TD: Trajectoires

1 Introduction

Ce TD a pour sujet la planification de trajectoires. S'il est en partie indépendant des précédents, il s'appuie tout de même sur leur contenu, particulièrement pour vous permettre d'appliquer directement les trajectoires générées à un robot dans un simulateur. Par conséquent, quelque soit votre niveau d'avancement, il est requis que vous commenciez votre travail à partir de l'archive disponible sur la page du cours.

Le travail nécessaire pour terminer l'ensemble du TD est conséquent et il est donc naturel que vous n'en fassiez qu'une partie. À ce titre, il est important de noter qu'il n'est pas nécessaire de faire l'ensemble des exercices "dans l'ordre". Vous pouvez donc parfaitement commencer la partie 3 avant d'avoir fini d'explorer la partie 2.

L'objectif étant de vous permettre d'explorer la problématique des générations de trajectoires, quelques outils sont mis à votre disposition dans le code initial pour vous permettre de visualiser rapidement les différentes trajectoires sur lesquels vous travaillez.

Les principales modifications à noter par rapport au code utilisé pour le TD MGI sont les suivantes:

- Ajout d'un fichier `trajectories.py` qui sera le coeur du travail de ce TD. Ce script permet de générer des fichier `csv` décrivant aussi bien la position que la vitesse ou l'accélération pour des trajectoires à partir de fichiers `json` dont le format sera spécifié par la suite. L'utilisation de `python3 trajectories.py -h` permettra d'afficher un court descriptif des options proposées. Il est important de noter que vous pouvez utiliser ce fichier seul, sans lancer le simulateur `Webots`.
- Pour `motor_controller.py`
  - Remplacement de la cible fixe qui était donnée auparavant par une trajectoire qui peut être choisie en modifiant la variable `TRAJECTORY_PATH`
  - Modification du système de log pour afficher la vitesse et l'accélération selon chacun des dimensions

Votre rendu inclura un compte-rendu qui devra au moins contenir les réponses aux questions explicitement mentionnée dans cette feuille de TD.

2 Trajectoires en dimension 1

Dans le fichier `trajectories.py`, la fonction `buildTrajectoryFromDictionary` permet de créer facilement une trajectoire de manière générique à partir d'un fichier `json` chargé (voir `json.load`).

Le format attendus pour les trajectoires à une dimension est le suivant:

`type_name` Le nom de la classe qui sera utilisée pour générer les trajectoires

`knots` Un tableau à deux dimensions contenant les différents points de passage de la trajectoire.

`start` Le moment auquel débute la trajectoire. Permet de facilement décaler l'ensemble de la trajectoire.

`parameters` Permet de spécifier la valeur d'éléments spécifiques à certaines génération de trajectoire. Par exemple, la vitesse maximale et l'accélération maximale pour `TrapezoidalVelocity`.

Le dossier `1d_trajectories` contient quelques exemples de trajectoires que vous pouvez utiliser pour tester votre implémentation.

Les squelettes de nombreuses classes héritant de `Trajectory` sont disponibles dans `trajectories.py`. Chacune d'entre elle correspond à une version mentionnée dans les slides et le même travail vous est demandé:

- Implémenter les méthodes manquantes en pensant bien à vérifier la validité des paramètres fournis.
- Tester différentes trajectoires pour visualiser les résultats que vous obtenez (n'hésitez pas à créer vos propres trajectoires).
- Déterminer quelle est la classe de continuité à laquelle appartient ce type de trajectoire. Écrivez cette information dans votre compte-rendu en la justifiant. N'hésitez pas à inclure des figure pour appuyez vos propos et pensez bien à comparer les résultats obtenus avec ceux que vous attendiez.

Afin de vous échauffer, commencer par la trajectoire `ConstantSpline` qui devrait être facile à implémenter. Pour obtenir une version fonctionnelle, il sera également nécessaire d'implémenter le constructeur de la classe `Spline` ainsi que sa méthode `getVal`. Vous pouvez noter que l'implémentation des différentes classes de `Spline` ne nécessite à priori que de redéfinir la méthode `updatePolynomials`. Effectivement, étant donné que peu importe le degré, il s'agit systématiquement de fonctions polynomiales par morceaux, il est à priori possible de factoriser grandement le code.

Une fois les méthodes implémentés vous pourrez vérifier graphiquement la pertinence des résultats que vous obtenez en exécutant la commande suivante depuis le dossier `controllers/motor_controller`:

```
./trajectories.py 1d_trajectories/constant_example.json > tmp.csv && ../../plot.py tmp.csv
```

N'hésitez pas à commencer par implémenter une méthode qui ne couvre qu'une partie des cas possibles (tout en vous assurant que votre programme plante de manière explicite si l'on est pas dans un de ces cas). Par exemple, pour `TrapezoidalVelocity`, vous pouvez commencer par implémenter une version qui ne fonctionne qu'avec 2 points, la source et la destination.

3 Trajectoires multidimensionnelles

La classe `RobotTrajectory` va permettre de stocker des trajectoires multidimensionnelles et d'accéder à la position et à la vitesse selon les différentes dimensions. Comme pour les trajectoires en une dimension, un format `json` est utilisé pour permettre de les charger à partir d'un fichier[1]. Ce format est le suivant:

`targets` Un tableau définissant les différentes cibles par laquelle passe la trajectoire. Chaque ligne correspond à un point différent et chaque colonne à une dimension. Dans le cas des splines, la première colonne définit le moment auquel ce point est sensé arrivé.

`trajectory_type` Similaire au `type_name` pour les trajectoires en dimension 1. Un seul type sera utilisé pour toutes les dimensions.

`target_space` L'espace dans lequel sont spécifiés les cibles.

`planification_space` L'espace au sein duquel est effectué la génération de trajectoire.

[1] Voir `buildRobotTrajectoryFromDictionary`.

`start` L'instant auquel débute la trajectoire.

`parameters` Similaire aux trajectoires en dimension 1.

L'implémentation de l'ensemble des fonctionnalités de cette classe peut se révéler relativement complexe. Pour cette raison, il est fortement conseillé de commencer par implémenter la gestion d'un sous ensemble des cas possibles, par exemple uniquement la gestion de la position dans le cas où `target_space` et `planification_space` sont identiques. Afin de ne pas générer d'erreur et de pouvoir visualiser la partie implémentée des résultats, le comportement par défaut des fonctions est de renvoyer `None` lorsqu'un cas n'est pas supporté actuellement.

La première étape à effectuer est d'être capable de générer des trajectoires à l'aide de la commande `./trajectoires.py --robot <trajectoire.json>` et de les afficher en utilisant l'outil `plot.py`. Pour ce faire, il *suffit* d'implémenter la méthode `getVal` (en ayant bien sûr implémenté le constructeur adapté).

Pour pouvoir exécuter les trajectoires à l'aide de `simulation.py` il sera nécessaire d'implémenter la méthode `getJointTarget` afin de fournir des cibles au simulateur.

Pour comparer les trajectoires attendues et les trajectoires mesurées lors de la simulation, il faudra implémenter l'ensemble des méthodes de `RobotTrajectory` et compléter la méthode `updateModel` dans `simulation.py`.

Finalement, suivant votre implémentation vous constaterez des problèmes de synchronisation entre les différentes dimensions de `rrr_trapezoidal`, essayez d'approfondir ce problème et de trouver une solution satisfaisante.

4 Consignes de rendu

1. Éditez le contenu du fichier `to_pack.txt` pour qu'il contienne tous les fichiers demandés (sans oublier votre compte-rendu)
2. Créer une archive en utilisant la dernière version du script `group_work_packer.py`, présente sur la page web du cours.
3. Vérifiez le contenu de l'archive `tar tzf XXX.tar.gz`
4. Envoyer votre archive par mail avec le sujet suivant: "(4TSD904U) Rendu TD Trajectoires". N'oubliez pas de mettre en copie tous les membres du groupe.