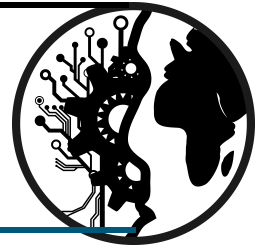


ROS2: Modélisation de robots

Visualisation & simulation

Laurent Beaudoin & Loïca Avanthey

Epita



Avant propos

La robotique mélange de l'informatique de l'électronique et de la mécanique, soit des éléments logiciels et des éléments physiques ! Pour concevoir un robot sous ROS, on va être amené à décrire un certain nombre d'informations relatives aux éléments physiques (positions relatives des différents capteurs par exemple) de ce dernier pour que les briques logicielles puissent fonctionner correctement. C'est ce qu'on appelle modéliser un robot. Cette version virtuelle de notre robot nous offre également la possibilité de tester nos programmes développés grâce aux logiciels de simulation robotique. À travers ces derniers, on peut modéliser un environnement virtuel et réaliste qui pourra prendre en compte la dynamique, la cinématique, les frottements, les collisions, etc. Ainsi nous serons en mesure de vérifier le comportement attendu de notre robot et le cas échéant, de relever des erreurs et pouvoir les corriger avant d'effectuer les tests en réel sur le matériel. Par ailleurs, certains simulateurs permettent de mettre en place des scénarios suffisamment réaliste pour être utilisés pour entraîner des IA de robots.

1 Modélisation du robot

1.1 Architecture



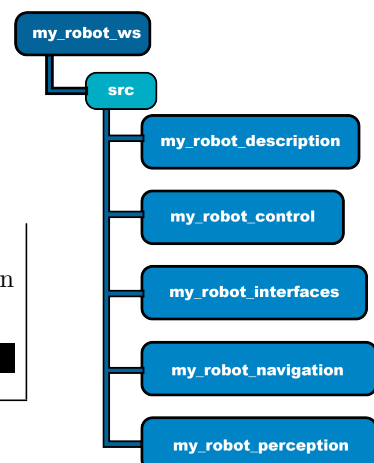
D'un point de vue de l'architecture logicielle, un modèle de robot dans ROS2, c'est tout simplement... un nouveau paquet !

EXERCICE 1 (*Paquet description*)



Pour créer un nouveau robot, on commence donc par créer un nouveau paquet comme on l'a vu précédemment :

```
ros2 pkg create --build-type ament_cmake my_robot_description
```



1.2 Description d'un robot en URDF

La description du robot virtuel se fait dans un fichier unique au format URDF (*Unified Robot Description Format*). C'est intrinsèquement un fichier XML, donc basé sur des balises. Notre fichier `my_robot_description.urdf`, qui contiendra toute la description de notre robot, doit être placé dans le répertoire `src/description/` du paquet `my_robot_description`.

La balise `robot` est la balise racine : tout ce qui suit s'intégrera à l'intérieur de cette balise. La structure minimale de notre fichier URDF est donc :

```
<?xml version="1.0"?>
<robot name="my_robot" xmlns:xacro="http://ros.org/wiki/xacro">
  <!-- tags to describe the robot -->
</robot>
```

EXERCICE 2 (*Fichier URDF*)



Créez le fichier `my_robot_description.urdf` dans le répertoire `src/description/` de votre nouveau paquet et ajoutez-lui le code minimaliste.

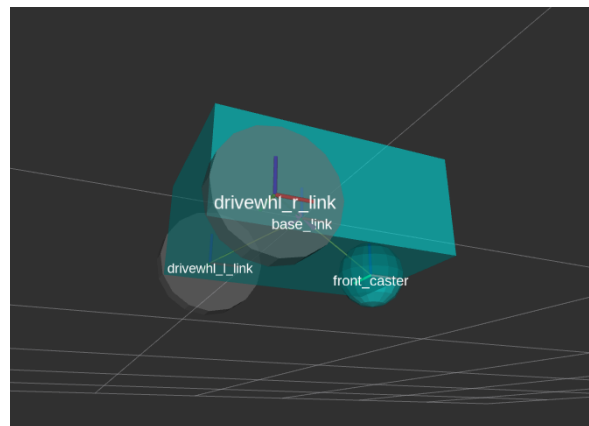
1.2.1 Le robot à modéliser

Le robot que l'on souhaite modéliser est un robot mobile composé d'un châssis rectangulaire, de deux roues à l'arrière et d'une sphère à l'avant. Le modèle que l'on va créer est illustré ci-contre.



La description du robot se fait comme un LEGO®

C'est-à-dire que l'on va décrire chaque élément ou brique (comme un châssis, une roue, une roulette) puis va indiquer comment chaque brique est reliée avec les autres briques (positionnement, nature du lien, etc.).






Le modèle virtuel du robot doit rassembler toutes les caractéristiques du matériel réel.

Même si la forme n'est pas très "ressemblante visuellement", on doit avoir renseigné toutes les caractéristiques physiques du robot.

1.2.2 Description des éléments

En URDF, un élément est décrit par la balise `link`. Dans un premier temps, on ne va s'intéresser qu'à la description qui ne concerne que la visualisation du robot et qui est contenue dans la balise `visual` de l'élément `link`, dont un exemple est donné ci-après :

```
<link name="my_link">
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <box size="1 2 3" />
    </geometry>
    <material name="Cyan">
      <color rgba="0 1.0 1.0 1.0"/>
    </material>
  </visual>
</link>
```

-  **origin** permet de positionner le centre du repère de l'élément en précisant sa position **xyz** et son orientation **rpy** (en roulis, tangage et lacet, en radians). Par défaut, c'est à zéro, c'est-à-dire que le centre du repère est aligné sur le centre de la pièce.
-  **geometry** définit la forme de la pièce qui peut-être **box** pour une boîte rectangulaire, **cylinder** pour un cylindre ou **sphere** pour une sphère.
-  **material** permet de préciser la couleur **color** à utiliser pour la visualisation. Elle est exprimée en **rgba** (rouge, vert, bleu et transparence) avec chaque valeur comprise entre 0 et 1.




Dans la balise **geometry**, en fonction du type de boîte choisi, il faut ajouter des informations supplémentaires (paramètres) :

- ⇒ Pour **box**, il faut préciser **size** qui donne les dimensions en x, y et z comme dans l'exemple.
- ⇒ Pour **cylinder**, les paramètres sont **radius** pour le rayon et **length** pour sa longueur.
- ⇒ Pour **sphere**, un seul paramètre **radius**, qui indique le rayon.

EXERCICE 3 (*Le châssis - description*)



En vous servant de l'exemple, construisez dans votre fichier URDF l'objet **base_link** (notre châssis) de type boîte rectangulaire avec pour dimensions 0.42 en x, 0.31 en y et 0.18 en z, et de couleur cyan (0, 1.0,1.0,1.0 en rgba). Alignez l'origine du repère de cette pièce sur le centre de cette dernière (valeurs de position et d'orientation par défaut).

-  **La couleur cyan** en rgba est décrite par le quadruplet suivant : (0.0, 1.0, 1.0, 1.0).
-  **Ici**, le repère est orienté avec l'axe x qui va vers l'avant du robot, l'axe y qui part sur la gauche du robot et l'axe z qui va vers le haut.
-  **On vérifiera votre code plus loin**, au moment de la visualisation par un programme dédié... Suspense !

EXERCICE 4 (*La roulette - description*)



Construisez maintenant l'objet **front_caster** (notre roulette) de type sphère de rayon 0.06 et de couleur cyan. Alignez l'origine du repère de cette pièce sur le centre de cette dernière (valeurs de position et d'orientation par défaut).

EXERCICE 5 (*La roue gauche - description*)



Créez l'objet roue gauche nommée **drivewhl_l_link** à partir d'un cylindre de rayon 0.1 et de longueur 0.04 avec la couleur grise. Alignez l'origine du repère de cette pièce sur le centre de cette dernière et orientez-le à 90 degrés (1.5708 radian) selon l'axe x.

-  **La couleur grise** en rgba est décrite par le quadruplet suivant : (0.5, 0.5, 0.5, 1.0).

EXERCICE 6 (*La roue droite - description*)



Enfin, créez l'objet roue droite nommée **drivewhl_r_link** avec les mêmes paramètres que pour la roue gauche.

1.2.3 Description des joints

Une fois chaque élément décrit visuellement, l'étape d'après consiste à décrire les relations entre ces éléments, c'est-à-dire leur positionnements relatifs (entre les origines de leurs repères respectifs) et la nature du lien (fixe ou mobile).

Le positionnement de chaque élément s'établit par rapport à un élément de référence (élément "parent"). On indique ainsi la position de l'origine du repère de l'élément "enfant" dans le repère de l'élément parent. L'élément de référence de base est `base_link` : c'est le premier parent et il est lié à tous les éléments enfants directs (et à leurs enfants par leur intermédiaire, etc.).



Chaque élément ne peut avoir qu'un seul parent, mais un parent peut avoir plusieurs enfants. Par exemple, notre châssis va avoir plusieurs enfants (2 roues et la roulette), mais chaque élément (roues et roulette) ne va avoir comme parent que châssis. Seul le châssis (`base_link`) n'aura pas de parent puisqu'il sert de référence à tous les autres éléments.

Toute partie du robot qui est en mouvement par rapport à une autre partie du robot (segment de bras articulé, roue, etc.) aura un lien de nature mobile alors que toute pièce qui est immobile par rapport à d'autres parties (ie. qui pourrait être englobée) mais qui nécessite d'avoir son propre repère de référence (des capteurs comme les lidar ou les caméras pour passer facilement du repère du robot au repère du capteur et inversement) aura un lien de nature fixe.



Pour faire tourner une visualisation ou une simulation, il sera nécessaire de calculer toutes les transformations géométriques entre tous les repères des différents éléments : c'est un problème particulièrement complexe ! Heureusement, il y a dans ROS2 des outils qui vont se charger de ce travail fastidieux et que l'on verra un peu plus tard.

En URDF, une relation entre deux éléments est décrite par la balise `joint`, dont un exemple est donné ci-après :

```
<joint name="my_joint" type="continuous">
  <parent link="parent_link"/>
  <child link="child_link"/>
  <origin xyz="0 0 1" rpy="0 0 3.1416"/>
  <axis xyz="0 1 0"/>
</joint>
```



type définit le type de lien qu'il y a entre l'objet à positionner (`child`) par rapport à l'objet de référence (`parent`).



parent link et **child link** sont respectivement le nom de l'objet de référence et le nom de l'objet que l'on cherche à positionner (position et orientation) par rapport à lui.



origin précise la position `xyz` et l'orientation `rpy` de l'origine du repère de l'élément enfant dans le repère de l'élément parent.



axis est un paramètre lié au type de lien choisi :

- Si la **nature du lien** est **fixe**, on utilisera **principalement un seul type de joint** :
 - **fixed**, par exemple pour un capteur (enfant) fixé sur le châssis (parent).
- Si la **nature du lien** est **mobile**, on pourra choisir parmi **quatre types de joints principaux** :
 - ⇒ **continuous**, dans le cas où l'enfant tourne autour d'un axe de rotation dans un mouvement continu sur 360° (comme une roue attachée au châssis, ou une hélice). Il faudra dans ce cas préciser l'axe de rotation avec la balise `axis` comme dans l'exemple précédent (ou l'enfant tourne autour de l'axe y du parent).
 - ⇒ **revolute**, dans le cas où la rotation de l'enfant est limitée, comme avec un servo-moteur par exemple. En plus de l'axe de rotation il faudra aussi préciser les limites de la rotation (**lower** et **upper** en radians) avec la balise `limit`.
 - ⇒ **prismatic**, dans le cas où l'enfant présente un mouvement de glissement linéaire le long d'un axe avec une position minimum et maximum sur cet axe (moteur pas à pas par exemple sur une vis d'imprimante 3D).

⇒ **floating**, dans le cas où on veut un lien où tous les mouvements sont possibles (comme une boule).



Pour connaître toutes les options (les types et leurs paramètres obligatoires ou optionnels) n'hésitez pas à consulter la documentation de référence : <https://wiki.ros.org/urdf/XML/joint>.

EXERCICE 7 (*La roulette - position*)



Créez la balise `joint` nommée `caster_joint` de type `fixed` positionnant l'objet `front_caster` par rapport au châssis `base_link` aux coordonnées (0.14, 0.0, -0.09) et avec l'orientation par défaut (0, 0, 0).

EXERCICE 8 (*La roue gauche - position*)



Créez la balise `joint` nommée `drivewhl_l_joint` de type `continuous` positionnant l'objet `drivewhl_l_link` par rapport au châssis `base_link` aux coordonnées (-0.12, -0.18, -0.05) et avec l'orientation par défaut (0, 0, 0).

EXERCICE 9 (*La roue droite - position*)



Créez la balise `joint` nommée `drivewhl_r_joint` avec les mêmes paramètres que pour la roue gauche à l'exception de la position en (-0.12, 0.18, -0.05).

1.3 Xacro power

Le fichier qui contient la description du robot commence à être verbeux et on intuite vite que lorsque l'on voudra décrire des robots plus complexes, ça va devenir très lourd. On voit en particulier :

- Que l'on a mis en "dur" toutes les dimensions et positions de chaque éléments. Ainsi :
 - Ce n'est pas facile de comprendre comment les valeurs en "dur" ont été calculées.
 - Si on souhaite changer une dimension du robot, c'est complexe de tout modifier.



Le mieux serait de tout définir à partir de variables pour mieux comprendre la logique de positionnement des éléments les uns par rapport aux autres et de pouvoir changer facilement des dimensions sans risque d'oublier de répercuter le changement dans le code.

- Qu'il y a de la redondance que l'on pourrait simplifier au niveau de la description et du positionnement des roues (la roue gauche est fortement "identique" à la roue droite).



Le plus efficace serait d'utiliser un équivalent de constructeur pour créer génériquement les deux roues avec juste deux appels et des paramètres spécifiques.

Il existe un outil pour faire tout cela et se simplifier la vie, il s'agit de **Xacro** (*XML macro*) ! On va pouvoir ainsi définir des variables ou des prototypes de fonction pour factoriser et réutiliser le code, et le rendre *de facto* plus lisible et plus évolutif.

EXERCICE 10 (*Xacro installation*)



Au préalable, on va commencer par installer les paquets dont on va avoir besoin :

```
sudo apt install ros-humble-xacro
```



Pensez à adapter les lignes précédentes à votre propre distribution ROS2.

EXERCICE 11 (*Actualisation du manifeste*)



Puis, on va compléter le manifeste `package.xml` qui est à la racine de notre paquet `my_robot_description` en lui ajoutant la dépendance suivante (de préférence après le `<buildtool_depend>`) :

```
<exec_depend>xacro</exec_depend>
```

1.3.1 Variables

Xacro permet de définir des variables en XML selon le modèle :

```
<xacro:property name="my_variable" value="42"/>
```

Pour pouvoir l'utiliser, il faut l'évaluer dans le fichier XML par l'appel à `#{my_variable}` :

```
<sphere radius="#{my_variable}"/>
```



Simple non ?
À vous de jouer !

EXERCICE 12 (*Xacroatisation - variables*)



Dans votre fichier URDF, dans les éléments et dans les liens, remplacez les différentes dimensions physiques (dimensions du robot et positionnement des éléments sur le châssis) écrites en dur par des variables.

1.3.2 Macros

Enfin, pour faire l'équivalent d'un constructeur en macro, on utilise la balise `xacro:macro` comme dans l'exemple suivant :

```
<xacro:macro name="my_generic_element" params="prefix rx">
  <link name="${prefix}_link">
    <visual>
      <origin xyz="0 0 0" rpy="${rx} 0 0"/>
      ....
    </visual>
  </link>

  <joint name="${prefix}_joint" type="continuous">
    ....
  </joint>
</xacro:macro>
```

L'appel de la macro se fait en donnant les valeurs pour chaque paramètres comme dans l'exemple :

```
<xacro:my_generic_element prefix="sensor1" rx="${pi/2}" />
```



Toujours simple non ?
À vous de (re)jouer !

EXERCICE 13 (*Xacro*tisation - macros)



Dans votre fichier URDF, créez une macro pour décrire l'élément et le joint d'une roue, puis appelez-la pour créer la roue gauche puis la roue droite.



Le code qui suit est une variante de ce que vous avez écrit en utilisant les possibilités offertes par *Xacro*. Sa lecture ne devrait pas vous poser de problème et vous saurez en apprécier l'élégance.

```
<?xml version="1.0"?>
<robot name="my_robot" xmlns:xacro="http://ros.org/wiki/xacro">

  <!-- Define robot constants -->
  <xacro:property name="base_width" value="0.31"/>
  <xacro:property name="base_length" value="0.42"/>
  <xacro:property name="base_height" value="0.18"/>

  <xacro:property name="wheel_radius" value="0.10"/>
  <xacro:property name="wheel_width" value="0.04"/>
  <xacro:property name="wheel_ygap" value="0.025"/>
  <xacro:property name="wheel_zoff" value="0.05"/>
  <xacro:property name="wheel_xoff" value="0.12"/>

  <xacro:property name="caster_xoff" value="0.14"/>

  <!-- Robot base description -->
  <link name="base_link">
    <visual>
      <geometry>
        <box size="${base_length} ${base_width} ${base_height}"/>
      </geometry>
      <material name="Cyan">
        <color rgba="0 1.0 1.0 1.0"/>
      </material>
    </visual>
  </link>

  <!-- Wheel description (MACRO) -->
  <xacro:macro name="wheel" params="prefix x_reflect y_reflect">
    <!-- Wheel element description -->
    <link name="${prefix}_link">
      <visual>
        <origin xyz="0 0 0" rpy="${pi/2} 0 0"/>
        <geometry>
          <cylinder radius="${wheel_radius}"
            length="${wheel_width}"/>
        </geometry>
        <material name="Gray">
          <color rgba="0.5 0.5 0.5 1.0"/>
        </material>
      </visual>
    </link>

    <!-- Wheel joint description -->
    <joint name="${prefix}_joint" type="continuous">
      <parent link="base_link"/>
      <child link="${prefix}_link"/>
      <origin xyz="${x_reflect*wheel_xoff}
        ${y_reflect*(base_width/2+wheel_ygap)} ${-wheel_zoff}" rpy="0 0 0"/>
    </joint>
  </macro>

  <xacro:wheel prefix="left" x_reflect="0" y_reflect="0"/>
  <xacro:wheel prefix="right" x_reflect="1" y_reflect="0"/>
</robot>
```

```

        <axis xyz="0 1 0"/>
    </joint>
</xacro:macro>

<!-- Left wheel description (element + joint) -->
<xacro:wheel prefix="drivewhl_l" x_reflect="-1" y_reflect="1" />
<!-- Right wheel description (element + joint) -->
<xacro:wheel prefix="drivewhl_r" x_reflect="-1" y_reflect="-1" />

<!-- Caster wheel element description -->
<link name="front_caster">
    <visual>
        <geometry>
            <sphere radius="\${(wheel_radius+wheel_zoff-(base_height/2))}" />
        </geometry>
        <material name="Cyan">
            <color rgba="0 1.0 1.0 1.0" />
        </material>
    </visual>
</link>

<!-- Caster wheel joint description -->
<joint name="caster_joint" type="fixed">
    <parent link="base_link" />
    <child link="front_caster" />
    <origin xyz="\${caster_xoff} 0.0 \${-(base_height/2)}" rpy="0 0 0" />
</joint>

</robot>

```

1.3.3 Multi-fichiers

Si on a un robot complexe à décrire, toujours dans un soucis de lisibilité, de réemployabilité et d'évolutivité, il est intéressant de scinder ce fichier unique en plusieurs morceaux : Xacro nous le permet !

Depuis notre fichier URDF principal, on peut charger le contenu de fichiers XACRO répartis dans différents répertoires grâce à la balise suivante :

```
<xacro:include filename="\$(find package_name)/src/description/directory_name/filename.xacro" />
```

Par exemple, pour notre robot, nous pouvons envisager l'architecture suivante :

- description/
 - my_robot_description.urdf
 - parts/
 - * dimensions.xacro
 - * robot_base.xacro
 - * wheels.xacro

Par la suite, si on ajoute des capteurs, on pourra ajouter une dossier **sensors/** au même niveau que **parts/** et décrire chaque capteur dans un fichier Xacro.



Maintenant que nous avons décrit le modèle de notre robot,
nous allons voir dans la partie suivante comment le visualiser.

2 Visualisation du robot

2.1 État du robot



Pour pouvoir visualiser dynamiquement notre robot,
il faut calculer en permanence la position de chaque élément les uns par rapport aux autres.

C'est une tâche complexe et fastidieuse car elle manipule beaucoup de changements de repères puisque chaque objet est défini relativement à un parent, lui même défini par rapport à son propre parent, etc., et ce jusqu'à la pièce maîtresse, pour nous le châssis.



Heureusement, on a évoqué précédemment qu'il existait des outils pour nous simplifier la vie et faire le travail à notre place.

On va utiliser les nœuds `joint_state_publisher`, `joint_state_publisher_gui` et `robot_state_publisher` qui vont se charger d'actualiser toutes les positions de tous les éléments et de publier le résultat dans un forum qui pourra être lu par l'outil de visualisation.



Enfinement, ce n'est pas si compliqué.

EXERCICE 14 (*Installations Joint State Publisher*)



Au préalable, on va commencer par installer les paquets dont on va avoir besoin et qui ne sont pas déjà installés par défaut :

```
sudo apt install ros-humble-joint-state-publisher-gui
```



Pensez à adapter les lignes précédentes à votre propre distribution ROS2.

EXERCICE 15 (*Actualisation du manifeste*)



Puis, on va compléter le manifeste `package.xml` qui est à la racine de notre paquet `my_robot_description` en lui ajoutant les dépendances suivantes (de préférence après le `<buildtool_depend>`) :

```
<exec_depend>joint_state_publisher</exec_depend>
<exec_depend>joint_state_publisher_gui</exec_depend>
<exec_depend>robot_state_publisher</exec_depend>
```

2.2 Le visualiseur rviz2

Si on reprend l'équation fondamentale de ROS2 :



ROS2 nous fournit de la **tuyauterie** (le système de communication entre nœuds par exemple), des **outils de développement** (comme les fichiers de lancement qu'on a vu dans le précédent TD), des **capacités** (tous les packages qui sont à disposition et qui permettent d'ajouter facilement des fonctionnalités à nos robots sans avoir à tout recoder) ainsi qu'une **communauté** (qui assure le développement continu et permet d'avoir des développements fiables et *up-to-date*).



Et dans les outils de développement
se trouve un outil de visualisation nommé **rviz2**.

EXERCICE 16 (*Actualisation du manifeste*)



Rviz2 est déjà installé par défaut avec ROS2, on va donc seulement compléter le manifeste `package.xml` qui est à la racine de notre paquet `my_robot_description` en lui ajoutant la dépendance suivante (de préférence après le `<buildtool_depend>`) :

```
<exec_depend>rviz</exec_depend>
```

Puisque l'on a plusieurs nœuds (`robot_state_publisher`, `joint_state_publisher`, `joint_state_publisher_gui` et `rviz2`) de plusieurs paquets (le nom des paquets est identique aux noms des nœuds cités) à lancer pour pouvoir faire tourner notre visualisation, on va tout naturellement créer un fichier de lancement (voir TD "Outils opérationnels").

EXERCICE 17 (*Fichier de lancement*)



Créez le répertoire `launch` à la racine de `my_robot_description` et créez à l'intérieur le fichier `display.launch.py` qui contient le code ci-après.

Contenu du fichier `display.launch.py` :

```
import os
from launch import LaunchDescription
from launch_ros.actions import Node
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration, Command
from launch_ros.substitutions import FindPackageShare
from launch.conditions import UnlessCondition, IfCondition

def generate_launch_description():
    ld = LaunchDescription()

    pkg_share = FindPackageShare(package="my_robot_description").find("my_robot_description")
    default_model_path = os.path.join(pkg_share, "src/description/my_robot_description.urdf")
    default_rviz_config_path = os.path.join(pkg_share, "rviz/urdf_config.rviz")

    gui_arg = DeclareLaunchArgument(
        name="gui",
        default_value="True",
        description="Flag to enable joint_state_publisher_gui"
    )
    model_arg = DeclareLaunchArgument(
        name="model",
        default_value=default_model_path,
        description="Absolute path to robot urdf file"
    )
    rvizconfig_arg = DeclareLaunchArgument(
```

```

    name="rvizconfig",
    default_value=default_rviz_config_path,
    description="Absolute path to rviz config file"
)

robot_state_publisher_node = Node(
    package="robot_state_publisher",
    executable="robot_state_publisher",
    parameters=[{"robot_description": Command(["xacro ", LaunchConfiguration("model")])}]
)

joint_state_publisher_node = Node(
    package="joint_state_publisher",
    executable="joint_state_publisher",
    name="joint_state_publisher",
    condition=UnlessCondition(LaunchConfiguration("gui"))
)

joint_state_publisher_gui_node = Node(
    package="joint_state_publisher_gui",
    executable="joint_state_publisher_gui",
    name="joint_state_publisher_gui",
    condition=IfCondition(LaunchConfiguration("gui"))
)

rviz_node = Node(
    package="rviz2",
    executable="rviz2",
    name="rviz2",
    output="screen",
    arguments=["-d", LaunchConfiguration("rvizconfig")],
)

ld.add_action(gui_arg)
ld.add_action(model_arg)
ld.add_action(rvizconfig_arg)

ld.add_action(joint_state_publisher_node)
ld.add_action(joint_state_publisher_gui_node)
ld.add_action(robot_state_publisher_node)
ld.add_action(rviz_node)

return ld

```

EXERCICE 18 (*Paramètres rviz2*)



Pour que rviz2 se lance avec de bons paramètres de visualisation, on vous conseille de créer le fichier de configuration `urdf_config.rviz` avec le contenu ci-après, que vous placerez dans un répertoire `rviz` que vous créerez à la racine de `my_robot_description`.



Attention : après chaque tiret dans le fichier suivant, ce n'est pas une espace mais une tabulation pour respecter l'indentation, sinon `rviz` plante).

Contenu du fichier `urdf_config.rviz` :

```

Panels:
  - Class: rviz_common/Displays
    Help Height: 78
    Name: Displays
    Property Tree Widget:
      Expanded:
        - /Global Options1
        - /Status1

```

```

        - /RobotModel1/Links1
        - /TF1
    Splitter Ratio: 0.5
    Tree Height: 557
Visualization Manager:
  Class: ""
  Displays:
    - Alpha: 0.5
      Cell Size: 1
      Class: rviz_default_plugins/Grid
      Color: 160; 160; 164
      Enabled: true
      Name: Grid
    - Alpha: 0.6
      Class: rviz_default_plugins/RobotModel
      Description Topic:
        Depth: 5
        Durability Policy: Volatile
        History Policy: Keep Last
        Reliability Policy: Reliable
        Value: /robot_description
      Enabled: true
      Name: RobotModel
      Visual Enabled: true
    - Class: rviz_default_plugins/TF
      Enabled: true
      Name: TF
      Marker Scale: 0.3
      Show Arrows: true
      Show Axes: true
      Show Names: true
  Enabled: true
  Global Options:
    Background Color: 48; 48; 48
    Fixed Frame: base_link
    Frame Rate: 30
  Name: root
  Tools:
    - Class: rviz_default_plugins/Interact
      Hide Inactive Objects: true
    - Class: rviz_default_plugins/MoveCamera
    - Class: rviz_default_plugins/Select
    - Class: rviz_default_plugins/FocusCamera
    - Class: rviz_default_plugins/Measure
      Line color: 128; 128; 0
  Transformation:
    Current:
      Class: rviz_default_plugins/TF
  Value: true
  Views:
    Current:
      Class: rviz_default_plugins/Orbit
      Name: Current View
      Target Frame: <Fixed Frame>
      Value: Orbit (rviz)
  Saved: ~

```

EXERCICE 19 (*CMakefile*)



Enfin, complétons le `CMakefile.txt` à la racine de `my_robot_description` en y ajoutant les lignes suivantes (de préférence avant `if(BUILD_TESTING)`) :

```
install(  
  DIRECTORY src launch rviz  
  DESTINATION share/${PROJECT_NAME}  
)
```

EXERCICE 20 (*Compilation*)



Il ne reste plus qu'à lancer la compilation à la racine de notre espace de travail :

```
colcon build
```

Et ne pas oublier de sourcer si ce n'est pas déjà fait :

```
source install/local_setup.bash
```

EXERCICE 21 (*Lancer la visualisation*)

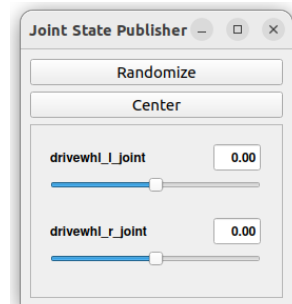
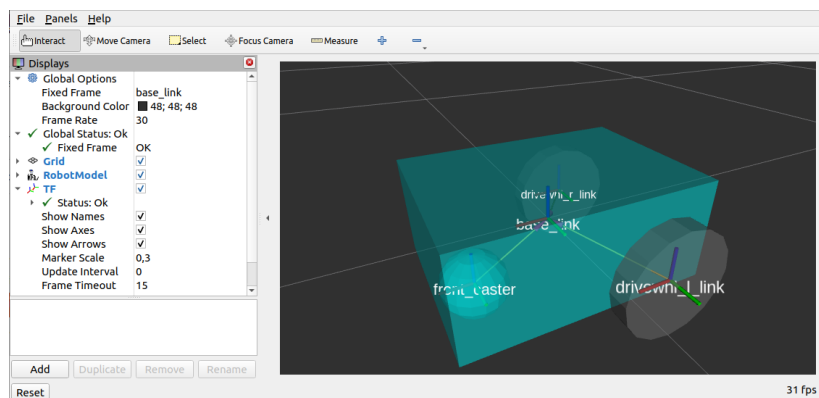


Il vous reste à lancer la simulation avec le fichier de lancement :

```
ros2 launch my_robot_description display.launch.py
```



Si tout fonctionne, vous devriez voir apparaître deux fenêtres : la première (la grande) contient le visualisateur `rviz2` et la seconde (la petite) nommée `joint state publisher` permet de modifier avec les curseurs les valeurs d'orientation des roues, et donc de les faire tourner.



La visualisation avec `rviz2` permet de vérifier le modèle URDF du robot mais aussi de s'assurer que tous les changements de repères sont bien actualisés et que tout est nominal. Sans cela, aucune simulation ne peut fonctionner !

3 Simulation du robot

3.1 Les principaux outils de simulation

Il existe de nombreux outils de simulation pour la robotique. Nous nous intéresserons dans ce TD à trois d'entre eux, qui sont parmi les plus connus dans la communauté robotique et compatibles avec ROS2 : Gazebo, Webots et CoppeliaSim. Nous ne parlerons pas de Unity (qui vient de la communauté des jeux vidéos), mais sachez qu'il est lui aussi compatible avec ROS2 et peut être utilisé en tant que simulateur robotique.

- **Gazebo** (<https://gazebo.org/home>) est un simulateur multi-robots populaire et bien établi dans la communauté ROS. Débuté en 2002, financé en partie au cours de son développement par Willow Garage (le laboratoire de recherche qui a créé OpenCV et ROS), actuellement hébergé par Open Robotics (comme ROS) il est open-source et tourne sous Linux. Il a été conçu pour reproduire avec précision la dynamique des environnements complexes, intérieur ou extérieur, qu'un robot pourrait rencontrer grâce à ces 4 moteurs physiques différents (ODE, Bullet, Simbody et DART). La courbe d'apprentissage peut-être un peu raide pour les débutants et toute la documentation n'est pas forcément à jour, notamment avec la transition de compatibilité avec ROS2.
- **Webots** (<https://cyberbotics.com/#webots>) est un simulateur multi-robots développé à l'EPFL à partir de 1996 et actuellement maintenu par l'entreprise Cyberbotics. Open-source et multi-plateforme, il a été rendu compatible avec ROS en 2019. Il utilise le moteur physique ODE (Open Dynamics Engine) et simule aussi bien des environnements intérieurs qu'extérieurs. L'environnement de développement et l'interface utilisateur qu'il fournit est simple, efficace et conviviale, ce qui le rend facile de prise en main.
- **CoppeliaSim** (anciennement V-REP <https://www.coppeliarobotics.com/>) est un simulateur multi-robots depuis 2010. Créé initialement au sein de Toshiba R&D et il est actuellement développé par l'entreprise Coppelia Robotics. Multiplateforme, il s'agit d'un logiciel propriétaire et payant mais qui dispose d'une licence éducative et de recherche gratuite. Compatible avec ROS (avec un peu de configuration), il propose quatre moteurs physiques différents (ODE, Bullet, Vortex Dynamics et Newton). Son installation et son fonctionnement sont assez faciles.

Chacun de ces trois simulateurs ont leurs **avantages** et **inconvénients**. Le tableau suivant propose une **comparaison** succincte :

Nom	Gazebo	Webots	CoppeliaSim
Structure	Open Robotics	Cyberbotics	Coppelia Robotics
License	Open-source (hébergé par ROS)	Open-source (Apache License 2.0) + license commerciale payante	License éducative gratuite (étudiants et chercheurs) + license commerciale payante
OS supportés	GNU/Linux (Ubuntu)	GNU/Linux (Ubuntu) Mac OS Windows	GNU/Linux (Ubuntu) Mac OS Windows
Langage de programmation	C++	C++	Lua
Moteur physique	ODE, Bullet, Simbody, DART	ODE (version propriétaire)	ODE, Bullet, Vortex, Newton
Multithreading	Oui	Oui	Non
Types de robots	Mobiles, humanoïdes, industriels	Mobiles, humanoïdes, industriels	Mobiles, humanoïdes, industriels
Facteur temps réel	***	*	**
Efficacité CPU	*	***	**

 Dans ce TD, nous nous concentrerons sur Gazebo qui est le simulateur historique de ROS2.

3.2 Ajout des paramètres physiques pour la simulation dans l'URDF

- ✓ **Pour que la simulation puisse donner un rendu compatible avec la réalité, il faut compléter notre fichier URDF en précisant des données physiques.**
- i **Toutes les données physiques utilisées dans les simulateurs sont exprimées en mètre, kilogramme, seconde et ampère (anciennement appelé le système MKSA, aujourd'hui système international d'unités).**

3.2.1 Une touche de physique : les collisions

Renseignons dans un premier temps les informations pour gérer les collisions. Il s'agit de décrire pour chaque élément le volume (centré sur l'origine du repère de l'élément) qui le délimite : ainsi notre robot ne passera pas à travers les objets dans la simulation (et pourra donc se "crasher" proprement sur ces derniers !).

On ajoute pour cela la balise `collision` dans chaque élément `link`. Il n'y a qu'une balise `collision` par élément. Elle contient un seul champ : `geometry` qui décrit le volume choisi :

```
<link name="base_link">
  <visual>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
    <material name="blue">
      <color rgba="0 0 .8 1"/>
    </material>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.6"
        radius="0.2"/>
    </geometry>
  </collision>
</link>
```



On peut soit choisir le même volume que l'on a décrit pour la géométrie de la balise `visual` ou bien choisir une **forme englobante plus large** en fonction des besoins de l'application ou de la complexité des formes.



Plus la forme indiquée pour les collisions sera **complexe**, plus le calcul de la gestion des collisions prendra du **temps**.

EXERCICE 22 (*Gérer les collisions*)



Pour chaque élément de notre robot (`base_link`, `front_caster`, `drivewhl_l_link` e `drivewhl_r_link`), ajouter la balise `collision` en utilisant les mêmes paramètres que ceux de la balise `visual`.

3.2.2 L'instant science physique : l'inertie

- i **Le deuxième type de paramètres physiques qu'il nous faut ajouter concerne l'inertie de notre objet, c'est-à-dire sa tendance à rester en mouvement (ou au repos) après une sollicitation mécanique comme lorsque l'on coupe (ou allume) un moteur de propulsion.**

Pour faire leurs calculs, les simulateurs ont besoin de connaître deux informations :

- [Inertie en translation] la **masse de l'objet** qui permet de calculer grâce à la deuxième loi de Newton le comportement dynamique de l'objet en terme de mouvement rectiligne.
- [Inertie en rotation] et le **tenseur d'inertie**, qui permet de calculer le comportement dynamique de l'objet en rotation autour de chaque axe (moments d'inertie) en prenant en compte les interactions entre les différents axes de rotation (produits d'inertie) et donc la manière dont la masse de l'objet est répartie dans les trois dimensions.



La combinaison de ces deux informations couvre tous les mouvements. Le lecteur intéressé pourra lire l'article [wik23] pour aller plus loin.



Le **moment d'inertie** est aussi utilisé dans le calcul de la **trajectoire de l'objet** (sa cinématique) : plus un objet est **lourd**, plus il sera **difficile** de lui faire prendre un **virage serré**. Un simulateur qui permettrait à un pétrolier de faire un virage à 90 degrés instantanément ne serait pas crédible (et les courses virtuelles de moustachus à casquette rouge sur des karts beaucoup moins fun).

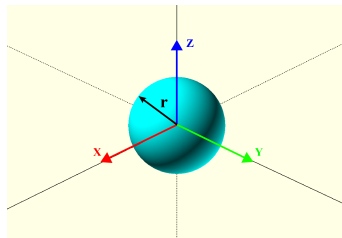
©Nintendo

Le tenseur (ou matrice) d'inertie I a pour forme générale (notez les symétries) :

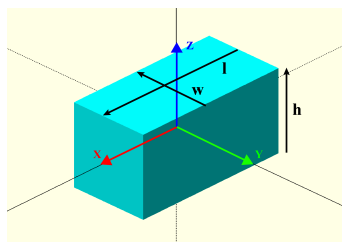
$$I = \begin{pmatrix} i_{xx} & i_{xy} & i_{xz} \\ i_{xy} & i_{yy} & i_{yz} \\ i_{xz} & i_{yz} & i_{zz} \end{pmatrix}$$



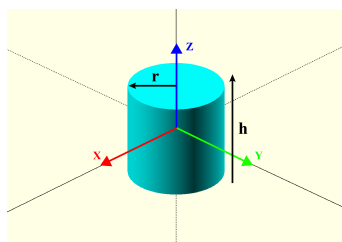
La bonne nouvelle, c'est que la matrice d'inertie se déduit directement de la forme géométrique de l'objet. Voici les matrices d'inertie pour les formes géométriques (pleines et uniformes) que vous pouvez utiliser en URDF (avec m la masse en kg) :



$$I_{sphere} = \begin{pmatrix} \frac{2}{5}mr^2 & 0 & 0 \\ 0 & \frac{2}{5}mr^2 & 0 \\ 0 & 0 & \frac{2}{5}mr^2 \end{pmatrix}$$



$$I_{box} = \begin{pmatrix} \frac{1}{12}m(h^2 + l^2) & 0 & 0 \\ 0 & \frac{1}{12}m(w^2 + h^2) & 0 \\ 0 & 0 & \frac{1}{12}m(w^2 + l^2) \end{pmatrix}$$



$$I_{cylinder} = \begin{pmatrix} \frac{1}{12}m((3r^2 + h^2)) & 0 & 0 \\ 0 & \frac{1}{12}m((3r^2 + h^2)) & 0 \\ 0 & 0 & \frac{1}{12}mr^2 \end{pmatrix}$$

Dans l'URDF, on ajoute nos deux informations à l'élément avec la balise `inertial` et les paramètres `mass` et `inertia` :

```
<link name="base_link">
  <visual>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
    <material name="blue">
      <color rgba="0 0 .8 1"/>
    </material>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="10"/>
    <inertia ixx="1e-3" ixy="0.0" ixz="0.0" iyy="1e-3" iyz="0.0" izz="1e-3"/>
  </inertial>
</link>
```

EXERCICE 23 (*Cinématique des éléments*)



Pour chaque élément de notre robot (`base_link`, `front_caster`, `drivewhl_X_link`), rajouter la balise `inertial` en utilisant pour `mass` la valeur 15 pour le châssis et 0.5 pour tous les autres éléments. Pour chaque balise `inertial`, on prendra la formule adéquate par rapport à la forme géométrique utilisée.



Xacro power : l'utilisation des `xacro:macro` est fortement recommandé pour ne pas se tromper dans les formules.

3.3 Gazebo classic

3.3.1 Installation

Dans ce qui suit on va travailler avec le simulateur historique de ROS, `gazebo` et on va utiliser sa version classique, car la toute dernière, `gazebo ignition` n'est pas encore tout à fait interfaçable avec ROS2 dans la pratique.



EXERCICE 24 (*Installation gazebo classic*)

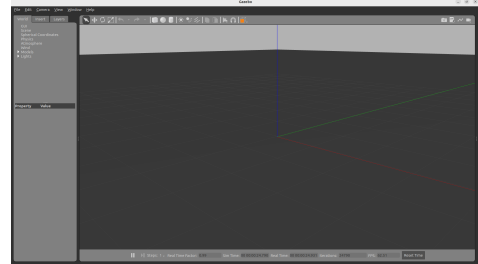


Installez **gazebo** à partir des paquets pour ROS2 Humble. Pour cela, lancez les commandes suivantes dans un terminal :

```
sudo apt install ros-humble-gazebo-ros
```

Vérifiez que l'installation s'est bien déroulée en lançant l'exemple par défaut fourni avec **gazebo** :

```
gazebo
```



Pour information, le lien de référence est le suivant : https://classic.gazebosim.org/tutorials?tut=install_ubuntu.



Attention, On cherche à installer une version de **gazebo** compatible avec ROS. N'oubliez pas que **gazebo** est un simulateur à part entière (indépendant de ROS) et qui peut être utilisé dans d'autres contextes. Il faut donc faire le tri dans la documentation.

EXERCICE 25 (*Installation autres paquets*)



Installez le paquet suivant pour pouvoir faire fonctionner **gazebo** et ROS2 de concert:

```
sudo apt install ros-humble-gazebo-plugins
```

3.3.2 Odométrie

Dans la simulation, la position du robot va évoluer au cours du temps et en fonction des consignes que vous allez lui donner. Pour pouvoir le positionner au bon endroit, le simulateur va partir de la première position du robot et recalculer sa position actuelle en cumulant toutes les positions intermédiaires passées à partir de cette position initiale.



Ce dont a besoin le simulateur c'est donc de connaître le déplacement entre deux instants élémentaires pour estimer le déplacement global.



C'est ce que l'on appelle l'odométrie.

Il y a plusieurs moyens de mesurer une odométrie, comme par exemple intégrer des données venant d'une centrale inertielle ou en mesurant le nombre de tours de chaque roues pour estimer le chemin parcouru par chacune d'elle ce qui permet de déduire l'orientation et la vitesse instantanée du robot. C'est cette dernière approche que l'on va utiliser dans notre exemple.



Pas de panique, c'est simple car tout est déjà intégré dans **gazebo**.



L'intégration se fait via un **plugin** : c'est du **code compilé** qui est **dédié** à une **tâche** précise comme décrire un capteur, estimer une position, faire de la navigation etc.

Il existe énormément de plugins (https://classic.gazebosim.org/tutorials?tut=ros_gzplugins), ce qui permet de facilement simuler un très large éventail de situations dans **gazebo**. Ainsi, l'ensemble des plugins peut être vu comme une bibliothèque de fonctionnalités. Dans notre cas, on va utiliser le plugin **libgazebo_ros_diff_drive.so** compris dans le paquet (**gazebo**) **diff_drive**. Il est nécessaire de faire le lien entre les éléments de notre robot et ce qu'attend comme paramètres notre plugin.

EXERCICE 26 (*Lien entre le robot et le plugin*)



Recopiez le code suivant dans votre fichier `my_robot_description.urdf` (ou dans un fichier xacro que vous incluez dans votre fichier URDF) pour faire le lien entre les éléments et les paramètres du plugin.

```
<gazebo>
  <plugin name='diff_drive' filename='libgazebo_ros_diff_drive.so'>
    <ros>
      <namespace>/demo</namespace>
    </ros>

    <!-- wheels -->
    <left_joint>drivewhl_l_joint</left_joint>
    <right_joint>drivewhl_r_joint</right_joint>

    <!-- kinematics -->
    <wheel_separation>0.4</wheel_separation>
    <wheel_diameter>0.2</wheel_diameter>

    <!-- limits -->
    <max_wheel_torque>20</max_wheel_torque>
    <max_wheel_acceleration>1.0</max_wheel_acceleration>

    <!-- output -->
    <publish_odom>true</publish_odom>
    <publish_odom_tf>false</publish_odom_tf>
    <publish_wheel_tf>true</publish_wheel_tf>

    <odometry_frame>odom</odometry_frame>
    <robot_base_frame>base_link</robot_base_frame>
  </plugin>
</gazebo>
```



Dans cet exemple, les informations d'odométrie indispensables pour gazebo seront publiées sur le forum `/demo/odom`.

3.3.3 Lancement

Pour simplifier, nous allons uniquement lancer la simulation **gazebo** dans ce qui suit et pas la visualisation sur **rviz**. Pour cela, on va faire un peu de ménage pour retirer **rviz** et la fenêtre associée qui simule la rotation des roues.

EXERCICE 27 (*Cleaning time*)



Dans le fichier manifeste `package.xml`, supprimez la ligne :

```
<exec_depend>joint_state_publisher_gui</exec_depend>
```

Puis dans le fichier `display.launch.py`, supprimez le code suivant :

```
joint_state_publisher_gui_node = launch_ros.actions.Node(
    package='joint_state_publisher_gui',
    executable='joint_state_publisher_gui',
    name='joint_state_publisher_gui',
    condition=launch.conditions.IfCondition(LaunchConfiguration('gui'))
)
```

ainsi que :

```
DeclareLaunchArgument(name='gui', default_value='True',
    description='Flag to enable joint_state_publisher_gui'
)
```

et la ligne suivante dans `joint_state_publisher_node` :

```
condition = launch.conditions.UnlessCondition(LaunchConfiguration("gui"))
```

et la suivante pour ne plus lancer **rviz** à chaque lancement de **gazebo** :

```
ld.add_action(rviz_node)
```



Vous pouvez aussi créer un autre fichier de lancement, spécialisé pour la simulation, tout en gardant celui pour la visualisation.

EXERCICE 28 (*Gazebo parameter*)



Pour pouvoir démarrer **gazebo**, il nous faut configurer son lancement dans le fichier de lancement. Pour cela, ajoutez les lignes suivantes avant les `ldd.actions` :

```
gazebo_sim = ExecuteProcess(
    cmd=['gazebo', '--verbose', '-s', 'libgazebo_ros_init.so', '-s', '
    libgazebo_ros_factory.so'],
)
```

Ajoutez ensuite la cible `gazebo_sim` dans la liste des actions de la description du lancement.

EXERCICE 29 (*Génération de l'avatar*)



Pour générer l'avatar de notre robot dans **gazebo**, complétez le fichier `display.launch.py` en créant (avant les `ldd.actions`) et lançant le nœud composé de l'exécutable `spawn_entity.py` du paquet `gazebo_ros` et qui a comme arguments `-entity, my_robot, -topic` et `robot_description`.

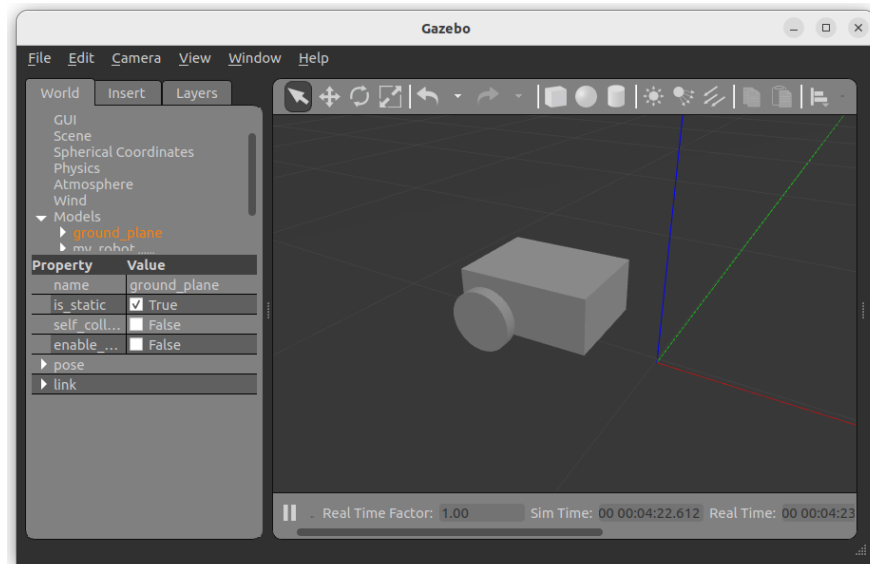
EXERCICE 30 (*Gazebo : 3... 2... 1... Launch*)



Lancez le fichier `display.launch.py`.



Vous devriez voir apparaître à l'écran notre robot dans le simulateur gazebo comme dans la figure suivante :



3.3.4 Rock'n roll

Maintenant, ajoutons un peu de mouvement et de fun dans notre simulation. Pour cela, commençons par rajouter le contrôle de notre robot par les touches du clavier. Sur le principe, comme pour `turtlesim`, il suffit de publier les instructions de vitesses linéaire et angulaire sur le bon forum pour que le robot reçoive et applique les nouvelles consignes de déplacement.



La tâche est simple car il existe un nœud qui fait cela, il suffit de le configurer correctement.

EXERCICE 31 (*Contrôle par clavier*)



Complétez le fichier `display.launch.py` en créant et lançant le nœud composé de l'exécutable `teleop_twist_keyboard` du paquet `teleop_twist_keyboard` qui ouvre une fenêtre `xterm -e`, qui est relancé dans un délai de 3 secondes si on le ferme et qui publie non pas sur le forum par défaut `/cmd_vel` mais sur `/demo/cmd_vel`.



Attention aux vitesses trop élevées : on vous conseille de vous mettre dans une gamme de vitesse linéaire de l'ordre de 0.2.

3.3.5 Création d'un monde



Pour finir, ajoutons un peu de décors à notre monde.

Gazebo n'utilise pas le format URDF, mais un autre format appelé SDF. La modélisation de notre robot est traduite d'URDF vers SDF, mais le monde étant inhérent à Gazebo, il nous faut le créer directement en SDF.

EXERCICE 32 (*Modélisation du monde*)

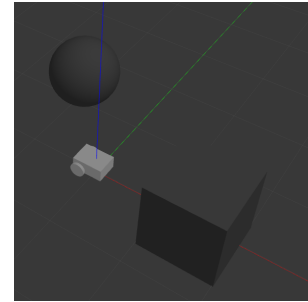


Créez un monde composé d'un cube et d'une sphère.

Pour cela, suivez les explications données sur ce site :

https://classic.gazebosim.org/tutorials?tut=build_world&cat=build_world#Next.

Puis sauvegardez votre création sous le nom de `my_world.sdf` dans notre paquet `my_robot_description` et dans le répertoire `world`



EXERCICE 33 (*Chargement du monde*)



Dans le fichier `display.launch.py`, ajoutez la variable `world` pour accéder à ce fichier :

```
world=os.path.join(pkg_share, 'world/my_world.sdf')
```

Et précisez de lancer gazebo en chargeant `my_world` en complétant la variable `gazebo_sim` ainsi :

```
gazebo_sim = ExecuteProcess(  
    cmd=['gazebo', '--verbose', '-s', 'libgazebo_ros_init.so', '-s', '  
        libgazebo_ros_factory.so', world]  
)
```



Launch your file and have fun!



Fin du quatrième TD

Nous avons découvert dans ce TD comment modeliser et visualiser un robot ainsi que la manière d'utiliser ce modèle dans un logiciel de simulation robotique.

References

[wik23] wikipédia. Moment d'inertie. https://fr.wikipedia.org/wiki/Moment_d'inertie, 2023. 16