TP de robotique - Introduction à ROS

1 Objectif

L'objectif de ce TP est de contrôler les mouvements d'un robot manipulateur dans l'espace. Pour cela la première étape est le calcul du modèle géométrique direct (MGD) qui donne la position de l'effecteur en fonction des variables articulaires. La jacobienne du robot sera ensuite calculée. Une fois ces deux modèles déterminés, le robot pourra être programmé pour suivre une trajectoire, ou bien pour suivre des consignes en vitesse.

La mise en pratique des modèles et des lois de commande s'appuie sur l'architecture ROS[1].

2 Travail préliminaire

L'étude porte sur un robot dont le schéma est donné à la figure 1.

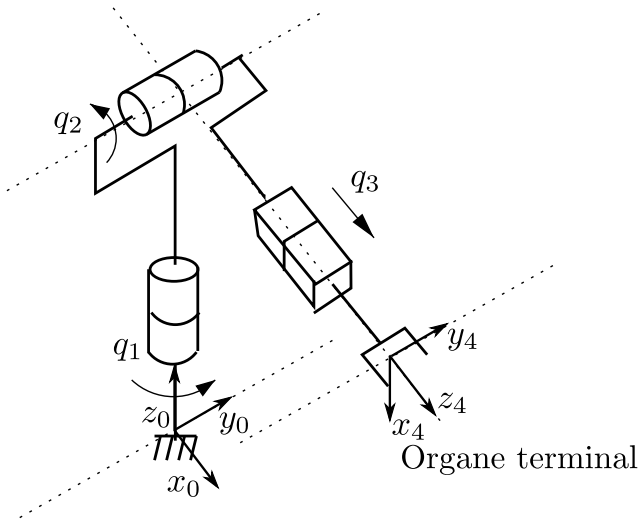

Figure 1: Schéma du robot

On adopte la convention des paramètres de Denavit-Hartenberg modifiés. On note $T_{i,j}$ la matrice de transformation homogène entre les repères $\mathcal{R}_i$ et $\mathcal{R}_j$ et $R_{i,j}$ la matrice de rotation entre les repères $\mathcal{R}_i$ et $\mathcal{R}_j$. On associe à l'organe terminal un repère, noté $\mathcal{R}_4$, dont l'origine $\mathcal{O}_4$

[1]Robot Operating System - http://www.ros.org

a pour coordonnées x_1, x_2 et x_3 (centre de la pince figurée) dans $\mathcal{R}_0$. L'orientation de l'organe terminal est définie par la matrice de rotation :

$$R_{0,3} = R_{0,4} = \begin{pmatrix} x_x & y_x & z_x \\ x_y & y_y & z_y \\ x_z & y_z & z_z \end{pmatrix}$$

entre $\mathcal{R}_0$ et $\mathcal{R}_4$.

Q1 Dessiner le robot dans sa configuration la plus simple et placer les repères selon la convention des paramètres de Denavit-Hartenberg modifiés.

Q2 On note b la distance de la base (hachures) à l'axe de la deuxième liaison et d la distance de l'axe de la deuxième liaison au centre de l'organe terminal (centre pince) pour $q_3 = 0$. On a : $b = 0,5\ m$ et $d = 0,1\ m$. Établir la situation (position + orientation) de l'effecteur en fonction de (q_1, q_2, q_3).

Vérifier pour trois jeux de variables articulaires que le modèle calculé est convenable.

Q3 Une fois le modèle géométrique direct validé, déterminer le modéle géométrique inverse (MGI) permettant de retrouver le jeu de variables articulaires correspondant à une situation donnée de l'effecteur. Les limites des différentes articulations sont :

$$q_1 \in [-\pi, \pi] \quad q_2 \in [-1.2, 1.2] \quad q_3 \in [0, 0.04]$$

Q4 Calculer de deux façons différentes la matrice jacobienne du robot.

3 Première séance

La première séance du TP est sous forme de cours intégré et permet de découvrir le fonctionnement de ROS et de faire des rappels de C++ et d'utilisation de la console Unix. On utilisera l'environnement de développement Qt Creator et la bibliothèque mathématique ViSP[2].

ROS, développé par Willow Garage[3] et maintenant maintenu par l'Open Source Robotics Foundation, permet d'interfacer très simplement des logiciels pouvant être écrits par des personnes différentes et dans des langages différents. Le but de son développement est de pouvoir mettre à disposition de chacun des briques permettant de construire un système complexe. À titre d'exemple, il existe des briques très performantes pour la vision, la plannification de trajectoire, la cartographie automatique, la commande avancée...

De façon assez logique, les logiciels utilisant ROS se présentent sous une forme élémentaire, le package. Sur un ordinateur donné, tous les packages de l'utilisateur sont situés dans le même dossier.

Pour ce TP nous utilisons l'avant-dernière version de ROS : *Hydro*. La dernière version, *Indigo*, sera directement compatible avec le code.

[2]Bibliothèque dédiée à l'asservissement visuel - http://www.irisa.fr/lagadic/visp

[3]Entreprise créée par des anciens du Stanford Artificial Intelligence Lab

3.1 Préparation de l'environnement de travail

1. Une fois connecté sur le PC, ouvrir une console[4] et aller dans le répertoire `~/ros`.
   Il s'agit du répertoire de travail de ROS, tous les paquets de ce répertoire seront donc détectés automatiquement pour la compilation et l'exécution. Ce répertoire contient les différents répertoires :

   - `build` : dossier de compilation
   - `devel` : dossier d'installation, c'est ici que ROS cherche les programmes
   - `src` : dossier des sources des différents packages

   Toutes les commandes du TP sont à entrer dans un terminal en ligne de commande.

2. Aller dans le répertoire `tps_robot` :

   - `cd src/tps_robot` (depuis `~/ros`)
   - ou plus simplement `roscd tps_robot` (depuis n'importe où)

   Ce package contient les simulateurs utilisés pour le TP. Pour le mettre à jour et initialiser l'environnement de la machine :

   - `git pull`[5]
   - `rosrun tps_robot init_tp.py`

   Afficher ce répertoire dans le gestionnaire de fichiers afin de voir la structure typique d'un package ROS :

   - Fichiers à personnaliser par l'utilisateur
     - `package.xml` : indique les dépendances à d'autres packages ROS
     - `CMakeLists.txt` : indique les sources C++ à compiler
   - Dossiers créés par l'utilisateur :
     - `scripts` : fichiers source Python
     - `src` : fichiers source C++
     - `include` : headers des fichiers C++
     - `launch` : contient les fichiers *launchfile* qui automatisent l'exécution de plusieurs programmes à la fois
     - `urdf` : contient les fichiers urdf (Universal Robot Description File), décrivant les modèles géométriques des robots

3. Compiler le package :

   - `cd ~/ros`
   - `catkin_make`

[4] L'utilisation de la console Unix est rappelée en Annexe A
[5] git est un outil de contrôle de version, semblable à svn.

3.2 Nodes et topics

1. On voit qu'un exécutable `simu_4points` a été compilé. Dans ROS les exécutables sont appelés des *nodes*. ROS intègre la complétion automatique. Pour exécuter un node, taper :

   - `rosrun tp<tab>` : complète avec le nom du package
   - `rosrun tps_robot <tab>` : indique les nodes disponibles

   La commande `rosrun` est juste un raccourci pour exécuter les programmes se trouvant dans un package ROS avec l'aide de l'autocomplétion. Cela permet de lancer des programmes se trouvant dans différents dossiers sans avoir besoin de naviguer dans le système de fichiers.

   On voit qu'en plus de `simu_4points`, ROS nous propose d'exécuter `arm_bridge.py`, `dh_code.py`, `init_tp.py` et `setpoint_4points.py`. Il s'agit de fichiers Python qui n'ont pas besoin d'être compilés pour être lancés[6]. On reconnaît le fichier `init_tp.py` qui a été lancé pour initialiser l'environnement.

2. Dans la prise en main nous nous intéressons à :

   - `simu_4points` : simule un asservissement visuel
   - `setpoint_4points.py` : génère des consignes de position

   Ces deux programmes doivent pouvoir communiquer entre eux si on veut que l'un suive les consignes de l'autre. Essayer d'exécuter chacun des programmes (dans une console différente) pour observer qu'ils attendent la présence d'un "master". Il s'agit du logiciel ROS chargé de mettre en relation les différents nodes : le lancer via la commande `roscore` (dans un nouveau terminal). Les deux nodes se lancent aussitôt.

3. Dans une nouvelle console, exécuter `rqt_graph` pour observer la structure élémentaire de l'architecture ROS :

   - Les nodes se sont enregistrés sous les noms `/setpoint_generator` et `/simulator`
   - Le premier envoie un message appelé `/setpoint` au second : ce canal de communication est appelé un *topic*

   Indépendamment du langage, utiliser ROS revient essentiellement à mettre en relation des programmes élémentaires qui souscrivent et publient chacun des informations sur différents topics.

   - Publier : déclarer au ROSmaster qu'on va envoyer des informations sur un topic, sans s'intéresser à la destination de ces données
   - Souscrire : déclarer au ROSmaster qu'on est intéressé par ce qui se passe sur un topic donné, sans s'intéresser à la source de ces données
   - Plusieurs nodes peuvent souscrire et publier sur le même topic : importance de se mettre d'accord pour éviter les embouteillages

[6] Pour être détecté par `rosrun` un fichier Python doit être exécutable

Le ROSmaster se charge de mettre les nodes en relation. Une architecture typique peut être :

- Un node interfacé avec une caméra, faisant l'aquisition des images et les publiant sur un topic
- Un node chargé de récupérer ces images, les traiter et publier la position d'un objet d'intérêt
- Un node récupérant ces positions et calculant la consigne à effectuer pour atteindre l'objet
- Un node interfacé avec un robot, qui ne fait qu'appliquer les consignes de position qu'il reçoit sur le topic auquel il souscrit

Chaque node pouvant être codé, exécuté et compilé indépendamment, cela permet une grande souplesse pour le développement de systèmes complexes.

3.3 ROS dans le code

1. On a pu voir que ROS disposait de nombreux outils en ligne de commande[7]. Pour pouvoir exploiter ces avantages dans le logiciel Qt Creator, il faut également le lancer depuis une console : `qtcreator`. Lors des prochaines séances il faudra également lancer Qt Creator via la console[8]. Pour ouvrir le package tps_robot :

   - Avec `File/Open file`, choisir le fichier `CMakeList.txt` du répertoire `~/ros`
   - Indiquer `~/ros/build` comme répertoire de compilation
   - Valider et exécuter CMake pour obtenir l'affichage des fichiers source
   - Avec `File/Open file`, ouvrir également le fichier `script/setpoint_4points.py`

2. Parcourir les fichiers `simu_4points.cpp` et `setpoint_4points.py` pour y repérer les lignes propres à ROS : déclaration du nom du node, souscription et publication des topics. On voit apparaître la notion de message : il s'agit du type d'information que des nodes s'échangent sur un topic (vecteur, image, chaîne de caractère...). Ici le message `/setpoint` est un `Float32MultiArray`, soit un vecteur de float. De nombreux types de message existent déjà dans ROS, mais il est toujours possible de créer le sien pour une utilisation particulière.

   On peut noter à la fin de `simu_4points.cpp` que ce node publie les positions et les erreurs de position. Ces topics ne sont écoutés par aucun node. Ce node publie également les images acquises par le robot, sur le topic `/camera/image`. Heureusement il n'y a pas besoin d'aller lire du code pour connaître les topics disponibles. Des outils en ligne de commande existent pour analyser les informations circulant sur les topics.

[7] Ces outils sont résumés en Annexe C

[8] Attention notamment à l'ouverture automatique des `.cpp` dans Qt Creator : le logiciel ainsi lancé ne sera pas interfacé avec l'environnement ROS.

3.4 Outils sur les topics - rostopic

1. La commande `rostopic` permet de manipuler les topics via la console :

   - `rostopic list` : liste les topics actifs
   - `rostopic echo <topic>` : affiche le contenu d'un topic
   - `rostopic info <topic>` : donne des infos sur un topic (nodes publiant et souscrivant, type de message)

   Vérifier la présence des autres topics publiés.

2. Si le topic contient des valeurs numériques, ce qui est le cas des `Float32MultiArray`, `rqt_plot` peut en tracer la courbe temporelle :

   - Commande : `rqt_plot <topic>/<attribut>[indice]`
   - Par exemple : `rqt_plot setpoint/data[0]`

3. Si le topic contient des images, un outil permet de les afficher. Vérifier que le topic `camera/image` contient des données de type `sensor_msgs/Image` puis les afficher :

   - `rosrun image_view image_view image:=camera/image`

   On reconnaît la syntaxe ROS : `image_view` est un node du package lui aussi appelé `image_view`. Ce node prend en argument le topic sur lequel recevoir des images pour simplement les afficher dans une fenêtre. Ici les images sont générées par le node de simulation, qui les affiche déjà dans sa propre fenêtre et les publie sur le topic.

3.5 Serveur de paramètres - rosparam

Le ROSmaster inclue également un serveur de paramètres. Tous les nodes peuvent ainsi récupérer et écrire des valeurs diverses et variées. Les paramètres sont à utiliser de préférence aux topics pour communiquer des valeurs qui changent peu souvent et qui n'ont pas besoin d'être diffusées à chaque itération sur le réseau.

Ces paramètres sont accessibles en ligne de commande :

- `rosparam list` : liste les paramètres disponibles
- `rosparam get Ts` : affiche la valeur de `Ts`
- `rosparam set Ts 1` : met à jour le paramètre

Ici les deux paramètres modifiables sont `Ts` (période des créneaux de consigne) et `Kp` (gain de l'asservissement). Utiliser `rosparam` pour obtenir la valeur de ces paramètres, puis changer la période des consignes et le gain de l'asservissement.

3.6 Gérer les exécutions simultanées - roslaunch

Pour un robot complexe il peut rapidement y avoir plusieurs nodes à exécuter et plusieurs paramètres à initialiser afin que tout tourne correctement. Le lancement simultané de plusieurs nodes est géré par l'outil `roslaunch` :

- `roslaunch <package> <launchfile>` : exécute le launchfile

Quitter les nodes ainsi que le `roscore`. Ouvrir le fichier `tps_robot/launch/simu.launch` afin de constater qu'il initialise les deux paramètres `Ts` et `Kp` et qu'il lance les deux nodes `setpoint_4points.py` et `simu_4points`. L'exécuter et vérifier que la simulation fonctionne : `roslaunch` lance automatiquement le ROSmaster s'il n'est pas déjà actif.

3.7 Enregistrer et rejouer des données - rosbag

La commande `rosbag` permet de sauvegarder dans un fichier tout ou partie des données circulant sur les topics.

1. Pour enregistrer l'ensemble des topics :

   - `rosbag record -a` : crée un fichier binaire `.bag` avec les contenus des topics.

   Arrêter l'enregistrement après quelques dizaines de secondes, et quitter les deux nodes (`Ctrl-C` dans les consoles).

2. Deux outils existent pour relire les topics :

   - `rosbag play <fichier.bag>` : republie les données sur les topics enregistrés
   - `rqt_bag` : ouvre une interface permettant de choisir les topics republiés

   Relancer le ROSmaster. Avec `rqt_bag`, lancer la republication de `/setpoint`. Relancer le node `simu_4points` : noter qu'il suit la consigne alors que le node `setpoint_4points` n'est pas lancé. En effet les consignes enregistrées sont republiées sur le même topic. Le node ne s'intéresse pas à la source du topic, seulement à son contenu.

3. Relancer le node `setpoint_4points`. Comment réagit la simulation ? Vérifier avec `rqt_graph` que deux nodes publient sur le même topic. Utiliser `rqt_plot` pour visualiser l'effet d'un conflit de publication (plusieurs nodes publiant sur le même topic sans se concerter) : `rqt_plot setpoint/data[0]`.

Pour approfondir

Des tutoriaux pour bien d'autres aspects de ROS sont disponibles en ligne

- `http://wiki.ros.org/ROS/Tutorials`

4 Deuxième séance

La deuxième séance est consacrée à la modélisation du robot. Pour cela un package sera créé par le binôme.

1. Création d'un package personnel : retourner dans le répertoire `~/ros`. On souhaite y créer un package qui sera dépendant de `tps_robot`
   - Création du package : `catkin_create_pkg <binôme> tps_robot`
   - Copier les fichiers `tps_robot/src/control.cpp` et `tps_robot/src/modeles.cpp` dans le nouveau package
   - Modifier le fichier `<binôme>/CMakeLists.txt` pour indiquer à ROS de compiler la source :
     ```
     add_executable(control control.cpp modeles.cpp)
     target_link_libraries(control ${catkin_LIBRARIES})
     ```
2. Ouverture du package personnel dans Qt Creator
   - Taper dans une ligne de commande : `qtcreator`
   - Dans `Fichier/Ouvrir un fichier`, choisir le fichier `CMakeList.txt` du dossier `~/ros`
   - Valider et exécuter CMake pour obtenir l'affichage des fichiers source

4.1 Fonctionnement du node de commande

Ouvrir le fichier `control.cpp`. On voit qu'il instancie une classe `Robot` avec un certain nombre de degrés de liberté :

```
// initialisation de la classe robot
 Robot robot(rosNH);
 const unsigned int N = robot.getDOFs();
```

Cette classe intègre les interfaces (publication et souscription de topics) avec les autres nodes ROS. Elle contient trois méthodes :

- `robot.getPosition(q)` : récupère les valeurs des variables articulaires
- `robot.setPosition(q)` : envoie une consigne de position `q`
- `robot.setVelocity(v)` : envoie une consigne de vitesse `v`

Les variables `q` et `v` sont de classe `vpColVector`, qui est un vecteur au sens mathématique. Son fonctionnement est détaillé en Annexe F.

L'objectif du TP est d'utiliser la mesure de position articulaire pour calculer le modèle géométrique, et envoyer des consignes de position ou de vitesse au simulateur. Par la suite la programmation se fera en modifiant :

- `modeles.cpp` pour les modèles géométriques, Jacobiens, etc.
- `control.cpp` pour l'algorithme de commande

4.2 Visualisation avec RViz

1. Lancer le simulateur :

   - `roslaunch tps_robot turret3.launch`

   L'interface RViz apparaît avec un modèle 3D. RViz permet d'afficher la situation globale du robot. La fenêtre de gauche donne des informations sur les différents éléments (links) du robot. Notamment, comme montré en Fig.2, il est possible de lire la position et l'orientation de l'effecteur (end_effector). On utilisera donc cette fenêtre pour valider le modèle géométrique.
   En cochant la case "Show Trail", RViz affiche également la trajectoire suivie par l'effecteur.

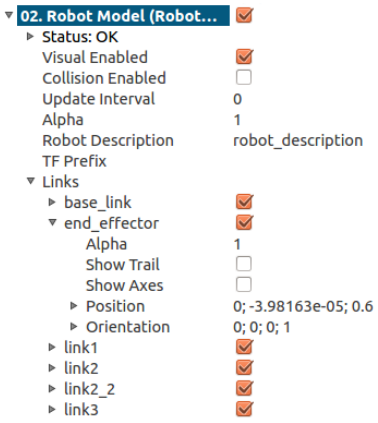

Figure 2: Lecture de la situation de l'effecteur dans RViz

2. Compilation du package personnel :

   - En ligne de commande, depuis le répertoire `~/ros`: `catkin_make`

3. Exécution du node personnel

   - En ligne de commande : `rosrun <binôme> control`
   - Sous ROS, le nom de ce node est `/main_control`
   - Il est possible de lancer le programme en mode debug avec le triangle vert surmonté d'un insecte

Les différents nodes et topics interagissant dans le TP sont détaillés en Annexe D. Une fois lancé, le node tel qu'il est ne fait qu'attendre la publication des positions articulaires du robot.

4.3 Travail demandé

Q1 Programmer la fonction `compDK`[9] qui calcule la position opérationnelle en fonction du vecteur des variables articulaires. La fonction est à coder dans `modeles.cpp`. Dans le

[9] L'Annexe B rappelle les différents prototypes de fonction en C++

code principal (`control.cpp`), afficher dans la console la position ainsi obtenue.
En cas d'erreur on pourra utiliser le script `dh_code.py` qui génère le code correspondant à un tableau de Denavit-Hartenberg, voir Annexe E.

Q2 En jouant sur les sliders (fenêtre `Joint State Publisher`), comparer en plusieurs endroits les valeurs de votre programme avec celles indiquées par le simulateur (Fig.2). Valider ainsi le modèle calculé.

Q3 Afficher le graphe des nodes et des topics. Appeler l'encadrant de TP pour décrire le fonctionnement de l'architecture affichée.

Q4 Programmer la fonction `compIK` qui calcule le modèle géométrique inverse de façon analytique. Coder la simulation alternant toutes les 10 secondes[10] entre les points $P_1 = (-0.065, -0.02, 0.61)$ et $P_2 = (-0.046, -0.046, 0.6)$.
Notez qu'il suffit de définir les points P_1 et P_2, le code inversant la consigne en fonction de la fréquence étant dejà écrit en début de boucle. Ce code fait qu'à tout instant, P_0 est l'ancienne consigne (le point d'où on vient) et P_d est la nouvelle (le point où on va).

Utiliser le paramètre `Q` pour passer d'une question à l'autre pendant l'exécution du code (block `switch` dans `control.cpp`).

Q5 Afficher le graphe des nodes et des topics. Sachant que le node `joint_control` convertit des consignes de position ou de vitesse et publie la position articulaire du robot sur `/joint_control/position`, commenter sur les conflits possibles avec l'interface des sliders `/joint_state_publisher`.

Q6 Vérifier dans une console que les positions obtenues sont bien celles demandées :

```
rosrun tf tf_echo base_link end_effector
```

Valider ainsi le modèle géométrique inverse.

On pourra aussi créer un nouveau *publisher* (Annexe C.3) permettant d'afficher avec `rqt_plot` l'évolution de l'erreur opérationnelle.

Q7 En cochant la case "Show Trail" de l'effecteur sous RViz, commenter la trajectoire obtenue.

[10] La fréquence en Hz est déjà indiquée dans le code par l'instruction `ros::Rate loop(30)`. 10 secondes correspondent donc à 300 itérations dans la boucle while.

5 Troisième séance

La troisième séance porte sur l'exploitation du jacobien dans la commande. On souhaite maintenant contrôler le robot dans l'espace opérationnel.

Q8 Programmer la fonction `compJacobian` qui calcule la matrice jacobienne (classe `vpMatrix`) en fonction du vecteur des variables articulaires. Le script décrit en Annexe E génère le code du jacobien en fonction du tableau des paramètres.

Q9 On souhaite toujours que le robot se déplace entre $P_1 = (0.065, -0.02, .61)$ et $P_2 = (-0.046, -0.046, 0.6)$., mais en allant de point en point de l'espace opérationnel, à l'aide du modèle géométrique inverse. Programmer cette stratégie, en utilisant l'une ou l'autre des procédure de calcul de MGI. Vérifier avec `tf` ou `rqt_plot` que les points sont bien atteints.

Q10 En cochant la case "Show Trail" de l'effecteur sous RViz, commenter la trajectoire obtenue.

Q11 On souhaite réaliser le même mouvement qu'à la question Q3. On s'inspirera de la partie 3.2.2 de la seconde partie du cours de Robotique. On fera le mouvement opérationnel à vitesse constante dans un premier temps. Programmer cette procédure, vérifier que les points sont bien atteints et afficher la trajectoire correspondante.

6 Pour aller plus loin

On a pu voir qu'une fois la modélisation géométrique validée, le schéma de commande est sensiblement le même.

1. Modéliser le robot suivant :

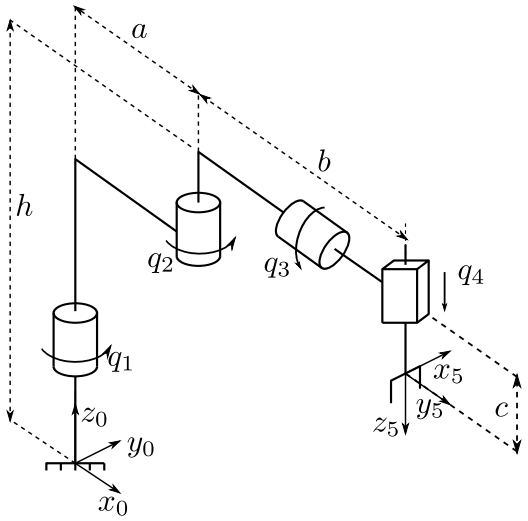

Figure 3: Schéma du robot 4 degrés de liberté

   Avec les distances :

   - Hauteur totale : $h = 0.27$
   - Longueur premier bras : $a = 0.20$
   - Longueur deuxième bras : $b = 0.15$
   - Distance entre l'axe du deuxième bras et le centre de l'effecteur pour $q_4 = 0$: $c = 0.07$

   On souhaite contrôler à la fois la position cartésienne de l'effecteur et l'angle pris par le 3ème axe. On note $\tilde{P} = (x, y, z, q_3)$.

2. Valider le modèle géométrique avec le simulateur :

   - `roslaunch tps_robot scara4.launch`

3. Comme en deuxième séance, utiliser le MGI pour osciller entre les positions $\tilde{P}_1 = (0.3, 0.1, 0.2, 0.4)$ et $\tilde{P}_2 = (0.3, 0.1, 0.2, -0.4)$. Les limites des articulations sont :

$$q_1 \in [-\pi/2, \pi/2] \quad q_2 \in [-\pi/2, \pi/2] \quad q_3 \in [-\pi/2, \pi/2] \quad q_4 \in [-0.04, 0.04]$$

4. Calculer la jacobienne associée à $\tilde{P}$. L'utiliser pour réaliser le même mouvement dans l'espace opérationnel. Que dire des trajectoires ?

A Syntaxe Unix

On rappelle ici quelques éléments de syntaxe de la console Unix.

- `cd <répertoire>` : va dans le répertoire indiqué
- `ls` : liste les fichiers du répertoire courant
- `./<exécutable>` : exécute le programme indiqué
- `Ctrl - C` : annule la commande actuelle, force la fermeture du programme qui a été lancé dans cette console
- `Ctrl - Shift - C` : Copie le texte sélectionné
- `Ctrl - Shift - V` : Colle le texte sélectionné

Même s'il est possible d'avoir plusieurs fenêtre de terminaux, il est parfois moins confus d'avoir plusieurs onglets dans une même fenêtre. Ces onglets sont gérés comme suit :

- `Ctrl - Shift - T` : ouvre un nouvel onglet
- `Ctrl - Shift - W` : ferme l'onglet courant
- `Ctrl - Shift - Q` : ferme la fenêtre courante et tous ses onglets

B Rappels de syntaxe C++

On rappelle ici les prototypages standards des fonctions en C++. Le but est de prendre l'habitude de se servir des références ou des objets en fonction de ce qu'on veut faire des arguments d'une fonction.

- Passage par copie : `f(vpMatrix M)`
  - La matrice `M` peut être modifiée dans la fonction
  - Elle ne sera pas modifiée à la sortie : la fonction travaille sur une copie de `M`
- Passage par référence : `f(vpMatrix &M)`
  - La matrice `M` peut être modifiée dans la fonction
  - Elle sera modifiée également en dehors de la fonction
  - Utile pour des fonctions qui mettent à jour certaines variables
- Passage par référence constante : `f(const vpMatrix &M)`
  - La matrice `M` ne peut pas être modifiée dans la fonction
  - L'appel à la fonction ne mobilise pas de ressources car il n'y a pas de recopie
  - Utile pour des fonctions qui utilisent une variable sans avoir le droit de la modifier

C Outils intégrés à ROS

ROS propose plusieurs outils en ligne de commande permettant d'aider le développement et le diagnostic.

C.1 Outils en ligne de commande

- `roscd <package>` : va dans le répertoire du package
- `rosmake <package>` : compile le package
- `rosrun <package> <program>` : exécute le programme contenu dans le package
- `rosnode list` : liste les nodes existant
- `rostopic list` : liste les topics existant
- `rostopic echo <topic>` : affiche les données circulant sur le topic

Matrice de passage

La valeur courante de la transformation entre l'effecteur et la base s'obtient par :

```
rosrun tf tf_echo base_link end_effector
```

`tf` est le package de ROS gérant la correspondante entre les variables articulaires et les matrices de transformation. `tf_echo` est un node de ce package qui se contente d'afficher la matrice de transformation entre deux repères (ici `base_link` et `end_effector`).

C.2 Outils graphiques

- `rqt_graph` : affiche le graphe des nodes
- `rqt_plot <topic1/valeurs1, topic2/valeurs2> -p <period>` : affiche le tracé des valeurs publiées sur les topics sur une période glissante

C.3 Créer d'autres publishers (C++)

Pour avoir accès à certains résultats il peut être utile de publier de nouvelles données au sein du contrôleur. Cela permet notamment de les afficher via `rqt_plot`. Un exemple de syntaxe est donné dans `control.cpp`. Avant la boucle un publisher et un message sont instanciés :

```
ros::Publisher examplePub;
std_msgs::Float32MultiArray example;
```

Dans la boucle l'attribut `data` de `example` est mis à jour et publié :

```
examplePub.publish(example);
```

On peut alors afficher le graphe des variables publiées en temps réel :

```
rqt\_plot /example/data[0],/example/data[1],/example/data[2] -p 10
```

D Nodes et topics de la simulation

1. Le simulateur et ses interfaces se lancent en exécutant un launchfile :

   - `roslaunch tps_robot turret3.launch`

   Ce launchfile lance les nodes suivants :

   - `/joint_state_publisher` : interface des sliders. Publie sur `/joint_states` les positions articulaires correspondant aux sliders.
   - `/robot_state_publisher` : reçoit les positions articulaires. À l'aide du modèle géométrique du robot, publie sur `/tf` les matrices de passage entre les différents éléments du robot.
   - `/rviz` : reçoit les matrices de passage pour afficher la position actuelle du robot
   - `/joint_control` : simule la commande bas niveau du robot.
     - Si ce node reçoit des consignes en position ou en vitesse sur `/mainControl/command`, il les transforme en suite de positions articulaires en prenant en compte les butées articulaires et les vitesses maximum. Ces positions sont publiées sur `/jointControl/position`. Ce topic prend la main sur les sliders de `/joint_state_publish`
     - S'il ne reçoit rien, il ne publie rien.

2. La boucle est fermée à l'exécution du node de commande :

   - `/main_control` : reçoit les positions articulaires sur `/joint_states`. Après calcul de la loi de commande, publie la consigne de position ou de vitesse sur `/main_control/command`.

E Génération de code avec paramètre de Denavit-Hartenberg

Le script `dh_code.py` permet de générer le code C++ du modele géométrique direct et du jacobien à partir du tableau des paramètres.
Un exemple est donné dans le package `tps_robot/urdf/dh_example.yml`. Pour le lancer :

- `roscd tps_robot/urdf`
- `rosrun tps_robot dh_code.py dh_example.yml`

N'hésitez pas a créer votre propre fichier YAML (extension `.yml`) dans votre package afin de ne pas avoir à faire les calculs a la main.

F Utilisation des classes de ViSP

ViSP est une bibliothèque proposant notamment des classes pour les objets mathématiques usuels (matrices, vecteurs, etc.). Une matrice `M` de dimension $n \times k$ et un vecteur colonne `V` de dimension k s'instancient avec les instructions :

```
vpMatrix M(n,k);
vpColVector V(k);
```

On accède aux éléments de `M` et `V` comme un tableau classique : `M[i][j]`. Toutes les opérations classiques sont permises, notamment :

- `M.t()` : transposée de `M`
- `M*V` : produit matriciel
- `M.pseudoInverse()` : inverse (ou pseudo-inverse si non carrée)
- `M.getCols()` : nombre de colonnes
- `M.getRows()` : nombre de lignes

Il existe également une classe pour les matrices de rotation et pour les vecteurs translation :

```
vpRotationMatrix R;
vpTranslationVector t;
```

Ces deux grandeurs permettent d'instancier une classe de matrice de changement de repère :

```
vpHomogeneousMatrix M;
M.buildFrom(t,R);
```

Dans ce cas on a :

$$\mathbf{M} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}_{3\times 1} & 1 \end{bmatrix}$$

À l'inverse, on peut aussi extraire les parties rotation et translation d'une matrice homogène

```
M1.buildFrom(t1,R1);
M2.buildFrom(t2,R2);
M = M1*M2;
M.extract(t);
M.extract(R);
```