour trouver un plan avec le maximum d'inliers associés.

Une fois ce plan trouvé, les inliers associés sont retirés des données du nuage et le processus se répète jusqu'à ce qu'il n'y ait plus assez de données pour ajuster un plan.

En fonctionnement, le programme permettait bien de trouver plusieurs plans, mais le problème des afficher sur l'image de profondeur était toujours présent. Pour le résoudre, nous avons décidé d'afficher les plans sur une autre fenêtre où nous pourrions projeter des données 3D.

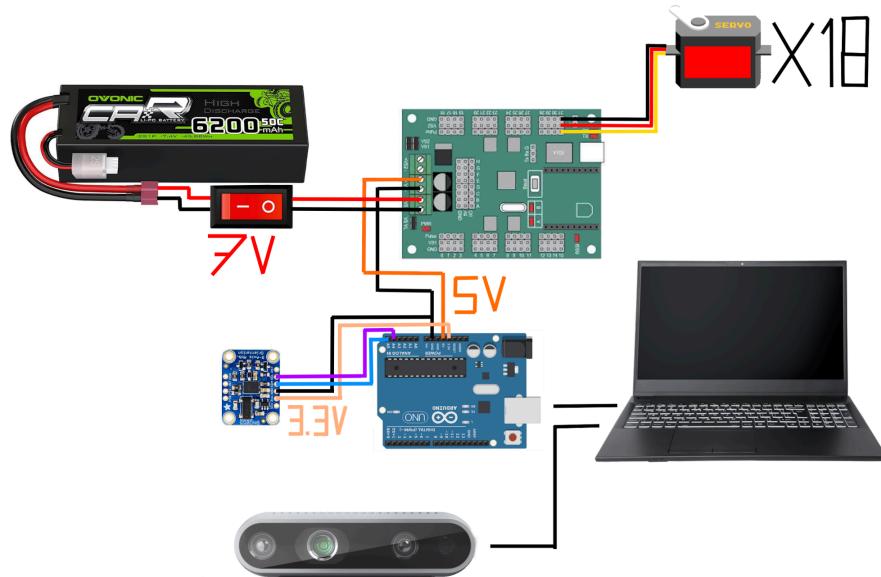
Voici un exemple d'affichage de plan lors du fonctionnement du programme :



Nous remarquons bien la présence de plusieurs plans et des outliers restants de nos données.



V. Schéma électrique du projet



VI. Coût du projet

Araignée	Nom	Prix(€)	Quantité	Total
Patte	Plastique PETG	30€/kg	650g	19,50 €
Corps	Aluminium 2mm	100€/m ²	0.065m ²	6.5 €
Microcontrôleur	Arduino UNO	10 €	1	10 €
Microcontrôleur	SSC-32	50 €	1	50 €
IMU	BNO055	20€	1	20€
Servomoteurs	TD8120MG	19 €	18	342€
Batterie	OVONIC 2s Lipo Batterie 6200mAh	23€	1	23€
Caméra	Intel Realsense D435	300€	1	300€
Autre	Visserie, entretoise, câbles...			5€
Total araignée				776€
	Matthias	23.75€/h	82h	1947,5€
	Koralie	23.75€/h	90h	2137,5€
Total				4861€

VII. Bibliographie

- Lien du projet sur github : https://github.com/llarak/Kumo_rob5
- Tutoriel Gazebo : <https://gazebosim.org/docs/harmonic/tutorials/>



- Lien du package ROS2 de la caméra:
<https://github.com/IntelRealSense/realsense-ros>

VIII. Conclusion

Ce projet de fin d'étude a été l'occasion de poursuivre et d'améliorer le développement du robot KUMO, un hexapode capable de se déplacer et de s'adapter à son environnement.

Grâce à ROS2, nous avons simulé le robot dans Gazebo et établi une communication entre les capteurs, les contrôleurs et les algorithmes de déplacement.

Pour réussir le déplacement, nous avons utilisé la cinématique inverse, le filtre de Kalman et le déplacement tripode pour permettre d'avoir un déplacement fluide et ainsi de s'adapter à l'inclinaison de son terrain.

L'intégration d'une caméra de profondeur et d'un programme de détection d'obstacles ont permis la perception du robot dans son environnement pour pouvoir naviguer sur tout type de terrain en autonomie.

Ce robot a des applications réelles comme pouvoir aider à la surveillance des zones comme des chantiers grâce à son déplacement pouvant s'adapter à son terrain.

Bien que le projet ait atteint ses principaux objectifs, certaines limites demeurent, comme la dépendance à un ordinateur externe pour la gestion des données de la caméra.

Il aurait aussi été intéressant de rajouter la caméra à la simulation Gazebo et de pouvoir détecter et éviter de chuter.

