

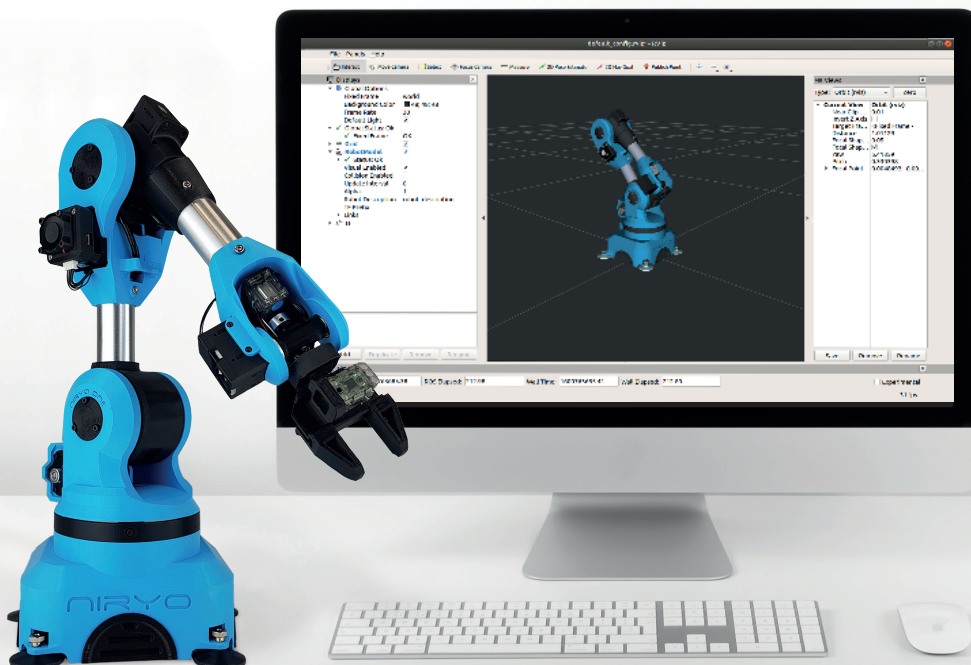
# NIRYO

HUMAN - MOTION - ROBOT

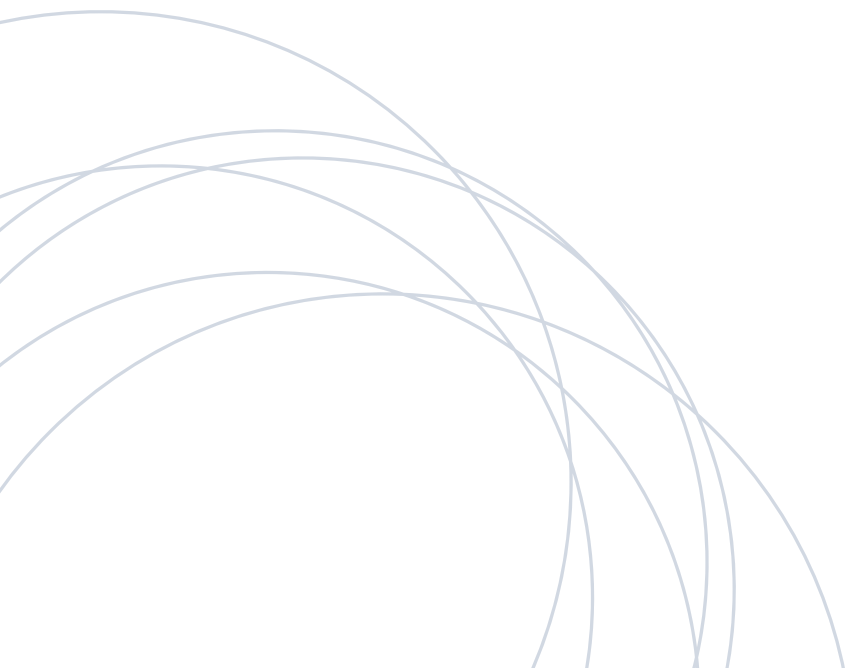
# ROS

# 1

Programmer le Niryo One avec  ROS



<b>OBJECTIFS</b>	<b>3</b>
<b>PRÉREQUIS</b>	<b>3</b>
<b>CE DONT VOUS AVEZ BESOIN</b>	<b>3</b>
<b>INSTALLATION</b>	<b>3</b>
<b>INTERAGIR AVEC ROS EN LIGNE DE COMMANDE</b>	<b>4</b>
SIMULATION DU NIRYO ONE AVEC RVIZ	4
NŒUDS, TOPICS ET SERVICES	5
MISE EN PRATIQUE : NŒUDS ET TOPICS	9
ESSAI : NŒUDS ET TOPICS	10
DÉCOUVERTE DE MOVEIT	11
EXERCICE DE DÉCOUVERTE : MOVEIT	11
MISE EN PRATIQUE : ROS EN LIGNE DE COMMANDE	13
<b>INTERAGIR AVEC ROS EN PYTHON</b>	<b>13</b>
EXERCICE DE DÉCOUVERTE : ROS EN PYTHON	14
<b>CORRECTION</b>	<b>21</b>



# OBJECTIFS

- Se familiariser avec l'environnement ROS
- Intégrer avec ROS en ligne de commande
- Intégrer avec ROS en Python
- Développer un nœud subscriber en Python
- Développer un nœud publisher en Python

# PRÉREQUIS

- Connaître le fonctionnement du Niryo One
- Connaître le terminal sous Linux

# CE DONT VOUS AVEZ BESOIN

- Un ordinateur équipé du Wi-Fi ou d'un port Ethernet
- Ubuntu 16.04 ou 18.04
- ROS Kinetic ou Melodic

# INSTALLATION

Durant ce TP, nous allons interagir avec ROS. Pour cela, vous aurez besoin d'installer ROS sur votre ordinateur.

Afin de l'installer, suivre le tutoriel à l'adresse suivante :

<http://wiki.ros.org/melodic/Installation/Ubuntu> (il est conseillé d'installer la version desktop-full)

- Ouvrir un terminal et entrer les commandes suivantes :  
(git doit être installé : *sudo apt-get install git*)


```
$ cd ~  
  
$ mkdir catkin_ws && cd catkin_ws  
  
$ mkdir src && cd src  
  
$ git clone https://github.com/NiryoRobotics/niryo_one_ros.git  
  
$ cd ..  
  
$ catkin_make
```

- Afin de pouvoir utiliser les paquets ROS de Niryo, il faut indiquer l'emplacement de ces derniers grâce à la commande suivante :

```
$ cd ~/catkin_ws  
$ source devel/setup.bash
```

- Afin de ne pas refaire cette manipulation à chaque fois, nous allons la rentrer dans le script `.bashrc` qui est lancé à chaque nouveau terminal :

```
$ echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
```

 Ce TP n'a pas pour but de détailler le fonctionnement de ROS dans son intégralité. Ici, nous survolerons quelques concepts de ROS afin de pouvoir l'utiliser avec le Niryo One. Pour plus d'explications sur ROS et ses concepts avancés, des tutoriels sont disponibles sur <http://wiki.ros.org/ROS/Tutorials>

# INTERAGIR AVEC ROS EN LIGNE DE COMMANDE

## SIMULATION DU NIRYO ONE AVEC RVIZ

### RVIZ

RViz est l'outil de visualisation 3D de ROS. L'objectif principal est de montrer les messages ROS en 3D, ce qui permet de vérifier visuellement les données. Nous allons donc nous en servir pour visualiser le modèle 3D du Niryo One

### VISUALISATION DU ROBOT NIRYO ONE SUR ROS

Tout d'abord, abordons la commande permettant d'exécuter une application ROS :

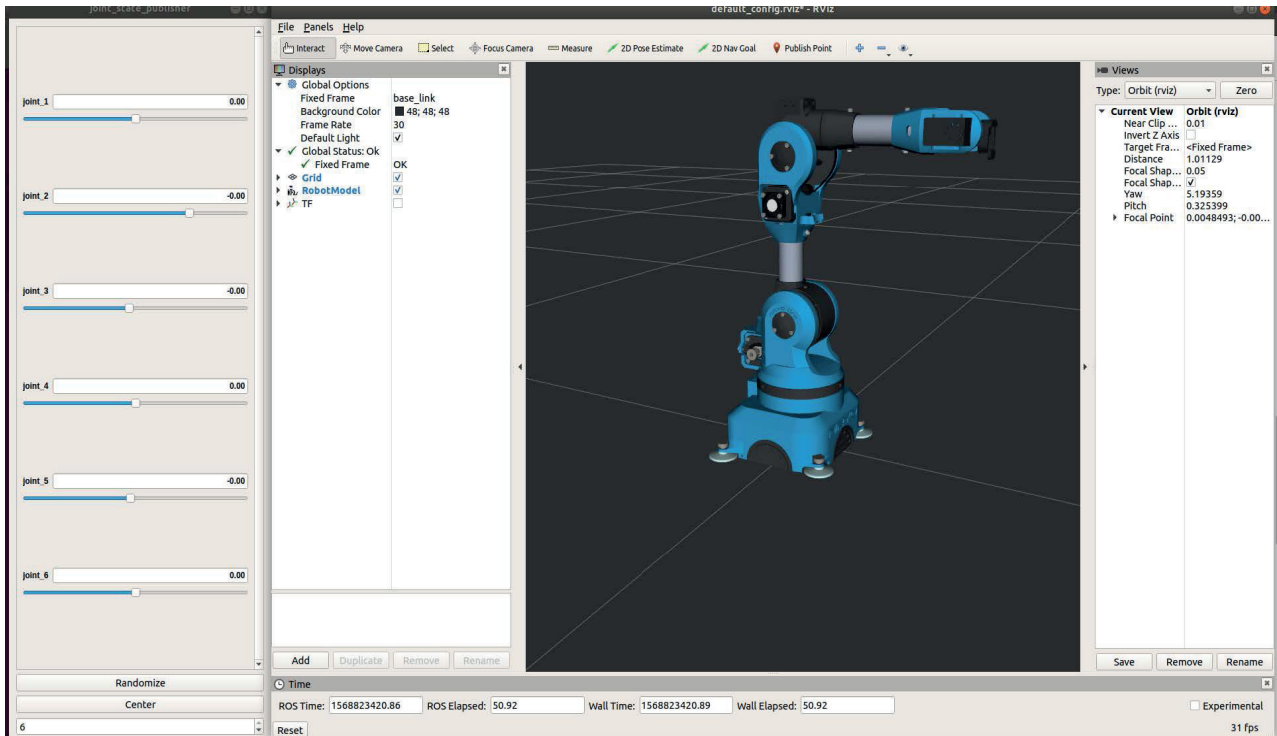
```
$ roslaunch package_name launch_file
```

`roslaunch` est une commande qui lit un fichier XML contenant l'ensemble des nœuds à exécuter ainsi que les paramètres associés à ces derniers.

Un fichier `.launch` (format xml) est déjà présent dans la stack Niryo One ROS qui s'occupe de lancer uniquement Rviz, nous allons donc l'utiliser pour visualiser le robot et son modèle 3D via la commande suivante. Cela va modifier le topic des joints du robot et donc le modèle du robot va bouger en fonction de ces valeurs :

```
$ roslaunch niryo_one_description display.launch
```

Si l'erreur suivante apparaît, "could not find the GUI, install the 'joint\_state\_publisher\_gui' package", faire la commande : `sudo apt install ros-melodic-joint-state-publisher-gui` (remplacer melodic par kinetic selon votre version de ROS).



L'interface permet de déplacer les curseurs des joints afin de faire bouger le robot et le modèle bougera en fonction de ces valeurs de joints. Rviz est majoritairement utilisé pour ses capacités de visualisation, que ça soit du robot, de trajectoires, de murs virtuels ou bien encore de repère spatial.

## NOEUDS, TOPICS ET SERVICES

### LEXIQUE DE BASE POUR ROS

- **Master** : Le Master est un service de déclaration et d'enregistrement des nœuds qui permet ainsi à des nœuds de communiquer entre eux et d'échanger de l'information. Il est nécessaire pour que les différents éléments ROS communiquent entre eux.
- **Nœuds** : Dans ROS, un nœud est une instance d'un exécutable. Un système de contrôle robotique comprendra généralement de nombreux nœuds. Par exemple, un nœud contrôle du bras, un nœud contrôle des moteurs, un nœud effectue la planification, etc... Chaque nœud qui se lance se déclare au Master (roscore) pour être disponible auprès des autres nœuds.
- **Messages** : Un message est une structure de données ROS, comprenant des champs typés. Les types primitifs standards (entier, float, boolean, etc.) sont supportés, de même que les tableaux de types primitifs. Les messages peuvent inclure des structures et des tableaux

imbriqués de manière arbitraire. Le message définit donc le "format" d'échange entre les nœuds via les différentes méthodes de communication.

- **Topics** : Le topic permet l'échange de l'information de manière asynchrone. Il est basé sur le système de l'abonnement / publication (subscribe / publish). Un ou plusieurs nœuds pourront publier de l'information sur un topic et un ou plusieurs nœuds pourront lire l'information sur ce topic, le topic est régulièrement utilisé pour publier de la data à une fréquence donnée.
- **Service** : Un service est défini par une paire de messages : un pour la requête, un pour la réponse. Un client appelle un service en envoyant la requête sous forme de message et attend la réponse. Le serveur, en recevant la requête, l'effectue et renvoie la réponse sous forme d'un message. Le service est destiné à des fonctions courtes en termes de temps de l'ordre du quasi instantané.
  - **Message-service / fichier .srv** : Ils désignent les fichiers utilisés pour décrire un service et sont construits sur le système de messages, un fichier .srv sera constitué d'une première partie (message) pour les arguments d'appels et d'une deuxième partie (message) pour le(s) valeur(s) de retours comme par exemple:

```
int8 arg_1
---
string ret_1
```

- **Action/actionlib** : Une action est semblable à un service, hormis qu'une action est un appel asynchrone (donc non bloquant) donc l'appelant peut effectuer d'autres tâches durant cet appel. Une action est constituée de 5 topics ; un topic pour annuler la commande envoyée de la forme */action\_name/cancel* ; pour appeler/démarrer une action, nous allons envoyer un goal (sorte de message pour les actions) sur le topic */action\_name/goal* ; si l'appelant souhaite être informé de l'état de l'action effectuée, il pourra écouter les différents topic */action\_name/feedback*, */action\_name/status*, et */action\_name/result*. De par ces caractéristiques, il est destiné à tous les usages prenant un temps supérieur à "l'instantané".

### Liste de commandes disponibles relatives aux nœuds

```
$ rosnod list
```

Obtenir la liste des nodes actifs.

```
$ rosnod info /nom_du_node
```

Obtenir des informations sur un node.

## Liste de commandes disponibles relatives aux topics

```
$ rostopic list
```

Affiche la liste des topics.

```
$ rostopic info / topic_name
```

Affiche des informations concernant le topic (type de message, subscriber, publisher, ...).

```
$ rostopic echo / topic_name
```

Affiche le contenu d'un topic en fonction du message transmis.

```
$ rostopic type /topic_name
```

Affiche le message (structure de données au sens de ROS) d'un topic.

```
$ rostopic pub /topic_name /message_type [message]
```

Permet de publier des données sur un topic.

## Liste de commandes disponibles relatives aux services

```
$ rosservice list
```

Liste les services disponibles.

```
$ rosservice call /service_name /message_type [data]
```

Affiche les informations concernant ce message / message-service et tous les champs qui le compose, le type de chaque données et nom pour chaque champs.

```
$ rosservice info /service_name
```

Permet de connaître certaines informations relatives au service\_name.

## Liste de commandes disponibles relatives aux messages et messages-service

```
$ [rosmg / rossrv] list
```

Affiche la liste de tous les différents messages / message-service disponible actuellement (pour le projet en cours).

```
$ [rosmg / rossrv] show /msg_name
```

Affiche les informations concernant ce message / message-service et tous les champs qui le compose, le type de chaque données et nom pour chaque champs.



*Il n'y a pas de commande spécifique pour les actions, en effet, les actions regroupent à chaque fois cinq topics cancel/goal/feedback/status/result donc interagir avec les différents topics d'une action se fera via les commandes de topics habituelles.*

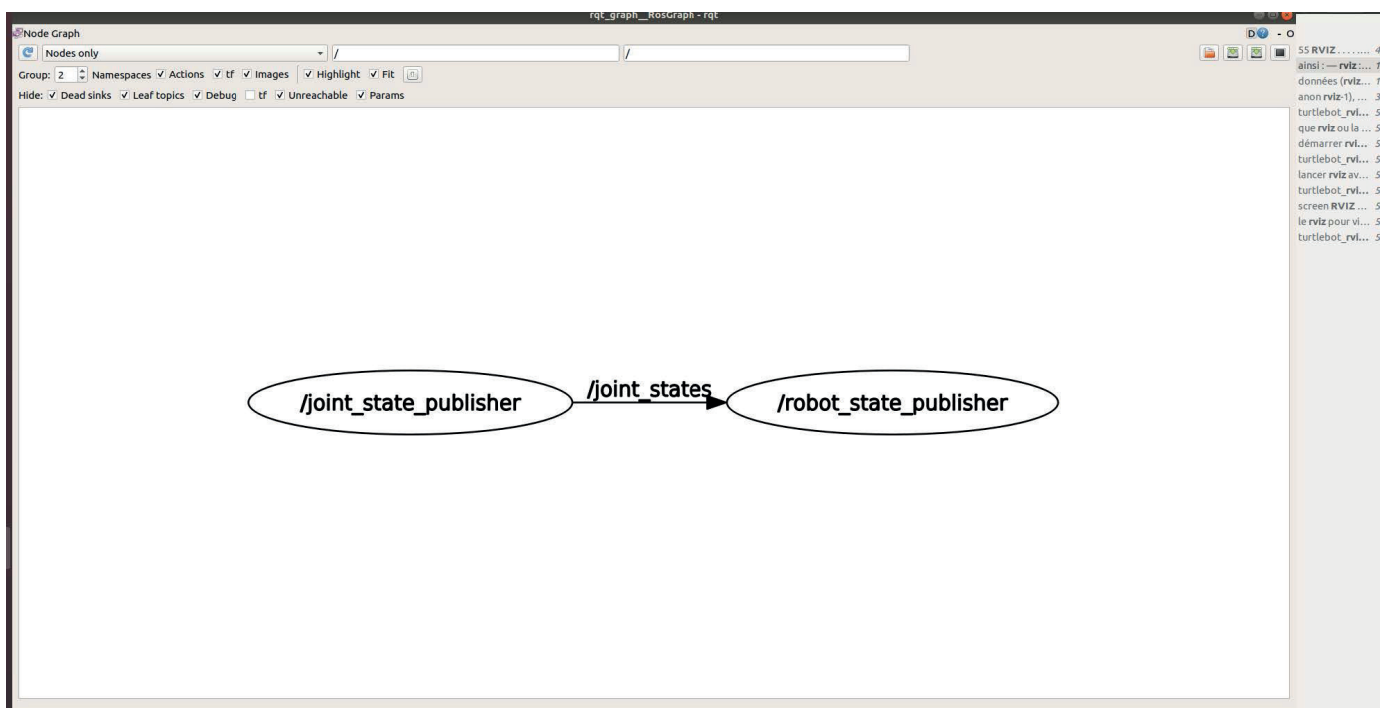
## VISUALISER LE GRAPHE D'INTERACTIONS ROS

Dans un nouveau terminal (laisser l'ancien actif), entrer la commande suivante :

```
$ rosrun rqt_graph rqt_graph
```

Une nouvelle fenêtre apparaît : *rqt\_graph*.

Celle-ci permet d'avoir une interface graphique des différents nœuds et topics (les liaisons entre deux nœuds dans lesquels circulent des données) actuellement actifs sous ROS. Vous pourrez vous en servir pour observer les différentes liaisons / interactions lors de vos développements sous ROS.



Dans rqt\_graph, les nœuds sont représentés par des ovales et les topics par des flèches reliant des nœuds.



## MISE EN PRATIQUE : NOEUDS ET TOPICS

Nous allons maintenant faire un petit cas pratique d'utilisation de service, nous allons observer la valeur du learning mode du robot (les moteurs sont coupés) et la modifier via un service.



*Le fichier `launch Desktop_rviz_simulation.launch` doit être en cours d'exécution (`roslaunch niryo_one_bringup desktop_rviz_simulation.launch`).*

Dans un nouveau terminal, lister les topics et remarquer le topic `"/niryo_one/learning_mode"` :

```
$ rostopic list
```

Ce topic renvoie la valeur, True ou False, suivant si le robot est en learning\_mode ou non. Le learning\_mode permet de désactiver les moteurs et de bouger librement le robot.

Ecouter ce topic et noter la valeur actuelle du learning mode publié par le topic.

```
$ rostopic echo /niryo_one/learning_mode
```

### La réponse sera de la forme : TRUE or FALSE

Dans un autre terminal, lister les services disponibles et remarquer le service `"/niryo_one/activate_learning_mode"`. Celui-ci permet d'activer ou de désactiver le learning mode.

Pour connaître le message envoyé à ce service et par conséquent quels paramètres donner à ce service, afficher les informations concernant ce service.

```
$ rosservice info /niryo_one/activate_learning_mode
```

Résultats obtenus :

```
Node: /niryo_one_driver
URI: rosrpc://hostname:port
Type: niryo_one_msgs/SetInt
Args: value
```

Ici le type de message-service est un SetInt. On peut observer dans args l'unique présence de `"value"`. Ainsi, le service ne prend en argument qu'un champs typé int `"value"` en paramètre.

Pour connaître les valeurs de retour, nous affichons via la commande :

```
$ rossrv show niryo_one_msgs/SetInt
```

Résultats obtenus :

```
int32 value
---
int32 status
string message
```

Dans ce même terminal, nous allons maintenant appeler ce service avec 0 pour désactiver ou 1 pour l'activer et remarquer le changement d'état du topic `"/niryo_one/learning_mode"` sur le précédent terminal.

```
$ rosservice call /niryo_one/activate_learning_mode "value: 0"
```



*Si vous avez une erreur de type "UnicodeEncodeError: 'ascii' codec can't encode character", veuillez effacer et retaper les guillemets manuellement, c'est un problème lors de la copie de la commande.*

## ESSAI : NOEUDS ET TOPICS

Alors que le launch file `"display.launch"` est encore lancé, afficher la liste des nœuds.

Afficher la liste de topics.

**Comparer les listes de topics et des nœuds avec le rqt graph obtenu dans la partie 1.**

Analyse de Topic `/joints_states` :

**Déterminez le type de message pour ce topic.**

**Quelle commande permet d'afficher le contenu de ce topic ?**

**Afficher le contenu de ce topic et commenter.**

**Bouger un ou plusieurs axes du robot, observer le contenu de topic puis commenter.**

Arrêter les topics lancés par la commande roslaunch en vous plaçant dans le terminal d'où il a été lancé, puis en faisant "Ctrl+c".

Exécuter la commande suivante dans un terminal (lancer la stack complète du Niryo one avec Rviz) :

```
$ roslaunch niryo_one_bringup desktop_rviz_simulation.launch
```

Donner la liste de nœuds.

Afficher la liste des topics.

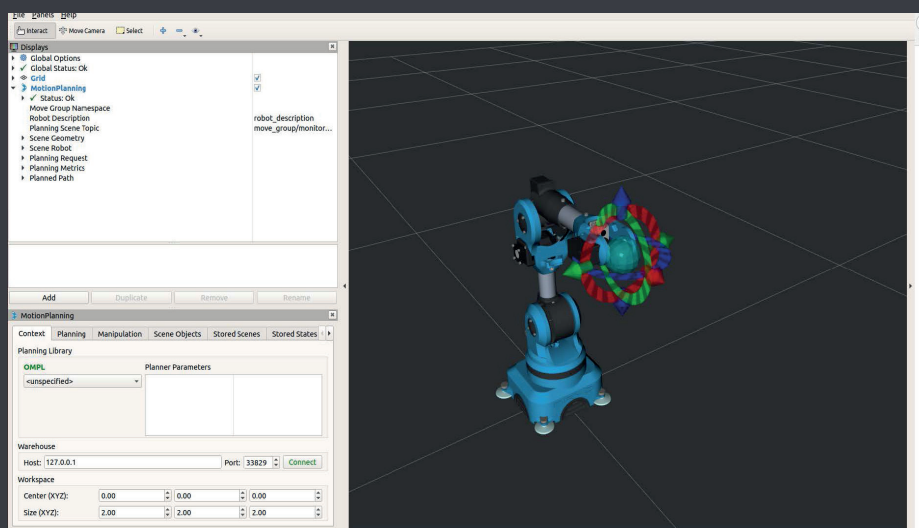
## DÉCOUVERTE DE MOVEIT

Moveit est un package servant principalement au contrôle de bras robots (calculs de trajectoires/exécution) et c'est un package orienté code / ROS mais il dispose tout de même d'une interface graphique de contrôle "rapide". Lorsque vous souhaitez obtenir des comportements spécifiques au niveau des trajectoires du robot, une familiarisation avec Moveit est nécessaire. Nous allons maintenant avoir une première approche de l'interface visuelle de Moveit.

## EXERCICE DE DÉCOUVERTE : MOVEIT

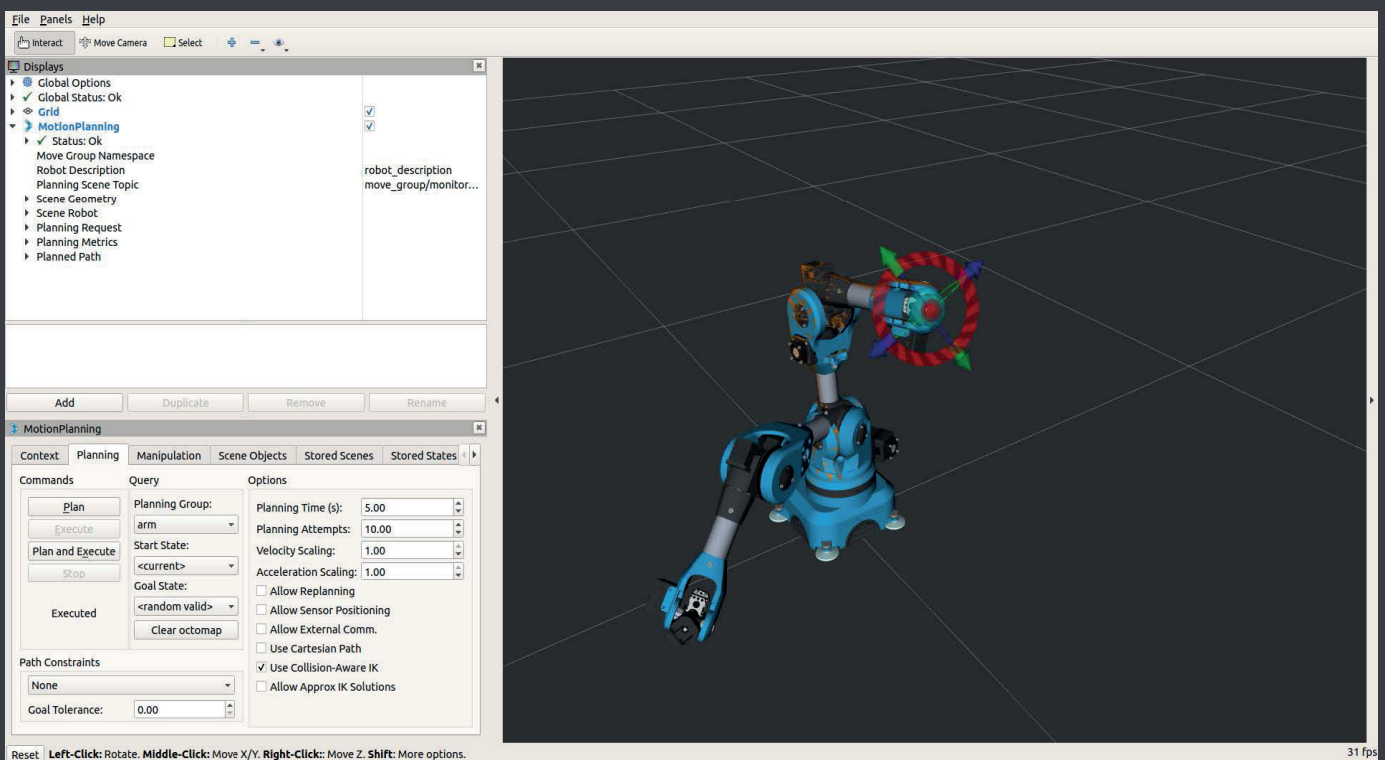
Nous allons maintenant arrêter tous les nœuds & launch en cours et lancer le launch file suivant pour ouvrir l'interface graphique de Moveit :

```
$ roslaunch niryo_one_moveit_config demo.launch
```



Il s'agit du plug-in MoveIt pour Rviz. Celui-ci nous permet de configurer des environnements virtuels (scènes), de créer des états de début et d'objectif pour le robot de manière interactive (aller d'un point A à un point B, de tester divers planificateurs de mouvements et de visualiser la sortie).

- Bouger le robot en utilisant le Rviz Motion planning. Pour ce faire, cliquer sur la boule bleue, maintenir enfoncé le bouton de la souris tout en déplaçant la boule. Relâcher le bouton à la position souhaitée.
- Lorsque vous faites cette manipulation, on peut voir en orange, la position finale du robot.
- Afin que le robot aille à la position demandée, dans la fenêtre "MotionPlanning", ouvrir l'onglet "Planning".



- Afin que le robot calcule la trajectoire pour atteindre la position et qu'il exécute la trajectoire calculée, cliquer sur "Plan and Execute".
- Modifier le planner à utiliser pour calculer la trajectoire dans l'onglet "Context", "Planning library" et choisir parmi la liste déroulante.

# MISE EN PRATIQUE : ROS EN LIGNE DE COMMANDES

Maintenant que nous avons vu les bases de ROS, nous allons les mettre en pratique lors de cet exercice pour effectuer quelques actions de mise en route sur le Niryo One via ROS.



*Il est important de vous connecter en SSH à votre robot (cf: notre TP Python). Pour la suite de l'exercice, les commandes seront effectuées par le biais de la connexion SSH établie avec le robot et non sur l'ordinateur.*

Lorsque notre robot s'allume, il n'est pas calibré et nous avons besoin de le calibrer pour effectuer des mouvements.

**Appeler le service `/niryo_one/calibrate_motors` avec la valeur "1" en argument pour calibrer le robot (penser à bien vérifier le(s) noms d'argument(s) d'entrée).**

**Écouter le topic `/niryo_one_/hardware_status` et observer la valeur du champs 'calibration\_needed' pour vérifier que la valeur passe à 0 une fois le robot calibré.**

Maintenant que notre robot est calibré, nous allons lui demander d'exécuter le programme autorun de test déjà présent dans le robot.

**Pour cela, appeler le service `/niryo_one/sequences/trigger_sequence_autorun` avec la valeur "1" (penser à bien vérifier le(s) argument(s) d'entrées).**

Après avoir vu les principaux aspects / termes de ROS et les outils & exécutable nécessaires pour pouvoir interagir avec une application ROS, nous allons ensuite interagir avec ROS en programmation Python.

## INTERAGIR AVEC ROS EN PYTHON

Dans la première partie de ce TP, nous avons vu comment interagir avec les services/topics ROS en ligne de commande.

Maintenant, nous allons interagir avec ROS via programmation en code. ROS étant un framework disponible dans plusieurs langages (Python, C++, Lisp, Lua, ...), nous choisirons ici Python car c'est le langage utilisé sur NiryoOne et qu'il est relativement abordable et permissif pour une première approche de ROS.

# EXERCICE DE DÉCOUVERTE : ROS EN PYTHON

Nous allons à présent créer un nœud en Python permettant d'envoyer une position au robot sur le topic `"/niryone_joint_trajectory_controller/command"` lorsque celui-ci reçoit un message sur un topic que nous allons créer. **Déterminer quel est le type de message du topic `"/niryone_joint_trajectory_controller/command"`.**

## CRÉATION DU NOEUD SUBSCRIBER

- Se rendre dans le dossier :

```
$ cd ~/catkin_ws/src/niryone_ros/niryone_commander/scripts
```

- Créer un fichier python nommé `my_node.py`
- Afin de rendre le fichier exécutable, taper les commandes suivantes sur un terminal :

```
$ chmod a+x my_node.py
```

- Puis ouvrir `my_node.py` avec un éditeur de texte (gedit par exemple) et copier/coller les lignes suivantes (veiller à la bonne indentation du code, celle-ci pouvant être affectée lors d'un copier/coller depuis le document PDF) :

```
#!/usr/bin/env python

import rospy

from std_msgs.msg import String

def myTopicCallback(msg):
    rospy.loginfo("I heard %s", msg.data)

def my_script():
    rospy.init_node('my_node', anonymous=True)
    rospy.Subscriber("my_topic", String, myTopicCallback)
    rospy.spin()
```

```
if __name__ == '__main__':  
    my_script()
```

Explications :

- Au lancement du script, la fonction `main` est appelée et appelle elle-même la fonction `my_script()`.
- `rospy.init_node` permet de créer un nouveau nœud que l'on nomme ici `my_node`.
- `rospy.Subscriber` permet de préciser le nom du topic ainsi que le type de message auquel nous souhaitons souscrire. Ici, nous créons un topic "`my_topic`" qui est de type `String` (une chaîne de caractères). La fonction "`myTopicCallback`" sera appelée à chaque nouveau message reçu sur ce topic.
- `rospy.spin()` permet de garder le nœud actif.
- `MyTopicCallback` est la fonction appelée à chaque nouveau message. Le message reçu est stocké dans le paramètre `msg` et donc `rospy.loginfo` permet d'afficher le message reçu et donc la valeur direct du champs "`data`".

Essayons à présent notre nœud :

- Dans un terminal, lancer le nœud master :

```
$ roscore
```

- Dans un autre terminal, lancer votre nœud :

```
$ rosrun niryo_one_commander my_node.py
```

- Dans un nouveau terminal, afficher la liste des nœuds et la liste des topics. Remarquez que votre nœud, ainsi que votre topic sont bien présents.
- Envoyer un message sur le topic `/my_topic` en rentrant la commande suivante :

```
$ rostopic pub /my_topic std_msgs/String "data: 'your_message'"
```

- Dans le terminal de votre nœud, vous devriez voir s'afficher :

```
[INFO] [1569318487.592693]: I heard your_message
```

## CRÉATION DU NOEUD PUBLISHER

Nous allons maintenant améliorer notre nœud pour qu'il envoie une commande au robot lorsqu'il reçoit un message.

- Ouvrir le script python *my\_node.py*
- Copier/coller les nouvelles lignes :

```
#!/usr/bin/env python

import rospy

from std_msgs.msg import String

from trajectory_msgs.msg import JointTrajectory

from trajectory_msgs.msg import JointTrajectoryPoint

pub = rospy.Publisher('/niryo_one_follow_joint_trajectory_\\
controller/command', JointTrajectory, queue_size=10)

def myTopicCallback(data):

    rospy.loginfo("I heard %s", data.data)

    joint_trajectory = JointTrajectory()

    joint_trajectory.header.stamp = rospy.Time.now()

    joint_trajectory.joint_names.append("joint_1")

    joint_trajectory.joint_names.append("joint_2")

    joint_trajectory.joint_names.append("joint_3")

    joint_trajectory.joint_names.append("joint_4")

    joint_trajectory.joint_names.append("joint_5")

    joint_trajectory.joint_names.append("joint_6")

    joint_trajectory_point = JointTrajectoryPoint()

    joint_trajectory_point.time_from_start.secs = 1
```



```

    joint_trajectory_point.positions.append(0.2)

    joint_trajectory_point.positions.append(0.5)

    joint_trajectory_point.positions.append(0.0)

    joint_trajectory_point.positions.append(0.1)

    joint_trajectory_point.positions.append(0.4)

    joint_trajectory_point.positions.append(0.0)

    join_trajectory.points.append(joint_trajectory_point)

    pub.publish(join_trajectory)

def my_script():

    rospy.init_node('my_node', anonymous=True)

    rospy.Subscriber("my_topic", String, myTopicCallback)

    rospy.spin()

if __name__ == '__main__':

    my_script()

```

#### Explications :

- *rospy.Publisher* permet de choisir le topic sur lequel la commande sera envoyée en spécifiant le type de message et la taille de la file d'attente (pas important dans ce TP). La variable *pub* est une instance de la fonction *rospy.Publisher* qui nous permettra de l'utiliser pour envoyer la commande.
- Le topic */niryo\_one\_follow\_joint\_trajectory\_controller/command* est de type *JointTrajectory*. Les variables *joint\_trajectory* et *joint\_trajectory\_point* permettent de créer un message du même type que le topic et de définir les commandes que l'on souhaite envoyer. Afin que la commande soit correctement interprétée, on doit remplir certains paramètres comme :
  - *joint\_trajectory.joint\_name* avec les noms des différentes articulations du robot.
  - *joint\_trajectory.points* avec la valeur des angles que chaque articulations doit

atteindre.

Essayons à présent notre nœud. Dans un terminal, lancer le launch file :

```
$ roslaunch niryo_one_bringup desktop_rviz_simulation.launch
```

- Dans un autre terminal, lancer votre nœud :

```
$ rosrun niryo_one_commander my_node.py
```

- Dans un nouveau terminal, afficher la liste des nœuds et la liste des topics. Remarquez que votre nœud, ainsi que votre topic sont bien présents.
- Envoyer un message sur le topic `/my_topic` en rentrant la commande suivante :

```
$ rostopic pub /my_topic std_msgs/String "data: 'your_message'"
```

- Dans le terminal de votre nœud, vous devriez voir s'afficher :

```
[INFO] [1569318487.592693]: I heard your_message
```

- Dans la fenêtre RViz, vous devriez voir le robot bouger.

## AMÉLIORATION DE VOTRE NŒUD

Maintenant, nous allons modifier notre subscriber pour qu'il prenne en paramètre un double et modifier notre publisher pour qu'il ajoute cet offset à chaque joint de position envoyé.

Commencer par arrêter l'exécution de votre nœud précédent.

Ensuite, nous allons devoir importer le type 'Float64' (double pour ROS) et adapter la fonction de callback de notre subscriber de la manière suivante :

```
#!/usr/bin/env python

import rospy

from std_msgs.msg import Float64

from trajectory_msgs.msg import JointTrajectory

from trajectory_msgs.msg import JointTrajectoryPoint
```

```

pub = rospy.Publisher(\
'/niryo_one_follow_joint_trajectory_controller/command',\
JointTrajectory, queue_size=10)

def myTopicCallback(msg):
    offset_value = msg.data
    rospy.loginfo("Offset value: {}".format(offset_value))

    joint_trajectory = JointTrajectory()
    joint_trajectory.header.stamp = rospy.Time.now()

    joint_trajectory.joint_names.append("joint_1")
    joint_trajectory.joint_names.append("joint_2")
    joint_trajectory.joint_names.append("joint_3")
    joint_trajectory.joint_names.append("joint_4")
    joint_trajectory.joint_names.append("joint_5")
    joint_trajectory.joint_names.append("joint_6")

    joint_trajectory_point = JointTrajectoryPoint()
    joint_trajectory_point.time_from_start.secs = 1

    joint_trajectory_point.positions.append(0.2 + offset_value)
    joint_trajectory_point.positions.append(0.5 + offset_value)
    joint_trajectory_point.positions.append(0.0 + offset_value)
    joint_trajectory_point.positions.append(0.1 + offset_value)

    joint_trajectory_point.positions.append(0.4 + offset_value)
    joint_trajectory_point.positions.append(0.0 + offset_value)

```

```
join_trajectory.points.append(joint_trajectory_point)
pub.publish(join_trajectory)
```

**Modifier le reste du code pour que votre subscriber prenne un Float64 en type de message publié et non une String comme précédemment.**

Une fois ces changements effectués, si votre nœud est relancé comme précédemment :

```
$ rosrun niryo_one_commander my_node.py
```

Et que le subscriber est publié de la manière suivante, votre robot devrait bouger en fonction de l'offset rentré et la valeur d'offset sera affichée (l'offset doit être de petite valeur entre  $\sim -0.4/0.4$  vu qu'il est appliqué à tous les joints).

```
$ rostopic pub /my_topic std_msgs/Float64 "data: 0.1"
```

Retrouvez tous nos supports pédagogiques sur

**[www.niryo.com](http://www.niryo.com)**

# ESSAI : NOEUDS ET TOPICS

## CORRECTION

Alors que le launch file "*display.launch*" est encore lancé, afficher la liste des nœuds.

Afficher la liste de topics.

```
rostopic list
```

```
/clicked_point  
/initialpose  
/joint_states  
/move_base_simple/goal  
/rosout  
/rosout_agg  
/tf  
/tf_static
```

Analyse de Topic */joints\_states* :

```
cmd : rostopic info /joint_states  
Type: sensor_msgs/JointState
```

```
rostopic echo /joint_states
```

```
header:  
  seq: 1989  
  stamp:  
    secs: 1600763254
```

```
nsecs: 427649974
  frame_id: "
name:
- joint_1
- joint_2
- joint_3
- joint_4
- joint_5
- joint_6
position: [0.0, -0.00033675939999988636, -0.0003149794752399515, 0.0,
-0.0003330409999999784, 0.0]
velocity: []
effort: []
```

On voit l'ensemble des joints qui composent le robot. Pour chaque joint, on observe sa position. On constate aussi que la valeur de vitesse et d'effort sont à 0.

En modifiant les valeurs des axes, la valeur de position des joints sur le topics est modifiée.

Arrêter les topics lancés par la commande roslaunch en vous plaçant dans le terminal d'où il a été lancé, puis en faisant "Ctrl+c".

Exécuter la commande suivante dans un terminal (lancer la stack complète du Niryo one avec Rviz) :

```
$ roslaunch niryo_one_bringup desktop_rviz_simulation.launch
```

Donner la liste de nœuds.

```
cmd :
rosnode list
rsp :
/controller_spawner
/move_group/niryo_one_driver
```

```
/niryo_one_tools
/robot_state_publisher
/rosapi
/rosbridge_websocket
/rosout
/rviz
/tf2_web_republisher
/user_interface
```

Afficher la liste des topics.

```
cmd :
  rostopic list
rsp :
/attached_collision_object
/clicked_point
/client_count
/collision_object
/connected_clients
/execute_trajectory/cancel
/execute_trajectory/feedback
/execute_trajectory/goal
/execute_trajectory/result
/execute_trajectory/status
/initialpose
/joint_states
/joy
/move_base_simple/goal
/move_group/cancel
/move_group/display_contacts
/move_group/display_planned_path
/move_group/feedback
/move_group/goal
/move_group/monitored_planning_scene
/move_group/ompl/parameter_descriptions
/move_group/ompl/parameter_updates
/move_group/plan_execution/parameter_descriptions
/move_group/plan_execution/parameter_updates/move_group/planning_scene_monitor/
```

**parameter\_descriptions**  
**/move\_group/planning\_scene\_monitor/parameter\_updates**  
**/move\_group/result**  
**/move\_group/sense\_for\_plan/parameter\_descriptions**  
**/move\_group/sense\_for\_plan/parameter\_updates**  
**/move\_group/status**  
**/move\_group/trajectory\_execution/parameter\_descriptions**  
**/move\_group/trajectory\_execution/parameter\_updates**  
**/niryo\_one/blockly/break\_point**  
**/niryo\_one/blockly/highlight\_block**  
**/niryo\_one/blockly/save\_current\_point**  
**/niryo\_one/current\_tool\_id**  
**/niryo\_one/hardware\_status**  
**/niryo\_one/joystick\_interface/is\_enabled**  
**/niryo\_one/learning\_mode**  
**/niryo\_one/robot\_state**  
**/niryo\_one/rpi/digital\_io\_state**  
**/niryo\_one/sequences/execute/cancel**  
**/niryo\_one/sequences/execute/feedback**  
**/niryo\_one/sequences/execute/goal**  
**/niryo\_one/sequences/execute/result**  
**/niryo\_one/sequences/execute/status**  
**/niryo\_one/sequences/sequence\_autorun\_status**  
**/niryo\_one/software\_version**  
**/niryo\_one/steppers\_reset\_controller**  
**/niryo\_one/tool\_action/cancel**  
**/niryo\_one/tool\_action/feedback**  
**/niryo\_one/tool\_action/goal**  
**/niryo\_one/tool\_action/result**  
**/niryo\_one/tool\_action/status**  
**/niryo\_one\_follow\_joint\_trajectory\_controller/command**  
**/niryo\_one\_follow\_joint\_trajectory\_controller/follow\_joint\_trajectory/cancel**  
**/niryo\_one\_follow\_joint\_trajectory\_controller/follow\_joint\_trajectory/feedback**  
**/niryo\_one\_follow\_joint\_trajectory\_controller/follow\_joint\_trajectory/goal**  
**/niryo\_one\_follow\_joint\_trajectory\_controller/follow\_joint\_trajectory/result**  
**/niryo\_one\_follow\_joint\_trajectory\_controller/follow\_joint\_trajectory/status**  
**/niryo\_one\_follow\_joint\_trajectory\_controller/state**  
**/niryo\_one\_matlab/command**  
**/niryo\_one\_matlab/result**  
**/pickup/cancel/pickup/feedback**



```
/pickup/goal
/pickup/result
/pickup/status
/place/cancel
/place/feedback
/place/goal
/place/result
/place/status
/planning_scene
/planning_scene_world
/rosout
/rosout_agg
/tf
/tf2_web_republisher/cancel
/tf2_web_republisher/feedback
/tf2_web_republisher/goal
/tf2_web_republisher/result
/tf2_web_republisher/status
/tf_static
/trajjectory_execution_event
```

## MISE EN PRATIQUE : ROS EN LIGNE DE COMMANDES CORRECTION

Lorsque notre robot s'allume, il n'est pas calibré et nous avons besoin de le calibrer pour effectuer des mouvements.

Appeler le service `/niryo_one/calibrate_motors` avec la valeur "1" en argument pour calibrer le robot (penser à bien vérifier le(s) noms d'argument(s) d'entrée).

```
$ rosservice info /niryo_one/calibrate_motors
```

```
Node: /niryo_one_driver
```

```
URI: rosrpc://hostname:port
```

```
Type: niryo_one_msgs/SetInt
```

Args: value

```
$ rossrv show niryo_one_msgs/SetInt
```

int32 value

---

int32 status

string message

```
$ rosservice call /niryo_one/calibrate_motors "value: 1"
```

Écouter le topic `/niryo_one/hardware_status` et observer la valeur du champs 'calibration\_needed' pour vérifier que la valeur passe à 0 une fois le robot calibré.

```
$ rostopic echo /niryo_one/hardware_status
```

header:

seq: 10589

stamp:

secs: 1600987284

nsecs: 854874751

frame\_id: "

rpi\_temperature: 0

hardware\_version: 2

connection\_up: True

error\_message: "

calibration\_needed: 0

calibration\_in\_progress: False

motor\_names: []

motor\_types: []

temperatures: []

voltages: []

hardware\_errors: []

---

...

Maintenant que notre robot est calibré, nous allons lui demander d'exécuter le programme autorun de test déjà présent dans le robot.

Pour cela, appeler le service `/niryo_one/sequences/trigger_sequence_autorun` avec la valeur "1" (penser à bien vérifier le(s) argument(s) d'entrées).

```
$ rosservice info /niryo_one/sequences/trigger_sequence_autorun
```

```
Node: /user_interface
```

```
URI: rosrpc://hostname:port
```

```
Type: niryo_one_msgs/SetInt
```

```
$ rossrv
```

```
$ rosservice call /niryo_one/sequences/trigger_sequence_autorun "value: 1"
```

```
status: 200
```

```
message: "Sequence Autorun has been activated"
```

## EXERCICE DE DÉCOUVERTE : ROS EN PYTHON CORRECTION

Nous allons à présent créer un nœud en Python permettant d'envoyer une position au robot sur le topic `/niryo_one_joint_trajectory_controller/command` lorsque celui-ci reçoit un message sur un topic que nous allons créer.

Déterminer quel est le type de message du topic `/niryo_one_joint_trajectory_controller/command`

Commande :

```
rostopic type /niryo_one_follow_joint_trajectory_controller/command
```

Type du message :

```
trajectory_msgs/JointTrajectory
```

Déterminer quel est le type de message du topic `"/niryo_one_joint_trajectory_controller/command"`.

## CRÉATION DU NOEUD SUBSCRIBER

- Se rendre dans le dossier :

```
$ cd ~/catkin_ws/src/niryo_one_ros/niryo_one_commander/  
scripts
```

- Créer un fichier python nommé `my_node.py`
- Afin de rendre le fichier exécutable, taper les commandes suivantes sur un terminal :

```
$ chmod a+x my_node.py
```

- Puis ouvrir `my_node.py` avec un éditeur de texte (gedit par exemple) et copier/coller les lignes suivantes :

```
#!/usr/bin/env python  
  
import rospy  
  
from std_msgs.msg import String  
  
def myTopicCallback(msg):  
    rospy.loginfo("I heard %s", msg.data)  
  
def my_script():  
    rospy.init_node('my_node', anonymous=True)  
    rospy.Subscriber("my_topic", String, myTopicCallback)  
    rospy.spin()  
  
if __name__ == '__main__':  
    my_script()
```

Explications :

- Au lancement du script, la fonction `main` est appelée et appelle elle-même la fonction `my_script()`.
- `rospy.init_node` permet de créer un nouveau nœud que l'on nomme ici `my_node`.
- `rospy.Subscriber` permet de préciser le nom du topic ainsi que le type de message auquel nous souhaitons souscrire. Ici, nous créons un topic "`my_topic`" qui est de type `String` (une chaîne de caractères). La fonction "`myTopicCallback`" sera appelée à chaque nouveau message reçu sur ce topic.
- `rospy.spin()` permet de garder le nœud actif.
- `MyTopicCallback` est la fonction appelée à chaque nouveau message. Le message reçu est stocké dans le paramètre `msg` et donc `rospy.loginfo` permet d'afficher le message reçu et donc la valeur direct du champs "`data`".

Essayons à présent notre nœud :

- Dans un terminal, lancer le nœud master :

```
$ roscore
```

- Dans un autre terminal, lancer votre nœud :

```
$ rosrun niryo_one_commander my_node.py
```

- Dans un nouveau terminal, afficher la liste des nœuds et la liste des topics. Remarquez que votre nœud, ainsi que votre topic sont bien présents.
- Envoyer un message sur le topic `/my_topic` en rentrant la commande suivante :

```
$ rostopic pub /my_topic std_msgs/String "data: 'votre_
message'"
```

- Dans le terminal de votre nœud, vous devriez voir s'afficher :

```
[INFO] [1569318487.592693]: I heard votre_message
```

## CRÉATION DU NŒUD PUBLISHER

Nous allons maintenant améliorer notre nœud pour qu'il envoie une commande au robot lorsqu'il reçoit un message.

- Ouvrir le script python `my_node.py`

- Copier/coller les nouvelles lignes :

```
#!/usr/bin/env python

import rospy

from std_msgs.msg import String

from trajectory_msgs.msg import JointTrajectory

from trajectory_msgs.msg import JointTrajectoryPoint

pub = rospy.Publisher('/niryo_one_follow_joint_trajectory_\  
controller/command', JointTrajectory, queue_size=10)

def myTopicCallback(data):

    rospy.loginfo("I heard %s", data.data)

    joint_trajectory = JointTrajectory()

    joint_trajectory.header.stamp = rospy.Time.now()

    joint_trajectory.joint_names.append("joint_1")
    joint_trajectory.joint_names.append("joint_2")
    joint_trajectory.joint_names.append("joint_3")
    joint_trajectory.joint_names.append("joint_4")
    joint_trajectory.joint_names.append("joint_5")
    joint_trajectory.joint_names.append("joint_6")

    joint_trajectory_point = JointTrajectoryPoint()
    joint_trajectory_point.time_from_start.secs = 1

    joint_trajectory_point.positions.append(0.2)
    joint_trajectory_point.positions.append(0.5)
    joint_trajectory_point.positions.append(0.0)
    joint_trajectory_point.positions.append(0.1)
```

```

    joint_trajectory_point.positions.append(0.4)

    joint_trajectory_point.positions.append(0.0)

    join_trajectory.points.append(joint_trajectory_point)

    pub.publish(join_trajectory)

def my_script():

    rospy.init_node('my_node', anonymous=True)

    rospy.Subscriber("my_topic", String, myTopicCallback)

    rospy.spin()

if __name__ == '__main__':

    my_script()

```

#### Explications :

- *rospy.Publisher* permet de choisir le topic sur lequel la commande sera envoyée en spécifiant le type de message et la taille de la file d'attente (pas important dans ce TP). La variable *pub* est une instance de la fonction *rospy.Publisher* qui nous permettra de l'utiliser pour envoyer la commande.
- Le topic */niryo\_one\_follow\_joint\_trajectory\_controller/command* est de type *JointTrajectory*. Les variables *joint\_trajectory* et *joint\_trajectory\_point* permettent de créer un message du même type que le topic et de définir les commandes que l'on souhaite envoyer. Afin que la commande soit correctement interprétée, on doit remplir certains paramètres comme :
  - *joint\_trajectory.joint\_name* avec les noms des différentes articulations du robot.
  - *joint\_trajectory.points* avec la valeur des angles que chaque articulations doit atteindre.

Essayons à présent notre nœud :

- Dans un terminal, lancer le launch file :

```
$ roslaunch niryo_one_bringup desktop_rviz_simulation.launch
```

- Dans un autre terminal, lancer votre nœud :

```
$ rosrun niryo_one_commander my_node.py
```

- Dans un nouveau terminal, afficher la liste des nœuds et la liste des topics. Remarquez que votre nœud, ainsi que votre topic sont bien présents.
- Envoyer un message sur le topic `/my_topic` en rentrant la commande suivante :

```
$ rostopic pub /my_topic std_msgs/String "data: 'votre_
message'"
```

- Dans le terminal de votre nœud, vous devriez voir s'afficher :

```
[INFO] [1569318487.592693]: I heard votre_message
```

- Dans la fenêtre RViz, vous devriez voir le robot bouger

## AMÉLIORATION DE VOTRE NŒUD

Maintenant, nous allons modifier notre subscriber pour qu'il prenne en paramètre un double et modifier notre publisher pour qu'il ajoute cet offset à chaque joint de position envoyé.

Commencer par arrêter l'exécution de votre nœud précédent.

Ensuite, nous allons devoir importer le type 'Float64' (double pour ROS) et adapter la fonction de callback de notre subscriber de la manière suivante :

```
#!/usr/bin/env python

import rospy

from std_msgs.msg import Float64

from trajectory_msgs.msg import JointTrajectory

from trajectory_msgs.msg import JointTrajectoryPoint

pub = rospy.Publisher(\

'/niryo_one_follow_joint_trajectory_controller/command',\

JointTrajectory, queue_size=10)
```



```

def myTopicCallback(msg):
    offset_value = msg.data
    rospy.loginfo("Offset value: {}".format(offset_value))

    joint_trajectory = JointTrajectory()
    joint_trajectory.header.stamp = rospy.Time.now()

    joint_trajectory.joint_names.append("joint_1")
    joint_trajectory.joint_names.append("joint_2")
    joint_trajectory.joint_names.append("joint_3")
    joint_trajectory.joint_names.append("joint_4")
    joint_trajectory.joint_names.append("joint_5")
    joint_trajectory.joint_names.append("joint_6")

    joint_trajectory_point = JointTrajectoryPoint()
    joint_trajectory_point.time_from_start.secs = 1

    joint_trajectory_point.positions.append(0.2 + offset_value)
    joint_trajectory_point.positions.append(0.5 + offset_value)
    joint_trajectory_point.positions.append(0.0 + offset_value)
    joint_trajectory_point.positions.append(0.1 + offset_value)

    joint_trajectory_point.positions.append(0.4 + offset_value)
    joint_trajectory_point.positions.append(0.0 + offset_value)

    joint_trajectory.points.append(joint_trajectory_point)
    pub.publish(joint_trajectory)

```

Modifier le reste du code pour que votre subscriber prenne un Float64 en type de message

publié et non une String comme précédemment.

```
#!/usr/bin/env python

import rospy

from std_msgs.msg import Float64

from trajectory_msgs.msg import JointTrajectory

from trajectory_msgs.msg import JointTrajectoryPoint

pub = rospy.Publisher(\
'/niryo_one_follow_joint_trajectory_controller/command',\
JointTrajectory, queue_size=10)

def myTopicCallback(msg):

    offset_value = msg.data

    rospy.loginfo("Offset value: {}".format(offset_value))

    joint_trajectory = JointTrajectory()

    joint_trajectory.header.stamp = rospy.Time.now()

    joint_trajectory.joint_names.append("joint_1")

    joint_trajectory.joint_names.append("joint_2")

    joint_trajectory.joint_names.append("joint_3")

    joint_trajectory.joint_names.append("joint_4")

    joint_trajectory.joint_names.append("joint_5")

    joint_trajectory.joint_names.append("joint_6")

    joint_trajectory_point = JointTrajectoryPoint()

    joint_trajectory_point.time_from_start.secs = 1
```

```
joint_trajectory_point.positions.append(0.2 + offset_value)
joint_trajectory_point.positions.append(0.5 + offset_value)
joint_trajectory_point.positions.append(0.0 + offset_value)
joint_trajectory_point.positions.append(0.1 + offset_value)
joint_trajectory_point.positions.append(0.4 + offset_value)
joint_trajectory_point.positions.append(0.0 + offset_value)

join_trajectory.points.append(joint_trajectory_point)

pub.publish(join_trajectory)
```

```
def my_script():
    rospy.init_node('my_node', anonymous=True)
    rospy.Subscriber("my_topic", Float64, myTopicCallback)
    rospy.spin()

if __name__ == '__main__':
    my_script()
```

Une fois ces changements effectués, si votre nœud est relancé comme précédemment :

```
$ rosrun niryo_one_commander my_node.py
```

Et que le subscriber est publié de la manière suivante, votre robot devrait bouger en fonction de l'offset rentré et la valeur d'offset sera affichée (l'offset doit être de petite valeur entre ~ -0.4/0.4 vu qu'il est appliqué à tous les joints).

```
$ rostopic pub /my_topic std_msgs/Float64 "data: 0.1"
```