Projet Semestre 7
Planification et exécution de trajectoire pour la manipulation d'objets encombrants avec une plateforme mobile holonome

Tariq Berrada, Julien Perichon, Nicolas Risse

4 mars 2020

Introduction

Le but de ce projet est de planifier la trajectoire d'un robot, connaissant son environnement, pour se déplacer de son point de départ à un point d'arrivée. On veut ensuite lui faire suivre cette trajectoire avec, comme contrainte supplémentaire, la possibilité d'ajouter un objet encombrant (notamment par sa taille/forme) sur le robot.

Ce robot est la plateforme mobile Kuka Youbot, qui a la particularité d'être holonome : il possède trois degrés de liberté : une translation selon l'axe x, une translation selon l'axe y, et une rotation autour de l'axe z (axe vertical). Le nombre de degrés de liberté d'un robot mobile est défini comme le nombre de mouvements indépendants que ce robot peut faire par rapport à un système de coordonnées déterminé. Le robot holonome est donc capable de se déplacer dans n'importe quelle direction quelle que soit son orientation.

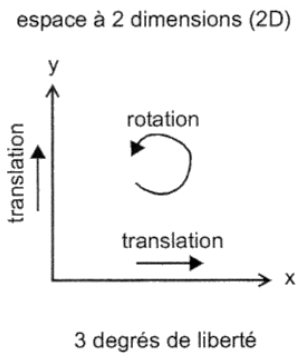


Figure 1 – Les trois degrés de liberté de la plateforme holonome

D'autre part, l'intention derrière le projet est de faire de la médiation scientifique, c'est-à-dire de rendre les tenants et aboutissants du projet accessibles au plus grand nombre. Pour cela, le meilleur moyen de procéder est de représenter nos résultats sous forme graphique, ou au moyen d'expériences concrètes.

Pour ce faire, nous allons d'abord construire une visualisation de la trajectoire planifiée du robot sous Python. Nous définissons d'abord un "monde" dans lequel sont décrits les limites du monde, les obstacles et le robot. Ensuite, nous utilisons l'algorithme RRT* pour planifier un chemin d'un point de départ à un point d'arrivée du monde. Nous visualisons ensuite le déplacement du polygone représentant le robot selon ce chemin. En parallèle, nous visualisons le graphe construit par l'algorithme de planification.

Dans une seconde partie, nous nous intéresserons à la simulation du robot sur ROS dans un environnement Gazebo. Nous utiliserons cette simulation en tant qu'intermédiaire entre la visualisation sous Python et le cas réel, afin de pouvoir comparer facilement les résultats obtenus aux résultats théoriques.

Dans une troisième partie, non traitée ici par manque de temps, nous aurions testé notre travail sur un cas réel, avec le vrai robot.

1 Définition d'un monde

Dans notre projet, le monde est une description géométrique de l'environnement du robot. Pour cette description, nous avons utilisé la bibliothèque Shapely de Python, qui permet notamment de vérifier aisément et efficacement l'intersection de deux polygones entre eux, qui est un problème reposant sur des algorithmes complexes (algorithmes de Vatti ou de Greiner-Hormann).

Le monde est un espace en deux dimensions, appelées x et y, où les distances sont exprimées en mètres. Notre monde est constitué des éléments suivants :

1. une Bounding Box : il s'agit d'un rectangle délimitant l'environnement atteignable par le robot. Comme l'exécution par le vrai robot d'une trajectoire planifiée devait se faire en intérieur, dans une salle fermée, l'environnement du robot était limité à la forme de cette salle.

   Nous avons donc choisi de définir les limites du monde sous la forme d'un quadruplet (x_{min},y_{min},x_{max},y_{max}) qui déterminent un rectangle (de sommets [x_{min},y_{min}],[x_{min},y_{max}],[x_{max},y_{max}],[x_{max},y_{min}]) qui représente les limites de la salle. Par convention, nous avons pris x_{min} et x_{max} de signe opposé, et de même pour y_{min} et y_{max}. Ainsi l'origine du monde (le point [0,0]) se trouve à l'intérieur de la Bounding Box.

2. le polygone représentant le robot : représenté comme un objet de type Polygon (importé de Shapely). Ce polygone est décrit par les positions de ses sommets par rapport à un point de référence par lequel passe son axe de rotation. En effet, nous ne pouvons pas exprimer les coordonnées absolues de ses sommets, puisque le robot est censé se déplacer dans le monde pour exécuter la trajectoire.

3. une liste de polygones représentant les obstacles : représenté comme une liste d'objets Polygon. Nous pouvons ici définir les polygones par les coordonnées absolues de leurs sommets, car nous admettons que tous les obstacles sont immobiles.

En réalité, nous stockons ces trois données dans un fichier de paramètres YAML, et nous importons ce fichier dans la classe World pour réécrire les données sous la forme décrite précédemment.

C'est également dans cette classe que nous vérifions la faisabilité d'un chemin entre deux points. En effet, ce problème est équivalent à celui de vérifier, à une position du robot donnée, si le polygone du robot est en intersection avec la Bounding Box ou avec un obstacle du monde. C'est ici que Shapely nous est utile puisqu'il gère ce calcul d'intersection entre polygones.

Pour faire cette vérification, nous interpolons le chemin avec une résolution de 5 cm, et nous utilisons la fonction `intersects` de Shapely pour nous assurer que le robot est bien à une pose libre de tout obstacle.

2 Planification de la trajectoire

Maintenant que nous avons une description précise du monde dans lequel le robot évolue, on veut que le robot soit capable de chercher automatiquement un chemin d'un point de départ à un point d'arrivée. Il existe de nombreux algorithmes de planification, dont l'algorithme RRT* que nous utiliserons dans ce projet. Notre espace de travail est l'ensemble des poses (x, y, θ) tel que : x et y sont bornés par la Bounding Box, et $\theta \in [0, 2\pi[$. On note cet ensemble $\mathcal{W}$, et on suppose qu'il existe une distance d sur $\mathcal{W}$. On note $\mathcal{C}$ l'espace des configurations, c'est à dire l'ensemble des éléments de $\mathcal{W}$ qui sont des poses possibles pour le robot. Nous supposerons dans la suite que les points de départ et d'arrivée sont des éléments de $\mathcal{C}$, et qu'il existe un chemin entre ces deux points que le robot peut emprunter.

2.1 Algorithme RRT

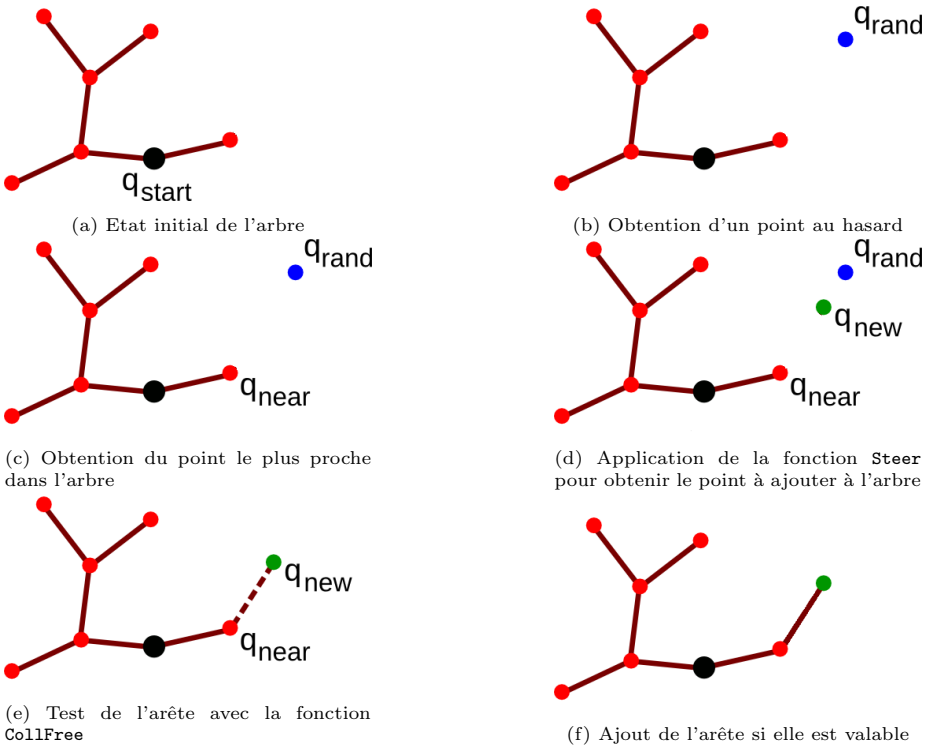
Nous allons d'abord présenter l'algorithme RRT, dont RRT* est une amélioration.

L'algorithme RRT nécessite 5 paramètres : $\mathcal{W}$, $\mathcal{C}$, x_{init} le point de départ, x_{goal} le point d'arrivée, et η la longueur maximale d'une arête de l'arbre. V est l'ensemble des noeuds de l'arbre construit, et E est l'ensemble des arêtes entre ces noeuds. Pour améliorer la vitesse de l'algorithme, on choisit aussi n le nombre d'arêtes que l'on va essayer d'ajouter à l'arbre avant de vérifier si le point d'arrivée est atteint. On présente ici l'algorithme d'extension de l'arbre.

$V \leftarrow \{x_{init}\}$; $E \leftarrow \emptyset$
for $i = 1, \ldots, n$ **do**
  $x_{rand} \leftarrow$`SampleFree`()
  $x_{nearest} \leftarrow$ `Nearest` (V, x_{rand})
  $x_{new} \leftarrow$ `Steer`$(x_{nearest}, x_{rand})$

```
    if CollFree(x_nearest, x_new) then
      V ← V ∪ {x_new}
      E ← E ∪ {(x_nearest, x_new)}
    end if
  end for
  return G = (V, E)
```

où `SampleFree` est une fonction permettant de tirer un point de $\mathcal{W}$ au hasard, `Nearest` (V, x) retourne le point de V le plus proche de x, et `CollFree` (x_1, x_2) vérifie que l'ensemble des poses définissant l'arête (x_1, x_2) soit contenu dans $\mathcal{C}$, c'est-à-dire qu'on vérifie l'absence d'obstacles entre ces deux poses. `Steer` (x_1, x_2) renvoie le point dans la direction de x_2 à une distance inférieure ou égale à η de x_1. Ainsi, si $d(x_1, x_2) < \eta$, la fonction retourne le point x_2. Sinon, elle retourne $x_1 + \eta \frac{x_2 - x_1}{d(x_1, x_2)}$.



(a) Etat initial de l'arbre

(b) Obtention d'un point au hasard

(c) Obtention du point le plus proche dans l'arbre

(d) Application de la fonction `Steer` pour obtenir le point à ajouter à l'arbre

(e) Test de l'arête avec la fonction `CollFree`

(f) Ajout de l'arête si elle est valable

FIGURE 2 – Illustration de l'ajout d'une arête dans l'espace (x, y)

Une fois cette extension de maximum n arêtes terminée (chacune de ces arêtes n'étant pas forcément valable), on vérifie s'il est possible de construire une arête de longueur inférieure ou égale à η reliant l'arbre actuel au point d'arrivée. On observe donc s'il existe $x \in V$ tel que $x \in \mathcal{B}_\eta(x_{goal})$, où $B_\eta(x_{goal})$ est la boule fermée de centre x_{goal} et de rayon η. Si c'est le cas, alors on peut

obtenir un chemin allant du point de départ au point d'arrivée. Sinon, on répète l'extension de l'arbre sur l'arbre actuel jusqu'à ce qu'un tel chemin soit possible.

Pour construire le chemin allant du point de départ au point d'arrivée, on utilise le fait que le graphe construit est un arbre. En effet, dans ce cas, chaque noeud possède un seul et unique parent, et la racine de cet arbre est x_{init}. Ainsi, on initialise le chemin au point d'arrivée x_{goal}. Ensuite, on remonte l'arbre incrémentalement jusqu'à arriver à x_{init}. On ajoute chacun des noeuds rencontrés au chemin, puis on l'inverse pour avoir le chemin dans le bon sens.

2.2 Amélioration de RRT avec RRT*

Dans certains cas, le chemin obtenu avec l'algorithme RRT n'est pas optimal au vu des noeuds de l'arbre. On obtient par exemple ce problème si les points tirés au hasard sont à une distance inférieure à η de l'arbre :

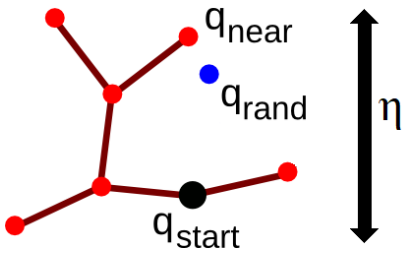


FIGURE 3 – Exemple de cas où RRT ne donnera pas un chemin optimal

Ici, on voit que l'arête qui va être ajoutée à l'arbre va relier les points q_{near} et q_{rand}. Pourtant, relier q_{rand} à q_{start} permettrait d'obtenir un meilleur chemin vers le point q_{rand}. Ici l'exemple est exagéré pour illustrer les propos, mais ce genre de problème peut avoir lieu lorsqu'on tire un point au hasard dans une zone déjà complètement couverte par l'arbre.

Pour résoudre ce problème, on utilise l'algorithme RRT*, qui est une modification de l'algorithme RRT.

$V \leftarrow \{x_{init}\}$; $E \leftarrow \emptyset$
for $i = 1, \ldots, n$ **do**
  $x_{rand} \leftarrow$`SampleFree`()
  $x_{nearest} \leftarrow$ `Nearest`(V, x_{rand})
  $x_{new} \leftarrow$ `Steer`$(x_{nearest}, x_{rand})$
  `if CollFree`$(x_{nearest}, x_{new})$ `then`
    $X_{near} \leftarrow$ `Near` $(G = (V, E), x_{rand}, \delta)$
    $x_{min} \leftarrow \arg\min_{x \in X_{near}}$ `C` $(x) + c(x, x_{new})$
    $V \leftarrow V \cup \{x_{new}\}$
    $E \leftarrow E \cup \{(x_{min}, x_{new})\}$
    `for all` $x \in X_{near}$ `do`
      `if C`$(x_{new}) + c(x_{new}, x) <$ `C`(x) `then`

$E \leftarrow E \setminus \{(\texttt{P}(x), x)\}$
$E \leftarrow E \cup \{(x_{new}, x)\}$
end if
end for
end if
end for
return $G = (V, E)$

où on a ajouté `Near` (G, x, d) l'ensemble des points de G à une distance inférieure ou égale à d de x, c'est-à-dire $V \cap \mathcal{B}_d(x)$, `C` (x) est le coût total du chemin entre x_{init} et x, $c(x_1, x_2)$ est le coût entre les points x_1 et x_2, et `P` (x) est le noeud parent de x dans l'arbre.

En prenant $\delta = \eta$, on obtient le chemin optimal entre x_{init} et x_{goal} au vu du graphe obtenu à la fin de l'algorithme. En effet, après l'ajout d'une nouvelle arête, les arêtes de l'arbre sont telles que les chemins entre x_{init} et les feuilles de l'arbre sont optimaux.

2.3 Preuve de la complétude probabilistique de RRT

Nous nous intéressons ici à prouver la complétude probabilistique de l'algorithme RRT (et donc de RRT*), c'est-à-dire la propriété de l'algorithme à avoir une probabilité d'échec tendant asymptotiquement vers 0 lorsque plus de travail est effectué [3] (lorsqu'on ajoute de nouveaux noeuds et de nouvelles arêtes à l'arbre). Bien sûr, on suppose pour cette preuve qu'un chemin existe entre le point de départ et le point d'arrivée.

En effet, on ne peut pas prouver que l'algorithme se termine, puisque l'ajout d'un noeud ne signifie pas forcément une augmentation de la couverture de $\mathcal{W}$ par l'arbre.

La preuve de la complétude de RRT, non triviale, est donnée dans le document [1] (l'algorithme RRT que l'on décrit ici y est nommé GEOM-RRT). Cette preuve nous informe par ailleurs (théorème 1, page 4) que cette probabilité d'échec tend à une vitesse exponentielle vers 0 en fonction du nombre de noeuds de l'arbre.

2.4 Implémentation de l'algorithme RRT*

On implémente ensuite l'algorithme RRT* dans notre librairie Python. On ajoute les fonctions `save` pour sauvegarder l'ensemble des arêtes ainsi que le chemin trouvé dans un fichier YAML, et `load` pour lire ces éléments depuis un fichier YAML.

Comme nous effectuons notre recherche dans l'espace des poses (x, y, θ), la distance utilisée doit prendre en compte la différence d'unités entre x et y en mètres, et θ en radians. De plus, il faut prendre en compte que $\theta_1 = \epsilon$ et $\theta_2 = 2\pi - \epsilon$ sont très proches pour $\epsilon \ll 2\pi$.

2.4.1 Calcul de $d\theta$

On veut calculer $d\theta$ la distance (en radians) entre θ_1 et $\theta_2 \in [0, 2\pi[$. On sait qu'on a $0 \leq d\theta \leq \pi$, car deux points opposés sur le cercle trigonométrique sont distants de π.

Ainsi, si on a $|\theta_1 - \theta_2| \leq \pi$, alors on a directement :

$$d\theta = |\theta_1 - \theta_2|$$

Cependant, si cette condition n'est pas remplie, on s'intéresse aux valeurs de θ_1 et θ_2. Dans ce cas, on a forcément $\theta_{min} = min(\theta_1, \theta_2) \in [0, \pi[$ et $\theta_{max} = max(\theta_1, \theta_2) \in [\pi, 2\pi[$. Ainsi, on a :

$$d\theta = (2\pi + \theta_{min}) - \theta_{max} < \pi$$

On remarque que dans ce cas, on peut réécrire cette formule comme :

$$d\theta = 2\pi - |\theta_1 - \theta_2|$$

Ainsi, on en déduit la formule générale pour la distance en radians entre deux angles :

$$d\theta(\theta_1, \theta_2) = min(|\theta_1 - \theta_2|, 2\pi - |\theta_1 - \theta_2|)$$

2.4.2 Distance dans l'espace des poses

Maintenant que nous savons calculer la distance entre deux angles, on peut s'intéresser à la distance entre deux poses (x_1, y_1, θ_1) et (x_2, y_2, θ_2). Le centre de ce problème est la conversion d'une distance angulaire $d\theta$ en radians vers une distance en mètres.

Pour effectuer cette conversion, on définit r_{robot} le rayon du robot comme la distance entre l'origine $(0, 0)$ et le sommet du polygone du robot le plus distant de l'origine (on rappelle que le polygone du robot est défini par rapport à une position de référence $(0, 0)$ par laquelle passe son axe de rotation). Ainsi, si on calcule $r_{robot} d\theta$, on obtient la distance maximale que vont parcourir les points du polygone du robot lors d'une rotation d'angle $d\theta$.

La distance d dans l'espace des poses (x, y, θ) peut donc être définie comme :

$$d((x_1, y_1, \theta_1), (x_2, y_2, \theta_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (r_{robot} d\theta(\theta_1, \theta_2))^2}$$

2.5 Autres techniques de planification

Il existe beaucoup d'autres techniques de planification, dont notamment des algorithmes dits "paresseux"[2]. L'idée derrière ces algorithmes est de d'abord construire l'arbre sans vérifier les collisions, puis de vérifier celles-ci pour un chemin candidat lorsqu'on en obtient un. En effet, les algorithmes permettant de savoir si deux polygones s'intersectent sont très coûteux par rapport aux autres opérations de l'algorithme. L'utilisation d'un algorithme "paresseux" permet donc d'améliorer grandement la rapidité de la planification, puisque les vérifications de collision se font seulement sur un chemin et pas sur l'arbre entier.

3 Visualisation d'un monde

Dans cette partie, nous implémentons la visualisation du monde en deux dimensions. Nous utilisons pour cela la bibliothèque graphique libre OpenCV. La classe VisuWorld prévue à cet effet doit être capable de créer un fichier image du monde avec tous les obstacles et également le robot à sa position exacte.

Nous avons donc besoin d'importer non seulement le monde (sous forme d'objet World comme nous l'avons vu plus haut), mais également la position du robot (qui n'est pas, elle, une position en deux dimensions, puisque la position angulaire du robot doit également être représentée en plus de sa position (x, y)). Nous pouvons aussi importer le chemin entier fourni par la classe RrtStar, qui a l'avantage de nous fournir une liste de positions, ce qui nous permet de créer ici plusieurs images qui, mises bout à bout, forment une vidéo du déplacement du robot dans son environnement.

Les positions successives du robot données par le chemin calculé par RRT* sont trop éloignées les unes des autres pour obtenir une vidéo convenable. Nous avons donc recréé ici, pour chaque arête du chemin, des positions intermédiaires du robot, en interpolant chaque arête (grâce à la fonction d'interpolation que nous avons créée dans la classe World pour vérifier la faisabilité d'un chemin) pour obtenir des positions plus rapprochées.

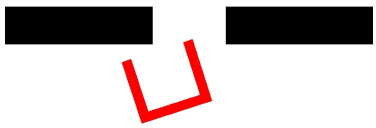


FIGURE 4 – Visualisation d'un monde : le robot (en rouge) et le obstacles (en noir)

Pour des raisons de tests, nous avons également implémenté une fonction qui dessine une suite d'images rapprochées de la même manière que précédemment, à la différence que la trace du robot est conservée sur sa trajectoire. On obtient cela en calculant une moyenne pondérée de l'image actuelle et de l'image précédente.

FIGURE 5 – Trace laissée par le robot lors de son déplacement

4 Visualisation 3D de l'arbre et des obstacles

Le but de cette partie est de visualiser l'arbre de RRT* ainsi que les obstacles dans l'espace d'état (x, y, θ). Pour cela on va utiliser la bibliothèque `Glumpy` pour créer un environnement en 3D auquel on va ajouter l'arbre de RRT* sous forme d'un graphe, ainsi que les poses du robot où il entre en collision avec son environnement, sous forme de nuages de points.

Pour le tracé de l'arbre, on va prendre la liste des arêtes sauvegardées dans le fichier YAML correspondant à cet arbre et en extraire les sommets dans une nouvelle liste.

Ces arêtes seront tracées comme des lignes alors que les sommets seront représentés par des sphères.

Pour le tracé d'un obstacle, on va extraire la Bounding Box de cet obstacle et lui ajouter dans chaque direction une longueur Δl correspondante à la diagonale de la Bounding Box du robot.

On obtient une surface $S_{x,y}$ où l'on est sûr que si le robot rentre en collision avec l'obstacle, alors il se trouve nécessairement dans cette surface.

Après, on va générer de manière aléatoire des poses dans cette surface et chacune de ces poses sera dessinée si le robot simulant cette pose est en collision avec l'obstacle, on s'arrête une fois ayant obtenu un nombre n de poses à dessiner

($n = 10000$ par défaut)

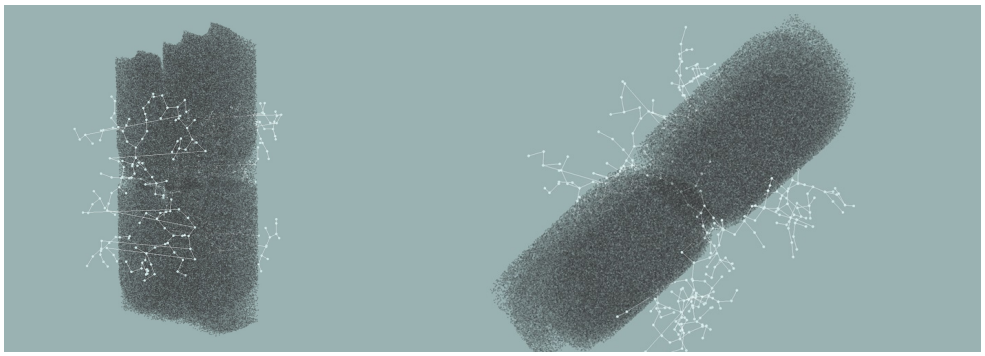


FIGURE 6 – Visualisation de l'arbre et des zones inaccessibles pour le robot dans le cas du monde présenté dans world1

5 Transformation d'un monde en OccupancyGridMap

Pour que l'algorithme de RRT* soit exploitable par le robot ou dans les simulations Gazebo, il est nécessaire d'assurer une conversion entre la classe World utilisée dans les scripts Python et les maps tel que reconnues par le robot, c'est-à-dire en une OccupancyGridMap.

Afin de transformer une instance de la classe World en une OccupancyGridMap, on procède de la manière suivante :

En ayant nos paramètres de résolution et la taille de notre monde, on crée une OccupancyGridMap ayant les dimensions correspondantes :

$$n_x = \frac{longueur_x}{resolution} \tag{1}$$

$$n_y = \frac{longueur_y}{resolution} \tag{2}$$

Puis, pour chaque case de notre OccupancyGridMap, on génère un point ayant comme position les coordonnées de cette case, la probabilité d'occupation est alors égale à 1 ou 0 selon que ce point rentre en collision avec un des obstacles du monde ou pas.

6 Transformation d'une OccupancyGridMap en monde

Pour permettre au robot de trouver une trajectoire à suivre dans un monde qu'il a généré par son télémètre laser (Hokuyo), on construit une classe qui fait la transformation d'une OccupancyGridMap en une instance de la classe World.

Cette classe va utiliser des méthodes de traitement d'images pour arriver à cette fin.

En effet, en ayant notre OccupancyGridMap, on va générer une image (en noir et blanc) ayant les mêmes dimensions que cette dernière. Après chaque pixel de l'image aura une valeur en gris proportionelle à la probabilité d'occupation de la case correspondante dans l'OccupancyGridMap.

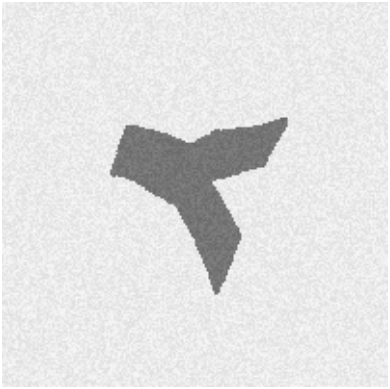


FIGURE 7 – Exemple d'une image obtenu d'après une OccupancyGridMap.

Cela étant fait, le problème est réduit à un problème de détection de polygones sur une image. N'ayant pas trouvé de manière facile de réaliser cette transformation, on va proposer un algorithme pour le faire.

6.1 Cas d'une image avec un seul obstacle

On essaye de se baser sur les fonctions fournies par OpenCV, notamment en utilisant une fonction de détection de coins sur une image et une fonction de détection de contour qui, pour une image avec une certaine forme à l'intérieur, renvoie une image avec seulement le contour de cette forme.

(a) Image du contour (b) Coins détectés sur l'image

Les coins étant donnés de manière désordonnée, il faut savoir comment les relier proprement pour pouvoir décrire le contour de manière non redondante. Pour cela, on passe par plusieurs étapes. Premièrement on va détecter tous les couples de points décrivant une arête de la manière suivante :

Pour chaque couple de deux points, on obtient l'équation du segment qu'ils décrivent, puis on va comparer ce segment dans l'image du contour pour voir quel est la proportion de points qui sont assez proches d'un pixel noir (soit un contour). On obtient une sorte de métrique nous permettant de savoir quelle est la probabilité que ce segment corresponde à une arête.

Le résultat est une liste d'arêtes décrivant le contour, mais qui sont redondantes et non triées. (Exemple de redondance : si $[a, b]$ est un segment détecté et c est un point de ce segment qui correspond à un coin, alors les segment $[a, c]$ et $[c, b]$ seront aussi détectés.

Pour trier les arêtes, on va les subdiviser de sorte à transformer les arêtes qui soient assez proches d'un coin en deux arêtes passants par ce coin. Cela étant fait, on va relier les arêtes de manière à pouvoir reconstruire le contour en utilisant un critère qui est la maximisation du nombre d'arêtes reliées. On procède de la manière suivante :

On sélectionne une arête $[a_0, b_0]$, on cherche une nouvelle arête ayant pour l'une de ses extrémités b_0, on continue cette procédure en gardant à chaque itération le nombre d'arêtes reliées. Si on n'arrive plus à relier des arêtes, on fait un back-tracking sur la dernière arête choisie et on continue. Une fois la liste d'arêtes parcourue, on garde le nombre maximal d'arêtes qu'on peut relier et on relance cet algorithme de liaison jusqu'à ce que le nombre d'arêtes reliées soit égal à ce nombre.

FIGURE 8 – Tracé de la liste d'arêtes obtenues après les procédures de subdivision et le tri

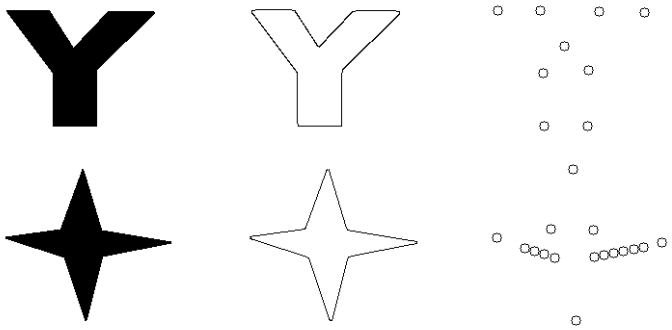
On peut ainsi extraire une liste de points ordonnés correspondants à notre polygone, la fonction `get_pos` de la classe `GridMap` nous permet d'obtenir les positions en mètres correspondant à chacun des points. Cette liste de positions ainsi que d'autres propriétés de la OccupancyGridMap sont alors utilisées pour créer une instance de la classe World contenant cette obstacle.

6.2 Cas d'une image avec plusieurs obstacles

Dans le cas plus pratique où on a plusieurs obstacles sur notre image, l'algorithme utilisé est sensiblement le même, hormis quelques changements. En particulier, l'algorithme de tri des arêtes est lancé à l'intérieur d'une boucle qui continue à tourner tant que la liste d'arêtes initiale est d'une longueur supérieure ou égale à 3 (c.à.d tant qu'on peut encore avoir un polygone dans cette liste).

Une fois ayant obtenu la liste d'arêtes de longueur maximale que l'on peut relier, cette liste sera sauvegardée dans une nouvelle variable puis ses arêtes seront supprimées de la liste d'arêtes initiale. Puis l'algorithme va être relancé avec les arêtes restantes.

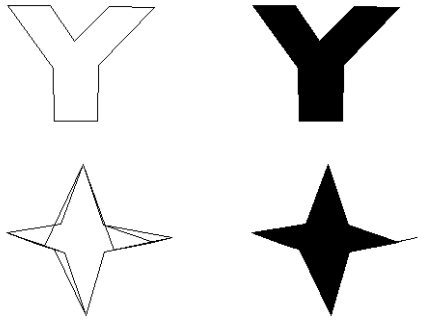
On obtiendra à la fin une liste dont chacun des éléments correspond à une liste d'arêtes décrivant un obstacle. La suite de la transformation est alors identique à celle du cas avec un seul obstacle.



(a) Image obtenue d'après une GridMap

(b) Image du contour

(c) Coins détectés



(d) Lignes détectés

(e) Polygones obtenus

7 Simulation Gazebo du robot

Avant de pouvoir expérimenter sur le robot réel, nous simulons d'abord le Kuka Youbot dans un environnement Gazebo. Cela nous permettra de connaître la précision exacte du robot et de la comparer à sa position estimée afin de connaître l'efficacité de l'algorithme de localisation que nous utiliserons.

Afin de connaître la description du robot et son comportement cinématique, nous avons besoin de 3 packages ROS disponibles sur GitHub :

— `https://github.com/youbot/youbot_driver`
— `https://github.com/mas-group/youbot_description`
— `https://github.com/mas-group/youbot_simulation`

Nous avons choisi des packages tiers pour les 2 derniers packages car ceux-ci sont plus à jour que leurs versions officielles. En effet, leurs versions officielles

provoquaient des erreurs (notamment lors de l'interprétation de la description du robot) sous ROS Melodic (version de ROS que nous utilisons dans ce projet).

Pour cette première version de la simulation du robot, nous recréons manuellement le monde "World1" utilisé précédemment pour la visualisation. Pour lancer la simulation Gazebo, on lance le fichier launch "empty_world.launch" du package gazebo_ros avec pour argument le fichier de description du monde Gazebo ainsi créé.

On utilise ce même package pour insérer le robot dans ce monde. Par défaut, sa position initiale est ($x = 0$, $y = 0$, $\theta = 0$, $z = 0.1$). θ est défini comme l'orientation en radians du robot par rapport à l'axe des x dans le plan (x, y). Cette définition ne correspond pas à celle utilisée par Shapely, et donc pas non plus à notre librairie Python, dans laquelle θ est défini comme l'orientation par rapport à l'axe des y. Ainsi, il faut penser à réaliser une conversion de l'orientation entre la librairie Python et le package ROS en ajoutant $\pm\frac{\pi}{2}$ radians selon le sens de conversion. La position du robot est définie par la position de son centre de masse, d'où la valeur de z non nulle.

On inclut aussi les fichiers launch permettant de simuler la cinématique du robot, un noeud pour téléopérer le robot à la manette, et une visualisation rviz. Enfin, on inclut le noeud "robot_state_publisher" permettant de publier les informations de position et d'orientation des différents éléments du robot sur le topic /tf.

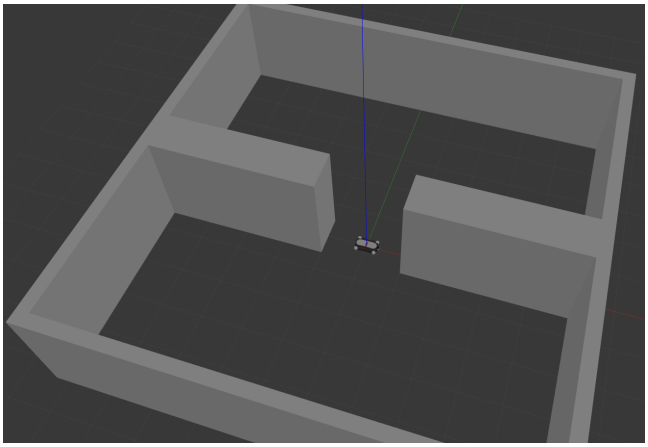


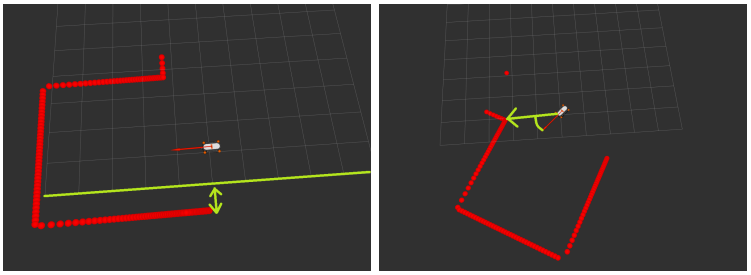
FIGURE 9 – Youbot simulé dans la version gazebo du monde "World1"

Les dimensions du monde "World1" étant bien plus grandes que les dimensions du robot, le "goulot" présent dans ce monde ne joue pas son rôle bloquant. En effet, dans ce cas, quelle que soit l'orientation du robot, celui-ci peut aisément passer d'un côté à l'autre du monde. On peut reproduire ce rôle bloquant en simulant le robot portant un objet encombrant de grande taille.

8 Bug de l'odométrie

Afin de passer aux étapes suivantes (construction du plan pour le robot simulé et suivi de celui-ci), il est nécessaire de connaître la position et l'orientation du robot. Dans une première version de cette simulation, on laisse le rôle de publication de l'état du robot à la simulation elle-même, qui doit donc fournir des valeurs exactes : la position du robot dans la simulation doit être la même que celle publiée par cette même simulation.

Cependant, en observant le scan du monde publié par le laser, on remarque que celui-ci correspond bien à la position dans la simulation mais pas à celle reçue dans la visualisation rviz.



(a) (b)

FIGURE 10 – Visualisation de la position du robot et du scan laser après (a) une translation de 3,50m selon l'axe y puis (b) une rotation de 4π radians dans le sens trigonométrique. En vert, on a (a) la position théorique du mur et son écart avec la position du mur captée par le scan laser, et en (b) on a l'orientation théorique du robot dans la visualisation, en comparaison avec son orientation telle que reçue par la visualisation.

Comme on peut l'observer sur la figure ci-dessus, il y a contradiction entre le scan laser du monde et la position visualisée du robot. Or, les scans laser correspondent à la position du robot dans la simulation. Il y a donc un problème dans la transmission de la position et de l'orientation du robot entre la simulation et la visualisation.

Puisque nous n'avons pas réussi à résoudre ce problème, il a été impossible d'implémenter les étapes suivantes du projet. Nous les présentons quand même à titre d'information.

9 Construction du plan pour le robot simulé

Maintenant que le robot est simulé, nous pouvons l'utiliser pour construire un plan d'un point à un autre du monde. Nous nous baserons sur l'algorithme de planification RRT* présenté précédemment.

Pour fonctionner, notre algorithme de planification a besoin d'une description du monde (donnée ici directement par "World1"), ainsi que d'une descrip-

tion du polygone du robot. Ici, ce polygone doit contenir le robot simulé (au minimum) ainsi que l'objet qu'il porte. Si ce n'était pas le cas, l'algorithme de planification pourrait permettre une position impossible à réaliser pour le robot simulé. On peut imaginer publier le polygone ainsi décrit, afin de le visualiser sur rviz pour vérifier qu'il n'y a bien aucune collision avec les obstacles.

La recherche est instanciée et lancée lorsqu'on publie un "goal" sur rviz. On peut aussi publier ce goal à la main en publiant sur le topic /goal directement depuis la console. Le noeud de planification sauvegarde alors la position actuelle du robot comme point de départ et le "goal" comme point d'arrivée. Ensuite, l'algorithme RRT* est lancé pour chercher un chemin entre ces deux points. Pour que l'algorithme se termine, il faut que la position du robot et le "goal" soient des positions possibles pour le polygone donné en argument à RRT*, qui peut être différent (plus grand) que le polygone du robot seul, et qu'il existe un chemin entre ces deux points praticable par le robot. Pour visualiser le résultat, on peut publier l'arbre sur un topic /rrt_tree et le chemin trouvé sur un topic /plan, et observer ces topics sur rviz.

On notera que pour que le chemin trouvé reste valable, il ne faut plus déplacer le robot une fois le "goal" publié.

10 Suivi du plan par le robot simulé

Une fois le chemin du point de départ au point d'arrivée obtenu, il faut faire en sorte que le robot suive ce chemin.

Un façon naïve de résoudre ce problème serait de suivre de manière exacte le chemin obtenu grâce à la planification. L'avantage de cette méthode est que le polygone du robot ne rentre jamais en collision avec les obstacles. Cependant, cette méthode est impossible à utiliser dans un cas non idéal. Tout d'abord, avec un robot réel, la position du robot ne reflète pas exactement la commande fournie au robot. De plus, dans le cas où la position du robot n'est pas connue de manière exacte, il est impossible de savoir si on suit bien le chemin de façon exacte.

Ainsi, pour suivre un chemin de manière réaliste, on va utiliser l'algorithme DWA (Dynamic Window Approach), qui permet d'obtenir un plan local à partir d'un plan global.

10.1 Algorithme DWA

On se place ici dans l'espace des vitesses $u = (\dot{x}, \dot{y}, \dot{\theta}) \in \mathcal{V}$. De part les limitations physiques du robot, ces vitesses sont bornées.

On veut prendre en compte deux paramètres : à quel point le robot est proche du chemin à suivre, et à quel point on est proche (en temps) de rentrer

en collision avec un obstacle. On évalue donc $\theta_{clear}(u)$ et $\theta_{path}(u)$:

$$\theta_{clear}(u) = \begin{cases} 0 & t_{col}(u) \leq T_b(u) \\ \frac{t_{col}(u) - T_b(u)}{T_{max} - T_b(u)} & T_b(u) < t_{col}(u) \leq T_{max} \\ 1 & t_{col}(u) > T_{max} \end{cases}$$

où u est la commande, $t_{col}(u)$ est le temps de première collision pour la commande, T_b est le temps de freinage et T_{max} est l'horizon temporel. Ici, la valeur optimale est donc $\theta_{clear}(u) = 1$.

$$\theta_{path}(u) = 1 - \frac{\sum_{i=1}^{N_t} \sum_{j=1}^{N_p} j d_{i,j} - D_{min}}{D_{max} - D_{min}}$$

où N_t est le nombre d'étapes de la trajectoire échantillonnée définie par la commande u, N_p est le nombre d'étapes du chemin de référence, $d_{i,j}$ est la distance entre le point i de la trajectoire échantillonnée et le point j du chemin de référence. D_{min} et D_{max} sont respectivement les minimum et maximum de la double somme et permettent de normaliser θ_{path} entre 0 et 1. Ici, la valeur optimale est $\theta_{path} = 1$.

Pour que θ_{path} reste cohérent, il faut oublier le chemin de référence petit à petit, afin que θ_{path} ne nous pousse pas à retourner en arrière.

On définit ensuite :

$$\Gamma = \lambda \theta_{clear} + (1 - \lambda) \theta_{path}$$

où λ est un poids à déterminer.

Ainsi, au vu des valeurs optimales de θ_{clear} et θ_{path}, on obtient que la valeur optimale de Γ est 1, c'est à dire son maximum.

Ainsi, l'algorithme DWA revient à déterminer à chaque étape le vecteur vitesse $\hat{u}$ tel que :

$$\hat{u} = \arg\max_{u \in \mathcal{V}} \Gamma(u)$$

On publie ensuite $\hat{u}$ sur le topic /cmd_vel pour déplacer le robot, puis on répète l'opération jusqu'à être à une distance relativement petite (c'est un paramètre à préciser) du point d'arrivée. Une fois dans ce cas, on n'utilise plus DWA. On publie de faibles vélocités jusqu'à obtenir un vecteur position assez proche de celui demandé au point d'arrivée.

11 Conclusion

Les objectifs du projet n'ont pas été complètement atteint : même si nous sommes parvenus à représenter le robot, son environnement et le chemin qu'il a à suivre sous différentes formes, afin de rendre la compréhension des enjeux et la manière de résoudre ces problèmes accessible au plus grand nombre ; il est clair que la partie la plus concrète et la plus importante du projet, à savoir de faire bouger le robot en lui faisant calculer une trajectoire et la lui faire suivre, n'est pas satisfaisante.

Références

[1] M. Kleinbort, K. Solovey, Z. Littlefield, K. E. Bekris and D. Halperin, *Probabilistic Completeness of RRT for Geometric and Kinodynamic Planning With Forward Propagation*, in IEEE Robotics and Automation Letters, vol. 4, no. 2, pp. x-xvi, April 2019, `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8584061&isnumber=8581687`

[2] Léonard Jaillet, *Méthodes probabilistes pour la planification réactive de mouvements*, Automatique Robotique, Université Paul Sabatier - Toulouse III, 2005, pages 12-43

[3] Wikipedia contributors. *Motion planning* [Internet]. Wikipedia, The Free Encyclopedia ; accédé le 24/01/2020, `https://en.wikipedia.org/wiki/Motion_planning`.