4ROB1 - Introduction à la robotique mobile

David Filliat - Elena Vanneaux - Thibault Toralba
ENSTA

28 mars 2025

Introduction

Ce document indique la trame générale prévue pour le développement du projet dans le cadre du cours 4ROB1 à l'ENSTA. Les travaux prévus dans chaque séance sont indicatifs, ils 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 dépôt 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 dépôt, 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 pushs réguliers vers votre dépôt 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 Profiler

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

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

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

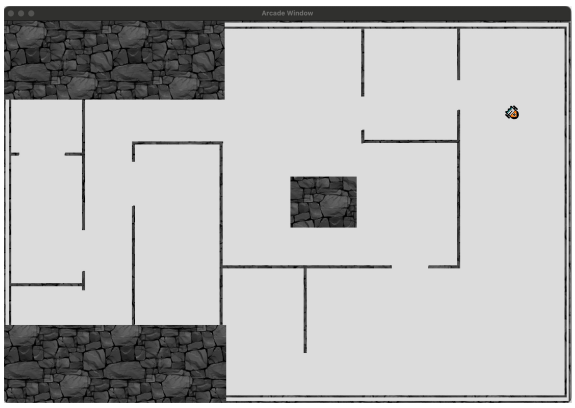

FIGURE 1 – Fenêtre du simulateur Place-bot

```
> 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. Parmi celles-ci, certaines sont dans les bibliothèques utilisées et ne peuvent pas être modifiées. Déterminez celles auxquelles vous avez accès et tentez de les optimiser.

1.3 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 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 fonctions dans le fichier `control.py` (la fonction `reactive_obst_avoid(lidar)` est déjà prévue pour cela), et importer les fonctions utiles dans le fichier `my_robot_slam.py`

Vous pouvez accéder aux données :

- du **télémètre laser** grâce à la fonction `self.lidar()` qui renvoie un objet 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, dans le repère robot, voir figure 2) de chaque rayon du télémètre laser par rapport à l'avant du robot. Le champ de vision du télémètre est de 2π (de $-\pi$ à π).
- de **l'odométrie** avec la fonction `self.odometer_values()` qui renvoie un `array` avec la position $[x_O, y_O, \theta_O]$ estimée dans le repère odométrie initialisé à $[0, 0, 0]$ au lancement de la simulation (voir figure 2). Le robot démarre avec une orientation dans le simulateur aléatoire par défaut, le repère absolu et le repère odométrie ont donc une orientation aléatoire

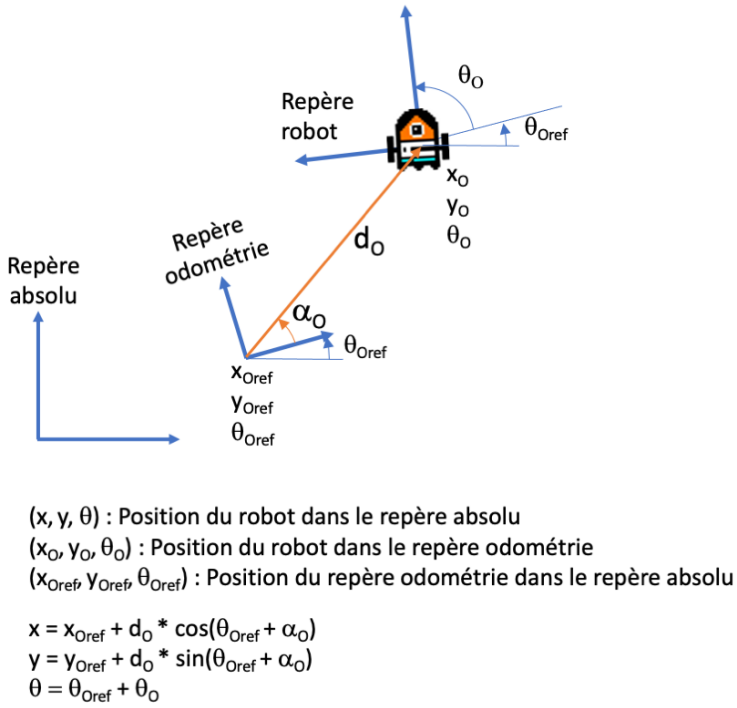

FIGURE 2 – Les différents repères utilisés dans le projet. Tous les angles sont en radians, dans le sens trigonométrique. Au début de la simulation, les trois repères (absolu, odométrie, robot) sont identiques.

par rapport à la fenêtre du simulateur à chaque lancement. Vous pouvez supprimer cette initialisation aléatoire pour mettre au point plus facilement si besoin.

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.

1.4 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`.

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

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.
- `current_pose`, contenant l'estimation q de la pose (`array` Numpy [x, y, theta]) actuelle du robot dans le repère odom ou world.
- `goal_pose`, pose q_{goal} (`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.

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`. Ses fonctions ne sont pas encore implémentées. Elle s'appuie sur la classe `OccupancyGrid` fournie dans le fichier `occupancy_grid.py` et utilise certaines de ses 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 (coordonnées dans le repère absolu).
- `add_value_along_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 (coordonnées dans le repère absolu).
- `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 (coordonnées dans le repère absolu).
- `conv_world_to_map(self, x_world, y_world)` : converti des coordonnées dans le repère absolu en coordonnées des cellules correspondantes dans la grille d'occupation.
- `conv_map_to_world(self, x_map, y_map)` : converti les coordonnées de cellules dans la grille d'occupation en coordonnées dans le repère absolu.
- `display_plt(self, robot_pose, goal=None, traj=None)` : affiche la carte avec `matplotlib`
- `display_cv(self, robot_pose, goal=None, traj=None)` : affiche la carte et la position du robot avec `opencv` (à 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

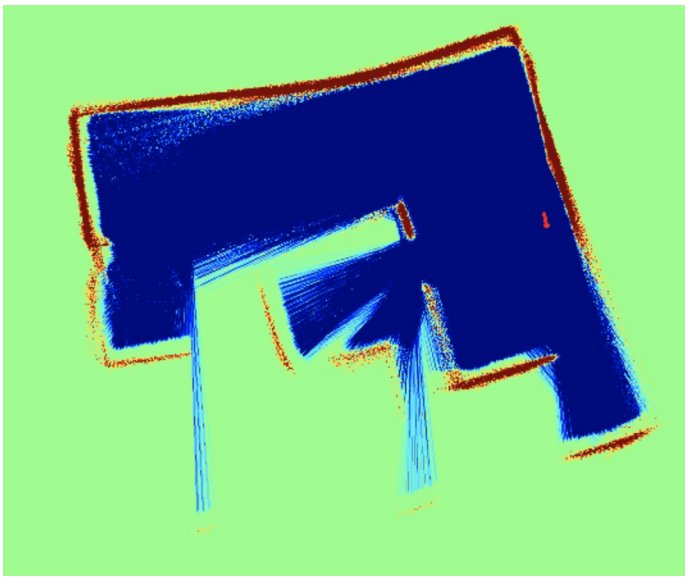

FIGURE 3 – Exemple de carte construite en utilisant le télémètre laser et la position donnée par l'odométrie.

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 détections du laser depuis les coordonnées polaires dans le repère robot (directions/distances) vers les coordonnées cartésiennes dans le repère absolu pour avoir les positions des points détectés par le laser dans la carte (voir figure 2).
- 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.

Vous pouvez tester cette cartographie à partir de la position du robot fournie par l'odométrie. Vous observerez une dérive normale due aux erreurs de l'odométrie (voir l'exemple de la figure 3). 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 3. En particulier la zone de probabilité 0.5 juste avant l'obstacle est utile pour avoir une cartographie stable.

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).

4 Séance 04 : Localisation

4.1 Travail demandé

Commencez par écrire la fonction `get_corrected_pose(self, odom_pose, odom_pose_ref=None)` qui renvoie la position du robot dans le repère absolu à partir de la position du repère odométrie dans le repère absolu (`odom_pose_ref`) et de la position du robot dans le repère odométrie (`odom`, voir figure 2). Si le paramètre `odom_pose_ref` n'est pas transmis, la fonction doit utiliser la valeur mémorisée `self.odom_pose_ref`.

Écrivez ensuite la fonction `_score(self, lidar, pose)` qui calcule pour une position de robot dans le repère absolu et un scan lidar la somme des valeurs des cellules dans lesquelles les points du lidar tombent. Pensez à bien optimiser son temps de calcul (utilisez des opérations sur des array `numpy` au lieu de boucles `for`) car cette fonction sera appelée de très nombreuses fois. Les opérations à réaliser sont :

- Supprimer les points à la distance maximale du laser (qui ne correspondent pas à des obstacles),
- Estimer les positions des détections du laser dans le repère absolu,
- Convertir ces positions en index de cellules dans la grille d'occupation et supprimer les points hors de la carte,
- Lire et additionner les valeurs des cellules correspondantes dans la carte pour calculer le score.

Écrivez enfin la fonction `localise(self, lidar, raw_odom_pose)` qui va modifier la position de référence de l'odométrie afin de maximiser le score correspondant au scan laser. Utilisez pour cela une simple recherche aléatoire. La fonction devra réaliser les opérations suivantes :

- Calculez le score du scan laser avec la position de référence actuelle de l'odométrie (`self.odom_pose_ref`),
- répétez tant que moins de N tirages sans amélioration :
  - Tirez un `offset` aléatoire selon une gaussienne de moyenne nulle et de variance σ et ajoutez le à la position de référence de l'odométrie
  - Calculez le score du scan laser avec cette nouvelle position de référence de l'odométrie,
  - Si le score est meilleur, mémorisez ce score et la nouvelle position de référence

La fonction doit mettre à jour la position de référence de l'odométrie `self.odom_pose_ref` et renvoyer le meilleur score trouvé.

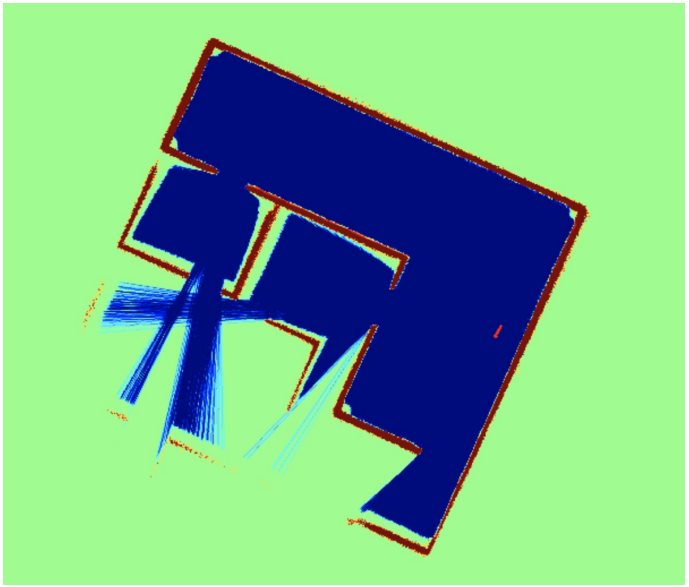

FIGURE 4 – Exemple de carte construite en utilisant le télémètre laser et la position donnée par la localisation.

Utilisez ensuite la fonction `localise` dans la fonction `control` de la classe `my_robot_slam` pour estimer la position du robot avant de faire la mise à jour de la carte si le score de localisation est

suffisamment élevé (au-dessus d'un seuil à déterminer). Optimisez ensuite le nombre de tests N, la variance σ utilisée pour la génération d'offset aléatoires, et éventuellement le modèle probabiliste du laser utilisé dans la méthode de cartographie pour que la cartographie soit stable. Le SLAM ainsi obtenu ne devrait plus dériver et fournir une carte cohérente (voir l'exemple de la figure 4).

4.2 Extensions possibles

Vous pouvez tenter d'implémenter une méthode de recherche plus efficace, telle que Cross Entropy Method (CEM) ou Covariance Matrix Adaptation Evolution Strategy (CMA-ES [2]).

Vous pouvez implémenter une interpolation bilinéaire des valeurs de la carte pour avoir une estimation plus fine du score.

2. Voir https://en.wikipedia.org/wiki/CMA-ES et le package python `cma` : `https://pypi.org/project/cma/`

5 Séance 05 : Planification de trajectoire

5.1 Travail demandé

Commencez par écrire dans la classe `Planner`, qui se trouve dans le fichier `planner.py`, la fonction `get_neighbors(self, current_cell)` qui, pour une position courante, renvoie ses 8 voisins sur la carte. Et la fonction `heuristic(self, cell_1, cell_2)` qui renvoie la distance euclidienne entre deux points de la carte. Vérifiez comment la structure de données `heapq` est implémentée en Python.

Écrivez ensuite la fonction `plan(self, start, goal)` qui prend la position de départ et d'arrivée comme arguments et renvoie le chemin le plus court entre eux. Pour trouver le chemin le plus court, implémentez l'algorithme A* avec une heuristique h égale à la distance euclidienne entre la position actuelle et le but. Attention au système de coordonnées : le départ et le but sont donnés dans le repère absolu. Le chemin doit être calculé en termes de cellules de la grille d'occupation, puis converti en coordonnées absolu. Vous pouvez utiliser les fonctions `conv_world_to_map(self, x_world, y_world)` et `conv_map_to_world(self, x_map, y_map)` de la classe `OccupancyGrid` pour basculer entre les systèmes de coordonnées. Vous pouvez baser votre algorithme sur le pseudocode suivant :

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path

// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach goal from node n.
function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    // This is usually implemented as a min-heap or priority queue rather than a
    hash-set.
    openSet := {start}

    // For node n, cameFrom[n] is the node immediately preceding it on the cheapest
     path from the start
    // to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from start to n
    currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0

    // For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our current
    best guess as to
    // how cheap a path could be from start to finish if it goes through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)

    while openSet is not empty
        // This operation can occur in O(Log(N)) time if openSet is a min-heap or a
     priority queue
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        for each neighbor of current
            // d(current,neighbor) is the weight of the edge from current to
    neighbor
            // tentative_gScore is the distance from start to the neighbor through
    current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
                // This path to neighbor is better than any previous one. Record it
    !
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := tentative_gScore + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)
```

```
// Open set is empty but goal was never reached
return failure
```

Pour utiliser votre fonction de planification, dans `my_robot_slam.py` :

— d'abord, faites de la cartographie/exploration avec les fonctions des TP précédents pendant un certain nombre d'itérations.

— à une itération choisie, calculez le plus court chemin entre la position courante du robot et la position initiale (0,0,0). Affichez le chemin avec la fonction `display_cv(self, robot_pose, goal, traj)` de la classe `OccupancyGrid`.

— pour les itérations suivantes, utilisez un contrôleur local pour suivre ce chemin. Arrêtez-vous lorsque le robot est revenu au point de départ.

5.2 Extensions possibles

Habituellement, les robots suivent le chemin avec une certaine erreur, de plus, chaque robot a une taille non nulle. Agrandissez les obstacles dans la carte de planification pour éviter les chemins situés trop proches des obstacles.

Implémenter un A* pondéré avec un coût f calculé comme $f = g + \mu * h$. Étudiez le rôle de l'heuristique, en conservant la carte par défaut, les positions de départ et d'arrivée et en modifiant le μ Avec le paramètre $\mu = 0.0$ quel est l'algorithme qui est exécuté ? Quels sont les avantages et les inconvénients par rapport à la version avec $\mu = 1.0$. Avec le paramètre $\mu = 5.0$ quel est le résultat de l'algorithme ? Quels sont les avantages et les inconvénients par rapport à la version avec $\mu = 1.0$? Quelle est la raison théorique de ce comportement ?

6 Séance 06 : Fonctions avancées

Lors de cette dernière séance, la priorité est de finaliser les fonctions demandées précédemment, debugger et nettoyer votre code, le commenter correctement. Ensuite vous pouvez tenter d'implémenter une ou plusieurs des fonctions suivantes.

6.1 Dynamic window

Implémentez la méthode de suivi de chemin et d'évitement d'obstacles *Dynamic Window* [Fox et al., 1997] à la place de la méthode de champ de potentiels.

6.2 Exploration

Implémentez la méthode *Frontier Based Exploration* [Yamauchi, 1997] afin de garantir une exploration complète de l'environnement.

Références

[Fox et al., 1997] Fox, D., Burgard, W., and Thrun, S. (1997). The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1) :23–33.

[Yamauchi, 1997] Yamauchi, B. (1997). A frontier-based approach for autonomous exploration. In *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97.'Towards New Computational Principles for Robotics and Automation'*, pages 146–151. IEEE.