TP 3

Animation d'un robot *Fanuc LR Mate 200iD/4s*

3.1 Objectifs du TP

Vous allez réaliser et implémenter les modèles qui permettent d'animer un robot en position articulaire, en position opérationnelle et en vitesse opérationnelle. Vous allez également utiliser différents outils proposés par *ROS*. Ce TP est découpé en 2 parties.

Première partie (préparation + séance de 4 heures)

- Paramétrage géométrique du robot avec la convention de Denavit-Hartenberg modifiée ;
- Implémentation du modèle géométrique direct pour le contrôle en position articulaire ;
- Implémentation du modèle géométrique inverse pour le contrôle en configuration opérationnelle (position et orientation de son effecteur par rapport au repère de base).

Deuxième partie (préparation + séance de 4 heures)

- Implémentation du modèle cinématique pour le contrôle en vitesse opérationnelle ;
- Contrôle du robot avec les outils de *ROS* ;
- Génération de trajectoire articulaire ;
- Génération de trajectoire opérationnelle :
  — Par modèle géométrique inverse le long d'une trajectoire rectiligne ;
  — Par modèle géométrique inverse le long d'une géodésique.

3.2 Architecture informatique du TP

Les différents scripts que vous allez compléter lors de ce TP s'intègrent dans une architecture qui utilise l'environnement de développement pour la robotique *ROS* (*Robot Operating System*). Une vue générale de l'architecture du TP est représentée sur la Figure 3.1.

Sur la Figure 3.1, on peut distinguer deux types d'objets :

- Les processus *ROS* (*nodes*) dans les ellipses : peuvent être programmés dans différents langages. Ici, le langage de programmation sera *Python*.
- Les canaux de communication entre processus (*topics*) dans les rectangles : le type de topic est défini par le type de message qu'il utilise.

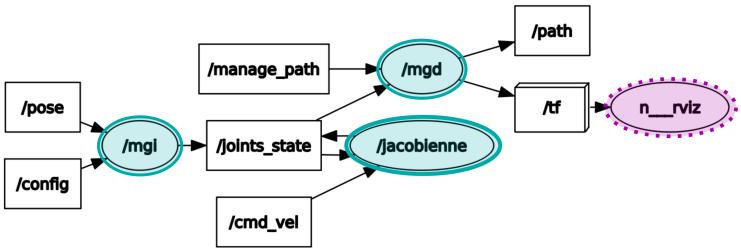


FIGURE 3.1 – Architecture informatique du TP

Un autre type d'objet *ROS* n'est pas représenté sur la figure et est utilisé dans le TP. Il s'agit des *services* qui permettent de faire un appel de fonction à un autre noeud.

Lors de ce TP, vous allez compléter des noeuds (entourés en bleu/vert avec trait plein sur la figure) qui ont été développés en *Python*.

Le noeud `n__rviz` (entouré en rose avec trait pointillé) permet d'afficher le robot et de le contrôler via un pupitre de commande (voir Figure 3.2).

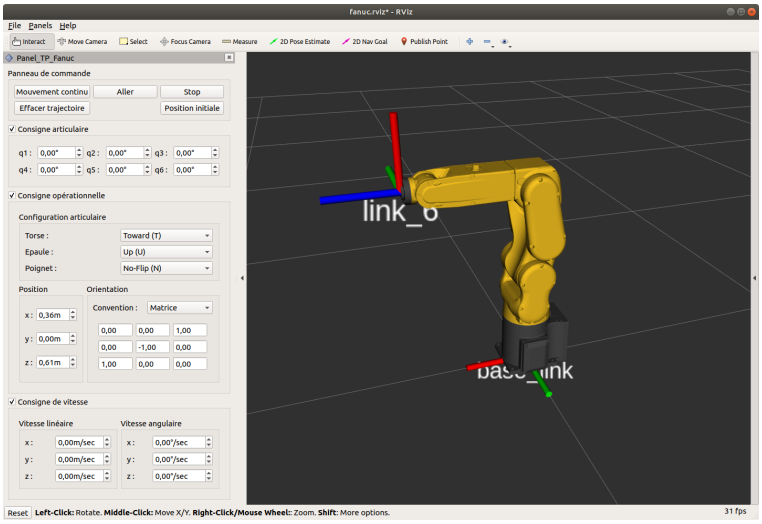


FIGURE 3.2 – Représentation du robot et du pupitre de commande dans *Rviz*

Plus de détails sur l'architecture informatique se trouvent en Annexe 3.9 à la page 29.

La bibliothèque *numpy* est utilisée pour l'utilisation des vecteurs et matrices ainsi que les opérations qui leurs sont appliquées. Il faudra bien veiller à respecter les dimensions attendues pour que les méthodes que vous complétez s'intègrent bien dans l'intégralité du projet.

Quelques remarques et astuces sur l'utilisation de *numpy* se trouvent en Annexe 3.10 à la page 32.

3.3 Lancement du TP

Environnement de travail à disposition

Vous avez à disposition un PC de l'école sur lequel *ROS* est installé.

Booter sur Ubuntu et se connecter sur le compte `etudiant` (mot de passe : `etudiant`).

Les logiciels suivants sont installés et seront vos meilleurs alliés dans l'accomplissement de votre tâche :

- `Terminator` : il y a beaucoup de terminaux à utiliser simultanément. Ce logiciel permet de partager la fenêtre et ainsi organiser plus simplement les différentes fenêtres ;
- `Visual Studio Code` : pour programmer les scripts Python. Il est possible d'ouvrir tout un répertoire et l'afficher dans un volet ;

Lancement de la première séance

1. Récupérer les sources (**Attention**, à ne faire qu'au début de la séance pour ne pas perdre votre travail).

```
cd ~/catkin_ws
./recuperer_sources_tp_fanuc
```

2. Taper les commandes suivantes dans un terminal pour que *ROS* soit bien fonctionnel.

```
cd ~/catkin_ws
./debut_tp_fanuc
```

Vous pouvez maintenant commencer à programmer ce robot pour le faire bouger !

Le **répertoire de travail** est `catkin_ws/src/robotique_experimentale/tp_fanuc`.

Lancement de la seconde séance

1. **Remplacer** le répertoire `catkin_ws/src/robotique_experimentale/tp_fanuc` par le répertoire `tp_fanuc` que vous avez précieusement **conservé** ;
2. Taper les commandes suivantes dans un terminal pour que *ROS* soit bien fonctionnel.

```
cd ~/catkin_ws
./debut_tp_fanuc
```

Sauvegarde du projet à la fin des séances

- Sauvegarde personnelle de votre projet ;
  Copier le répertoire `catkin_ws/src/robotique_experimentale` sur une clé USB pour la prochaine séance.
- Copie locale (au cas où vous perdriez votre copie ou tout autre désagrément) :

```
cd ~/catkin_ws
./sauvegarde_tp_fanuc NOM1_NOM2
```

3.4 Paramétrage géométrique du robot

Après avoir fait le paramétrage géométrique du robot en suivant la convention de Denavit-Hartenberg modifiée, compléter le fichier `config_DHm.yaml` situé dans le répertoire `config`.

Les distances sont en **[m]** et les angles en **[deg]**. Entrer uniquement les **paramètres constants**, les variations q_i sont prises en compte dans le calcul du modèle géométrique direct.

3.5 Modèle géométrique direct

On cherche ici à contrôler le robot en position articulaire, c'est-à-dire qu'on cherche à calculer la pose (position + orientation dans le repère de base) de chacun des corps le constituant en fonction de sa configuration articulaire.

Des outils *ROS* existent déjà pour cet usage, mais vous allez faire tout de vos propres mains.

Tapez la commande suivante dans un terminal pour faire appel aux outils en question, à titre d'exemple.

```
roslaunch tp_fanuc control_robot_ros_tools.launch
```

Puis `Ctrl+C` pour arrêter l'exécution.

Maintenant, c'est à vous de jouer ! Compléter le fichier `mgd.py` situé dans le répertoire `script`.

3.5.1 Détermination de la matrice de transformation rigide d'un corps $i-1$ à un corps i

Compléter la méthode `compute_Ti(self,dh,q)`.

Cette méthode prend en entrée les paramètres de Denavit-Hartenberg du corps $i-1$ au corps i (sous la forme $[\sigma, \alpha, \mathbf{a}, \mathbf{d}, \theta]$) et la position articulaire en **[m ou rad]**.

3.5.2 Détermination de la matrice de transformation rigide d'un corps i à un corps j

Compléter la méthode `compute_T(self,Q,i,j)`.

Cette méthode prend en entrée le vecteur des positions articulaires en **[m ou rad]** des articulations i à j ainsi que les indices i et j en question.

Remarque : La paramétrisation de Denavit-Hartenberg est stockée dans le tableau `self._DH` à l'initialisation du noeud. Pour récupérer la ligne correspondant à la transformation entre le corps 0 et le corps 1, il faut utiliser `self._DH[0,:]`.

Le test en console permet de vérifier la validité du code. Des exemples avec le robot test et avec le robot *Fanuc LR Mate 200iD/4s* sont proposés dans la section 3.5.4.

3.5.3 Détermination de l'état du robot

Compléter la méthode `compute_robot_state(self)`.

Cette méthode prend en entrée le vecteur des positions articulaires en **[m ou rad]**.

Elle est appelée à chaque réception d'une configuration articulaire sur le *topic* `/joints_state` et retourne :

- la matrice de transformation rigide entre la base du robot et le corps terminal ;
- la liste des matrices de transformation rigide entre les corps i et i+1 ;
- la liste des matrices de transformation rigide entre la base et le corps i.

Remarques :

1. L'état des articulations du robot est enregistré dans l'attribut `self._joints_state` ;
2. Le nombre d'axes du robot est enregistré dans l'attribut `self._nb_axis`.

Le test avec affichage sous *Rviz* permet de vérifier la validité du code.

3.5.4 Test

Test du modèle géométrique pour un robot simple

On teste notre modèle géométrique sur un robot constitué d'une rotoïde et d'une prismatique, dont les paramètres géométriques sont présentés dans le tableau 3.1.

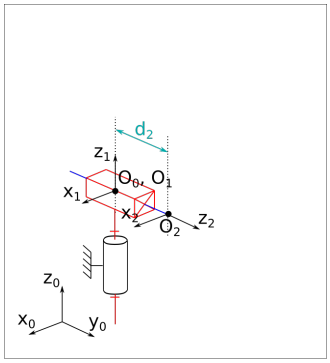
i	σ_i	a_{i-1} [m]	α_{i-1} [°]	d_i [m]	θ_i [°]
1	0	0	0	0	q_1
2	1	0	-90	q_2 + 0.5	0

TABLE 3.1 – Paramètres géométriques du robot test en suivant la convention de Denavit-Hartenberg modifiée

```
# charger la configuration du robot dans un terminal
roslaunch tp_fanuc load_config.launch

# executer le noeud du modele geometrique direct en mode test avec le robot test
# dans un autre terminal
rosrun tp_fanuc mgd.py --test --robot=robot_test
```

Puis `Ctrl+C` dans chaque terminal pour arrêter l'exécution.

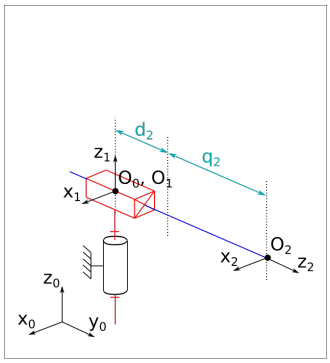
- $q = [0, 0]$




```
Coordonnees articulaires (rotoides en [rad], prismatiques
en [m]) :
0.0 0.0
Q =
[ 0.  0.]

Repere de base
0
i = 0

Repere d arrivee
2
j = 2

Matrice de transformation rigide entre le corps 0 et le
corps 2 :
[[ 1.  0.  0.  0. ]
 [ 0.  0.  1.  0.5]
 [ 0. -1.  0.  0. ]
 [ 0.  0.  0.  1. ]]
```

Vérifier votre modèle et commenter.

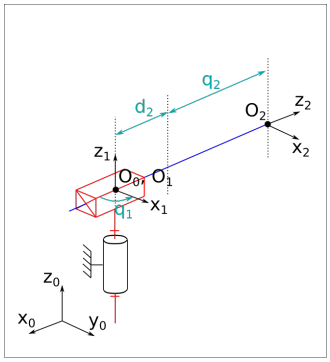
- $q = [0, 1]$




```
Coordonnees articulaires (rotoides en [rad], prismatiques
en [m]) :
0.0 1.0
Q =
[ 0.  1.]

Repere de base
0
i = 0

Repere d arrivee
2
j = 2

Matrice de transformation rigide entre le corps 0 et le
corps 2 :
[[ 1.  0.  0.  0. ]
 [ 0.  0.  1.  1.5]
 [ 0. -1.  0.  0. ]
 [ 0.  0.  0.  1. ]]
```

Vérifier votre modèle et commenter.

- $q = [\frac{\pi}{2}, 1]$




```
Coordonnees articulaires (rotoides en [rad], prismatiques
en [m]) :
1.571 1.0
Q =
[ 1.571  1.   ]

Repere de base
0
i = 0

Repere d arrivee
2
j = 2

Matrice de transformation rigide entre le corps 0 et le
corps 2 :
[[ 0.  -0.  -1.  -1.5]
 [ 1.   0.   0.   0. ]
 [ 0.  -1.   0.   0. ]
 [ 0.   0.   0.   1. ]]
```

Vérifier votre modèle et commenter.

Test du modèle géométrique avec le robot *Fanuc LR Mate 200iD/4s*

On teste le modèle géométrique direct pour des configurations simples.

```
# charger la configuration du robot dans un terminal
roslaunch tp_fanuc load_config.launch

# executer le noeud du modele geometrique direct en mode test
# dans un autre terminal
rosrun tp_fanuc mgd.py --test
```

Puis `Ctrl+C` dans chaque terminal pour arrêter l'exécution.

- $q = [0, 0, 0, 0, 0, 0]$

```
Coordonnees articulaires (rotoides en [rad], prismatiques
en [m]) :
0.0 0.0 0.0 0.0 0.0 0.0
Q =
[ 0.  0.  0.  0.  0.  0.]

Repere de base
0
i = 0

Repere d arrivee
6
j = 6

Matrice de transformation rigide entre le corps 0 et le
corps 6 :
[[ 0.   0.   1.   0.36]
 [-0.  -1.   0.   0.  ]
 [ 1.  -0.  -0.   0.61]
 [ 0.   0.   0.   1.  ]]
```

Représenter le robot dans cette configuration articulaire en suivant le paramétrage géométrique. Vérifier votre modèle et commenter.

- $q = [\frac{\pi}{2}, 0, 0, 0, 0, 0]$

```
Coordonnees articulaires (rotoides en [rad], prismatiques
en [m]) :
1.571 0.0 0.0 0.0 0.0 0.0
Q =
[ 1.571  0.     0.     0.     0.     0.    ]

Repere de base
0
i = 0

Repere d arrivee
6
j = 6

Matrice de transformation rigide entre le corps 0 et le
corps 6 :
[[ 0.    1.    0.    0.  ]
 [ 0.    0.    1.    0.36]
 [ 1.   -0.   -0.    0.61]
 [ 0.    0.    0.    1.  ]]
```

Représenter le robot dans cette configuration articulaire en suivant le paramétrage géométrique. Vérifier votre modèle et commenter.

Test du modèle géométrique en contrôlant le robot avec le pupitre de commande sous *Rviz*

```
# execution du modele geometrique direct uniquement
roslaunch tp_fanuc tp_DHm.launch mgi:=false jacobienne:=false
```

Puis `Ctrl+C` pour arrêter l'exécution.

3.6 Modèle géométrique inverse

On cherche ici à contrôler la configuration opérationnelle du robot (position et orientation de son effecteur par rapport au repère de base). C'est-à-dire qu'on cherche à déterminer sa position articulaire.

Compléter le fichier `mgi.py` situé dans le répertoire `script`.

3.6.1 Détermination de la position articulaire du robot

Compléter la méthode `compute_MGI(self,T,config)`.

Cette méthode prend en entrée la matrice de transformation rigide de la base du robot à l'effecteur ainsi que la configuration articulaire du robot parmi les 8 possibles. Elle retourne le vecteur des positions articulaires en **[m ou rad]**.

Remarques :

1. Comme pour `mdg.py`, la paramétrisation de Denavit-Hartenberg est stockée dans le tableau `self._DH` à l'initialisation du noeud. Pour récupérer le paramètre a_0, il faut utiliser `self._DH[0,1]` ; pour récupérer le paramètre d_1, il faut utiliser `self._DH[0,3]` ; etc.
2. La configuration articulaire du robot est une chaîne de caractères. Le premier correspond à la configuration du poignet (*Flip/No-Flip*), le deuxième à la configuration du coude (*Up/Down*) et le troisième à la configuration du torse (*Toward/Backward*) ;
3. La méthode `compute_homogeneous_transform(self,q,ind1,ind2)` de la classe `MGI` fait appel à la méthode `compute_T(self,Q,i,j)` du noeud `mgd` au moyen d'un *service ROS*. Vous pouvez y faire appel pour calculer les différentes matrices de transformation rigide dont vous avez besoin pour implémenter le modèle géométrique inverse. Par exemple :

```
T01 = self.compute_homogeneous_transform([q1],0,1)
```

3.6.2 Test

Test du modèle géométrique inverse en ligne de commande

```
# charger la configuration du robot dans un terminal
roslaunch tp_fanuc load_config.launch

# executer le noeud du modele geometrique direct en mode test
# dans un autre terminal
rosrun tp_fanuc mgd.py --test

# executer le noeud du modele geometrique inverse en mode test
# dans un autre terminal
rosrun tp_fanuc mgi.py --test
```

Puis `Ctrl+C` dans chaque terminal pour arrêter l'exécution.

Exemple avec la position articulaire $q = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]$:

```
Coordonnees articulaires (rotoides en [rad], prismatiques en [m]) :
np.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
Q =
[ 0.1  0.2  0.3  0.4  0.5  0.6]

Pose cible =
[[ 0.639 -0.551  0.537  0.352]
```

```
  [-0.742  -0.625   0.242   0.048]
  [ 0.203  -0.553  -0.808   0.407]
  [ 0.      0.      0.      1.   ]]

-

config : NUT

Q =
[ 0.1   0.2   0.3   0.4   0.5   0.6]

T =
[[ 0.639  -0.551   0.537   0.352]
  [-0.742  -0.625   0.242   0.048]
  [ 0.203  -0.553  -0.808   0.407]
  [ 0.      0.      0.      1.   ]]

[...]

config : FDB

Q =
[ 3.242  -2.142   0.3     0.242  -2.251  -2.032]

T =
[[ 0.639  -0.551   0.537   0.352]
  [-0.742  -0.625   0.242   0.048]
  [ 0.203  -0.553  -0.808   0.407]
  [ 0.      0.      0.      1.   ]]
```

Vérifier votre modèle et commenter.

Test du modèle géométrique inverse avec le pupitre de commande sous *Rviz*

```
# execution des modeles geometriques direct et inverse
roslaunch tp_fanuc tp_DHm.launch jacobienne:=false
```

Puis `Ctrl+C` pour arrêter l'exécution.

Vérifier que la configuration opérationnelle reste inchangée lorsque vous changez de configuration articulaire.

Vérifier également que la configuration articulaire du robot est respectée, comme représenté dans la préparation.

3.7 Modèle cinématique

On cherche ici à animer le robot *Fanuc LR Mate 200 iD/4s* entre deux configurations opérationnelles par modèle cinématique inverse.

Compléter le fichier `jacobienne.py` situé dans le répertoire `script`.

3.7.1 Détermination de la jacobienne du robot

Compléter la méthode `compute_J(self,Q)`.

Cette méthode prend en entrée le vecteur des coordonnées articulaires et retourne la matrice jacobienne.

Remarques :

1. Le nombre d'axes du robot est enregistré dans l'attribut `self._nb_axis`.
2. La méthode `get_links_pose(Q)` fait appel à la méthode `compute_robot_state(self,Q)` du noeud `mgd` au moyen d'un *service ROS*. Elle permet de récupérer la liste des matrices de transformation rigide de chacun des corps par rapport à la base du robot.

Le test en console permet de vérifier la validité du code. Des exemples avec le robot test et avec le robot *Fanuc LR Mate 200iD/4s* sont proposés dans la section 3.7.4.

3.7.2 Détermination des vitesses articulaires du robot

Compléter la méthode `compute_q_velocity(self,linear,angular,q)`.

Cette méthode prend en entrée les vecteurs de vitesse linéaire et angulaire ainsi que le vecteur des coordonnées articulaires. Elle retourne le vecteur des vitesses articulaires.

Remarque : Voir l'Annexe 3.10 sur l'utilisation de *numpy* pour la multiplication de matrices.

3.7.3 Détermination des positions articulaires du robot

Compléter la méthode `compute_q(self,linear,angular,q0,dt)`.

Cette méthode prend en entrée les vecteurs de vitesse linéaire et angulaire ainsi que le vecteur des coordonnées articulaires courantes et le temps d'échantillonnage pour l'intégration des vitesses articulaires. Elle retourne le nouveau vecteur des positions articulaires.

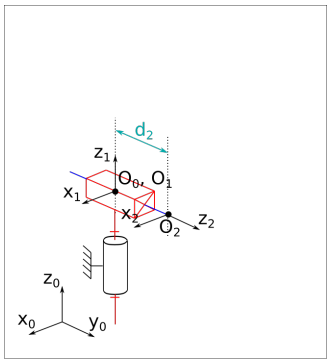
3.7.4 Test

Test du modèle cinématique inverse pour un robot simple

On teste notre modèle cinématique sur le même robot test que pour le modèle géométrique direct.

```
# charger la configuration du robot
roslaunch tp_fanuc load_config.launch

# executer le noeud du modele geometrique direct en mode test avec le robot test
rosrun tp_fanuc mgd.py --test --robot=robot_test

# executer le noeud du modele cinematique en mode test avec le robot test
rosrun tp_fanuc jacobienne.py --test --robot=robot_test
```

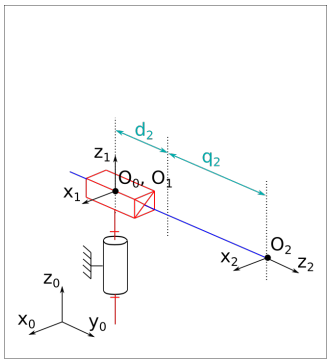
- $q = [0, 0]$




```
Coordonnees articulaires (rotoides en [rad], prismatiques
en [m]) :
0.0  0.0
Q =
[ 0.   0.]

Jacobienne :
[[ 0.   0.]
 [ 0.   0.]
 [ 1.   0.]
 [-0.5  0.]
 [ 0.   1.]
 [ 0.   0.]]
```

Vérifier votre modèle et commenter.

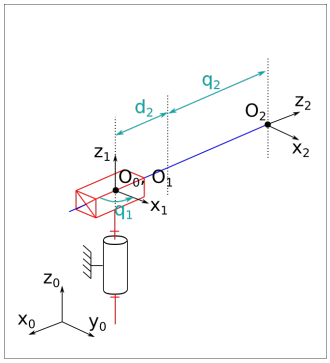
- $q = [0, 1]$




```
Coordonnees articulaires (rotoides en [rad], prismatiques
en [m]) :
0.0  1.0
Q =
[ 0.   1.]

Jacobienne :
[[ 0.   0.]
 [ 0.   0.]
 [ 1.   0.]
 [-1.5  0.]
 [ 0.   1.]
 [ 0.   0.]]
```

Vérifier votre modèle et commenter.

- $q = [\frac{\pi}{2}, 1]$



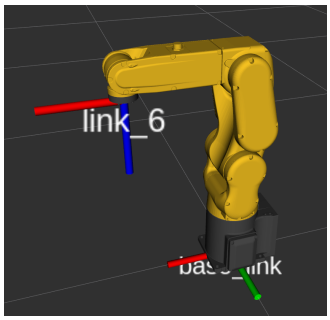

```
Coordonnees articulaires (rotoides en [rad], prismatiques
en [m]) :
1.571  1.0
Q =
[ 1.571   1.    ]

Jacobienne :
[[ 0.   0.]
 [ 0.   0.]
 [ 1.   0.]
 [ 0.  -1.]
 [-1.5  0.]
 [ 0.   0.]]
```

Vérifier votre modèle et commenter.

Test du modèle cinématique inverse avec le robot *Fanuc LR Mate 200iD/4s* en ligne de commande

```
# on lance tous les processus sans le modele cinematique, sans l'affichage
roslaunch tp_fanuc tp_DHm.launch jacobienne:=false gui:=false

# on lance le modele cinematique dans un terminal a part pour tester
rosrun tp_fanuc jacobienne.py --test
```

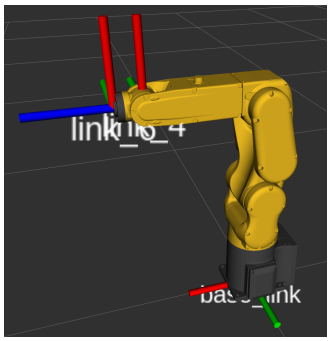
- $q = [0, 0, 0, 0, \frac{\pi}{2}, 0]$




```
Coordonnees articulaires (rotoides en [rad], prismatiques
en [m]) :
0.0 0.0 0.0 0.0 1.571 0.0
Q =
[ 0.     0.     0.     0.     1.571  0.   ]

Jacobienne :
[[ 0.     0.     0.     1.     0.     0.   ]
 [ 0.     1.     1.     0.     1.     0.   ]
 [ 1.    -0.    -0.     0.     0.    -1.   ]
 [-0.     0.21  -0.05  -0.    -0.07   0.   ]
 [ 0.29  -0.    -0.     0.07   0.     0.   ]
 [ 0.    -0.29  -0.29   0.     0.     0.   ]]
```

Vérifier le résultat et commenter. Dans quelle configuration se trouve le robot ? Peut-on le contrôler en utilisant le modèle cinématique inverse ?

- $q = [0, 0, 0, 0, 0, 0]$




```
Coordonnees articulaires (rotoides en [rad], prismatiques
en [m]) :
0.0 0.0 0.0 0.0 0.0 0.0
Q =
[ 0.   0.   0.   0.   0.   0.]

Jacobienne :
[[ 0.     0.     0.     1.     0.     1.   ]
 [ 0.     1.     1.     0.     1.     0.   ]
 [ 1.    -0.    -0.     0.    -0.     0.   ]
 [-0.     0.28   0.02   0.     0.     0.   ]
 [ 0.36  -0.    -0.     0.    -0.     0.   ]
 [ 0.    -0.36  -0.36   0.    -0.07   0.   ]]
```

Vérifier le résultat et commenter. Dans quelle configuration se trouve le robot ? Peut-on le contrôler en utilisant le modèle cinématique inverse ?

Test du modèle cinématique inverse en contrôlant le robot avec le pupitre de commande sous *Rviz*

À ce stade, tous les modes de déplacement du robot sont implémentés.

```
# execution des modeles geometriques direct, inverse et cinematique
roslaunch tp_fanuc tp_DHm.launch
```

3.7.5 Animation du robot avec le modèle cinématique inverse

Vous allez, dans cette partie, utiliser les outils que fournit *ROS* pour envoyer des messages sur les différents *topics* et les afficher en fonction du temps.

Interfaçage avec *ROS*

On s'interface avec l'architecture *ROS* du contrôleur et on vient écrire une consigne de vitesse sur le topic `/cmd_vel` (entouré en bleu/vert sur la Figure 3.3).

```
# pour afficher l'architecture ROS
rqt_graph

# pour afficher la liste des topics dans le terminal
rostopic list

# pour afficher la liste des noeuds dans le terminal
rosnode list
```

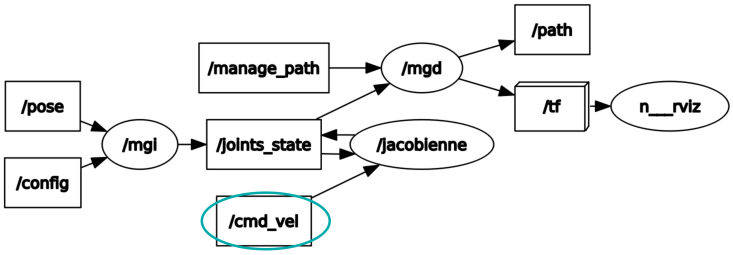


Figure 3.3 – *Topic* `/cmd_vel` sur lequel on vient publier des messages

On utilise l'outil graphique `rqt_gui` (voir Figure 3.4) qui permet d'envoyer des messages, afficher des données, etc.

- Pour publier un message, utiliser le plug-in `Message Publisher` (Plugins > Topics > Message Publisher).
- Pour afficher des mesures, utiliser le plug-in `Plot` (Plugins > Visualization > Plot).

```
# ouverture de rqt_gui pour envoyer les consignes de vitesse
rosrun rqt_gui rqt_gui
```

On note qu'il est également possible de publier ou souscrire sur/à un *topic* au moyen de lignes de commande dans le terminal mais par souci de simplicité, on utilise ici les outils graphiques.

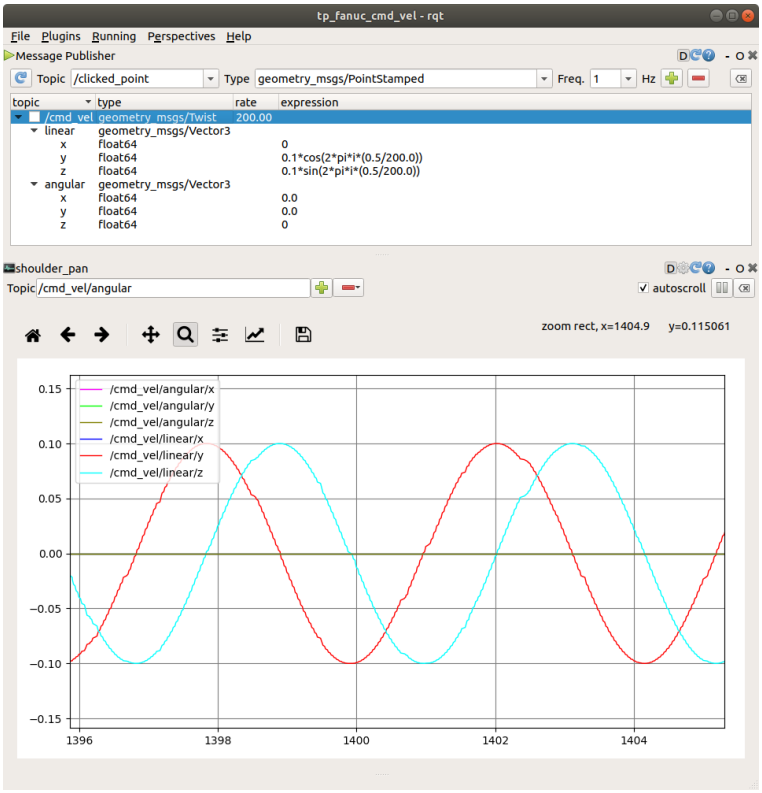


FIGURE 3.4 – Outil graphique `rqt_gui` avec les deux Plugins `Message Publisher` et `Plot` pour publier sur le topic `/cmd_vel` et afficher la consigne

Consigne de vitesse

On se propose d'envoyer une consigne de vitesse qui permet à l'effecteur de décrire un cercle dans l'espace opérationnel.

$$\left| \begin{array}{l} v_y = A\cos(\omega t) = A\cos(2\pi f_s \frac{i}{f_e}) \\ v_z = A\sin(\omega t) = A\sin(2\pi f_s \frac{i}{f_e}) \end{array} \right. \quad \text{avec :} \quad \left| \begin{array}{lcl} A & : & \text{amplitude de la sinusoide} \\ f_s & : & \text{fréquence de la sinusoide} \\ i & : & \text{échantillon} \\ f_e & : & \text{fréquence d'échantillonnage (i.e. de publication sur le } \textit{topic ROS}) \end{array} \right.$$

En partant de la position initiale $q_0 = [0°, 40°, 0°, 0°, -40°, 0°]$, appliquer la consigne de vitesse avec les paramètres $A = 0.1m/s$, $f_s = 0.5Hz$ et $f_e = 200Hz$.

Remarques

- Pour envoyer les consignes de vitesse, il faut cocher la case devant le *topic* `/cmd_vel` dans le Plugin `Message Publisher`.
- Pour arrêter le mouvement, décocher la case et appuyer sur Stop sur le pupitre de commande dans *Rviz* afin d'envoyer une consigne de vitesse nulle.

Jouer avec les différents paramètres pour décrire une ellipse ; ajouter une consigne de vitesse v_x pour décrire une sinusoïde le long du cercle ou de l'ellipse ; démarrez le mouvement en partant d'une autre position articulaire ; etc.

Que se passe-t-il si on part de la position initiale $q_0 = [0°, -10°, 0°, 0°, 10°, 0°]$? Pourquoi ?

Que se passe-t-il si on part de la position initiale $q_0 = [0°, 0°, 0°, 0°, 0°, 0°]$? Pourquoi ?

Influence de la période d'échantillonage

On souhaite étudier l'influence de la période d'échantillonnage sur le contrôle du robot par modèle cinématique inverse.

1. Exécuter les différents processus du TP sans `jacobienne` ;

```
# execution des modeles geometriques direct et inverse (inutile de relancer
cette commande pour chaque periode d echantillonnage)
roslaunch tp_fanuc tp_DHm.launch jacobienne:=false
```

2. Puis exécuter le noeud `jacobienne` pour différentes périodes d'échantillonnage.

```
# execution du modele cinematique avec une periode d echantillonnage donnee
rosrun tp_fanuc jacobienne.py --dt=0.1
```

Appliquer la consigne de vitesse précédente pour différentes périodes d'échantillonnage du modèle cinématique `jacobienne` : 0.1s, 0.5s et 0.02s par exemple.

Observer la dérive et commenter. Quelle est l'influence du temps d'échantillonnage du modèle cinématique sur la trajectoire du robot ? Expliquer pourquoi.

Remarques :
- Le noeud `jacobienne` ne peut pas être exécuté à une fréquence supérieure à 100Hz en raison des temps de calcul.
- Pour ne pas effacer la trace de la trajectoire pour les différents tests, faire tourner la base du robot (q_1).

3.8 Génération de trajectoires

Dans cette partie, vous allez vous interfacer avec l'architecture *ROS* pour générer différentes trajectoires :

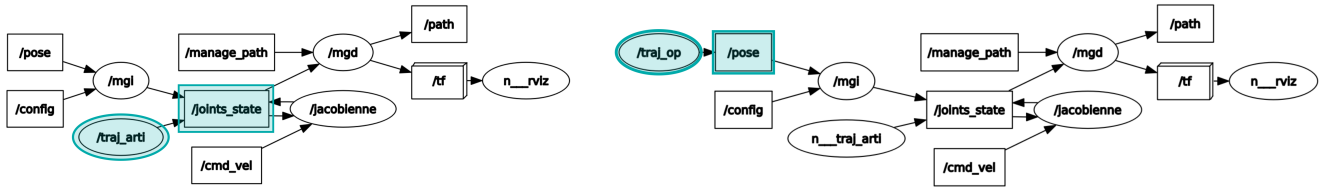
- Génération de trajectoire articulaire ;
- Génération de trajectoire opérationnelle :
  - Par modèle géométrique inverse le long d'une trajectoire rectiligne ;
  - Par modèle géométrique inverse le long d'une géodésique.

3.8.1 Interfaçage avec *ROS*

On s'interface avec l'architecture *ROS* pour envoyer soit des positions articulaires soit des poses opérationnelles.

Pour ce faire, vous allez programmer des noeuds *ROS* qui publient sur les *topics* encadrés sur la Figure 3.5 avec les messages définis suivants :

- Trajectoire articulaire (voir Figure 3.5a) :
  - *topic* : `/joints_state`
  - message : `sensor_msgs/JointState`
- Trajectoire opérationnelle (voir Figure 3.5b) :
  - *topic* : `/pose`
  - message : `geometry_msgs/Pose`



(a) Le noeud `/traj_arti` publie un message de type `sensor_msgs/JointState` sur le topic `/joints_state`

(b) Le noeud `/traj_op` publie un message de type `geometry_msgs/Pose` sur le topic `/pose`

Figure 3.5 – Interfaçage avec l'architecture *ROS* pour la génération de trajectoires

Ressources utiles :

- Écrire un script Python pour publier ou lire un message sur un / depuis un *topic ROS* : http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python)
- Page de référence du message `sensor_msgs/JointState` : http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/JointState.html
- Page de référence du message `geometry_msgs/Pose` : http://docs.ros.org/en/melodic/api/geometry_msgs/html/msg/Pose.html

3.8.2 Génération de trajectoire articulaire

Compléter le script `traj_arti.py`.

La partie *ROS* de ce script est, en partie, déjà faite. Vous devez uniquement programmer la partie *TRAJECTOIRE*.

Exécuter le noeud comme suit.

```
# execution des modeles geometriques direct, inverse et cinematique (si ce n'est
pas deja fait)
roslaunch tp_fanuc tp_DHm.launch

# execution du noeud traj_arti
rosrun tp_fanuc traj_arti.py
```

3.8.3 Génération de trajectoire opérationnelle

Par modèle géométrique inverse le long d'une trajectoire rectiligne

Compléter le script `traj_op.py`.

1. Compléter la méthode `eulerZYX_to_matrix(alpha,beta,gamma)`.
2. Compléter le programme principal :
   (a) Configuration de *ROS* : configuration du *node*, configuration du *publisher*, configuration de la fréquence de publication ;
   (b) Programmation de la trajectoire.
   Remarque : le message de type `geometry_msgs/Pose` est constitué d'une position (`geometry_msgs/Point`) et d'une orientation (`geometry_msgs/Quaternion`) sous la forme d'un quaternion. Vous pouvez utiliser la bibliothèque `tf` pour transformer une matrice de transformation rigide en quaternion comme suit.

```
quaternion = tf.transformations.quaternion_from_matrix(R)
```

Éxécuter le noeud comme suit.

```
# execution des modeles geometriques direct, inverse et cinematique (si ce n est
pas deja fait)
roslaunch tp_fanuc tp_DHm.launch

# execution du noeud traj_op
rosrun tp_fanuc traj_op.py
```

Par modèle géométrique inverse le long d'une géodésique

Compléter le script `traj_geodesique.py`.

1. Compléter la méthode `eulerZYX_to_matrix(alpha,beta,gamma)`.
2. Compléter la méthode `axis_angle_to_matrix(u,theta)`.
3. Compléter la méthode `matrix_to_axis_angle(R)`.
4. Compléter le programme principal :
   (a) Configuration de *ROS* ;
   (b) Programmation de la trajectoire.

Éxécuter le noeud comme suit.

```
# execution des modeles geometriques direct, inverse et cinematique (si ce n est
pas deja fait)
roslaunch tp_fanuc tp_DHm.launch

# execution du noeud traj_geodesique
rosrun tp_fanuc traj_geodesique.py
```

3.8.4 Comparaison des différents types de trajectoires

Exemple de trajectoire articulaire

Tester votre programme avec les positions articulaires initiales $q_0 = [-\frac{\pi}{4}, \frac{\pi}{4}, 0, 0, \frac{\pi}{4}, -\frac{\pi}{4}]$ et $q_f = [\frac{\pi}{4}, \frac{\pi}{4}, 0, 0, \frac{\pi}{4}, \frac{\pi}{4}]$ comme suit.

```
Coordonnees articulaires initiales (rotoides en [rad], prismatiques en [m]) :
np.array([-np.pi/4, np.pi/4, 0, 0, np.pi/4, -np.pi/4])
q0 =
[-0.785  0.785  0.     0.     0.785 -0.785]

Coordonnees articulaires cible (rotoides en [rad], prismatiques en [m]) :
np.array([np.pi/4, np.pi/4, 0, 0, np.pi/4, np.pi/4])
q_cible =
[ 0.785  0.785  0.     0.     0.785  0.785]
```

Exemple de trajectoire opérationnelle

Tester les deux programmes de trajectoire opérationnelle `traj_op.py` et `traj_geodesique.py` avec les poses initiales et finales $\text{pose}_0 = [0.3, 0.2, 0.4, 0.0, 0.0, \frac{5\pi}{4}]$ et $\text{pose}_f = [0.3, -0.2, 0.4, 0.0, 0.0, \frac{5\pi}{4}]$.

```
Pose initiale (position en [m], angles d Euler ZYX en [rad]) :
np.array([0.3, 0.2, 0.4, 0.0, 0.0, 5*np.pi/4])
pose initiale =
[ 0.3    0.2    0.4    0.     0.     3.927]

Pose cible (position en [m], angles d Euler ZYX en [rad]) :
np.array([0.3, -0.2, 0.4, 0.0, 0.0, -5*np.pi/4])
pose cible =
[ 0.3   -0.2    0.4    0.     0.    -3.927]
```

C'est beau la géodésique, n'est-ce-pas ?

Questions

Commenter ces différences.

Dans quelle contexte utilise-t-on la trajectoire articulaire plutôt que la trajectoire opérationnelle et inversement ?

3.9 Annexe : Architecture informatique du TP

L'utilisation de *ROS* permet de découper le code en briques et de les interconnecter entre elles avec, notamment, le principe du *Publisher/Subscriber*.

3.9.1 Arborescence du répertoire `tp_fanuc`

```
~/catkin_ws/src/robotique_experimentale/tp_fanuc$ tree -L 1
.
├── CMakeLists.txt
├── config
├── doc
├── include
├── launch
├── meshes
├── package.xml
├── rviz
├── scripts
├── src
├── srv
└── urdf
```

Figure 3.6 – Arborescence du répertoire `tp_fanuc`

3.9.2 Les noeuds

Les noeuds (*nodes*) sont les processus *ROS*. Ils peuvent être programmés en divers langages comme par exemple C++, Python ou Matlab.

Dans le cadre du TP, les noeuds sont des scripts Python. Ils sont situés dans le répertoire `scripts`.

mgd (`mgd.py`)

Permet de traduire les consignes articulaires en *tf* (=transformation rigides).

- **souscriptions**
  - **/joints_state** [sensor_msgs/JointState]
  - **/manage_path** [std_msgs/String]
- **publications**
  - **/tf** [tf2_msgs/TFMessage]
  - **/tf_static** [tf2_msgs/TFMessage]
  - **/path** [nav_msgs/Path]
- **services**
  - **/get_pose** : calcule la pose de l'effecteur pour une position articulaire donnée ; le message est envoyé sous la forme de [geometry_msgs/Pose].
  - **/move** : fait bouger le robot à une position articulaire donnée.
  - **/get_homogeneous_transform** : calcule la transformation entre 2 repères, le message est mis sous forme de pose ; le message est envoyé sous la forme de [geometry_msgs/Pose].
  - **/manage_path** : permet d'effacer la trajectoire (remise à zéro, suppression des poses).
  - **/get_links_pose** : calcule la liste des matrices de transformation rigide de la base du robot à chacun des corps ; le message est envoyé sous la forme de [tf2_msgs/TFMessage].

mgi (`mgi.py`)

Calcule la consigne articulaire permettant d'atteindre une pose donnée, avec une configuration articulaire définie.

- **souscriptions**
  - **/pose** [geometry_msgs/Pose]
  - **/config** [std_msgs/String]
- **publications**
  - **/joints_state** [sensor_msgs/JointState]
- **services**
  - **/get_joints** : calcule la consigne articulaire pour une pose et une configuration données ; le message est envoyé sous la forme de [sensor_msgs/JointState].

jacobienne (`jacobienne.py`)

Calcule les vitesses articulaires pour une vitesse opérationnelle donnée. Le noeud fait l'intégration à une certaine fréquence et renvoie directement les coordonnées articulaires.

- **souscriptions**
  - **/cmd_vel** [geometry_msgs/Twist]
  - **/joints_state** [sensor_msgs/JointState]
- **publications**
  - **/joints_state** [sensor_msgs/JointState]
- **services**
  - **/get_joints_velocity** : calcule les vitesses articulaires pour une vitesse opérationnelle donnée ; le message est envoyé sous la forme de [sensor_msgs/JointState].

RViz

L'interface de commande permet de contrôler facilement le robot dans les différents mode, pour tester rapidement les différents modèles. Cependant, elle ne permet pas de vérifier numériquement la validité des modèles.

3.9.3 Les fichiers de configuration ou de description du robot

Ils sont situés dans les répertoires `config`, `meshes` et `urdf`.

- `config` : On y trouve un fichier qui contient les paramètres géométriques de deux robots : le *Fanuc LR Mate 200iD/4s* et un robot test ;
  La configuration de chacun de ces robots est chargée dans le serveur des paramètres de *ROS* lors du TP. Les paramètres de Denavit-Hartenberg du robot utilisé sont remis sous la forme d'une matrice à l'initialisation de chacun des noeuds.
- `meshes` : On y trouve les fichiers 3D des différents corps du robot *Fanuc LR Mate 200iD/4s* ;
- `urdf` : On y trouve les fichiers de description du robot *Fanuc LR Mate 200iD/4s*. Lorsqu'on charge ce fichier, la description du robot est chargée dans le serveur des paramètres de *ROS*.
  - *links* (corps) du robot : Géométrie des différents corps du robot.
    [base_link, link_1, link_2, link_3, link_4, link_5, link_6]
  - *joints* (articulations) du robot : Elles permettent de spécifier les mouvements d'un corps enfant (*child*) par rapport à son parent (*parent*). Elles contiennent donc le paramétrage géométrique du robot.
    [joint_1, joint_2, joint_3, joint_4, joint_5, joint_6]
    Remarque : le paramétrage du robot est nécessaire lorsqu'on utilise les outils mis à disposition par *ROS* pour contrôler les articulations du robot et afficher les corps du robot. Dans le cadre du TP, vous allez faire vous-même ces outils.

3.9.4 Les fichiers d'exécution

Des fichiers d'exécution, appelés *launchfiles*, sont situés dans le répertoire `launch`. Ils permettent de charger les différents paramètres du robot et d'exécuter des noeuds. Ils peuvent prendre des arguments en entrée.

Dans le cadre du TP, vous allez essentiellement utiliser le *launchfile* `tp_DHm.launch` avec les arguments suivants :

- `mgd/mgi/jacobienne` pour exécuter ou non les noeuds ;
- `robot` pour sélectionner le robot dont on veut charger les paramètres géométriques
- `gui` pour lancer ou non la visualisation sous *Rviz*.

3.10 Annexe : Utilisation de *numpy*

N'hésitez pas à tester vos opérations dans une invite de commande python (il faut taper `python` dans le terminal). Vous pouvez y afficher les variables comme c'est le cas avec Matlab.

3.10.1 Différences avec Matlab

Quelques différences notables avec Matlab :

- Les indices commencent à 0 ;
- La création d'une matrice est moins *user-friendly* ;
- Un vecteur ligne peut être un vecteur ou une matrice. Cependant un vecteur colonne ne peut être qu'une matrice (voir exemple ci-dessous).

Synthèse de ces différences : https://numpy.org/doc/stable/user/numpy-for-matlab-users.html

3.10.2 Remarques pour le TP

- Respecter bien les dimensions attendues pour que les méthodes que vous complétez s'intègrent bien dans l'intégralité du projet ;
- Faites bien attention à multiplier des matrices entre elles. Notamment lors du calcul du modèle cinématique, vous ne pouvez pas multiplier un vecteur avec une matrice.

```
# Par exemple x = A.q

# definir q comme une matrice a une ligne
>>> q = np.array([[0., 1., 2.]])
>>> q =
array([[ 0.,  1.,  2.]])

# definir A
>>> A = np.identity(3)
>>> A
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])

# x = A.q
>>> x = np.dot(A,q.T)
>>> x
array([[ 0.],
       [ 1.],
       [ 2.]])

# on veut retourner x comme un vecteur
>>> x[:,0]
array([ 0.,  1.,  2.])
```