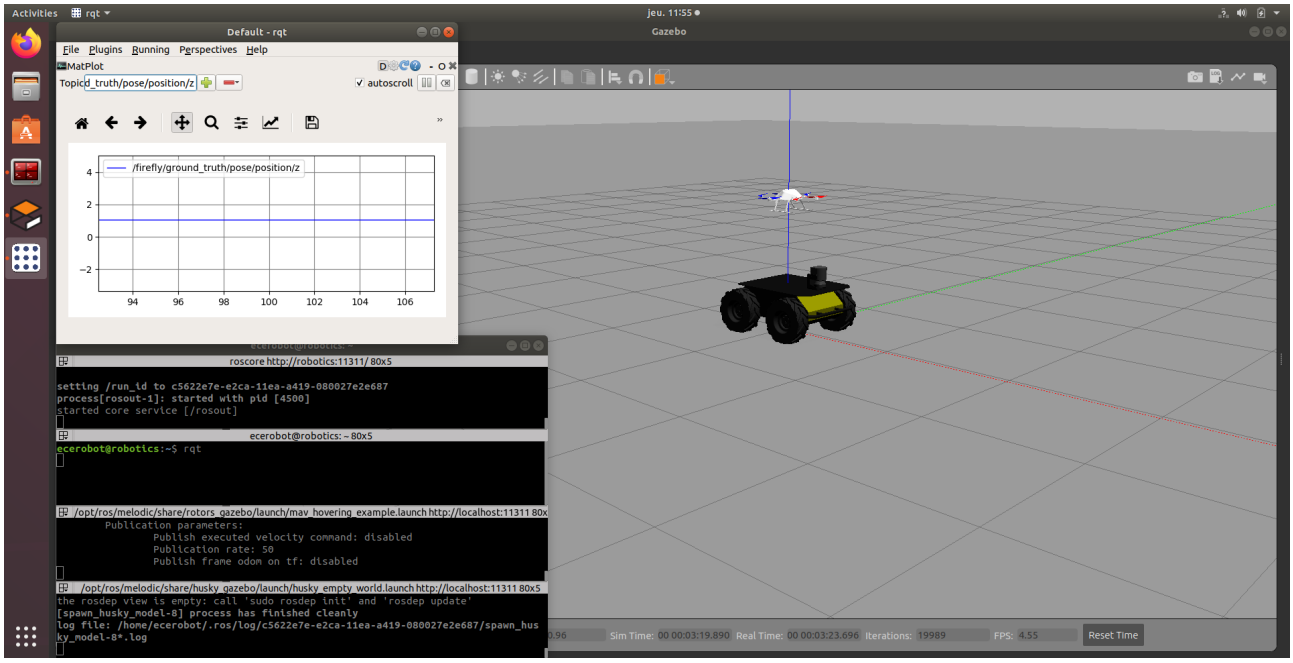
Introduction à ROS



1 Introduction

Cet atelier vise à prendre en main l'outil informatique open-source ROS (Robotic Operating System), devenu un standard dans les laboratoires universitaires et chez les industriels de la robotique.

ROS permet de faire communiquer le matériel (capteurs, actionneurs) et des algorithmes embarqués via des interfaces standardisées. De nombreux outils de visualisation, simulation et interaction sont disponibles, ainsi qu'une large communauté d'utilisateurs qui diffuse en ligne des outils utilisables par tous.

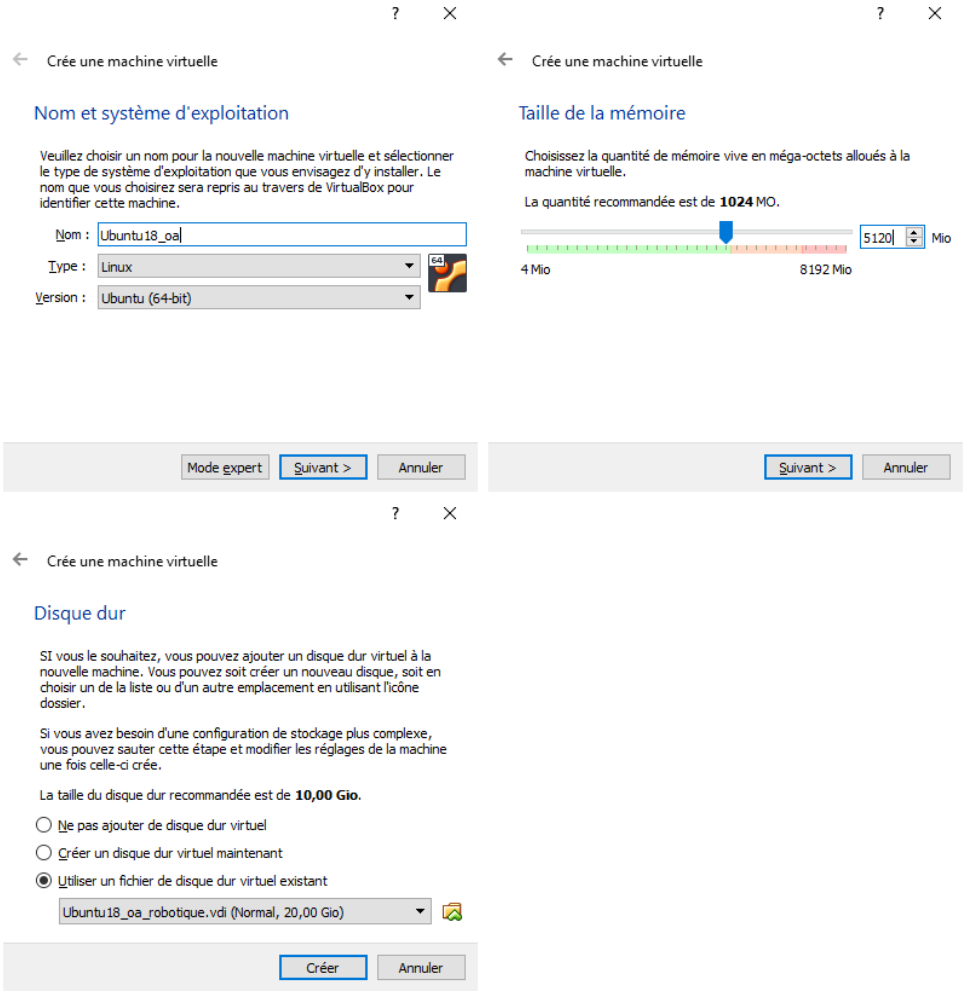
ROS supporte les développements en **Python et C++**. L'une des forces de ce middleware est par ailleurs de pouvoir faire communiquer des programmes écrits en différents langages, sans modification de ceux-ci.

Quelques liens utiles :

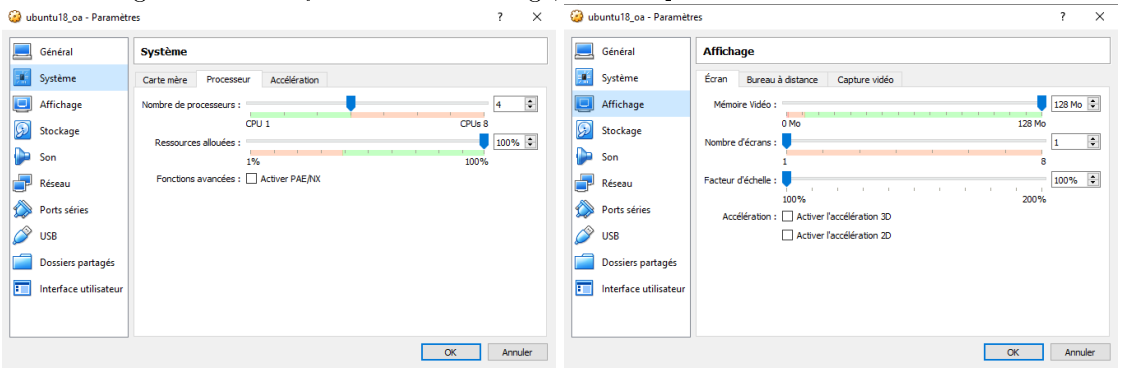
- Le wiki ROS : `http://wiki.ros.org/`
- Quelques généralités : `http://wiki.ros.org/ROS/Introduction` ; `http://wiki.ros.org/ROS/Concepts`
- Le TP suit en partie les tutoriels ROS : `http://wiki.ros.org/ROS/Tutorials#Beginner_Level`
- Le système de compilation catkin-tools : `http://catkin-tools.readthedocs.io/`
- L'interface graphique Node Manager : `http://wiki.ros.org/node_manager_fkie`

2 Installation Machine Virtuelle (tous OS)

1. Installer Oracle VirtualBox.
2. Télécharger la machine virtuelle **Ubuntu18_rob.vdi** à l'adresse fournie.
3. Dans VirtualBox, créer une nouvelle machine virtuelle avec les options suivantes :



4. Dans Configuration ⟶ Système ⟶ Processeurs, augmenter le nombre de CPU au maximum autorisé. Dans Configuration ⟶ Système ⟶ Affichage, vérifier que la mémoire vidéo est au maximum.



5. Lancer la machine virtuelle. Le login est **robot** et le mot de passe **robot**.
6. Une fois la machine lancée, activer *Périphériques* ⟶ *Insérer l'image CD des Additions invitées* et suivre les étapes demandées (*Run*, rentrer le mot de passe *robot*, et redémarrer).

3 Prise en main ROS

La machine virtuelle fournie est une distribution Ubuntu, avec ROS pré-installé.

La plupart des travaux se lancent depuis un terminal Linux (raccourci clavier : **Ctrl** + **Alt** + **T**).
Liste des commandes Linux usuelles (regarder en particulier **cd**, **pwd**, **ls**, **ll**, **gedit**) : `http://doc.ubuntu-fr.org/tutoriel/console_commandes_de_base`.

L'outil « Terminator » permet d'avoir plusieurs terminaux dans une seule fenêtre. Utiliser **Ctrl**+ **Shift** + **E** pour séparer verticalement, **Ctrl** + **Shift** + **O** pour séparer horizontalement et **Ctrl** + **Shift** + **W** pour fermer une des sous-fenêtres.

Autocomplétion : avec la touche **Tabulation**, les commandes existantes sont proposées (les programmes ROS disponibles également).

Pour arrêter un programme dans un terminal, utiliser **Ctrl** +**C**.

Pour conserver une fenêtre toujours visible (l'interface graphique des simulateurs, par exemple), faire un clic droit sur la barre de celle-ci et sélectionner « Always on top ».

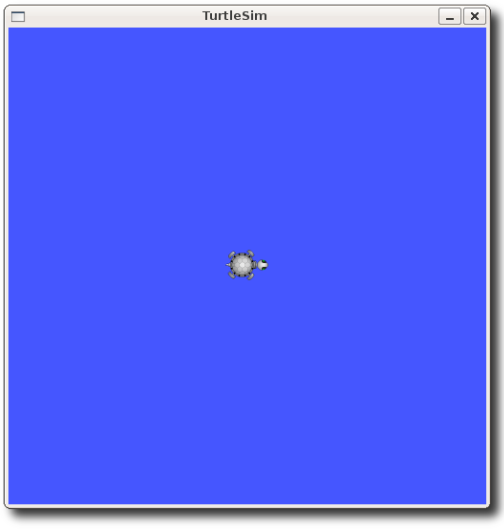
3.1 Nœuds

Pour que les commandes ROS puissent s'exécuter, il faut avoir lancé la commande **roscore** dans un terminal, et le laisser actif (travailler dans un autre terminal).

On peut visualiser la liste des nœuds actifs avec la commande **rosnode list** (dans un autre terminal).

Nous allons tout d'abord travailler avec le programme « turtlesim » qui permet de déplacer une tortue dans un espace 2D.

La commande **rosrun [package] [node]** permet de lancer le nœud **[node]** du package **[package]**, (un package peut contenir plusieurs nœuds). Dans un nouveau terminal, lancer le nœud **turtlesim_node** du package **turtlesim**, cela devrait faire apparaître la fenêtre suivante.



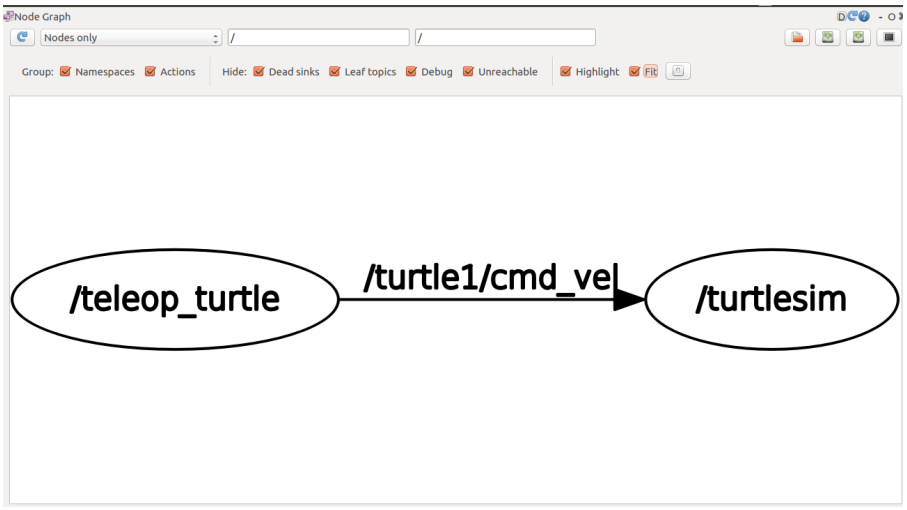
Dans un nouveau terminal, lancer le nœud **turtle_teleop_key**, toujours du package **turtlesim**. Lorsque ce terminal est actif (clic gauche souris dedans si ce n'est pas le cas), il est possible de déplacer la tortue du **turtlesim_node** ouvert précédemment.

3.2 Topics et visualisation

Il y a donc maintenant 2 nœuds qui s'exécutent et communiquent entre eux via des **topics**.

L'outil graphique de visualisation ROS se lance en exécutant la commande **rqt** dans un nouveau terminal.
De nombreux plugins sont disponibles, permettant de suivre l'évolution des programmes.

Ouvrir Plugins ⟶ Introspection ⟶ Node Graph, cela permet de visualiser les nœuds actifs et les messages échangés. Avec les commandes de la section précédente, le résultat devrait être le suivant, où le topic **/turtle1/cmd_vel** est envoyé depuis le nœud de téléopération clavier vers le nœud de simulation du déplacement de la tortue. Ce topic est de type **geometry_msgs/Twist**, il contient une vitesse linéaire et une vitesse angulaire.



Pour publier sur ce topic ou lire son contenu, utiliser dans **rqt** les outils du groupe Plugins ⟶ Topics (Message Publisher, Topic Monitor). Le topic **/turtle1/pose** (de type **turtlesim/Pose**) contient notamment la position et l'orientation courante de la tortue.

On peut également écouter un topic en ligne de commande dans un terminal : **rostopic echo /turtle1/pose** ou publier sur un topic : **rostopic pub /turtle1/cmd_vel** et appuyer sur tabulation pour l'autocomplétion du message qui est ainsi modifiable.

Pour tracer des courbes en temps réel, utiliser dans **rqt** Plugins ⟶ Visualization ⟶ Plot, où il est possible d'ajouter n'importe quel topic (connus à partir de la liste précédente).

3.3 Rosbag : enregistrement de données

La commande **rosbag** permet d'enregistrer des fichiers de données au format **.bag** qui peuvent ensuite être rejoués dans ROS, ou lus dans rqt comme si la simulation ou le robot réel étaient présents.

Pour réaliser un enregistrement, par exemple des variables de la simulation turtlesim, lancer la commande **rosbag record [topics à enregistrer]**.
Indiquer par exemple **/turtle1/cmd_vel /turtle1/pose**. Pour stopper l'enregistrement : **Ctrl + C**.

Pour rejouer les données, exécuter dans un terminal **rosbag play [bag_name]**. Les données peuvent être visualisées dans **rqt**.

3.4 Premier nœud ROS

Nous allons maintenant créer un nœud permettant de **publier** une commande à un robot (par exemple sur le topic **/turtle1/cmd_vel**).

1. Les nœuds ROS créés par l'utilisateur se trouvent dans le dossier **catkin_ws/src**
   (situé dans **/home/robot/** pour la machine virtuelle fournie).
   Dans ce dossier, créer un nouveau package (du nom de votre choix) avec la commande
   `catkin create pkg [package_name] --catkin-deps rospy roscpp`
   `std_msgs geometry_msgs nav_msgs sensor_msgs`
   Les indications après le nom du package sont les dépendances logicielles souhaitées. Il sera possible de les modifier manuellement dans les fichiers **CMakeLists.txt** et **package.xml**.

   Pour compiler le workspace, y compris le nouveau package, lancer **catkin build** dans le dossier catkin_ws ou un de ses sous-dossiers.
   Pour que ROS connaisse le workspace, il faut ajouter dans le fichier **.bashrc** la ligne suivante (modification active au lancement d'un nouveau terminal) :
   `source ~/catkin_ws/devel/setup.bash`
2. **NB : Choisir Python ou C++**
   (a) **Python**
   Créer un dossier scripts (commande **mkdir** scripts) dans le dossier du package nouvellement créé.
   Télécharger le code suivant et le déposer dans ce dossier **[package_name]/scripts** et le rendre exécutable **chmod +x talker.py** :
   `https://raw.github.com/ros/ros_tutorials/kinetic-devel/rospy_tutorials/001_talker_listener/talker.py`

   Afin de pouvoir compiler ce nouveau code, éditer le fichier **CMakeLists.txt** présent à la racine du package, et y ajouter les lignes suivantes en fin de fichier :
   ```
   catkin_install_python(PROGRAMS scripts/talker.py
   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
   ```
   (b) **C++**
   Télécharger le code suivant et le déposer dans ce dossier **[package_name]/src** :
   `https://raw.github.com/ros/ros_tutorials/kinetic-devel/roscpp_tutorials/talker/talker.cpp`

   Afin de pouvoir compiler ce nouveau code, éditer le fichier **CMakeLists.txt** présent à la racine du package, et y ajouter les lignes suivantes en fin de fichier :
   ```
   include_directories(include ${catkin_INCLUDE_DIRS})

   add_executable(talker src/talker.cpp)
   target_link_libraries(talker ${catkin_LIBRARIES})
   ```
3. Lancer **catkin build**. Il est également possible de compiler uniquement un package, se placer dans son dossier et utiliser **catkin build --this**.
   Le nouveau nœud peut maintenant être lancé avec la commande **rosrun [package_name] talker** (.py pour un nœud Python)
4. Modifier le code source (talker.cpp ou .py) pour publier un message de type **geometry_msgs/Twist** sur le topic **/turtle1/cmd_vel** du nœud turtlesim_node utilisé précédemment.
   Placer des valeurs arbitraires dans les champs **linear.x** (vitesse linéaire) et **angular.z** (vitesse angulaire), les autres valeurs ne sont pas utilisées en 2D.
   NB : en Python, pré-déclarer le message tel que **var=Twist()**

3.5 Roslaunch

Lorsque le nombre de nœuds à lancer est important, il est fastidieux de taper à plusieurs reprises la commande rosrun dans un terminal. C'est pourquoi il existe la commande **roslaunch**, qui permet de charger un script xml contenant les informations de plusieurs nœuds à lancer ainsi que leurs paramètres.

Créer un dossier **launch** dans le dossier **src** du package de travail. Créer dans ce nouveau dossier un fichier nommé **turtle.launch** et y copier les lignes suivantes :

```
<launch>
<node pkg="turtlesim" name="sim" type="turtlesim_node" >
</node>
</launch>
```

En exécutant **roslaunch turtle.launch** dans un terminal, la simulation précédente va se lancer.
Ajouter une ligne similaire dans ce fichier pour que le nœud qui a été créé dans la section précédente se lance également (attention en Python, préciser l'extension **.py** dans le type).

3.6 Rosparam

Il est possible de passer des paramètres à un nœud, via un fichier de configuration (.yaml) ou directement depuis le launch. Par exemple, si on ajouter dans la définition de notre **node** dans le fichier launch la ligne suivante :

```
<param name="~vel" value="2.0"/>
```

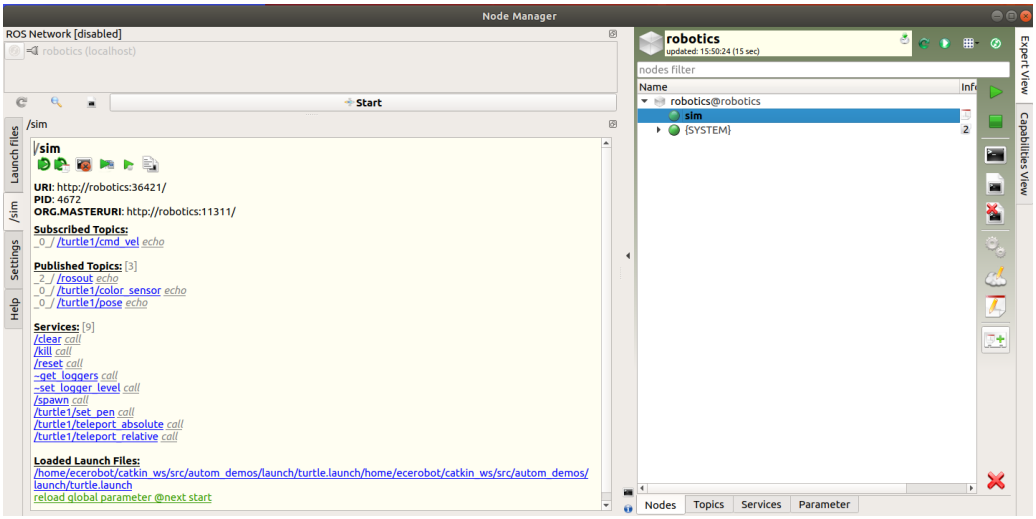
Le « ~ » indique que le paramètre est privé (connu du nœud uniquement). Dans le code source, on peut alors récupérer la valeur de ce paramètre via la commande :

- En C++, `rosparam.get("~vel")`
- En Python, `rospy.get_param('~vel')`, après la commande d'initialisation rospy.init_node().

3.7 Node manager

Node manager se lance via la commande **node_manager** dans un terminal. Remarque : inutile de lancer un roscore, ce programme s'en charge.

Cet outil permet de charger des fichiers de launch et d'effectuer la plupart des actions précédentes via une interface graphique : la liste des nœuds actifs, lancer ou arrêter des nœuds, écouter et publier des topics, lancer rqt, etc...
A titre d'exercice, charger le fichier launch défini dans la section précédente, et utiliser l'interface pour lancer les nœuds, les arrêter, publier sur un topic.

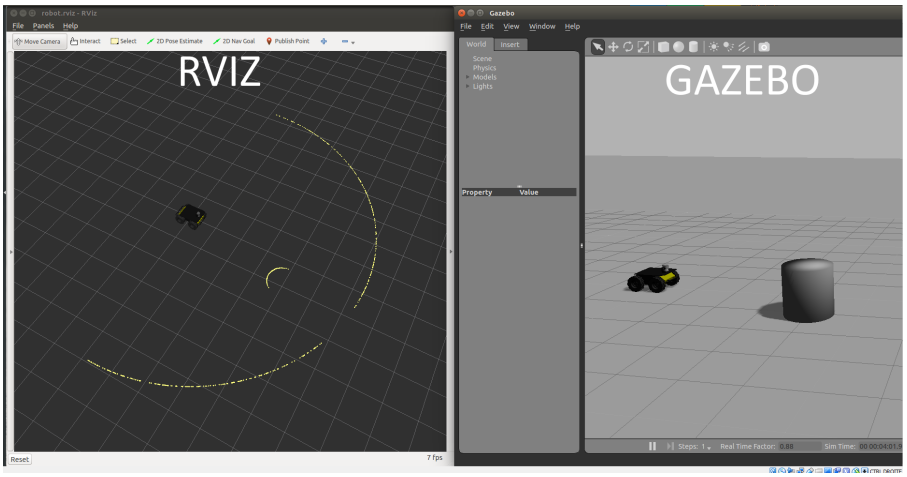


4 Simulateur réaliste Gazebo

ROS communique avec le simulateur physique Gazebo, qui permet de modéliser finement les robots, leurs capteurs et actionneurs en interaction avec un environnement virtuel.
NB : après avoir arrêté les nœuds ROS, le service Gazebo a parfois du mal à se fermer proprement. Pour l'aider, exécuter `killall gzclient gzserver` dans un terminal.

4.1 Robot mobile Husky

Dans un terminal ou dans node_manager, charger :
`roslaunch husky_gazebo husky_empty_world.launch`
L'interface graphique du simulateur va alors s'ouvrir dans une nouvelle fenêtre, montrant la vue du monde physique simulé. Un obstacle peut être ajouté à la souris en choisissant un cube, un cylindre ou une sphère dans la barre du haut. Dans un nouveau terminal lancer la vue **rviz** adaptée au robot via **roslaunch husky_viz robot_view.launch** (Remarque : on peut également lancer rviz depuis un terminal). Cela permet de visualiser en 3D les informations recueillies et calculées par le robot (nappe LaserScan par exemple). Dans Node manager, examiner les nœuds qui ont été lancés, les topics et les paramètres disponibles.



4.2 Simulation drone : package *rotors*

Dans un terminal ou dans node_manager, charger :
`roslaunch rotors_gazebo mav_hovering_example.launch`
Comme pour le robot mobile, l'interface gazebo se lance avec un drone qui décolle et s'asservit à une position fixe (hovering). Dans Node manager, examiner les nœuds qui ont été lancés, les topics et les paramètres disponibles. Un contrôleur en boucle fermée est déjà actif dans cette simulation, il est possible de fournir un nouveau point de référence au drone via le topic `/firefly/command/pose`.