ROB201 - Introduction à la robotique mobile

David Filliat - Elena Ivanova - Thibault Toralba
ENSTA Paris

28 mars 2023

Introduction

Ce document indique la trame générale prévue pour le développement du projet dans le cadre du cours ROB201. Les travaux prévus dans chaque séance sont indicatifs, il peuvent être améliorés si les travaux proposés sont terminés avant la fin, mais chaque séance doit être finie avant de passer à la suivante.

1 Séance 01 : Installation et prise en main

Nous vous conseillons d'utiliser l'environnement de développement Visual Studio Code (mais un autre est possible si vous êtes à l'aise avec). Installez l'extension Python qui fournit des outils utiles pour le développement python.

1.1 Installation du code de base

Commencez par créer votre propre copie du dépot github :

https://github.com/emmanuel-battesti/ensta-rob201

Pour cela vous devez posséder votre propre compte GitHub (créez le si besoin), puis, sur la page du dépot, sur le bouton `Fork`, sélectionnez `Create a new Fork`.

Clonez ensuite sur votre ordinateur le squelette de code que vous venez de dupliquer. Dans un terminal, tapez :

```
> git clone git@github.com:votre_login_github/ensta_rob201.git
```

Vous pouvez ensuite commencer à travailler sur le code téléchargé. Pendant toute la durée du cours, pensez à faire des `commit` fréquents, avec un message court et informatif. Pensez à faire des push réguliers vers votre déport github (à chaque fin de séance par exemple).

Suivez ensuite la procédure d'installation décrite dans le fichier `INSTALL.md`. Vous devrez adapter les commandes en fonction de votre version de python 3 (si ce n'est pas la 3.8). En particulier, ne sautez pas la phase de création de l'environnement virtuel et pensez bien à l'activer.

Une fois l'installation terminée, dans VSCode, ouvrez le répertoire `ensta_rob201`. Sélectionnez ensuite l'interpréteur python de l'environnement virtuel que vous venez de créer (avec `(Ctrl+Shift+P) Python: Select Interpreter`). Vous pouvez lancer le projet en exécutant le fichier `main.py`.
La fenêtre de simulation doit apparaître (Figure 1), et il ne se passe rien d'autre, c'est normal, la fonction de contrôle de votre robot est vide. Pour quitter la simulation, tapez `q` dans la fenêtre de la simulation.

1.2 Linter

Pour utilisez le linter, il faut le configurer en tapant `(Ctrl+Shift+P)` puis rechercher la commande `Python: Select Linter`. Sélectionnez `pylint`. A chaque sauvegarde de fichier python, le résultat de l'analyse apparaît dans l'onglet `Problems` (par défaut sous le code).

Pour vous entraîner, corrigez les problèmes relevés dans le fichier `main.y`.

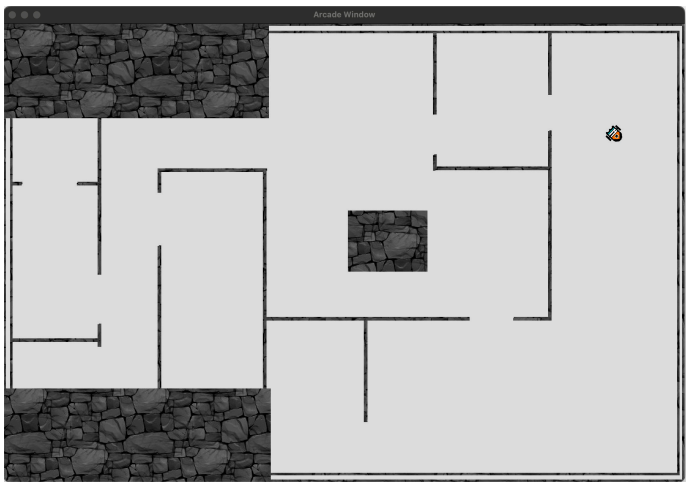

FIGURE 1 – Fenêtre du simulateur Place-bot

1.3 Profiler

Pour utiliser le profiler, utilisez d'abord le programme `cProfile` pour lancer votre script et enregistrer un fichier de statistiques :

```
python3 -m cProfile -o mon_script.prof mon_script.py
```

Utilisez ensuite l'interface graphique `snakeviz` pour visualiser plus facilement les résultats :

```
> python3 -m pip install snakeviz
> python3 -m snakeviz mon_script.prof
```

Au travers de l'interface graphique, vous pouvez alors déterminer les fonctions prenant le plus de temps d'exécution et tenter de les optimiser.

Utilisez le profiler sur le fichier `main.py` fourni (tapez `q` dans la fenêtre de la simulation pour l'arrêter au bout de quelques secondes), déterminez les fonctions les plus chronophages. Parmis celles-ci, certaines sont dans les bibliothèques utilisée et ne peuvent pas être modifiées. Déterminez celles auxquelles vous avez accès et tentez de les optimiser.

1.4 Prise en main du simulateur

Nous utilisons dans ce cours un simulateur simple développé à l'ENSTA : Place-bot (il a déjà été installé avec le code du cours, il n'est pas utile de le réinstaller vous-même).

Pour utiliser ce simulateur, il faut créer :

— une classe qui dérive de la classe `RobotAbstract` pour définir votre robot, en particulier une fonction `control(self)` qui va donner les commandes à votre robot en fonction des données capteurs,

— une classe qui dérive de la classe `WorldAbstract` pour définir l'environnement de simulation avec votre robot en paramètre,

— un simulateur dérivant de la classe `Simulator` avec votre monde en paramètre.

L'exécution de la fonction `Simulator.run()` va ensuite simuler le fonctionnement de votre robot avec sa fonction de contrôle. Il est également possible de contrôler le robot avec le clavier en passant l'argument `use_keyboard=True` au simulateur.

L'ensemble de ces étapes a été fait dans le fichier `main.py`, avec les classes définies dans les fichiers `my_robot_slam.py` et `worlds/my_world.py`.

Programmez ensuite un premier comportement pour le robot. Pour cela, adaptez la fonction `control(self)` dans la classe `MyRobotSlam` afin de faire un évitement d'obstacles simple. Pour une bonne organisation du code, vous pouvez mettre votre code dans une ou plusieurs fonction dans le fichier `control.py`, et importer les fonctions utilse dans le fichier `my_robot_slam.py`

Vous pouvez accéder aux données :

— du **télémètre laser** grâce à la fonction `self.lidar()` qui renvoie un object de type `Lidar`. Cet objet a une méthode `get_sensor_values()` qui renvoie les distances mesurées par le

télémètre laser sous forme d'un `array` Numpy, et une méthode `get_ray_angles()` qui renvoie la direction (angle en radians, sens trigonométrique) de chaque rayon du télémètre laser par rapport à l'avant du robot. Le champ de vision du télémètre est de 2π.
— de **l'odométrie** avec la fonction `self.odometer_values()` qui renvoie un `array` avec la position $[x, y, \theta]$ estimée dans le repère odométrie initialisé à $[0, 0, 0]$ au lancement de la simulation.

Votre fonction doit renvoyer une commande de vitesses de translation et de rotation, dans l'intervalle $[-1, 1]$, sous forme d'un dictionnaire python :

```
command = {"forward": 0,
           "rotation": 0}
```

Implémentez un comportement qui avance en ligne droite quand il n'y a pas d'obstacles devant le robot, et qui tourne d'un angle aléatoire quand un obstacle est présent.

1.5 Extensions possibles

Vous pouvez ensuite améliorer le comportement d'évitement d'obstacles, par exemple en gérant un historique des directions de rotation pour éviter de tourner toujours du même coté, ou en choisissant la direction de rotation en fonction de l'angle de l'obstacle (pour éviter de tourner face à un mur). Vous pouvez également essayer de réaliser un algorithme de suivi de mur, par exemple en vous inspirant du TP 'Wall Follow' de F1TENTH[1]. Essayez de faire en sorte que le robot explore tout l'environnement.

Vous pouvez également définir un nouvel environnement de simulation en remplacement de celui fournit dans le fichier `worlds/my_world.py`.

2 Séance 02 : Navigation réactive

2.1 Travail demandé

La séance consiste à créer un contrôle réactif du robot basé sur un algorithme de champ de potentiel. Pour cela, nous créons une nouvelle fonction de contrôle `potential_field_control()` dans `control.py`, prenant en argument :
— `lidar`, dont les méthodes `get_sensor_values()` (`array` Numpy des distances mesurées) et `get_ray_angles()` (angle de la mesure en radians, sens trigonométrique) seront utilisées pour l'évitement d'obstacles.
— `pose`, contenant la meilleure estimation de la pose (`array` Numpy [x, y, theta]) actuelle du robot dans le repère odom ou world.
— `goal`, pose (`array` Numpy [x, y, theta]) de la cible à atteindre dans le repère odom ou world.

Pour une position q_{goal} cible donnée, calculez le gradient de potentiel attractif au niveau de la pose actuelle q du robot sous sa forme linéaire :

$$\nabla f = \frac{K_{goal}}{d(q, q_{goal})}(q_{goal} - q)$$

Proposez une commande de déplacement en vitesse linéaire et en vitesse de rotation selon l'orientation et la norme du vecteur calculé.

Ajoutez une condition d'arrêt à proximité de l'objectif pour gérer les imprécisions de pose et le cas problématique de la distance très proche de 0.

Testez votre fonction `potential_field_control()` en l'appelant dans la fonction `control(self)` de la classe `MyRobotSlam` avec une pose objectif statique judicieusement choisie, et constatez le comportement.

Pour lisser les déplacements du robot, changez le comportement à proximité de l'objectif avec un potentiel quadratique. Choisissez le coefficient du gradient pour assurer une continuité avec le comportement précédent.

1. Disponible sur https://f1tenth-coursekit.readthedocs.io/en/latest/assignments/labs/lab3.html

A l'aide des données du LIDAR, calculez le gradient de potentiel répulsif de l'obstacle le plus proche du robot, et ajoutez le comportement d'esquive d'obstacle à la commande du robot.

$$\nabla f = \frac{K_{obs}}{d^3\left(q,\, q_{obs}\right)} \left(\frac{1}{d\left(q,\, q_{obs}\right)} - \frac{1}{d_{safe}} \right) \left(q_{obs} - q\right)$$

Gardez à l'esprit que les consignes de vitesses linéaire et angulaire doivent être normées sur [-1,1].

Maintenant que votre esquive d'obstacle est opérationnelle, vous pouvez éditer la fonction `control(self)` de la classe `MyRobotSlam` pour lui ajouter une condition de validation, afin de tirer un nouvel objectif une fois le précédent atteint. Selon le comportement du robot, ajustez les différents paramètres de votre contrôle réactif (rayon de changement de potentiel attractif, rayon de répulsion des obstacles, coefficients attractifs / répulsifs, etc.)

2.2 Extensions possibles

La prise en compte seule de l'obstacle le plus proche dans le calcul de répulsion peut conduire à des comportements oscillatoires dans certaines situations (coins, passages étroits). Proposez une solution pour segmenter les points issus du LIDAR en obstacles distincts, et appliquer un potentiel répulsif à chacun d'entre eux.

Vous pourrez rencontrer des cas de minimums locaux immobilisant le robot avant sa cible. Implémentez une détection de ces points et proposez un comportement de récupération pour l'en extraire.

3 Séance 03 : Cartographie

3.1 Code fourni

La classe `TinySlam` est fournie dans le fichier `tiny_slam.py`, avec certaines fonctions :

- `__init__(self, x_min, x_max, y_min, y_max, resolution)` : le constructeur de la classe qui initialise la grille d'occupation et définit les bornes et la résolution de la carte.
- `add_map_line(self, x_0, y_0, x_1, y_1, val)` : ajoute la valeur `val` à tous les points d'une ligne définie par ses extrémités avec l'algorithme de Bresenham.
- `add_map_points(self, points_x, points_y, val)` : ajoute la valeur `val` à un ensemble de points dont les coordonnées sont fournies sous forme de liste.
- `_conv_world_to_map(self, x_world, y_world)` : converti les coordonnées du monde en coordonnées dans la carte (indexes de la cellule).
- `_conv_map_to_world(self, x_map, y_map)` : converti les coordonnées de la carte en coordonnées dans le monde
- `display(self)` : affiche la carte avec `matplotlib`
- `display2(self, robot_pose)` : affiche la carte et la position du robot avec `opencv` (a préférer car plus rapide que la version matplotlib)
- `save(self, filename)` : sauvegarde la carte sous forme d'image png et l'ensemble des données (carte, résolution, origine...) sous forme de fichier pickle.

3.2 Travail demandé

Complétez la classe `TinySlam` en écrivant la fonction `update_map(self, lidar, pose)` qui va intégrer les données du télémètre laser dans la grille d'occupation. Pour cela, il vous faut réaliser les opérations suivantes :

- Conversion des coordonnées polaires locales des détections du laser (directions/distances) à partir de la position absolue du robot en coordonnées cartésiennes absolues dans la carte pour avoir les positions des points détectés par le laser.
- Mise à jour de la carte avec chaque point détecté par le télémètre en fonction du modèle probabiliste :
  - mise à jour des points sur la ligne entre le robot et le point détecté avec une probabilité faible
  - mise à jour des points détectés par le laser avec une probabilité forte
- Seuillage des probabilités pour éviter les divergences

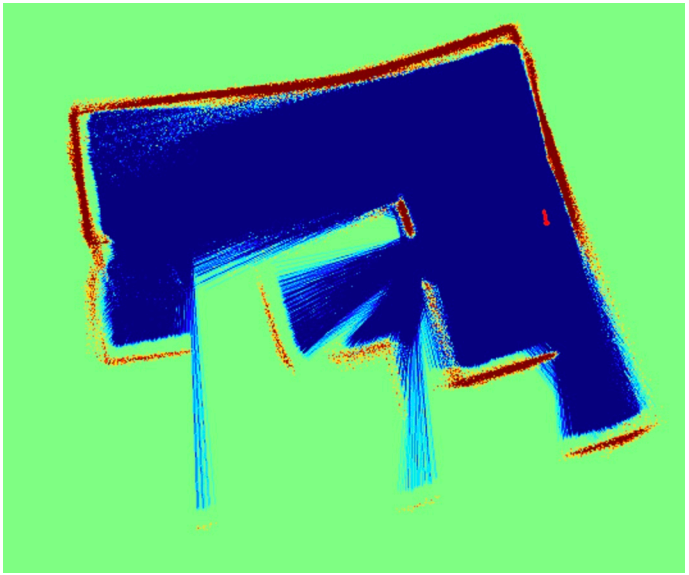

FIGURE 2 – Exemple de carte construite en utilisant le télémètre laser et la position donnée par l'odométrie.

Vous pouvez tester cette cartographie à partir de la position de l'odométrie. Vous observerez une dérive normale due aux erreurs de l'odométrie (voir l'exemple de la figure 2). Pour cela, dans la fonction `control(self)` de la classe `MyRobotSlam`, lancez la mise à jour de la carte, puis un affichage (avec une fréquence réduite éventuellement, c'est à dire afficher seulement une fois sur 10 par exemple). Ajuster le modèle probabiliste jusqu'à avoir une image ressemblant à la figure 2.

3.3 Extensions possibles

Vous pouvez tenter d'implémenter des modèles probabilistes plus complexes afin de mieux représenter les obstacles. Vous pouvez également diminuer le nombre de points pris en compte, soit de manière régulière (1 sur 2), soit de manière adaptative (seulement des points dont l'espacement est supérieur à un seuil : il n'est pas utile de mettre à jour avec deux points tombant dans la même cellule).