IFT608 / IFT702

Planification en intelligence artificielle

Devoir 2 – Planification de trajectoires

Objectifs

- Comparer des algorithmes de planification de trajectoires.
- Utiliser et étendre des domaines et des problèmes de planification de trajectoires.
- Utiliser des outils standards en planification de trajectoires.

Consignes.

- Le devoir est le même pour les étudiants de IFT608 et IFT702.
- Le devoir 1 se fait en équipe d'au plus 4 étudiants.

- La remise se fait dans Moodle sous la forme de trois archives (***d2-etape1.zip***, ***d2-etape2.zip*** et ***d2-etape3.zip***).
- Les remises doivent respecter scrupuleusement les consignes données dans l'énoncé. Un non-respect des consignes qui entraîne une complication dans la correction sera pénalisé en fonction de l'écart des consignes. Dans le pire cas, si une tâche ne s'exécute pas et qu'il devient difficile de déterminer en un temps raisonnable si c'est à cause du non-respect des consignes ou à cause des erreurs, elle risque d'être évaluée à zéro.

- Pour réaliser ce devoir, vous devez utiliser la machine virtuelle qui vous a été fournie. Cette machine fonctionne sous Ubuntu 16.04. Elle possède tous les logiciels nécessaires pour mener à terme ce devoir : ROS, MoveIt! et ROSPlan.
- Si vous avez des questions techniques, référez-vous aux manuels et forums de ROS.

Grille d'évaluation

Étape 0. Exercices préparatoires	Non évalué
Étape 1. Planification de trajectoires dans MoveIt	/30
Étape 2. Implémentation d'actions primitives pour un robot	/30
Étape 3. Planification et exécution de plan avec ROSPlan	/40
Total	**/100**

A la fin du document, vous trouverez

- La grille détaillée de correction
- La description de deux problèmes techniques fréquemment rencontrés et leur solution.

0 Exercices préparatoires

Les deux exercices suivants ont pour but de vous familiariser avec les outils utilisés. Ces exercices ne sont pas évalués. Cela étant dit, il est fortement conseillé de faire les manipulations décrites.

0.1 Machine virtuelle

La machine virtuelle est disponible dans Moodle. Vous pouvez aussi la télécharger à l'adresse suivante

http://planiart.usherbrooke.ca/cours/ift608/TPs/ProjetROS/ubuntu-16.04.3-desktop-amd64-clone-VMWARE.zip

Le mot de passe est : rosplan123

Remarque. La machine virtuelle est configurée en QWERTY. Si vous avez de la difficulté à entrer le mot de passe, assurez-vous que vous utilisez le bon clavier.

0.2 Manipulations sans obstacles

Le but de cet exercice consiste à déplacer un robot et lui faire faire des manipulations simples dans un environnement sans obstacles.

Le robot utilisé sera un PR2 (Figure 1), un robot mobile fabriqué par Willow Garage[1].

Figure 1: Un robot PR2

0.2.1 Tâches à réaliser

Les manipulations se feront dans l'outil RViz (« ROS 3D Robot Visualizer »). Ce dernier permet de visualiser des robots et de lancer des requêtes de planification.

1. Dans un terminal, lancez le simulateur à l'aide de la commande suivante :
   roslaunch pr2_moveit_base demo.launch

> **Informations complémentaires**
> *pr2_moveit_base* est un « package » ROS situé à l'endroit suivant : *~/catkin_ws/src*. Tous les logiciels développés avec ROS sont organisés en package. Ces derniers peuvent contenir un nœud[2], une librairie, des fichiers de configuration, etc. Dans le cas présent, *pr2_moveit_base* est un package contenant tout le nécessaire pour utiliser un robot PR2 avec le logiciel MoveIt![3]. Il a été créé, en partie, en suivant ce tutoriel: http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/setup_assistant/setup_assistant_tutorial.html. Des modifications ont aussi été apportées afin de rendre mobile la base du robot (autrement, seul les bras du robot peuvent se déplacer lors de l'exécution d'un plan).
>
> Le dossier *catkin_ws*, est un espace de travail ROS, aussi appelé « *Catkin Workspaces* »[4]. L'outil *catkin* permet de construire et de gérer l'ensemble des packages ROS au sein d'un même espace de travail.

[1] http://wiki.ros.org/Robots/PR2
[2] http://wiki.ros.org/Nodes
[3] http://moveit.ros.org/
[4] http://wiki.ros.org/catkin/workspaces

2. Dans la fenêtre *Display*s, le groupe *Motion Planning* permet de lancer des requêtes de planification reliant un état initial à un état final désiré. **Ces états sont définis par rapport au *Planning Group* courant** (onglet *Planning Request*). Sélectionnez *left_arm* afin de pouvoir déplacer le bras gauche du robot.

   Les états initial et final (du *Planning Group* courant) peuvent être spécifiés de deux façons différentes :

   a. En déplaçant les marqueurs interactifs situés dans la fenêtre de visualisation du robot (disponibles seulement pour *left_arm*, *right_arm* et *base*).

   b. En changeant les *Start State* et *Goal State* dans la fenêtre *Motion Planning* (onglet *Planning*). En particulier, l'option *random* permet de spécifier un état aléatoire.

   L'algorithme de planification de trajectoire utilisé (tiré de la librairie OMPL) peut être spécifié dans l'onglet *Context* de la fenêtre *Motion Planning*. Si aucun algorithme n'est spécifié, MoveIt! utilisera un algorithme par défaut.

   Le bouton *Plan* de l'onglet *Planning* appelle l'algorithme afin de trouver une solution. **La solution inclut seulement le *Planning Group* courant**.

   - Si aucune solution n'est trouvée, la mention *Planning Failed* s'affichera. **De l'information supplémentaire sera affichée dans le terminal**.
     - Aucune solution n'est trouvée si l'état initial ou l'état final du *Planning Group* est en collision avec le robot lui-même ou avec l'environnement (des segments du robot sont alors affichés en rouge).
     - Aucune solution n'est trouvée si l'état initial ou l'état final se trouve en dehors des limites de l'environnement (*out of bounds*). Dans ce cas, il pourrait être avisé de rapprocher la base du robot du centre de l'environnement.
   - Si une solution est trouvée, la case à cocher *Loop Animation* dans l'onglet *Planned Path* permet de répéter l'animation.

   Effectuez différentes requêtes de planification en faisant varier :

   - Les états initial et final;
   - L'algorithme de planification utilisé;

3. Planifiez la trajectoire du robot complet à l'aide du *Planning Group* « *whole_body* » en suivant la démarche suivante :

   a. Sélectionnez un des *Planning Group* suivant : *left_arm*, *right_arm* ou *base*.
   b. Changez l'état final du groupe en utilisant les marqueurs interactifs.
   c. Effectuez les modifications voulues sur les deux autres *Planning Group.*
   d. Sélectionnez le *Planning Group* « *whole_body* ».
   e. Cliquez sur le bouton *Plan*.

0.3 Évitement d'obstacles

Le deuxième exercice consiste à exécuter des requêtes de planification pour le PR2 dans un environnement parsemé d'obstacles.

0.3.1 Tâches à réaliser

1. Visualisez le robot dans RViz, comme à l'exercice précédent.
2. Localisez le répertoire *Projet1-exemple* (*~/Desktop*). Ce dernier contient des fichiers dont vous aurez besoin dans cette tâche (voir ci-après).

3. Importez l'environnement *Projet1-exemple/example-scene/moveit.scene* dans RViz à partir de l'onglet *Scene Objects* de la fenêtre *Motion Planning*. La procédure est la suivante:
    a. Dans l'onglet *Scene Objects* de la fenêtre *Motion Planning,* cliquez sur le bouton *Import From Text*.
    b. Sélectionnez le fichier texte (*.scene*) de la scène. Les objets importés devraient apparaître dans la section *Current Scene Objects*.
    c. En sélectionnant un objet, il est possible d'ajuster sa position, son orientation et sa taille dans la scène. Il est également possible de dupliquer les objets en utilisant les raccourcis *ctrl-c* et *ctrl-v*.
    d. Le bouton *Export As Text* permet d'enregistrer la scène dans un fichier texte.
4. Modifiez l'environnement de façon à reproduire celui apparaissant dans le vidéo *Projet1-exemple/pr2-scene-model.avi*, ou créez un environnement de complexité similaire.
5. Dans l'onglet *Context* de la fenêtre *Motion Planning*, cliquez sur *Publish Current Scene* après chaque modification de l'environnement. Ceci permet de notifier l'algorithme de planification afin qu'il tienne compte de ces changements lors des calculs de trajectoires.
6. Testez des requêtes de planification dans votre environnement et assurez-vous que les chemins calculés évitent les obstacles.
7. Exportez l'environnement sous format texte sous le nom *pr2.scene*.

1 Planification de trajectoires dans MoveIt – Étape 1

1.1 Robonaut

La première tâche pratique du devoir consiste à exécuter des requêtes de planification pour le robot Robonaut (2) dans un environnement parsemé d'obstacles. Robonaut est un robot mobile fabriqué par la NASA et destiné à assister les astronautes dans la station spatiale internationale[5].

Figure 2: Robonaut

1.1.1 Format URDF

URDF est un format XML permettant de représenter des modèles de robots. Les chaînes cinématiques y sont modélisées à l'aide entre autres de liaisons (*joints*) et de pièces (*links*). Un exemple est représenté sur la Figure 3.

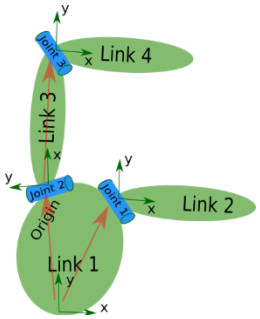

Figure 3: Exemple de chaîne cinématique[6]

1.1.2 Tâches à réaliser

1. Contrairement aux exercices préparatoires, aucun package vous permettant d'utiliser directement le robot n'est fourni. Vous devez utiliser l'outil *MoveIt! Setup Assistant* pour créer ce package de configuration pour le robot Robonaut.

   Pour créer le package de configuration, il est nécessaire d'avoir un fichier de type *urdf* pour le robot Robonaut. Le fichier se trouve à l'endroit suivant :

   ~/catkin_ws/src/r2_description/urdf/r2c5.urdf.

   À l'aide du *MoveIt! Setup Assistant*, générez un package nommé *r2_moveit_generated* dans le répertoire *src/* de l'espace de travail catkin.

   Le tutoriel suivant peut servir de guide pour cette tâche :
   http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/setup_assistant/setup_assistant_tutorial.html

   (Vous pouvez ignorer la section *Getting Started* et les étapes 8 à 10 inclusivement.)

[5] http://www.nasa.gov/mission_pages/station/multimedia/robonaut_photos.html

[6] http://wiki.ros.org/urdf/Tutorials/Create%20your%20own%20urdf%20file

Bien que ce tutoriel s'applique au robot *Panda* de la compagnie *Franka Emika*, la démarche est sensiblement la même pour le robot Robonaut. Il y a quelques différences à prendre en considérations :

- Lors de la création des *Planning Groups* pour les bras, utilisez pour le bras droit et pour le bras gauche une chaine cinématique *Chain Group* (bouton *Add Kin. Chain*).
- L'ajout des Planning Groups pour les mains du robot est optionnel. De ce fait, l'ajout des *Ends Effectors* est aussi optionnel.

2. Visualisez le robot dans RViz en utilisant le package *r2_moveit_generated* généré : *roslaunch r2_moveit_generated demo.launch*

   Il est possible qu'aucun marqueur interactif ne soit affiché.

3. Modifiez l'environnement (voir l'exercice préparatoire #2) de façon à reproduire celui apparaissant dans le vidéo *Projet1-exemple/r2-scene-model.avi*, ou créez un environnement similaire.

4. Ajoutez des objets sur la table qui se situe devant le robot (des bouteilles par exemple). Effectuez des requêtes de planification impliquant l'évitement de collision entre les bras du robot et ces objets.

1.1.3 Critères d'évaluation

Les points seront distribués de la façon suivante :

- Création du package *r2_moveit_generated* : **3 points**
- Création du fichier *scene* : **2 points**
- Exécution de la solution : **5 points**
- Vidéo de démonstration : **5 points**

L'évaluation de chacun de ces points se fera selon les critères suivants :

- Solution fonctionnelle;
- Démonstration réussie.

1.1.4 Livrable

L'archive ***d2-etape1.zip*** doit inclure un dossier nommé ***Robonaut*** ayant le contenu suivant:

1. Package ROS ***r2_moveit_generated*** où les opérations suivantes sont possibles:
   - Sélectionner un Planning Group nommé *whole_body* contenant au moins les sous-groupes suivants : left_arm, right_arm. Il n'est pas nécessaire de faire apparaître des marqueurs interactifs pour ces groupes.
   - Résoudre des requêtes de planification reliant un état initial à un état final.
2. Environnement ***r2.scene*** pouvant être importé à partir de RViz. L'environnement doit être similaire à celui apparaissant dans le vidéo *Projet1-exemple/r2-scene-model.avi* et inclure les objets spécifiés dans la tâche 4 décrite précédemment.
3. Vidéo ***tache1.avi*** illustrant la résolution d'une requête de planification de trajectoire impliquant des changements dans les groupes suivants : *left_arm*, *right_arm*, au sein de l'environnement *r2.scene.*
4. Un fichier ***noms.txt*** comportant les prénoms, noms, matricules et courriels des membres de votre équipe.

1.2 Comparaison de différents algorithmes de la librairie OMPL

L'objectif de cet exercice est de comparer les performances de quelques algorithmes de la librairie OMPL. Pour ce faire, vous devrez compléter le développement du script Python *OMPL_algorithm_benchmark_2019.py*. Ce dernier est développé en utilisant *rospy*[7] et *moveit_commander*[8]. Ces librairies Python nous permettent d'interagir directement avec ROS et MoveIt!

Les algorithmes que vous devrez comparer sont les suivants :

- RRTkConfigDefault;
- RRTConnectkConfigDefault;
- PRMkConfigDefault;
- ESTkConfigDefault.

Tous ces algorithmes se retrouvent dans RViz, plus précisément dans l'onglet *Context* de la fenêtre *Motion Planning*. Vous devrez comparer ces algorithmes selon les quatre métriques suivantes :

- Le temps de planification;
- La longueur du trajet;
- La durée du trajet;
- Le nombre d'échecs (le nombre de fois où le planificateur a été incapable de trouver un chemin).

Pour cet exercice, vous utiliserez un nouveau package pour le robot PR2 : *pr2_moveit_base2*. Quelques corrections ont été apportées à la base du robot. Vous pourrez le récupérer dans l'archive "nouveau_package_pr2.zip" disponible dans Moodle.

1.2.1 Installation du nouveau package

1. Copiez le dossier *pr2_moveit_base2* dans l'espace de travail *catkin (~/catkin_ws/src)*.
2. Copiez le fichier *base.urdf.xacro* à l'endroit suivant :
   ~/catkin_ws/src/pr2_common/pr2_description/urdf/base_v0
   (Avant de remplacer le fichier déjà existant, faites une sauvegarde du fichier original. Si vous voulez utiliser à nouveau le package *pr2_moveit_base*, vous devrez aussi utiliser le fichier *base.urdf.xacro* original.)
3. Pour lancer le simulateur avec le nouveau package, utilisez la commande suivante :
   roslaunch pr2_moveit_base2 demo.launch

1.2.2 Mise en place de l'environnement de développement

Pour le développement, il est fortement recommandé d'utiliser l'IDE PyCharm. Vous pourrez facilement déboguer et exécuter directement votre script. Les étapes suivantes vous permettront de mettre en place votre environnement de développement.

1. Téléchargez PyCharm. Pour ce faire, allez à cette adresse : https://www.jetbrains.com/pycharm/download/#section=linux. Choisissez la version *Community*.
2. Pour lancer PyCharm, accédez au dossier */pycharm-community-2017.3.3/bin/* avec un terminal. Ensuite, lancez la commande : *./pycharm.sh*

[7] http://wiki.ros.org/rospy
[8] http://wiki.ros.org/moveit_commander

3. À l'aide de l'option *Open*, ouvrez le fichier *OMPL_algorithm_benchmark.py*.
4. Pour être capable d'exécuter le script, vous devez configurer un interpréteur Python. Après avoir ouvert le fichier, vous devriez voir l'option *Configure Python Interpreter* dans le haut de la fenêtre.
    a. Cliquez sur ce bouton.
    b. Dans la nouvelle fenêtre, localisez l'option *Project Interpreter*. Cliquez sur le bouton qui se situe à la droite de cette option (icône en forme d'engrenage).
    c. Cliquez sur *Add*.
    d. Dans la nouvelle fenêtre, dans la section de gauche, choisissez *System Interpreter*. Finalement, pour l'interpréteur, choisissez */usr/bin/python2.7*. **Important : Il est nécessaire que l'interpréteur soit Python 2.7.**
    e. Cliquez sur *OK*, puis *Apply*.

    PyCharm effectuera le téléchargement de diverses mises à jour. En effet, on peut lire « *process running…* » dans le bas de l'application.
5. Avant de pouvoir exécuter le script, il est nécessaire d'installer certains packages dont dépend ce script.
    a. Installer *pip*.
    Dans un terminal, exécutez la commande suivante : *sudo apt install python-pip*
    b. Installer le module *statistics*[9].
    Dans un terminal, exécutez la commande suivante : *pip install statistics*
    Ce module contient des fonctions très utiles pour calculer des moyennes, des écarts types, etc.
6. Pour finir, **pour que le script soit fonctionnel, le package *pr2_moveit_base2* doit être en exécution**. Par conséquent, dans un terminal, lancez la commande :
roslaunch pr2_moveit_base2 demo.launch

À ce point-ci, votre environnement de développement est prêt. Pour exécuter ou pour déboguer votre script, cliquez sur l'option *Run* de la barre de menu puis sur *Run* ou *Debug*. Essayez d'exécuter le script *OMPL_algorithm_benchmark_2019.py*; vous devriez voir dans RViz l'environnement se transformer autour du PR2.

1.2.3 Tâches à réaliser

Le travail consiste à compléter le script *OMPL_algorithm_benchmark_2019.py*. L'objectif de ce script est de calculer en boucle de nombreux plans avec chacun des quatre algorithmes d'OMPL pour différentes configurations. Une configuration correspond à l'environnement du robot, qui est généré de façon aléatoire (Figure 4). Les coordonnées du point de départ et du point d'arrivée du robot PR2 seront fixées de cette façon :

- Point de départ (point A) : **(-5.0; -5.0)**
- Point d'arrivée (point B) : **(5.0; 5.0)**

[9] https://pypi.python.org/pypi/statistics/1.0.3.5

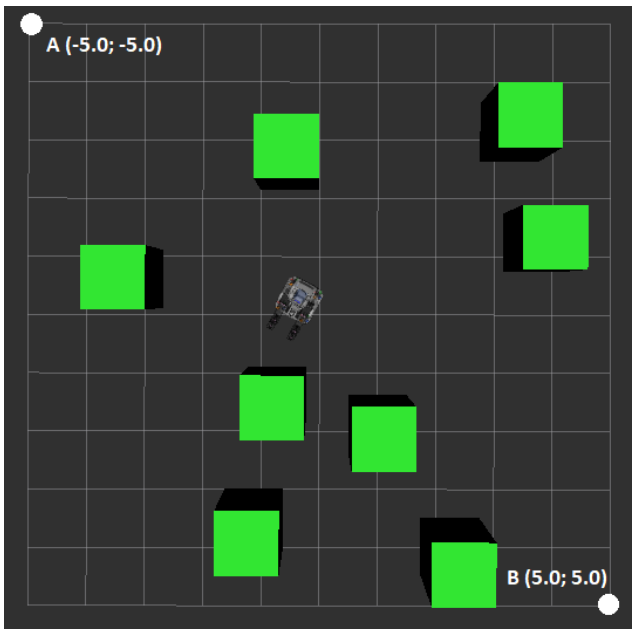

Figure 4: Exemple d'environnement aléatoire

À chaque itération, on enregistre les métriques suivantes :

- Si le planificateur a réussi ou non à trouver un plan;
- Le temps de planification;
- La longueur du trajet;
- La durée du trajet.

À la toute fin, on compile les données, puis on les présente sous forme graphique. N'oubliez pas que pour évaluer adéquatement les algorithmes, il faut accumuler une quantité suffisante de données (par conséquent, évaluer les algorithmes avec une seule configuration n'est pas valide).

Pour bien comprendre comment réaliser cette tâche, commencez par lire attentivement le code du script et les commentaires déjà présents. Les sections de codes à compléter sont clairement identifiées avec des *TODOs*. Il y a 5 fonctions qui doivent être complétées :

Benchmark::run()

Cette fonction est la plus importante du script. Elle devra contenir toute la logique nécessaire pour faire la comparaison des algorithmes. Cette logique devrait correspondre à celle décrite ci-dessous.

1. Générez une configuration, ce qui veut dire générer un environnement aléatoire (déjà fait pour vous);
2. Sélectionnez un des quatre algorithmes d'OMPL;
3. Configurez l'objet de type *MoveGroupCommander* pour utiliser cet algorithme;
4. Configurez les paramètres de l'algorithme à l'aide du service *set_planner_params*;
5. Calculez un plan;
6. Si le plan est valide, enregistrer les métriques de temps, durée et longueur. Si le plan est invalide, enregistrer cette information (n'enregistrez pas les métriques de temps, durée et longueur lorsque le plan est invalide ou si l'algorithme n'a pas réussi à trouver de plan).

7. Retournez à l'étape 5 pour calculer à nouveau un plan. Si le nombre de plans calculé est égal au nombre de plan que doit calculer chaque algorithme pour chaque configuration, passez à l'étape suivante;
8. Retourner à l'étape 2 pour sélectionner le prochain algorithme. Si tous les algorithmes ont déjà été sélectionnés pour cette configuration, passez à l'étape suivante;
9. Supprimez l'environnement;
10. Retourner à l'étape 1 pour générer une nouvelle configuration. Si le nombre de configurations générées est égal au nombre maximal de configurations, l'algorithme de comparaison des algorithmes est alors complété.

Benchmark::displayStatistic()

Cette fonction devra afficher le résultat de la comparaison des algorithmes. Vous devez, au minimum, présenter les informations suivantes **pour chaque algorithme et pour chaque métrique** :

- La moyenne;
- La médiane;
- L'écart type;
- La valeur minimale et la valeur maximale.

Exception: pour la métrique "nombre d'échecs", n'affichez que le nombre d'échecs total pour chaque algorithme.

L'information doit être présentée sous forme de graphique. Pour une métrique, on doit être en mesure de comparer aisément les performances des quatre algorithmes d'OMPL.

Benchmark::__computePathLength(plan)

Cette fonction devra calculer et retourner la longueur d'un plan (la longueur du trajet). Une trajectoire est constituée de plusieurs points espacés dans le temps. Chaque point correspond à une position de tous les joints du robot. Vous devez utiliser la distance euclidienne pour calculer la longueur totale du plan.

Benchmark::__callbackGetPlanningTime(data)

Cette fonction est utilisée pour recueillir le temps de planification. Elle devra donc extraire le temps de planification de l'objet *data* et l'ajouter à la variable membre *planningTimeList*. L'objet *data* contient un champ *error_code* qui vous indique si la planification a réussi ou non. Comme il a été mentionné plus haut, dans le cas où la planification a échoué, il ne faut pas enregistrer les métriques de temps, de durée et de longueur.

La fonction *main* est le point d'entrée du script. Vous devez créer des objets de type *Planner* pour chacun des algorithmes et les ajouter à l'objet de type *BenchmarkSettings*. Vous êtes libre de modifier les paramètres de l'algorithme de comparaison (nombre de plans, nombre de configurations, etc.).

1.2.4 Guide

Pour vous aider dans votre travail, cette section présente diverses ressources et explications.

1.2.4.1 Ressources

Documentation de *rospy* : https://docs.ros.org/en/diamondback/api/rospy/html/module-tree.html

Survol de *rospy* : http://wiki.ros.org/rospy/Overview

Documentation de *moveit_commander* :
http://docs.ros.org/kinetic/api/moveit_commander/html/annotated.html

Tutoriel sur l'utilisation de *moveit_commander* :
http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/move_group_python_interface/move_group_python_interface_tutorial.html

1.2.4.2 Comment récupérer les métriques

Cette section décrit de quelle façon nous pouvons récupérer les métriques suite à un calcul de plan.

Temps de planification

Lorsqu'un plan est calculé, un message de type *MoveGroupActionResult* est publié sur le *topic*[10] */move_group/result*. Ce message inclut le temps de planification. La fonction __callbackGetPlanningTime(data), qui devra être complétée, est utilisée pour récupérer cette métrique. La réception des messages se fait sur un thread dédié.

Longueur du trajet

Le calcul d'un plan se fait à l'aide de la fonction *plan()*[11] de *moveit_commander*. Cette fonction retourne un objet de type *RobotTrajectory*[12]. Dans le cas où l'algorithme n'a pas été en mesure de trouver un plan, cet objet sera vide. Autrement, ce dernier contiendra une trajectoire constituée de multiples points. La fonction __computePathLength(plan) devra être développée pour calculer la longueur totale du plan.

Durée du trajet

Toujours avec l'objet de type *RobotTrajectory*, la durée totale du trajet peut être récupérée en consultant la variable *time_from_start*[13] du dernier point de la trajectoire.

1.2.4.3 Création des graphiques

Vous pouvez créer facilement des graphiques à l'aide de la librairie Python matplotlib, qui est déjà installée sur la machine virtuelle. Vous trouverez des exemples d'utilisation ici :
https://matplotlib.org/stable/gallery/index.html

Vous êtes libres de choisir une autre librairie si matplotlib ne vous convient pas.

1.2.4.4 Configuration des algorithmes de planification

La sélection de l'algorithme de planification se fait avec la fonction *set_planner_id()*[14].

La configuration de l'algorithme se fait avec la fonction *set_planner_params()*, qui est un *service proxy* déjà déclaré dans le constructeur de la classe *Benchmark*. Les *services proxies* sont nécessaires pour envoyer des requêtes à des services ROS[15]. Pour plus de détails sur les *services proxies*, consultez cette page : http://wiki.ros.org/rospy/Overview/Services

---

[10] http://wiki.ros.org/Topics

[11] http://docs.ros.org/kinetic/api/moveit_commander/html/classmoveit__commander_1_1move__group_1_1MoveGroupCommander.html#a79a475263bffd96978c87488a2bf7c98

[12] http://docs.ros.org/kinetic/api/moveit_msgs/html/msg/RobotTrajectory.html

[13] http://docs.ros.org/kinetic/api/trajectory_msgs/html/msg/JointTrajectoryPoint.html

[14] http://docs.ros.org/kinetic/api/moveit_commander/html/classmoveit__commander_1_1move__group_1_1MoveGroupCommander.html#a362bbe2b8418e1e64bc04b5162e75e11

[15] http://wiki.ros.org/Services

La fonction *set_planner_params()* nécessite quelques paramètres. Les paramètres correspondent au contenu du message de type *SetPlannerParams* :
http://docs.ros.org/kinetic/api/moveit_msgs/html/srv/SetPlannerParams.html

Les paramètres des algorithmes de planification sont visibles dans RViz, plus précisément dans l'onglet *Context* de la fenêtre *Motion Planning*.

1.2.4.5 Dernières remarques

N'hésitez pas à séparer votre code en plusieurs fonctions. De plus, vous êtes libres de retravailler le code déjà présent dans le script.

1.2.5 Critères d'évaluation

Les points seront distribués de la façon suivante :

- Exécution de la solution : **4 points**
- Rapport : **7 points**
- Vidéo de démonstration: **4 points**

L'évaluation de chacun de ces points se fera selon les critères suivants :

- Exécution sans erreur et fonctionnelle;
    - Script remis complété et fonctionnel
    - Exécution du script
- Clarté de la documentation;
- Explication des résultats obtenus;
- Vidéo de démonstration bien structuré et montrant clairement les étapes et que la solution est fonctionnelle.

1.2.6 Livrable

L'archive ***d2-etape1.zip*** doit inclure un dossier nommé ***OMPL*** ayant le contenu suivant:

1. Le script *OMPL_algorithm_benchmark_2019.py*, complété et fonctionnel. Le script doit produire en sortie des graphiques présentant les performances de chacun des algorithmes pour chacune des métriques.
2. Un court rapport (maximum 5 pages), nommé *Rapport_VosNoms.pdf*, présentant vos résultats. Il devrait contenir tous vos graphiques. Tentez d'expliquer les différences ou les similitudes obtenues entre les algorithmes, et ce, pour chacune des métriques.

2 Étape 2 : Implémentation d'actions primitives du robot

Dans la deuxième partie du devoir, le robot PR2 devra effectuer des manipulations dans un environnement complexe. Cet environnement est formé de 7 pièces. De plus, certaines de ces pièces contiennent un ballon.

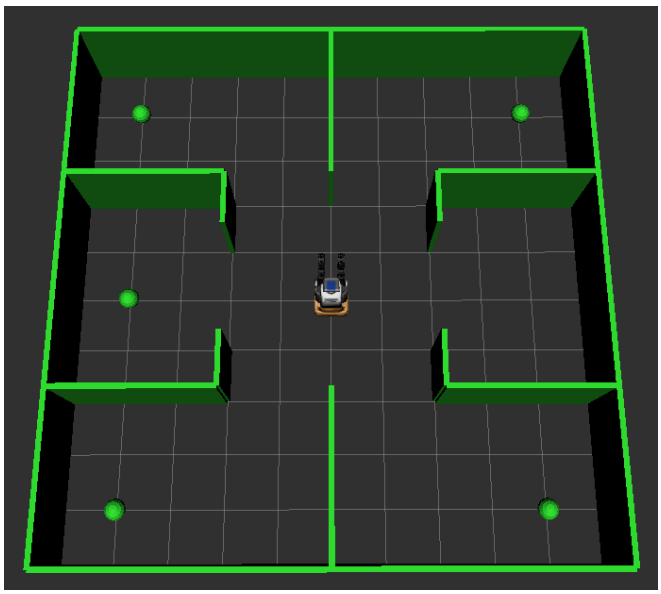

Figure 5: Environnement dans lequel le robot devra évoluer (état initial)

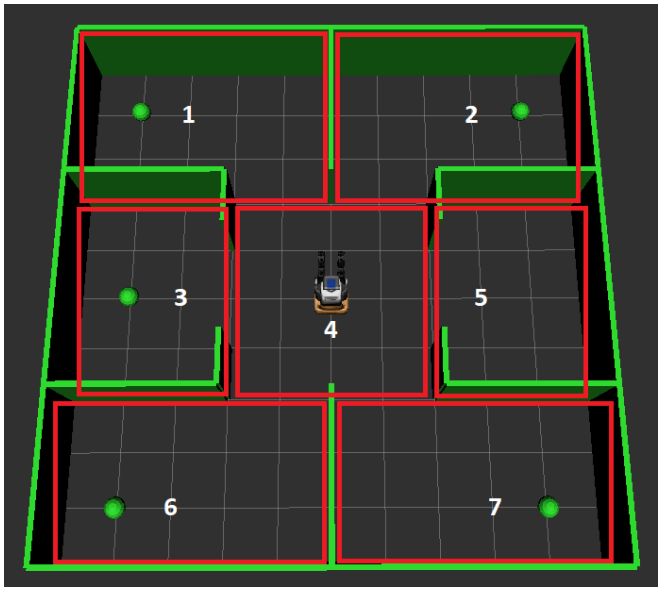

Figure 6: Les 7 pièces de l'environnement

2.1 Tâches à réaliser

L'objectif de cette partie est d'apprendre à programmer des actions primitives dans un contexte de planification de trajectoires. En guise d'exemple, une implémentation des actions primitives `Move(Room)`, `PickBall(Ball)` et `DropBall(Ball)` est déjà disponible dans le fichier `primitives_solution.py`.

- **Move(Room)** // Le robot se déplace à la pièce indiquée
  Précondition : Aucune
- **PickBall(Ball)** // Le robot prend le ballon
  Préconditions : Le robot est dans la même pièce que le ballon. Le robot n'a aucun ballon en sa possession.
- **DropBall(Ball)** // Le robot pose le ballon
  Précondition : Le robot possède le ballon.

Vous pouvez utiliser ces implémentations ou développer vos propres implémentations. Vous devrez par contre implémenter une nouvelle action primitive :

- **Kick(Ball)** // Le robot frappe le ballon avec son « pied »
  Précondition : Le robot est dans la même pièce que le ballon. Le robot peut frapper un ballon s'il a ce ballon en sa possession ou si sa plateforme est suffisamment proche du ballon.

  Effets : le ballon se déplace de 2 cases dans direction perpendiculaire à l'orientation de la plateforme. Le ballon ne peut pas passer à travers les murs et ne rebondit pas.

Une fois les actions primitives opérationnelles, vous devrez programmer et exécuter une suite d'actions qui permet d'atteindre l'état final suivant (Figure 7):

- Tous les ballons sont dans la zone du centre;
- Le robot PR2 est dans la pièce #5 (pièce de droite).

Vous devez essayer de trouver le plan le court possible en termes d'action et indiquer le coût de ce plan dans votre rapport. Est-ce plus efficace de prendre les ballons et de les déplacer, de les frapper pour les amener au centre, ou de combiner ces approches ?

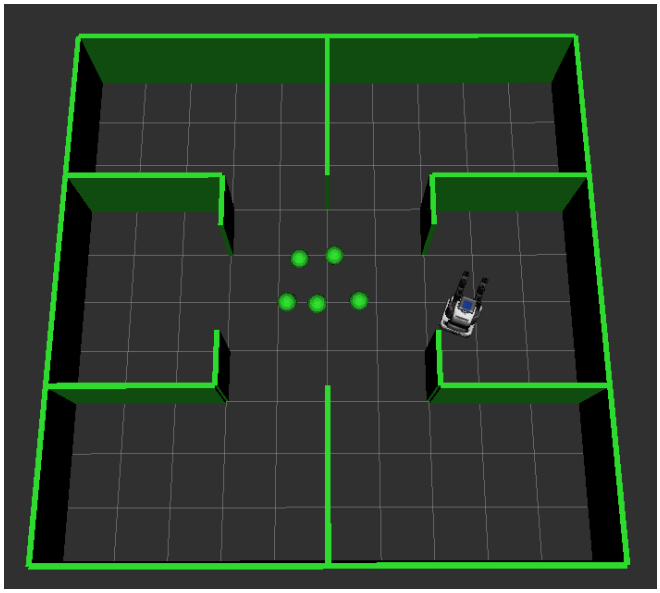

Figure 7: État final

Comme pour la question 1.2, vous utiliserez les librairies Python *moveit_commander* et *rospy*. Vous devrez compléter le script Python nommé *primitives.py* ou *primitives_solution.py* (selon que ayez décidé de réimplanter ou pas les trois premières actions primitivives. Le script, dans son état actuel, ne fait qu'initialiser certains objets et générer l'environnement.

2.2 Mise en place de l'environnement de développement

L'environnement de développement est le même que pour la question précédente. Vous utiliserez encore une fois le package pour le robot PR2 *pr2_moveit_base2*. Rappel : utilisez la commande suivante pour lancer le simulateur avec le nouveau package :
roslaunch pr2_moveit_base2 demo.launch

2.3 Guide

Pour vous aider dans votre travail, cette section présente quelques explications.

- Pour simplifier le travail, le robot PR2 n'aura pas à utiliser ses bras pour prendre possession des balles. Plutôt, il est possible d'attacher (et de détacher) directement un objet à un *planning group* du robot. Lorsqu'un objet est attaché, l'algorithme de planification prend en compte cette nouvelle masse. Dans le cas présent, puisque les déplacements du robot se font à l'aide du *planning group* nommé *base*, les ballons seront attachés à la base du robot.

  Important : il n'y a pas de distance minimale pour attacher un objet au robot. Cela signifie que le robot peut être dans une pièce et un ballon dans une autre. Après avoir attaché le ballon, les mouvements de la base du robot entraineront un mouvement au niveau du ballon, comme si une tige invisible reliait la base et le ballon. À ce niveau, une contrainte devra être respecté (voir la section Livrable).

- Pour obtenir la position du robot et son orientation dans l'espace, regardez la valeur des joints suivants :

  - *base_footprint_joint_y*
  - *base_footprint_joint_x*
  - *base_footprint_joint_rev*

  Explication : En temps normal, MoveIt! et le simulateur RViz ne permettent pas à la base mobile du robot de se déplacer dans l'environnement. Même s'il est possible de calculer un plan, son exécution est impossible. Pour contourner cette limitation, nous avons remplacé le joint de la base mobile par les joints suivants :

  - Un joint linéaire selon l'axe des y;
  - Un joint linéaire selon l'axe des x;
  - Un joint en rotation.

  Ces joints sont invisibles et se retrouvent sous le robot. Avec cette configuration, il est alors possible de simuler les déplacements du robot. C'est pourquoi, lorsque l'on consulte la valeur des joints *base_footprint_joint_y* et *base_footprint_joint_x*, il est possible de connaître la position du robot.

  Notez que les deux packages, *pr2_moveit_base* et *pr2_moveit_base2* contiennent cette modification. *pr2_moveit_base2* inclut une correction supplémentaire.

- Les fonctions suivantes de *moveit_commander* peuvent vous être utiles :

  - get_current_joint_values()
  - set_joint_value_target()
  - set_goal_tolerance()
  - attach_object()
  - detach_object()
  - get_objects()

  Consultez la documentation de *moveit_commander* pour plus de détails.

2.4 Critères d'évaluation

Les points seront distribués de la façon suivante :

- Implémentation des primitives : **8 points**
- Exécution de la solution : **8 points**
- Vidéo de démonstration: **8 points**
- Utilisation des bras du robot : **6 points**

L'évaluation de chacun de ces points se fera selon les critères suivants :

- Implémentation correcte;
- Exécution sans erreur et fonctionnelle;
- Respect des contraintes;
- Vidéo de démonstration bien structuré et montrant clairement les étapes et que la solution est fonctionnelle.

2.5 Livrable

L'archive ***d2-etape2.zip*** doit inclure le contenu suivant:

1. Le script *primitives.py*, complété et fonctionnel. L'exécution de ce script doit mettre en mouvement le robot PR2 (package *pr2_moveit_base2*). Ce dernier doit effectuer les manipulations nécessaires pour arriver à l'état final présenté à la Figure 7. Les contraintes suivantes doivent être respectées :
    a. Tous les ballons se retrouvent dans la salle #4 (voir Figure 6);
    b. Le robot termine son parcours dans la salle #5 (voir Figure 6);
    c. Lorsque le robot effectue l'action **PickBall**, ce dernier doit se positionner face au ballon. Le robot doit aussi être le plus près possible du ballon (Fig. 8).
    d. **Défi** : Au lieu que l'action PickBall attache le ballon à la base du robot, faites-en sorte que le robot prenne possession du ballon en utilisant ses bras (il doit soulever le ballon en utilisant un ou deux bras). Dans un même ordre d'idée, le robot devrait déposer le ballon au sol lorsque l'action DropBall est appelée. Ceci n'est pas évident à faire. C'est pour cela que c'est un défi. Vous ne recevrez pas d'aide. Ce défi compte pour seulement 6 points sur 30 points. Essayez, mais n'y mettez pas trop de temps si ça s'avère être trop difficile (pas plus de 10% du temps pour cette question).
    e. Assurez-vous de respecter les préconditions de chacune des actions (section 2.1).
2. Une vidéo ***tache2.avi*** illustrant le script en exécution (on doit voir le PR2 résoudre le problème).

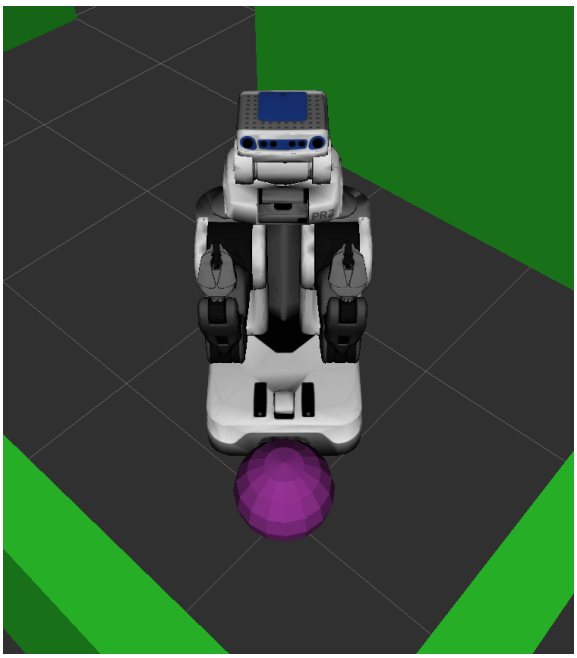

Figure 8: Position du robot par rapport au ballon pour l'action PickBall

3 Étape 3 : Planification et exécution de plan avec ROSPlan

Grâce à la deuxième partie de ce travail, vous disposez des actions primitives Move, Pick, Drop et Kick. Vous savez aussi établir manuellement une liste d'actions (un plan) qui, lorsque exécutée, permet d'atteindre un but précis. Pour la dernière partie de ce devoir, vous utiliserez ROSPlan pour calculer et exécuter automatiquement ce plan. Vous devrez tout d'abord modéliser le domaine et le problème avec PDDL. Ensuite, vous devrez modifier votre script *primitives.py* pour que celui-ci communique avec ROSPlan afin de générer et exécuter un plan.

3.1 Tâches à réaliser

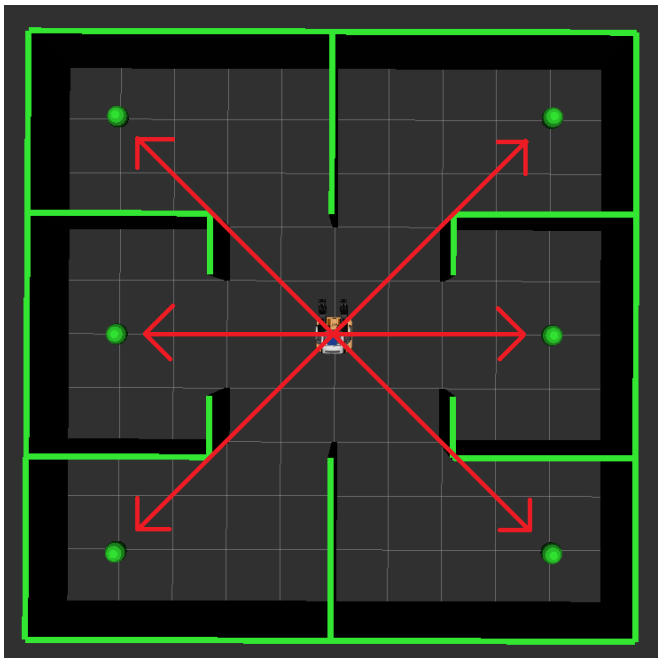

Figure 9: Problème à résoudre

1. Modélisez le domaine, les types (**nécessaire pour un bon fonctionnement de ROSPlan**), les prédicats et les actions Move, Pick, Drop et Kick, ainsi que le problème à résoudre (les objets, l'état initial et l'état final) selon la syntaxe PDDL 2.1. Vous devriez donc avoir deux fichiers : *pr2-domain.pddl* et *pr2-problem.pddl*.

   Le problème à résoudre est celui présenté à la Figure 9 : le robot PR2 doit échanger chaque ballon avec celui qui se situe dans la pièce opposée. Après avoir effectué les échanges, le robot doit se placer dans la pièce du centre (pièce #4 selon la Figure 6). La contrainte suivante doit être respectée à tout moment : **il ne peut y avoir qu'un seul ballon libre dans chaque pièce de l'environnement (un ballon libre n'est pas contrôlé par le PR2)**.

2. Préparez un *launch file* nommé *rosplan.launch*. Ce fichier, avec la commande roslaunch, s'occupe de démarrer tous les nœuds nécessaires au bon fonctionnement de ROSPlan.

3. Copiez le code développé à la partie 2 (*primitives.py*) dans un nouveau fichier, nommé *planification.py*.

4. Modifiez la fonction *generateEnvironment()* pour ajouter un ballon dans la pièce de droite.

5. Modifiez vos actions primitives afin que les **effets** des actions PDDL correspondantes soient appliqué à la base de connaissance (*Knowledge Base*) de ROSPlan. De plus, **après l'exécution**

de chaque action, vous devez afficher dans la console l'état actuel du monde, c'est-à-dire afficher la liste des faits (aussi appelée *Propositions* dans ROSPlan) qui sont vrais.

```
Executing action (pickball room1 ball1)
Execution succeeded!
New state:
(ball-at ball2 room2)
(ball-at ball3 room3)
(ball-at ball4 room4)
(ball-at ball5 room5)
(ball-at ball6 room6)
(carry ball1)
(is-empty room0)
(is-empty room1)
(robot-at room1)
-----
```

Figure 10: Exemple de message affiché dans la console

6. Modifiez le reste du script afin que ROSPlan soit utilisé pour générer et exécuter un plan (aussi appelé *plan dispatch* dans ROSPlan). **Le plan qui a été généré doit être affiché dans la console avant son exécution**. De plus, **vous devez afficher un message dans la console indiquant si l'exécution du plan a réussi ou bien si l'exécution a échoué** (peut survenir si l'exécution d'une action a échoué).

3.2 Mise en place de l'environnement de développement

L'environnement de développement est le même que pour la question 1.2 et l'étape 2. Vous utiliserez encore une fois le nouveau package pour le robot PR2 : *pr2_moveit_base2*. Vous devez cependant mettre à jour ROSPlan. Dans un terminal, exécutez les commandes suivantes :

1. sudo apt-get install flex ros-kinetic-move-base-msgs ros-kinetic-mongodb-store ros-kinetic-tf2-bullet freeglut3-dev python-catkin-tools bison libbdd-dev
2. cd ~/catkin_ws/src/ROSPlan
3. git fetch
4. git pull
5. cd ~/catkin_ws
6. catkin_make

3.3 Guide

Pour vous aider dans votre travail, cette section présente quelques explications.

- Pour ce travail, vous devrez utiliser vos connaissances de *rospy*, développées dans la partie 1 de ce devoir. Plus particulièrement, vous devrez utiliser *rospy* pour communiquer avec les services ROS, publier des messages sur certains *topics* et souscrire à d'autres. Le tout est nécessaire pour être en mesure de communiquer avec ROSPlan.
- Il est fortement recommandé de consulter le tutoriel et la documentation sur le site web de ROSPlan : http://kcl-planning.github.io/ROSPlan/documentation/ . Le tutoriel explique comment démarrer et utiliser ROSPlan.
- Pour chaque action primitive, vous devrez appliquer les effets des actions PDDL correspondantes à la base de connaissance (*Knowledge Base*). Cela est nécessaire car le service d'exécution de plan (*plan dispatch*) valide que les préconditions d'une action sont respectées

avant de l'exécuter. Pour faire cette validation, le service d'exécution interroge la base de connaissance pour obtenir l'état actuel.

- Deux *topics* sont particulièrement importants pour l'exécution d'un plan : *action_dispatch* et *action_feedback*. Les actions qui doivent être exécutées sont publiées sur le topic *action_dispatch* par le service d'exécution. De votre côté, vous devez donner un retour d'information au service d'exécution en publiant des messages de type *ActionFeedback*[16] sur le topic *action_feedback*. Les statuts possibles sont :
    - *"action enabled"* : Indique que votre programme a bien reçu l'action à exécuter.
    - *"action achieved"* : Indique que l'action a été exécutée correctement.
    - *"action failed"* : Indique que l'action n'a pas été exécutée correctement.

  Pour chaque action du plan, le service d'exécution s'attend à recevoir un message *ActionFeedback* ayant comme statut *"action enabled"*, puis un autre message du même type indiquant si l'exécution de l'action a réussie ou échouée.

- Lors de l'évaluation, les fichiers *pr2-domain.pddl*, *pr2-problem.pddl*, *planification.py* et *rosplan.launch* seront copiés directement dans le dossier *~/catkin_ws*. Assurez-vous que votre solution fonctionne lorsque les fichiers sont à cet endroit (ajuster en conséquence les *paths* dans le fichier *rosplan.launch*). Pour tester votre solution, *rosplan.launch* ainsi que *demo.launch* (du package *pr2_moveit_base2*) seront lancés chacun dans un terminal. Ensuite, vous exécuterez votre programme *planification.py*.

3.4 Critères d'évaluation

Les points seront distribués de la façon suivante :

- Implémentation des fichiers PDDL : **10 points**
- Exécution de la solution : **20 points**
- Vidéo de démonstration : **10 points**

L'évaluation de chacun de ces points se fera selon les critères suivants :

- Implémentation correcte;
- Exécution sans erreur et fonctionnelle;
- Vidéo de démonstration bien structuré et montrant clairement les étapes et que la solution est fonctionnelle.

3.5 Livrable

L'archive ***d2-etape3.zip*** doit inclure le contenu suivant:

1. Le script *planification.py*, complété et fonctionnel.
2. Le fichier *pr2-domain.pddl* et le fichier *pr2-problem.pddl*.
3. Le fichier *rosplan.launch*.
4. Un fichier *plan.pddl*, contenant le plan trouvé par le planificateur (ROSPlan enregistre automatiquement chaque plan calculé dans un fichier. Il suffit de copier ce fichier et de le déposer dans l'archive).
5. Une vidéo ***tache3.avi*** illustrant le script en exécution (on doit voir le PR2 résoudre le problème).

[16] https://github.com/KCL-Planning/ROSPlan/blob/master/rosplan_dispatch_msgs/msg/ActionFeedback.msg

Grille d'évaluation détaillée

	Total par étapes	Détails
Étape 0. Exercices préparatoires	Non évalué	
Étape 1. Planification de trajectoires dans MoveIt	/30	
1.1 Robonaut • Création du package *r2_moveit_generated* : **3 points** • Création du fichier *scene* : **2 points** • Exécution de la solution : **5 points** • Vidéo de démonstration : **5 points**		/15
1.2 Comparaison de différents algorithmes de la librairie OMPL • Exécution de la solution : **4 points** • Rapport : **7 points** • Vidéo de démonstration: **4 points**		/15
Étape 2. Implémentation d'actions primitives pour un robot	/30	
• Implémentation des primitives : **8 points** • Exécution de la solution : **8 points** • Vidéo de démonstration: **8 points** • Utilisation des bras du robot : **6 points**		
Étape 3. Planification et exécution de plan avec ROSPlan	/40	
• Implémentation des fichiers PDDL : **10 points** • Exécution de la solution : **20 points** • Vidéo de démonstration : **10 points**		
Total	**/100**	

Problèmes techniques.

- Si votre équipe rencontre un problème avec l'étape 3 du devoir, au niveau de la commande pour mettre à jour l'environnement de travail : sudo apt-get install flex ros-kinetic-move-base-msgs ros-kinetic-mongodb-store roskinetic-tf2-bullet freeglut3-dev python-catkin-tools bison libbdd-dev
    - Solution : retirer "ros-kinetic-mongodb-store" règle le problème
- À la partie 3 du devoir de session ROSPLAN, si votre équipe rencontre un problème dans la définition du domaine et du problème en pddl.
    - Solution. Le planificateur ne prend pas de negative-preconditions dans les préconditions des actions et dans les buts. Il les accepte par contre dans les effets des actions. On ne peut donc pas avoir de (not (variable x y)) dans le but, ni dans les préconditions des actions.