

Ros Melodic - URDF - RVIZ - Ros Control - Gazebo

Nicolas Le Gall

Juin 2020



Table des matières

1	Introduction	4
2	L'installation de Ros Melodic	4
2.1	Les prérequis	4
2.2	Installation de Ros Melodic Morenia	4
2.3	Installer les packages	4
3	Navigation à partir de la console	5
4	Présentation rapide de Ros	5
4.1	La philosophie de Ros	5
4.2	Créer un projet sous Ros	6
4.3	Créer un package	6
4.4	Rqt-Graph	6
5	URDF	7
5.1	Présentation de l'URDF	7
5.2	Les xacros	10
5.2.1	leenby.launch	10
5.2.2	display_robot.launch	11
5.2.3	display_robot.rviz	12
5.2.4	description.launch.xml	12
5.2.5	leenby.urdf.xacro	13
5.2.6	L'inclusion d'autres fichiers xacro	15
5.3	Le résultat	19
6	ROS control avec simulation sous Gazebo	20
6.1	Présentation de Ros Control	20
6.2	Utilisation de Ros Control et de Gazebo avec l'exemple du RRBot	21
6.2.1	Installer l'exemple du RRBot	21
6.2.2	Lancer l'exemple du RRBot	21
6.3	Architecture du projet RRBot	22
6.4	Présentation des fichiers	23
6.4.1	Les includes et les appels de fonction	23
6.4.2	rrbot_gazebo / launch / rrbot_world.launch	24
6.4.3	rrbot_control / launch / rrbot_control.launch	25
6.4.4	rrbot_control / config / rrbot_config.yaml	26
7	Controller manager, hardware interface et DiffDriveController	27
7.1	Rappel sur le fonctionnement de ROS control	27
7.2	Les différents fichiers	28
7.3	diff_drive_.launch	28
7.4	BaseMotors.h	29
7.5	BaseMotors.cpp	31
7.5.1	Les includes	31
7.5.2	Le créateur	31
7.5.3	Le destructeur de la classe BaseMotors	32
7.5.4	La fonction initialize()	32
7.5.5	La fonction read()	33
7.5.6	La fonction write()	33
7.5.7	La fonction main()	34
7.6	diff_drive_controller_params.yaml	35
7.7	CMakeLists	37

8	ROS Network	38
8.1	Sur le PC principal - RosMaster	38
8.2	Sur le PC secondaire	38

1 Introduction

ROS signifie Robot Operating System. C'est un environnement permettant de développer, créer, simuler et contrôler des robots. On peut également associer à ces robots différents capteurs, des programmes de traitements d'images, ou encore des connexions avec des appareils externes, comme une télécommande par exemple.

Son système repose sur le fait que ses programmes, que l'on appelle des **nodes**, ne peuvent communiquer entre eux directement. A la place, ils écrivent sur une sorte de tchat, appelé **topic**, et peuvent également lire ce qui s'y trouve. De cette manière, chacun des nodes a accès aux infos qui l'intéressent en lisant les informations sur les topics souhaités. Toute cette architecture est contrôlée par un programme général appelé **master**.

Dans un premier temps, nous verrons comment installer Ros Melodic sur Ubuntu 18.04, ainsi que les packages nécessaires, et les outils pratiques au développement sous Linux, ainsi que la navigation en commande par le terminal. Nous verrons ensuite la philosophie générale de Ros, avant de créer un premier fichier URDF, i.e. un format dédié à la robotique représentant les pièces et les liaisons entre ces pièces. Nous terminerons par la simulation du robot sous Gazebo, par l'intermédiaire de Ros Control.

Ce document est basé sur le robot Leenby. Il peut être adapté pour d'autres projets sous ROS, mais il faudra modifier convenablement des parties afin d'atteindre le projet souhaité.

2 L'installation de Ros Melodic

2.1 Les prérequis

Afin d'installer Ros Melodic, il faut avoir Ubuntu 18.04 comme OS. Si le PC est sous Windows ou un autre OS, il est possible d'installer une machine virtuelle comme VirtualBox et utiliser le fichier d'installation .iso (**lien**) pour créer une version d'Ubuntu 18.04 (**lien**).

2.2 Installation de Ros Melodic Morenia

Il suffit de suivre la procédure disponible sur le site de wiki.ros (**lien**). Toutes les commandes sont à taper dans le terminal, que l'on peut ouvrir en tapant Ctrl+Alt+t sous Ubuntu. L'installation peut prendre jusqu'à quelques heures. Pensez à désactiver du PC ou de la machine virtuelle la veille automatique, afin que l'installation se déroule sans encombre. En cas de problèmes lors de l'installation, il existe de nombreux forums sur internet détaillant les différentes erreurs et leur solution.

2.3 Installer les packages

La commande à effectuer pour installer un package est : `sudo apt-get install ros-melodic-NomDuPackage`

3 Navigation à partir de la console

Sous Linux, et lorsque l'on utilise Ros, il convient la plupart du temps d'utiliser les lignes de commande et le clavier pour se déplacer plus rapidement dans cet environnement.

Pour ouvrir un terminal, on peut faire Ctrl+Alt+t. Il sera alors basé sur le fichier racine de l'utilisateur du PC. Pour ouvrir un terminal directement à un emplacement donné dans le PC, on peut faire clic droit->Open Terminal. Si besoin, on peut ouvrir un autre terminal dans un autre onglet en allant dans File->New Tab. Quelques commandes en vrac :

- pwd : Connaître son répertoire courant
- ls : Voir la liste des fichiers et répertoires dans le répertoire courant
 - ls -a : Voir également les fichiers cachés
 - ls Desktop/dossier : Voir tous les fichiers dans le dossier "dossier", qui est sur le bureau
- cd repertory : Se déplacer dans le dossier repertory
 - cd .. : Revenir au répertoire parent (possible de faire cd ../../.. pour deux fois, cd ../../.. etc)
 - cd ~ : revenir dans le fichier racine de l'utilisateur PC
- mkdir MonRepertoire : Créer un répertoire à l'endroit où l'on est
- touch FichierACreer.txt : Créer un fichier A l'endroit où l'on est, avec l'extension que l'on souhaite.
- cat MonFichier.py : Avoir un aperçu du contenu du fichier
- rm -r Fichier : permet de supprimer Fichier
- sudo : Commencer une ligne de commande par sudo permet de l'exécuter en tant qu'administrateur, ce qui est parfois obligatoire. Le mot de passe de l'utilisateur est alors demandé
- man ls : permet d'ouvrir le manuel correspondant à la commande ls. Marche également avec d'autres commandes.
 - sudo -s : permet d'utiliser le mode super-utilisateur
- apt : A ne pas utiliser seul. Permet la gestion des paquets pour un utilisateur final
 - sudo apt install zoom : permet d'installer le logiciel zoom
 - apt-get : Permet la gestion des paquets pour un script

Dans la console, on peut arrêter la procédure en cours en faisant Ctrl+C.

4 Présentation rapide de Ros

4.1 La philosophie de Ros

Ros fonctionne sur le principe des packages. Ce package peut contenir un ensemble de sous-programmes, des fichiers Stl pour modéliser les éléments du robot, des libraires réservées aux capteurs, au contrôle des actionneurs, ou encore des éléments de filtrages, de modélisation, etc. Un peu de vocabulaire :

- Master : Le master est le programme supervisant tous le reste sous Ros
- Node : Une node est un programme. Elle peut être écrite en C++, en python, en Java, en HTML, en XML, etc.
- Topic : Un Topic est un espace dédié à la réception et à la lecture de messages par les nodes qui sont intéressées. On peut par exemple y trouver la position du robot, les relevés d'odométrie ou encore les informations de commande
- Node publisher : Une node publisher est une node qui publie un message sur un topic. Cette node est publisher pour ce topic, mais pas nécessairement pour un autre topic
- Node Subscriber : Une node subscriber est une node qui lit les messages présents sur un topic. Elle va donc lire le message qui vient d'être envoyé par une node publisher. Une node peut à la fois être publisher et subscriber sur un ou plusieurs topics.

Lorsque l'on démarre notre projet sur Ros, on lance tout d'abord dans un terminal la commande : roscore. Une node rosout est alors créée et le terminal ouvert se concentrera sur le fait de faire tourner cette node. On ouvre ensuite un autre terminal pour lancer le premier programme qui sera une autre node et ainsi de suite. On utilise pour cela la commande : rosrn my_package my_node.

4.2 Créer un projet sous Ros

La première étape est de créer le workspace c'est-à-dire le répertoire de travail en utilisant la commande : `mkdir NomDuProjet`. On se place dedans avec la commande : `cd NomDuProjet`. Puis on crée le répertoire `src` avec toujours : `mkdir src`.

En se plaçant dans `src`, on tape : `catkin_init_workspace`. Si la commande n'est pas reconnue, il faudra faire : `source /opt/ros/melodic/setup.bash`. On revient au niveau du workspace avec de faire : `catkin_make` pour compiler tous les éléments du projet. Le projet est créé. Des fichiers sont générés, contenant par exemple les fichiers à compiler, dans `CMakeList.txt`.

4.3 Créer un package

A ce stade, on doit avoir cette architecture de programme (on ne s'intéressera pas au contenu de `build` et de `devel`, mais ces derniers ont du être automatiquement générés à la suite de la commande `catkin_make`.

- `build`
- `devel`
- `src`
 - `CMakeLists.txt`

Pour créer un nouveau package, qui pourra ensuite contenir nos différents programmes, on se place dans le fichier `src` qui se trouve dans le workspace et on utilise la commande : `catkin_create_pkg MonNomDePackage std_msgs rospy roscpp`.

`std_msgs` permet de renseigner sur le type de message de ce package, qui seront ici des messages de type standart. `Rospy` et `roscpp` informe sur les langages de programmation des nodes incluses dans ce package, soit ici du python et du C++. A ce stade, des fichiers supplémentaires ont été créés. On a donc cette nouvelle architecture :

- `build`
- `devel`
- `src`
 - `CMakeLists.txt`
 - `MonNomDePackage`
 - `include`
 - `MonNomDepackage`
 - `src`
 - `CMakeLists.txt`
 - `package.xml`

Si l'on souhaite créer de nouveaux packages, on peut le faire de la même manière que précédemment, mais également inclure des packages dans celui que l'on vient de créer en se plaçant dans le répertoire `src` inclus dans ce package. Il faut penser à recompiler le projet en se plaçant dans le workspace principal, avec la commande : `catkin_make`.

⚠ A chaque fois qu'une console est redémarrée, il est nécessaire de se placer dans la console et de sourcer `setup.bash` à l'aide de la commande : `source devel/setup.bash`

4.4 Rqt-Graph

`Rqt Graph` est un package permettant de visualiser toutes les nodes et de tous les topics en train de tourner sur Ros. Ce package est très utile pour vérifier la bonne organisation d'un projet et avoir facilement une vue d'ensemble. Pour l'installer, il faut faire les lignes de commandes suivantes :

```
sudo apt-get install ros-melodic-rqt ros-melodic-rqt-tf-tree
sudo apt-get install ros-melodic-rqt-graph
sudo apt-get install ros-melodic-rqt-common-plugins
```

Lorsqu'un projet est lancé, il suffit d'ouvrir une nouvelle console et de taper la commande : `roslaunch rqt_graph rqt_graph` pour afficher sous forme de schéma les nodes et les topics en cours.

5 URDF

5.1 Présentation de l'URDF

L'URDF (Unified Robot Description Format) est un assemblage de deux principaux types d'éléments, qui sont les links et les joints. Les links correspondent aux pièces physiques du robot, tandis que les joints sont les articulations ou les actionneurs, c'est-à-dire qu'ils représentent les mouvements relatifs entre deux links adjacents. Il est possible d'utiliser les pièces au format Stl que l'on peut générer avec Catia, SolidWorks, ... afin de représenter les links. Les links sont également représentés par le repère de base du fichier Stl. Il est donc important avant de commencer à ajouter les pièces au format Stl à l'URDF de placer le repère dans le bon sens, dès lors de la phase de conception mécanique du système. En général, en robotique, on choisit de placer l'axe Z selon le degré de liberté de la liaison, et l'axe X orienté vers la liaison suivante, lorsque cela est possible.

Il est préférable de rédiger ses codes de la manière la plus vague possible, en incluant le maximum de paramètre, afin de pouvoir adapter notre travail au maximum de situation possible. On n'aura plus qu'à chargé un fichier contenant l'ensemble des variables nécessaires en fonction du projet sur lequel on souhaite travailler.

La structure adoptée pour l'URDF et le RVIZ de la leenby est la suivante. Tous ces fichiers se trouve dans le package sur lequel on souhaite travaillant, qui lui-même se trouve dans le src du workspace. URDF permet de définir le robot, tandis que RVIZ permettra d'obtenir un rendu visuel à l'écran.

- urdf
 - leenby.urdf.xacro
 - parameters_leenby.xacro
 - include
 - materials.xacro
 - inertial.xacro
 - wheels.xacro
 - links_chain.xacro
 - kinematic_chains.xacro
- rviz
 - display_robot.rviz
- meshes
 - leenby
 - base.stl
 - elbow.stl
 - head.stl
 - hip_link_pitch.stl
 - neck.stl
 - shoulder.stl
 - shoulder_pitch.stl
 - shoulder_roll.stl
 - shoulder_yaw.stl
 - skirt.stl
 - thumb.stl
 - wrist_pitch.stl
 - wrist_roll.stl
 - wrist_yaw.stl
- launch
 - leenby.launch
 - display_robot.launch
 - include
 - description.launch.xml

légende : Répertoires, Fichiers xacro, Fichiers Stl, Fichiers launch, Fichier XML, Fichiers RVIZ

La description globale du robot est incluse dans le répertoire urdf, dans le fichier `leenby.urdf.xacro`. Il comprend par exemple la mise en place de pièce au format Stl sous forme de link, l'insertion de joints, etc. Ce fichier récupère des

paramètres physiques et mécaniques du robot sous forme de variables, qui sont comprises dans le fichier `parameters_leenby.xacro`. Les fichiers Xacro contenus dans `include` contiennent également des éléments de modélisation. Par exemple, `materials.xacro` contient les différents matériaux, les couleurs à appliquer etc. On sait ainsi que si on souhaite modifier une couleur, il suffit de se rendre dans ce fichier. `wheels.xacro` contient la création de roue sous forme de cylindre. Il est appelé par `leenby.urdf.xacro` afin de définir les roues en déportant les lignes de description dans un autre programme. Cela permet également de pouvoir utiliser ce modèle de roue dans un autre projet si on le souhaite. `kinematic_chains.xacro` contient la description d'un bras de robot. Comme le robot possède deux bras, on peut en inclure un, puis un autre en faisant un effet miroir afin de respecter la symétrie du robot. Cet effet miroir dépend d'une variable qui est définie directement dans le xacro principal.

Le fichier `display_robot.rviz` permet de lancer RVIZ, c'est-à-dire l'affichage de l'URDF que l'on vient de créer. On notera que ce fichier ne comporte pas le nom "leenby". Il peut être utilisé pour l'affichage de différents projets, et c'est pourquoi il est important de rester le plus vague possible en définissant ces sous-programmes.

Le fichier `launch` contient les fichiers qu'on démarre pour lancer le projet. On lance tout simplement `leenby.launch` qui fait ainsi appel à `display_robot.launch`. Une fois de plus, on peut voir que le nom de ce dernier est neutre, car les paramètres se référant à la leenby sont compris dans `leenby.launch`. Il peut donc être adapté à plusieurs projets différents. `description.launch.xml` sert quant à lui à appeler le fichier `leenby.urdf.xacro`.

Dans le projet Leenby, on lance le système en tapant la commande : `roslaunch robot_description leenby.launch`. C'est ce fichier `leenby.launch` qui va appeler d'autres programmes, qui en appelleront eux-mêmes d'autres. Voyons cette structure d'appels et d'incluses. Voyons cette liste d'appels et d'inclusions sous forme d'un diagramme :

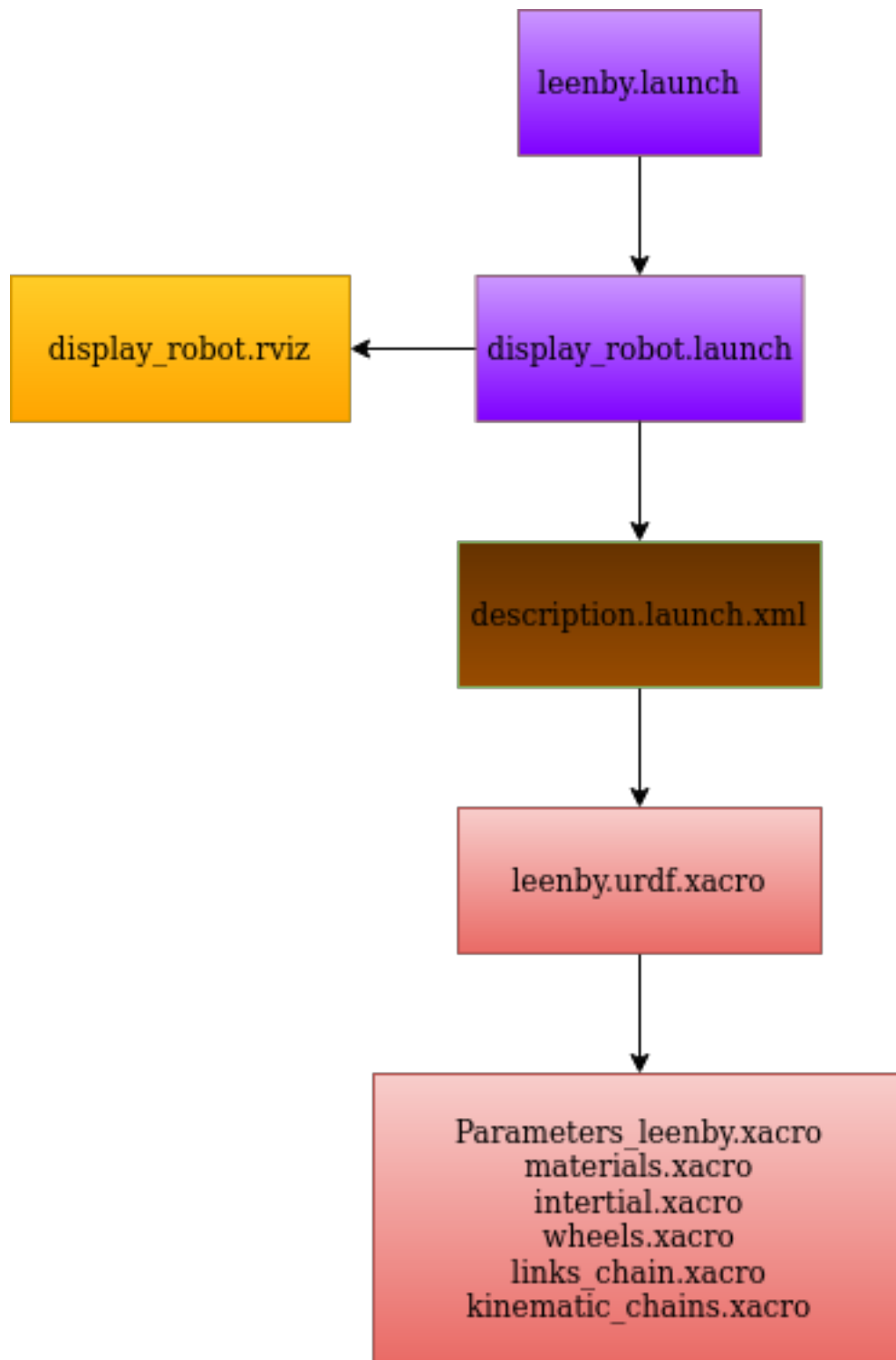


FIGURE 1 – Diagramme des appels et des inclusions pour le projet Leenby

5.2 Les xacros

L'URDF se programme sous forme de commandes Macro. Les Macros que nous utilisons ici sont au format xacro et ressemblent à du langage XML. Je vous conseille d'utiliser l'éditeur Atom pour éditer vos programmes sous Linux.

Une des particularités de ce langage est d'utiliser des tags, ou balises, < et >. Ces tags sont la base de ce langage et permettent de définir les variables, les paramètres, lancer des nodes, include des fichiers, etc.

⚠ Tous les fichiers dans ce langage doivent commencer par la ligne : `<?xml version="1.0"?>`

5.2.1 leenby.launch

Commençons par voir le contenu du fichier `leenby.launch`, qui est le fichier qu'on lance pour démarrer le projet.

```
leenby.launch
1  <?xml version="1.0"?>
2  <launch>
3    <arg name="model" value="leenby"/>
4
5    <include file="$(find robot_description)/launch/display_robot.launch">
6      <arg name="model" value="$(arg model)"/>
7      <arg name="pos_z" value="0.0"/>
8    </include>
9  </launch>
10
```

FIGURE 2 – Contenu du programme `leenby.launch`

A la première, on commence par la ligne obligatoire mentionnée plus tôt. On peut voir à la ligne 2 que l'on ouvre une balise `<launch>`. On définit au coeur de cette balise les éléments qui s'exécuteront lorsque le fichier sera lancé. Cette balise est refermée à la ligne 9 par la balise fermante `</launch>`. Lorsqu'une balise ne fait qu'une ligne, on peut l'ouvrir et la fermer aussitôt comme on peut le voir à la ligne 3.

La ligne 3 permet de définir un argument du nom de `model`, dont la valeur est égale à `leenby`. Cela permettra par la suite d'écrire la commande `$(model)` pour faire référence à `"leenby"`, et ce même dans un path pour rechercher par exemple les pièces au format `stl`. C'est avec ce type de définition que l'on peut rendre les programmes qui seront lancés par la suite les plus vagues possibles, en faisant appel au contenu de l'argument `model`, et non directement en tapant `"leenby"`.

La balise `<include>` ligne 5 suivi de la fermeture de balise ligne 8 permettent de définir l'élément que l'on va inclure. Il s'agit d'un fichier d'où le `file=""`. En XML, les commandes, les calculs et les références à des variables ou arguments se font avec l'opérateur `${}`. On inclut dans les accolades ce que l'on souhaite. Ici la commande `$(find robot_description)` permet de chercher et de se placer dans le fichier mentionné. On choisit le fichier que l'on souhaite lancer, qui se situe dans le répertoire `launch`.

Aux lignes 6 et 7, on définit des arguments qui sont utilisés lors de l'appel du fichier. Dans ce fichier appelé, on aura de cette manière accès à la valeur `"arg model"` en tapant `$(arg model)`. Comme on a défini plus tôt la valeur de l'argument `model`, il aura alors la valeur `"leenby"`.

Pour récapituler, ce fichier permet d'inclure le fichier que l'on souhaite. On y définit également des arguments qui seront utiles dans ces sous-programmes, et qui n'auront donc pas nécessairement comporter le nom `"leenby"` par exemple, car ce dernier a été défini plus tôt.

5.2.2 display_robot.launch

Le fichier `display_robot.launch` a été appelé par celui que nous venons de voir. Rappelons qu'il dispose de deux arguments : `model` qui contient la valeur "leenby" et `pos_z` qui contient la valeur "0.0".

```
display_robot.launch
1  <?xml version="1.0"?>
2  <launch>
3    <arg name="model" default="leenby"/>
4    <arg name="gui" default="true" />
5    <arg name="pos_x" default="0.0" />
6    <arg name="pos_y" default="0.0" />
7    <arg name="pos_z" default="0.15" />
8    <arg name="rvizconfig" default="$(find robot_description)/rviz/display_robot.rviz" />
9
10  <!--
11    <arg name="home_gps" default="$(find robot_description)/launch/rviz_satellite/home.gps"/>
12    <arg name="gui" default="true" />
13    <param name="robot_description" type="String" value="$(arg model)/robot_description"/>
14    <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher" />
15  -->
16  <include file="$(find robot_description)/launch/include/description.launch.xml">
17    <arg name="model" value="$(arg model)" />
18  </include>
19
20
21  <param name="use_gui" value="$(arg gui)"/>
22
23  <node pkg="tf" type="static_transform_publisher" name="static_tf_drone"
24    args="$(arg pos_x) $(arg pos_y) $(arg pos_z) 0 0 0 1 world base_footprint 100" />
25
26  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher"/>
27  <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher" />
28
29  <node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)" required="true"/>
30
31  </launch>
```

FIGURE 3 – Contenu du programme `display_robot.launch`

Une nouvelle fois, on commence le programme par la première ligne habituelle, et on ouvre la balise `<launch>` que l'on ferme à la fin.

De la ligne 3 à la ligne 8, on définit des arguments, et on leur rentre une valeur par défaut, au cas où elle n'aurait pas été initialisée auparavant. Si toutefois on choisit une valeur différente de celle par défaut, cette valeur sera bien sûr conservée et celle par défaut ne servira pas. On voit ici par exemple que seuls les arguments `model` et `pos_z` ont été renseignés. Ils conservent donc la valeur qu'ils avaient auparavant (même si la valeur par défaut est la même). Les autres arguments que l'on n'avait pas définis auparavant existent désormais et ont tous par défaut la valeur attribuée ici. Une fois de plus, on voit à la ligne 8 la commande `$(find robot_description)/rviz/display_robot.rviz` permettant de viser le fichier `display_robot.rviz` en particulier, en recherchant automatiquement le répertoire `robot_description`.

Les lignes 10 à 15 permettent de voir la notation des commentaires en XML. Il suffit d'ouvrir le commentaire avec `<!--` et le finir avec `-->`.

Avec les lignes 16 à 18, on lance le fichier `description.launch.xml`, et on y inclut l'argument `model`, qui a été paramétré à la valeur "leenby" dans le programme précédent. La ligne 21 permet de définir le paramètre "use_gui" à la valeur "arg gui", qui a été définie ligne 4 par défaut à la valeur "true". Contrairement aux arguments que l'on crée avec la commande `arg` et qui sont des éléments locaux, les éléments définis avec `param` sont globaux et seront utilisables dans les autres programmes sans avoir à les rappeler.

Enfin, les lignes 23 à 29 permettent de lancer les différents nodes nécessaires pour ce robot sous RVIZ. Elles sont incluses dans des packages définies lors de l'installation de Ros et de ses packages. Le chemin de la node ligne 29 rviz a été défini en argument par défaut ligne 8.

5.2.3 display_robot.rviz

Ce fichier permet de décrire les paramètres au lancement de RVIZ, qui permet l'affichage à l'écran de l'URDF, et de pouvoir déplacer comme on le souhaite les différents joints de l'assemblage. Il n'est pas programmé en XML mais dans un langage se rapprochant plus du système de Python dans l'utilisation des tabulations. Il est disponible dans le package sur forge.xlim.fr et comporte dans le cas de la Leenby plus de 400 lignes. Les premières lignes peuvent être copiées-collées, voire réadaptée si besoin. Ensuite, il vous faudra définir l'ensemble des caractéristiques pour tous les links qui seront créés dans l'URDF.

5.2.4 description.launch.xml

```
description.launch.xml
1 <?xml version="1.0"?>
2 <launch>
3   <arg name="model" />
4
5   <arg name="urdf_file" default="$(find xacro)/xacro --inorder '$(find robot_description)/urdf/${arg model}.urdf.xacro'" />
6   <param name="robot_description" command="$(arg urdf_file)" />
7 </launch>
```

FIGURE 4 – Contenu du programme `description.launch.xml`

Les premières lignes sont habituelles. Une fois de plus, on vient lancer un autre programme en incluant l'argument `model` défini au tout début. On définit l'argument `"urdf_file"` par défaut initialisé à sa valeur. `$(find xacro)/xacro --inorder` permet d'autoriser la commande suivante à rechercher dans tous le fichier actuel.

La ligne 6 permet de définir le paramètre `"robot_description"`, qui a pour action de commander l'ouverture du fichier URDF dont le chemin a été spécifié juste au-dessus, d'où l'action `command="$(arg urdf_file)"`. On aurait pas pu faire un `include` comme dans les cas précédent car le format du fichier n'est ici pas le même.

5.2.5 leenby.urdf.xacro

```
leenby.urdf.xacro
1  <?xml version="1.0"?>
2  <robot name="leenby" xmlns:xacro="http://ros.org/wiki/xacro">
3    <xacro:include filename="$(find robot_description)/urdf/parameters_leenby.xacro" />
4    <xacro:include filename="$(find robot_description)/urdf/include/materials.xacro" />
5    <xacro:include filename="$(find robot_description)/urdf/include/inertial.xacro"/>
6    <xacro:include filename="$(find robot_description)/urdf/include/wheels.xacro"/>
7    <xacro:include filename="$(find robot_description)/urdf/include/links_chain.xacro"/>
8    <xacro:include filename="$(find robot_description)/urdf/include/kinematic_chains.xacro"/>
9
```

FIGURE 5 – Début du programme `leenby.urdf.xacro`

C'est dans ce programme que le robot va être décrit. On peut voir que pour commencer à décrire le robot, une balise `<robot>` est ouverte au début. On y renseigne le nom du robot, et on ajoute toujours l'élément `xmlns:xacro="http://ros.org/wiki/xacro"`. Cette balise robot sera fermée à la toute fin du programme, lorsqu'on aura fini de décrire tous le robot.

Les lignes 3 à 8 permettent d'inclure le contenu des différents fichiers cités. Ils contiennent par exemple des grandeurs caractéristiques, des bibliothèques de couleurs, des chaînes cinématiques prêtes à poser dans cet URDF, etc. La commande à utiliser est `<xacro:include filename=PATH />` car il s'agit de fichier Xacro à inclure, contrairement à ce que l'on faisait auparavant.

On peut maintenant commencer à décrire le robot, en utilisant si on le souhaite des éléments décrits dans les fichiers inclus.

```
11  <!-- Used for fixing robot to Gazebo 'base_link' -->
12  <link name="base_footprint"/>
```

FIGURE 6 – Déclaration d'un link minimal dans `leenby.urdf.xacro`

Ceci est le minimum pour définir une link, c'est-à-dire une pièce. Elle n'a aucune géométrie associée, ni de masse. C'est tout simplement une pièce virtuelle qui servira de projection de l'axe verticale du robot sur le repère du monde. On peut voir que pour définir une link, on utilise la balise `<link>`. Il faut au minimum donner un nom à cette pièce, qui sera utilisé par la suite pour y faire référence.

```
13  <joint name="base_joint" type="fixed">
14    <parent link="base_footprint"/>
15    <child link="base_link"/>
16    <origin xyz="0 0 ${diam_frontwheels/2+0.02}" rpy="0 0 0"/>
17  </joint>
```

FIGURE 7 – Déclaration d'un joint minimal dans `leenby.urdf.xacro`

La déclaration des joints se fait à l'aide de la balise du même nom. De la même manière que pour les links, il faut définir un nom. On doit également définir le type de liaison dont il s'agit. Ici il s'agit d'une liaison `fixed`. C'est-à-dire qu'il n'y aura pas de mouvement relatif entre les deux pièces. Nous verrons plus tard deux autres types de liaisons. Il faut définir un link parent et un link child. Cela signifie que l'on vient créer une articulation entre deux repères : le repère de la link parent et de la link child. Le link child n'existe ici pas encore mais sera créé juste après.

Dans la balise `origin`, on définit la position relatif du repère de la base parent et de la base. Les coordonnées sont données par rapport au repère de la link parent. Tout d'abord on définit le déplacement selon les trois axes. Ici on a 0 selon x et y , et $\text{diam_frontwheels}/2 + 0.02$ selon z . Cette valeur est définie dans le fichier inclus `parameters_leenby.xacro`. On peut également ajouter des rotations autour des trois axes. Il faut noter que toutes les rotations se font selon les axes du repère parent et que l'unité est en radians.

```

20 <!-- BASE -->
21 <link name="base_link">
22   <visual>
23     <geometry>
24       <mesh filename="package://robot_description/meshes/leenby/base.stl" scale="0.001 0.001 0.001" />
25     </geometry>
26     <material name="cream" />
27     <origin xyz="0 0 0" rpy="0 ${-pi/2} 0"/>
28   </visual>
29   <collision>
30     <geometry>
31       <mesh filename="package://robot_description/meshes/leenby/base.stl" scale="0.001 0.001 0.001" />
32     </geometry>
33     <origin xyz="0 0 0" rpy="0 ${-pi/2} 0"/>
34   </collision>
35   <inertial>
36     <mass value="${mass_base}"/>
37     <origin xyz="0 0 ${diam_frontwheels/2 + 0.02}"/>
38     <inertia ixx="1e-3" ixy="0.0" ixz="0.0" iyy="1e-3" iyz="0.0" izz="1e-4"/>
39   </inertial>
40 </link>
41

```

FIGURE 8 – Déclaration d'une link dans `leenby.urdf.xacro`

On définit ici la link "base_link", qui correspond au child de la joint précédemment créée. On ouvre comme d'habitude la balise link, que l'on ferme une fois la link décrite. Cette link se décompose en trois caractéristiques différentes : visual, collision et inertial. Visual correspond, comme son nom l'indique, au visuel qui va ressortir, ce qu'on verra sous RVIZ. Il est composé d'une géométrie correspondant à la forme physique de l'objet, un matériau faisant référence à la couleur "cream" présentée dans le fichier inclus `materials.xacro`, et d'une origine.

Le fichier stl qu'on implémente ligne 24, et dont l'échelle est définie en millimètre (le mètre est l'unité par défaut si on ne met pas ce paramètre) possède un repère qui lui est propre. Il est cependant possible de dire que le repère de la pièce n'est pas son repère de base en déplaçant l'origine du repère, et en le faisant tourner. Comme pour le joint, tous les déplacements se font selon le repère d'origine. Il est cependant préférable d'avoir une pièce dont le repère d'origine est le bon. Cela facilitera le changement d'un des fichiers stl si besoin.

La balise collision permet à Ros de connaître la géométrie de l'objet. On y met généralement les mêmes valeurs que dans la balise visual. La balise inertial sert quant à elle à définir le centre de gravité de l'élément, sa masse, son inertie selon chaque axe. Cette caractéristique sera importante par la suite lorsqu'on modélisera le comportement du robot. Par exemple, un contrôleur avec un régulateur PID sur un servomoteur déplaçant le bras ne se comportera pas de la même manière selon la masse ou le centre du gravité de chacune des pièces, de l'épaule au poignée.

```

188 <joint name="head_joint" type="revolute">
189   <parent link="neck_link_yaw"/>
190   <child link="head_link"/>
191   <limit effort="50" lower="${-m_pi}" upper="${m_pi}" velocity="0.2"/>
192   <axis xyz="0 0 1"/>
193   <origin xyz="0 0 ${lenz_neck_head}" rpy="${pi/2} 0 ${pi/2}"/>
194   <dynamics damping="0.1"/>
195 </joint>

```

FIGURE 9 – Déclaration d'une joint pivot dans `leenby.urdf.xacro`

Afin de définir une liaison pivot, on utilise cette structure. Comme pour le joint précédent, on définit un nom, et un type qui est dans le cas de la pivot "revolute". On choisit le link parent et le link child. On choisit l'axe de révolution de la pivot. Il s'agit ici de l'axe Z (comme on aura la plupart du temps à cause de la norme robotique). On définit également une origin comme fait pour le joint base_joint, avec des rotations et translations si besoin.

Cette liaison permet également de définir des limites d'efforts. Elle est ici fixée à 50 N.m (unité à vérifier). Les cases lower et upper permettent de choisir l'amplitude de chaque côté de la liaison de la liaison pivot. Le paramètre velocity permet de définir la vitesse maximale pour cette liaison.

```
<joint name="right_plier_joint" type="prismatic">
  <parent link="n1_wtool"/>
  <child link="right_plier_link"/>
  <origin rpy="{-pi/2} 0 0" xyz="0 0 0"/>
  <axis xyz="0 0 1"/>
  <limit effort="1000.0" lower="-200e-3" upper="0.0" velocity="0.5"/>
</joint>
</xacro:macro>
```

FIGURE 10 – Déclaration d'une joint glissière dans le projet Rose

Voici un exemple de définition d'une liaison glissière. La structure demeure similaire à ce qui a été vu précédemment. Cet exemple n'est pas dans le projet Leenby car ce robot ne possède pas de liaison glissière.

5.2.6 L'inclusion d'autres fichiers xacro

Tous les éléments sont maintenant à votre disposition pour créer un fichier URDF et le visualiser sous RVIZ ensuite. Il est cependant possible de définir des chaînes cinématiques compliquées de pièces dans d'autres fichiers, ou encore de créer des pièces simples telles que des cylindres pouvant représenter des roues.

Commençons par une macro permettant d'inclure des roues que l'on définit dans un autre fichier nommé **wheels.xacro**, qui se trouve dans le répertoire **include**.

```
53 <xacro:caster_wheel prefix="rear_left" base="base_link"
54     xyz="{-len_wheelbase/2} {-len_casterwheeltrack/2} {diam_frontwheels/2}"
55     radius="{diam_frontwheels/2}" mass="0.1"/>
```

FIGURE 11 – Inclusion d'une roue dans **leenby.urdf.xacro** depuis **wheels.xacro**

On voit que pour appeler cette xacro, on utilise la balise <xacro, dont le namespace est caster_wheel. Cela signifie simplement que la macro que l'on cherche se nomme "caster_wheel"

Cette xacro a besoin de différents arguments en entrée, qui sont prefix, base, xyz, radius et mass. Encore une fois, ces arguments permettent de rendre la macro caster_wheel modulable afin de servir à plus de programmes.

Voici le contenu de cette macro `wheels.xacro` :

```
98 <xacro:macro name="caster_wheel" params="prefix base xyz radius mass">
99   <link name="${prefix}_caster_wheel_link">
100     <visual>
101       <geometry>
102         <sphere radius="${radius}" />
103       </geometry>
104       <material name="tire" />
105     </visual>
106     <collision>
107       <geometry>
108         <sphere radius="${radius}" />
109       </geometry>
110     </collision>
111     <inertial>
112       <mass value="${mass}" />
113       <xacro:sphere_inertia m="${mass}" r="${radius}" />
114     </inertial>
115   </link>
116
117   <joint name="${prefix}_wheel_hinge" type="fixed">
118     <parent link="${base}" />
119     <child link="${prefix}_caster_wheel_link" />
120     <origin xyz="${xyz}" rpy="0 0 0" />
121   </joint>
122
123   <gazebo reference="${prefix}_caster_wheel_link">
124     <mu1>0.0</mu1>
125     <mu2>0.0</mu2>
126     <kp>100000000.0</kp>
127     <kd>1.0</kd>
128     <material>Gazebo/Gray</material>
129   </gazebo>
130
131 </xacro:macro>
```

FIGURE 12 – Déclaration de la macro `caster_wheel` dans `wheels.xacro`

Ceci n'est qu'un extrait du fichier entier. Il commence bien entendu toujours par `<?xml version="1.0"?>` suivi de la balise `<robot xmlns:xacro="http://ros.org/wiki/xacro">` (cette balise est à refermer à la fin du fichier par `</robot>`).

Après avoir défini le nom de la macro, on rappelle quels sont les paramètres à avoir obligatoirement en entrée. On peut ensuite définir les links et les joints de la même manière que dans le programme principal. Il ne faut pas faire attention pour l'instant à la définition de la balise gazebo qui ne sert à rien dans notre cas. On ferme ensuite notre macro qui n'a plus qu'à être appelée dans tous les programmes que l'on veut.

On peut également utiliser un fichier inclus pour définir les variables du système, comme ce qui est fait dans notre fichier `parameters_leenby.xacro` dont voici un extrait :

```
1  <?xml version="1.0"?>
2  <robot xmlns:xacro="http://ros.org/wiki/xacro">
3    <xacro:property name="model" value="leenby" />
4    <xacro:property name="rad_base" value="0.32" />
5    <xacro:property name="mass_base" value="15" />
```

FIGURE 13 – Déclaration de variables dans `parameters_leenby.xacro`

On voit ici clairement la structure pour définir des variables et leur associer une valeur. Les valeurs contenues dans les variables sont accessibles dans le programme qui les inclut uniquement, en tapant `${model}` pour prendre comme exemple la variable `model`.

Pour finir, nous allons voir comment ajouter une chaîne cinématique que l'on définit dans un fichier à part, comme dans le cas de la roue. Voici comment on inclut c'est chaîne cinématique depuis le programme principal :

```
205 <xacro:kchain_arm_stl_7R
206   base="left_arm_mount" id="left" reflect="1"
207   link1="shoulder_pitch"
208     stl1="${model}/shoulder_pitch.stl"
209     x1="0" y1="0" z1="${lenlink_shoulder}" m1="0.2" r1="${pi}"
210     lx1="0" ly1="0" lz1="0"
211     jx1="0" jy1="${y_poitaine/2}" jz1="${lenz_shoulder/2}"
212   link2="shoulder_roll"
213     stl2="${model}/shoulder_roll.stl"
214     x2="${len_arm*len_arm_percentage}" y2="0.05" z2="0.05" m2="0.2" r2="${pi}"
215     lx2="0" ly2="0" lz2="0"
216     jx2="0" jy2="0" jz2="${lenlink_shoulder}"
217   link3="shoulder_yaw"
218     stl3="${model}/shoulder_yaw.stl"
219     x3="0.05" y3="0.05" z3="${len_arm*(1-len_arm_percentage)}" m3="0.2" r3="${pi}"
220     lx3="0" ly3="0" lz3="${(len_arm*(1-len_arm_percentage))/2}"
221     jx3="${len_arm*len_arm_percentage}" jy3="0" jz3="0"
```

FIGURE 14 – Appel d'une chaîne dans `leenby.urdf.xacro`

Dans cet exemple, on ne verra une chaîne que pour 3 pièces à la suite, mais la méthodologie est la même pour un nombre de pièce supérieur. La balise `<xacro:kchain_arm_stl_7R` ayant été ouverte, elle est bien entendue fermée à la fin de la déclaration par `</>`. Cet appel est exactement le même que dans le cas de la `caster_wheel` vu avant, mais avec plus de paramètres en entrée. Allons voir la définition de cette chaîne cinématique dans le fichier `kinematic_chains.xacro` :

```

194 <xacro:macro name="kchain_arm_stl_7R" params="
195   base id reflect
196   link1 stl1 x1 y1 z1 m1 r1 lx1 ly1 lz1 jx1 jy1 jz1
197   link2 stl2 x2 y2 z2 m2 r2 lx2 ly2 lz2 jx2 jy2 jz2
198   link3 stl3 x3 y3 z3 m3 r3 lx3 ly3 lz3 jx3 jy3 jz3
199   link4 stl4 x4 y4 z4 m4 r4 lx4 ly4 lz4 jx4 jy4 jz4
200   link5 stl5 x5 y5 z5 m5 r5 lx5 ly5 lz5 jx5 jy5 jz5
201   link6 stl6 x6 y6 z6 m6 r6 lx6 ly6 lz6 jx6 jy6 jz6
202   link7 stl7 x7 y7 z7 m7 r7 lx7 ly7 lz7 jx7 jy7 jz7
203   effort velocity">
204
205 <xacro:link_element_stl
206   prefix="{link1}" stl="{stl1}" suffix="{id}" base="{base}"
207   x="{x1}" y="{y1}" z="{z1}" mass="{m1}"
208   link_xyz="{lx1*reflect}" "{ly1}" "{lz1*reflect}" joint_xyz="0 0 0" joint_rpy="0 0 0"
209   j_max="{r1}" j_min="{-r1}" j_effort="{effort}" j_velocity="{velocity}"/>
210
211 <xacro:link_element_stl
212   prefix="{link2}" stl="{stl2}" suffix="{id}" base="{link1}_link_{id}"
213   x="{x2}" y="{y2}" z="{z2*reflect}" mass="{m2}"
214   link_xyz="{lx2}" "{ly2}" "{lz2*reflect}" joint_xyz="{jx2}" "{jy2}" "{jz2}" joint_rpy="{-reflect*pi/2} 0 0"
215   j_max="{r2}" j_min="{-r2}" j_effort="{effort}" j_velocity="{velocity}"/>
216
217 <xacro:link_element_stl
218   prefix="{link3}" stl="{stl3}" suffix="{id}" base="{link2}_link_{id}"
219   x="{x3}" y="{y3}" z="{z3*reflect}" mass="{m3}"
220   link_xyz="{lx3}" "{ly3}" "{lz3}" joint_xyz="{jx3}" "{jy3}" "{jz3*reflect}" joint_rpy="0 {pi/2} 0"
221   j_max="{r3}" j_min="{-r3}" j_effort="{effort}" j_velocity="{velocity}"/>

```

FIGURE 15 – Déclaration d'une chaîne cinématique `kinematic_chains.xacro`

Ce cas est très similaire à celui de la roue. Dans un premier temps on définit le nom de la chaîne cinématique et on mentionne tous les paramètres requis. Ayant dans le cas de la Leenby 7 pièces différentes, il y a plus de paramètres. Dans le cas présenté, on ne se sert que des paramètres correspondant au 3 premiers links. Les links sont créés ici en utilisant une autre xacro nommée `link_element_stl`. Cette macro est dans le fichier `links_chains.xacro` et contient exactement la même structure que pour la définition de la macro `caster_wheel`, aux joints et noms de paramètres prêts.

5.3 Le résultat

Pour lancer le tout, on se rend dans le workspace via un terminal, on le compile avec `catkin_make` et on lance la commande `roslaunch robot_description leenby.launch`. On peut faire `Ctrl+C` pour stopper le terminal, ce qui fermera tous les fichiers lancés par ce terminal. Dans le cas du projet sur le leenby, on obtient le résultat suivant :

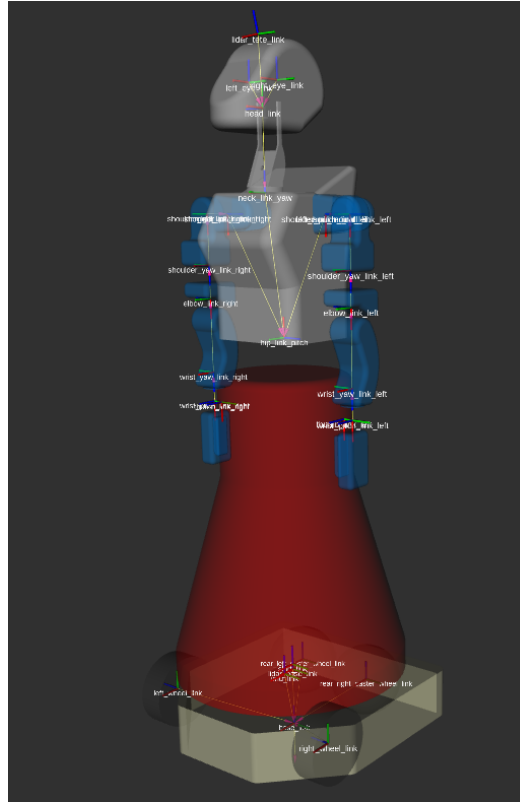


FIGURE 16 – Résultat sous RVIZ de l'URDF de la Leenby

A l'ouverture de RVIZ, une petite fenêtre s'ouvre permettant d'articuler le robot selon les joints que l'on a définis. Les repères sont également affichés : rouge pour l'axe des X, vert pour Y et bleu pour Z.

On peut aller dans Panels et cocher la case Displays afin de pouvoir cacher ou afficher ce que l'on souhaite.

On déplace l'affichage du robot à l'aide de la souris : clic gauche maintenu et glisser pour tourner autour, molette maintenu et glisser pour déplacer, et enfin clic droit et glisser pour zoomer et dézoomer.

6 ROS control avec simulation sous Gazebo

Nous venons de voir comment établir un fichier URDF et le visualiser à l'aide d'RVIZ. Avec la petite fenêtre qui apparaît, on peut déplacer les articulations, mais cela ne permet en fait que la visualisation, et ne pourrait pas déplacer le robot réel. Il faut pour cela utiliser Ros Control, qui permettra d'envoyer des consignes directement aux actionneurs, par l'intermédiaire de régulateur PID paramétrable.

Comme nous n'allons pas contrôler directement le robot réel, nous simulerons le robot réel à l'aide de Gazebo. Gazebo se basera alors sur le fichier URDF que nous avons créé pour connaître la géométrie, les limites des joints, les centres de gravité, les inerties, etc. Le robot virtuel pourra être déplacé, et Gazebo pourra même simuler le bruit de mesure, les retours capteurs, la gravité, les perturbations, etc. Si la simulation fonctionne correctement avec Gazebo, il n'y aura qu'à remplacer le robot virtuel fonctionnant sous Gazebo par le robot réel : Ros Control n'y verra aucune différence!

6.1 Présentation de Ros Control

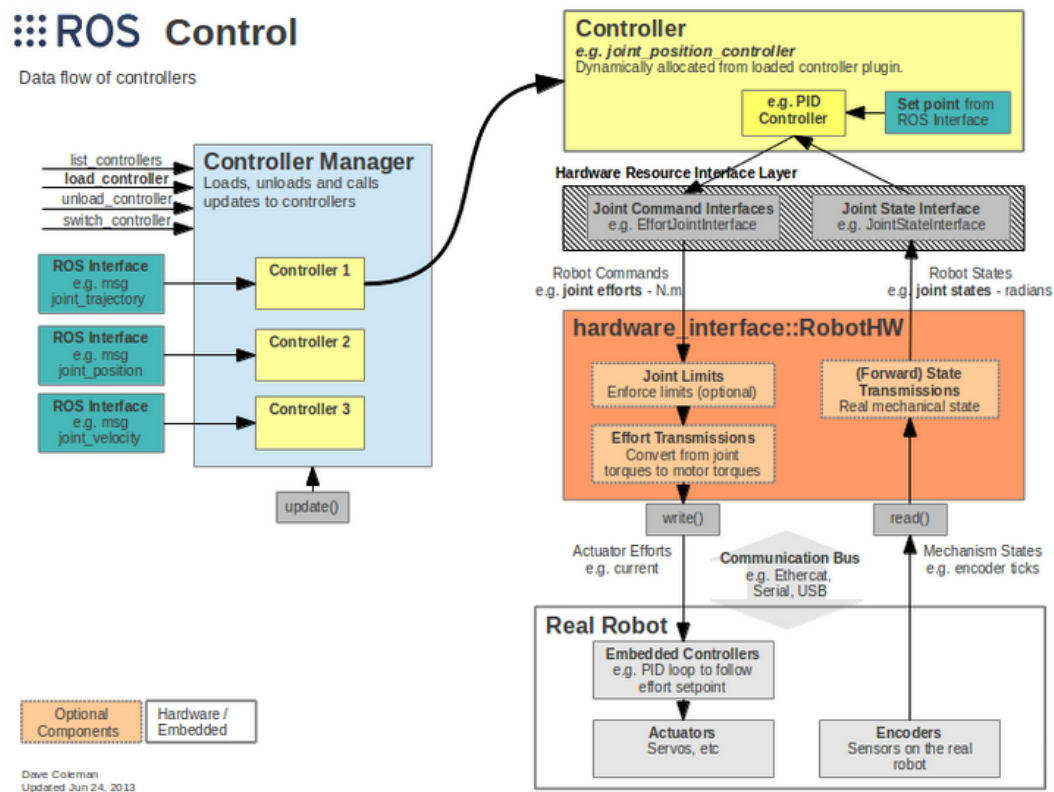


Diagram source in `ros_control/documentation`

FIGURE 17 – Schéma de fonctionnement de ROS control ([lien Ros Wiki](#))

Afin de commander le robot, le controller manager gère la commande du robot. Il peut s'agir d'un controller que l'on a créé, ou d'un controller qu'on utilise déjà "tout prêt" et disponible dans les packages ROS. Ce point sera davantage abordé dans la section suivante. Ce controller manager envoie la commande qu'il faut au controller qui prend en charge la régulation PID. L'"hardware interface" permet de choisir ce que l'on veut envoyer ou recevoir. Ce sont par exemple un effort, une vitesse, une position, ... L'hardware interface permet aussi l'effort transmission, c'est-à-dire la conservation de la puissance en fonction de la commande qu'on applique. Vient ensuite le robot réel (ou dans notre cas le robot que l'on simule sous Gazebo) qui déplacera le robot. Les capteurs permettent de fermer la boucle et créer une commande régulée pour le PID.

6.2 Utilisation de Ros Control et de Gazebo avec l'exemple du RRBot

6.2.1 Installer l'exemple du RRBot

Il faut tout d'abord se placer dans un workspace, dans le fichier src. On fait alors :

```
git clone https://github.com/ros-simulation/gazebo_ros_demos.git
cd ..
catkin_make
```

Le robot RRBot est alors installé dans un nouveau package nommé gazebo_ros_demos.

6.2.2 Lancer l'exemple du RRBot

On peut lancer la simulation du robot avec la commande : `roslaunch rrbot_gazebo rrbot_world.launch`. Comme dans le cas du lancement de RVIZ pour voir l'URDF dans la section précédente, faire un launch de ce fichier en ouvre d'autre en tâche de fond, qui en ouvre d'autres, etc. Il démarre donc gazebo, et affiche le robot qui est décrit dans l'URDF du projet. Une fenêtre sous Gazebo s'ouvre avec la simulation du robot. Pour contrôler le robot, il faut lancer Ros Control avec la commande `roslaunch rrbot_control rrbot_control.launch`. Pour déplacer le robot, on peut alors envoyer des informations depuis une autre console ou une node. Une node existe dans ce projet pour le déplacer de manière visuelle. On la démarre avec : `roslaunch rrbot_control rrbot_rqt.launch`.

Il est également possible de lancer automatiquement le fichier `rrbot_control.launch` en lançant `rrbot_world.launch`. Pour cela, il faut décommenter la ligne qui l'appelle à la fin du fichier `rrbot_world.launch`, en prenant bien soin de laisser les tags `<et >`. Dans la suite, on considérera que cette manipulation a été faite.

Après avoir utilisé cette commande, lancer `rqt_graph` avec `roslaunch rqt_graph rqt_graph` pour voir les nodes et les topics qui ont été créés. On peut voir qu'il y a un topic `/rrbot/Joint1_position_controller/command` qui dialogue avec le node `/rrbot_rqt`, qui permettra de contrôler le robot. Ce topic est inclus dans `/rrbot/Joint1_position_controller`, lui même appartenant au robot `/rrbot`.

Dans la fenêtre `rrbot_rqt.perspective - rqt`, on peut voir différents éléments. Tout d'abord, commencer par cocher la case correspondant au topic `/rrbot.Joint1_position_controller/command`. Cela devrait déplacer le premier joint du robot dans la fenêtre Gazebo, en activant le déplacement qui est visible en déroulant cet objet (le petit triangle à gauche du topic). On note que l'on peut choisir la fréquence de publication des commandes, soit ici 100 Hz, le type de message et le type du contenu du message. On peut voir dans l'outil `rqt` un graph sur la droite représentant la consigne et l'état du robot. On peut ainsi voir l'erreur.

On a réussi de cette manière à contrôler la première articulation. Pour contrôler la seconde, il faut ajouter un topic en sélectionnant le bon topic. Il s'agit du même nom que le topic présent, mais en remplaçant "Joint1" par "Joint2". Le type de message et la fréquence sont les mêmes. On définit la position consigne que l'on souhaite, par l'intermédiaire d'un nombre `float64`, ou bien une fonction comme le $\sin(i/100)$ appliquée pour le premier topic. Si on coche la case, la deuxième articulation du robot devrait également se déplacer.

6.3 Architecture du projet RRBot

On se place ici dans le package `gazebo_ros_demos`. L'architecture des fichiers que vous avez également est la suivante :

- `rrbot_control`
 - `config`
 - `rrbot_control.yaml`
 - `launch`
 - `rrbot_control.launch`
 - `rrbot_rqt.launch`
 - `rrbot_rqt.perspective`
- `rrbot_description`
 - `launch`
 - `rrbot.rviz`
 - `rrbot_rviz.launch`
 - `meshes`
 - `hokuyo.dae`
 - `urdf`
 - `materials.xacro`
 - `rrbot.gazebo`
 - `rrbot.xacro`
 - `rrbot.xml`
- `rrbot_gazebo`
 - `launch`
 - `rrbot_world.launch`
 - `worlds`
 - `rrbot.world`

Le contenu du répertoire `robot_description` est assez similaire à celui de la section précédente.

`Robot_control` contient les fichiers nécessaire au lancement et au paramétrage de Ros Control. Le fichier `rrbot_control.yaml` permet de paramétrer les controllers de chaque liaison, le type de ce controller, le joint correspondant et les constantes du PID.

`Robot_control.launch` indique que le Ros Control se basera sur les paramètres présents dans le fichier yaml précédent et charge les différents controllers.

Quant à `rrbot_rqt.launch` et `rrbot_rqt.perspective`, ils ne servent qu'à décrire l'élément visuel qui contrôle le robot dans notre cas.

Le répertoire `robot_gazebo` contient, comme son nom l'indique, les éléments permettant de décrire la simulation Gazebo. Le fichier qu'on lance en premier `rrbot_world.launch` fait appel à différents fichiers permettant de lancer les nodes. Il définit au passage quelques arguments et indique que l'URDF est celui contenu dans le répertoire `robot_description`. Dans le cas d'un robot réel, gazebo serait remplacé par ce robot réel puisqu'il ne sert qu'à la simulation.

6.4 Présentation des fichiers

6.4.1 Les includes et les appels de fonction

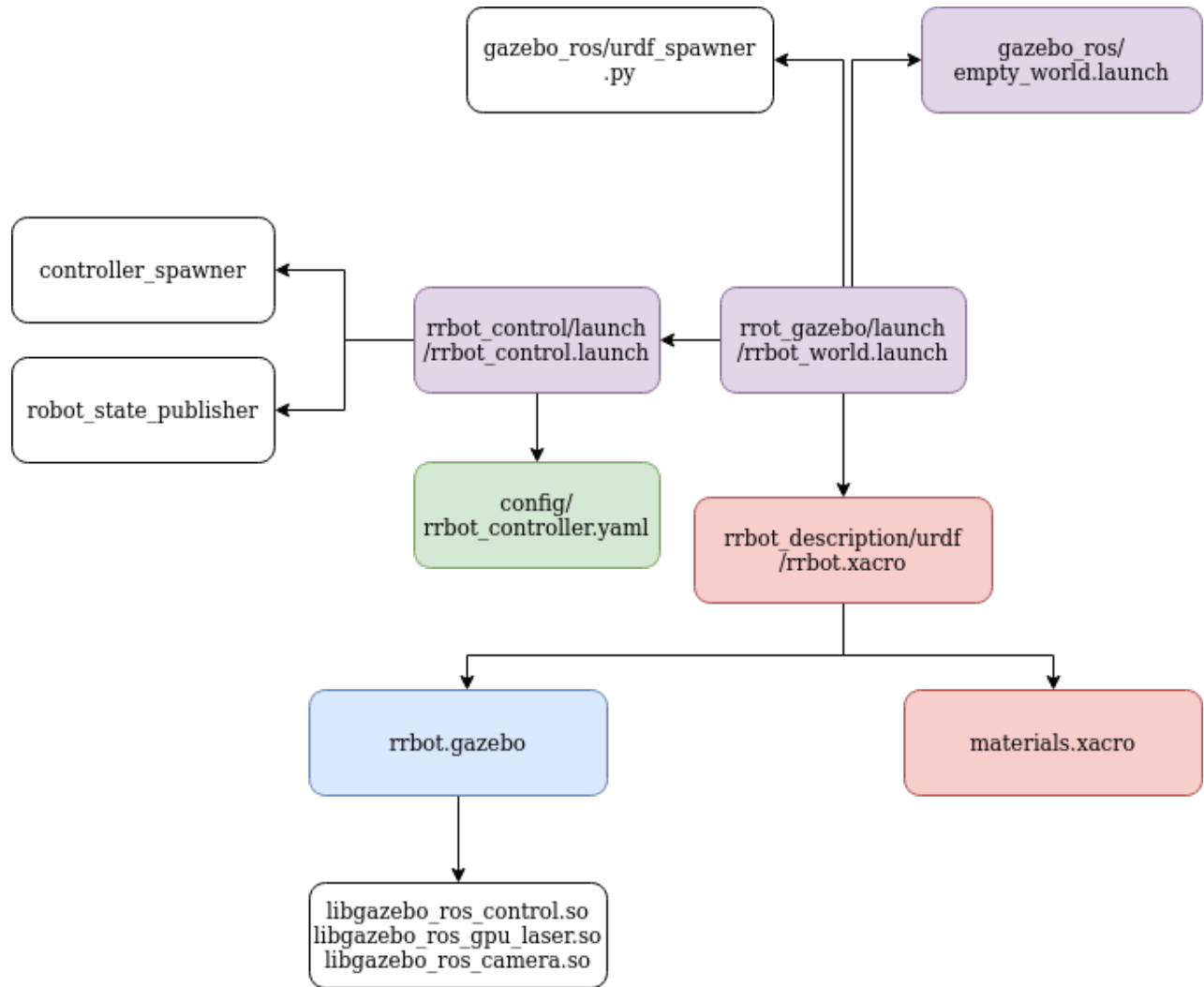


FIGURE 18 – Schéma des inclusions et appels de fonction pour l'exemple du RRBot

Les fichiers laissés en blanc sont inclus dans les fichiers récupérés en installant Ros Control ou Gazebo. Les autres sont ceux présents dans l'environnement, présentés dans l'arborescence au point 6.3. Regardons tous ces fichiers un à un, par ordre d'inclusion.

6.4.2 rrbot_gazebo / launch / rrbot_world.launch

```
rrbot_world.launch
1 <launch>
2
3 <!-- these are the arguments you can pass this launch file, for example paused:=true -->
4 <arg name="paused" default="false"/>
5 <arg name="use_sim_time" default="true"/>
6 <arg name="gui" default="true"/>
7 <arg name="headless" default="false"/>
8 <arg name="debug" default="false"/>
9
10 <!-- We resume the logic in empty_world.launch, changing only the name of the world to be launched -->
11 <include file="$(find gazebo_ros)/launch/empty_world.launch">
12   <arg name="world_name" value="$(find rrbot_gazebo)/worlds/rrbot.world"/>
13   <arg name="debug" value="$(arg debug)" />
14   <arg name="gui" value="$(arg gui)" />
15   <arg name="paused" value="$(arg paused)" />
16   <arg name="use_sim_time" value="$(arg use_sim_time)" />
17   <arg name="headless" value="$(arg headless)" />
18 </include>
19
20 <!-- Load the URDF into the ROS Parameter Server -->
21 <param name="robot_description"
22   command="$(find xacro)/xacro --inorder '$(find rrbot_description)/urdf/rrbot.xacro'" />
23
24 <!-- Run a python script to the send a service call to gazebo_ros to spawn a URDF robot -->
25 <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false" output="screen"
26   args="-urdf -model rrbot -param robot_description"/>
27
28 <!-- ros_control rrbot launch file -->
29 <include file="$(find rrbot_control)/launch/rrbot_control.launch" />
30
31 </launch>
32
```

FIGURE 19 – Contenu du programme rrbot_world.launch

Les premières lignes sont utilisées pour définir des arguments utiles à Gazebo, ils sont réutilisés à la ligne 11 pour lancer la simulation. L'URDF du robot est défini en tant que paramètre afin d'être retrouvable ensuite. On lance ensuite une node interne au package gazebo_ros et on lance le fichier rrbot_control.launch permettant de cette manière l'inclusion de Ros Control.

On ne détaillera pas l'ajout de l'URDF et les inclusions qui en découlent, ce point ayant été déjà abordé dans la partie précédente. Il faut seulement savoir que ce robot possède deux joints importants s'appelant respectivement joint1 et joint2.

6.4.3 rrbot_control / launch / rrbot_control.launch

```
rrbot_control.launch
1 <launch>
2
3 <!-- Load joint controller configurations from YAML file to parameter server -->
4 <roscpp param file="$(find rrbot_control)/config/rrbot_control.yaml" command="load"/>
5
6 <!-- load the controllers -->
7 <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false"
8   output="screen" ns="/rrbot" args="joint_state_controller
9     joint1_position_controller
10    joint2_position_controller"/>
11
12 <!-- convert joint states to TF transforms for rviz, etc -->
13 <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher"
14   respawn="false" output="screen">
15   <remap from="/joint_states" to="/rrbot/joint_states" />
16 </node>
17
18 </launch>
19
```

FIGURE 20 – Contenu du fichier rrbot_control.launch

On voit que ce programme commence par charger le contenu du fichier rrbot_control.yaml. Nous verrons juste après ce dernier. Il contient les joints que l'on souhaite contrôler et quelques paramètres.

Il permet également de définir trois contrôleurs joint_state_controller, joint1_position_controller et joint2_position_controller. De plus, il lance une node publisher qu'il place sous le robot. C'est la raison pour laquelle cet élément était sous le robot RRBot, lors de la commande rqt_graph durant la phase de test du package RRBot.

6.4.4 rrbot_control / config / rrbot_config.yaml

```
rrbot_control.yaml
1  rrbot:
2    # Publish all joint states -----
3    joint_state_controller:
4      type: joint_state_controller/JointStateController
5      publish_rate: 50
6
7    # Position Controllers -----
8    joint1_position_controller:
9      type: effort_controllers/JointPositionController
10     joint: joint1
11     pid: {p: 100.0, i: 0.01, d: 10.0}
12    joint2_position_controller:
13      type: effort_controllers/JointPositionController
14      joint: joint2
15      pid: {p: 100.0, i: 0.01, d: 10.0}
16
```

FIGURE 21 – Contenu de rrbot_control.yaml

Ce fichier place sous l'instance rrbot, que l'on appelle ici un namespace, trois controllers. Le premier est obligatoire et doit toujours exister. Les deux autres correspondent aux liaisons joint1 et joint2. C'est dans ce fichier que l'on fait le lien entre le controller Joint1_position_controller et l'articulation joint1. On définit également les constantes que l'on souhaite appliquer à la commande. Ces valeurs sont modifiables en fonction de la situation que l'on désire.

Vous avez désormais tous les éléments pour créer un robot, le visualiser sous RVIZ, le contrôler avec Ros Control tout en le simulant avec Gazebo.

7 Controller manager, hardware interface et DiffDriveController

Dans cette partie, nous allons voir comment il est possible de contrôler les moteurs d'un robot, en utilisant le controller manager. Nous nous intéresserons ici à un controller manager précis : le DiffDriveController, qui est un controller permettant le contrôle d'un robot différentiel. Un robot différentiel est un robot composé de deux roues dont l'axe de rotation est le même. Le plus souvent, une ou plusieurs roues folles permettent également de supporter le poids du robot. Le turtlebot est un exemple de robot différentiel. Dans notre cas, nous utiliserons le robot Leenby déjà vu pour expliquer le fonctionnement de l'URDF, dont la base se compose de deux roues permettant le déplacement, et de deux roues folles. La base de ce robot doit donc être contrôlée de manière différentielle. Par souci de simplicité, nous n'afficherons pas le reste du robot, on ne s'intéressera qu'à la base.

7.1 Rappel sur le fonctionnement de ROS control

Reprenons le schéma que nous avons déjà vu précédemment :

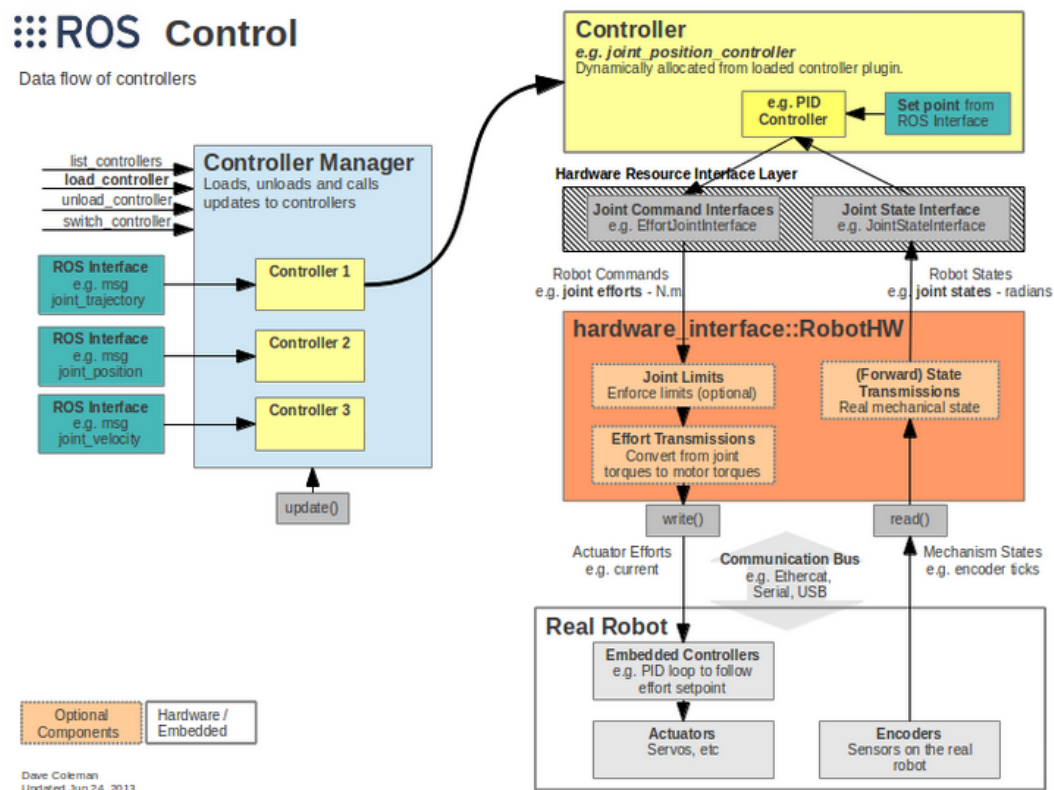


Diagram source in `ros_control/documentation`

FIGURE 22 – Schéma de fonctionnement de ROS control ([lien Ros Wiki](#))

Dans ce cas, nous utiliserons le DiffDriveManager comme controller manager. Ce dernier est normalement inclus dans les packages déjà installés. Il ne s'agit donc pas d'un programme que nous allons devoir rédiger, mais correctement utiliser afin de contrôler notre robot différentiel. Ce controller manager présente un seul controller, qui prendra en entrée un message du type Twist, composé d'une vitesse linéaire et angulaire, qu'il lira directement sur le topic `cmd_vel`. Connaissant l'écartement des roues et leur diamètre, le DiffDriveController sera en mesure de calculer la vitesse de commande de chacune des roues séparément pour appliquer les vitesses linéaire et angulaire que nous voulons. Il appellera directement les fonctions de l'hardware interface. L'hardware interface est un programme que nous écrirons (en C++)

faisant le lien entre le controller et le robot réel. Il proposera une classe comprenant les fonctions nécessaires au contrôle des moteurs, comme les protocoles de communication avec les variateurs pour chacune des roues par exemple.

Pour résumer, il nous faut créer un controller du type `DiffDriveController`, que nous appellerons `mobile_base_controller`. Ce controller lira sur le topic `/mobile_base_controller/cmd_vel` les commandes à effectuer, les traduira en vitesse de chacune des roues, et les enverra à l'hardware interface. Cette dernière se chargera du protocole de communication avec les moteurs pour faire appliquer la commande, et permettra également de récupérer les données des encodeurs pour effectuer la régulation PID.

7.2 Les différents fichiers

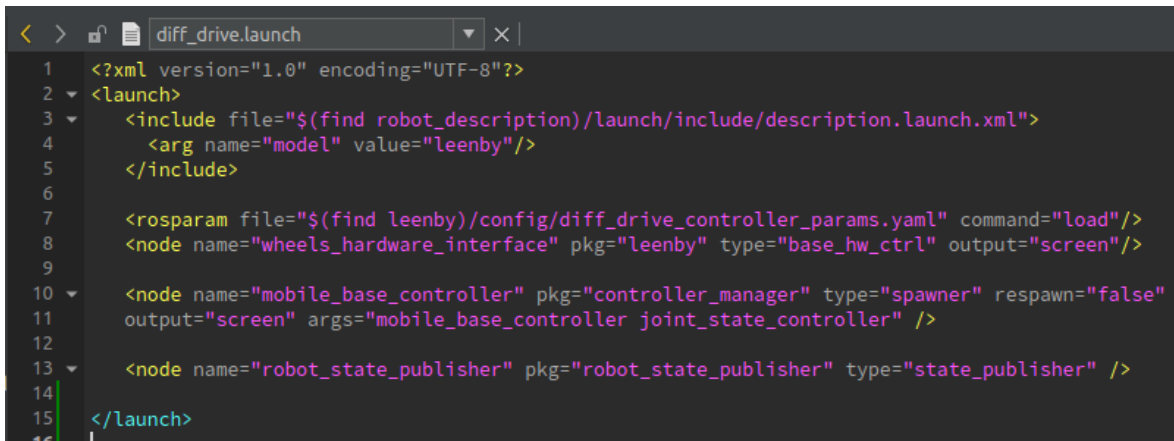
Voyons tout d'abord l'architecture du package :

- leenby
 - config
 - `diff_drive_controller_params.yaml`
 - cybedroid
 - include
 - `BaseMotors.h`
 - launch
 - `diff_drive.launch`
 - src
 - `BaseMotors.cpp`

le package se compose de différents fichiers : `config` contient un fichier `yaml` permettant de définir notre contrôleur `mobile_base_controller`. C'est lui qui sera du type `DiffDriveController`. `Cybedroid` est un paquet obtenu par la société `Cybedroid` qui a fabriqué la `Leenby`. Il contient des librairies pour le contrôle des moteurs brushless des roues de la base via les variateurs. Il y est définie la classe `EPOS2Controller` qui sera utilisée par la suite (`Epos2` est la gamme du variateur).

Dans l'`include`, on retrouve les fichiers `.h` associée à l'hardware interface, ainsi le `.cpp` dans `src`. Le répertoire `launch` contient le fichier qu'on lancera pour démarrer le contrôleur et l'hardware interface. Il ne restera donc plus qu'à publier que `/mobile_base_controller/cmd_vel` afin de contrôler la base du robot.

7.3 diff_drive.launch



```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
  <include file="$(find robot_description)/launch/include/description.launch.xml">
    <arg name="model" value="leenby"/>
  </include>

  <rosparam file="$(find leenby)/config/diff_drive_controller_params.yaml" command="load"/>
  <node name="wheels_hardware_interface" pkg="leenby" type="base_hw_ctrl" output="screen"/>

  <node name="mobile_base_controller" pkg="controller_manager" type="spawner" respawn="false"
    output="screen" args="mobile_base_controller joint_state_controller" />

  <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher" />
</launch>
```

FIGURE 23 – Contenu du fichier `diff_drive.launch`

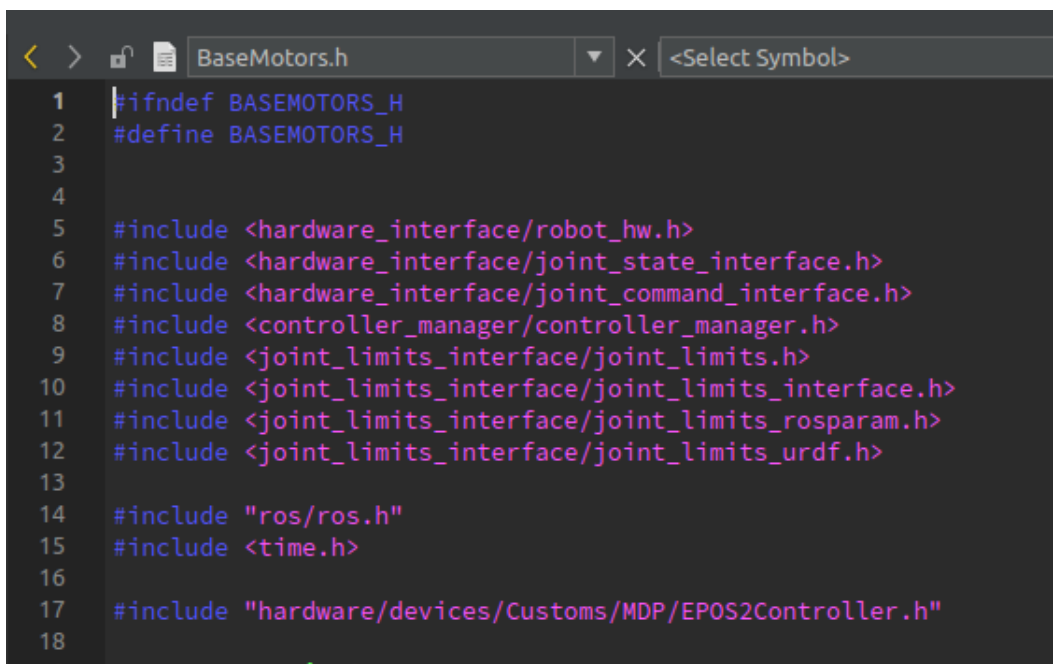
Dans un premier temps, on peut voir que l'on vient chercher le fichier `description.launch.xml`. Ce dernier avait été abordé dans la section sur l'URDF et contient la séparation entre les roues, ainsi que leur diamètre. Le `DiffDriveController` connaîtra ainsi ces valeurs et les utilisera pour établir la commande différentielle.

On vient ensuite charger le contenu du fichier `diff_drive_controller_params.yaml`. Il contient l'ensemble des paramètres pour charger le type de controller manager que l'on souhaite. La ligne suivante, on lance notre hardware interface qui est renommée `base_hw_ctrl` lors de la compilation, mais il est construit à partir de `BaseMotors.h` et de `BaseMotors.cpp`.

Il ne reste plus qu'à faire spawner les deux controllers dont on a besoin, c'est-à-dire le `mobile_base_controller`, défini dans `diff_drive_controller_params.yaml` et le `joint_state_controller`, obligatoire pour le contrôle sous ROS. On lance également le `state_publisher`, ce qui sera obligatoire si on souhaite également visualiser le robot sur le PC, sous RVIZ par exemple. Le PC connaîtra ainsi également l'état du robot.

7.4 BaseMotors.h

Ce fichier contient la description de la classe `BaseMotors`. Cette classe disposera de différentes fonctions qui seront utilisées pour contrôler le robot et en tirer des informations. Elle sera appelée par le controller manager `DiffDriveController`.



```
1 #ifndef BASEMOTORS_H
2 #define BASEMOTORS_H
3
4
5 #include <hardware_interface/robot_hw.h>
6 #include <hardware_interface/joint_state_interface.h>
7 #include <hardware_interface/joint_command_interface.h>
8 #include <controller_manager/controller_manager.h>
9 #include <joint_limits_interface/joint_limits.h>
10 #include <joint_limits_interface/joint_limits_interface.h>
11 #include <joint_limits_interface/joint_limits_rosparm.h>
12 #include <joint_limits_interface/joint_limits_urdf.h>
13
14 #include "ros/ros.h"
15 #include <time.h>
16
17 #include "hardware/devices/Customs/MDP/EPOS2Controller.h"
18
```

FIGURE 24 – Début du contenu du fichier `baseMotors.h`

On commence toujours un fichier `.h` par les deux premières lignes. Cela permet de ne pas lancer par erreur deux fois ce fichier. La première qu'il est lancé, on définit une variable système. La deuxième fois, comme la variable a déjà été définie, on saute tout le contenu du fichier. Le fichier `.h` se termine par la ligne `#endif`.

On voit ici la liste des includes. On ajoute les 8 bibliothèques nécessaires à l'hardware interface et au controller manager, puis la bibliothèque ROS, et la bibliothèque `time` pour une fonction ROS obligatoire. On ajoute ensuite la bibliothèque permettant le contrôle des moteurs via les variateurs Epos2 70/10.

```

19 namespace remix {
20
21
22 #define RES_MOTOR 0.0000164 // rad/counts per rotation -> 2*pi/380928
23
24 using namespace hardware;
25 using namespace std;

```

FIGURE 25 – Suite du contenu du fichier baseMotors.h

On indique dans un premier temps qu'on utilisera ici les namespace remix, hardware et std et on définit la résolution des encodeurs.

```

27 class BaseMotors : public hardware_interface::RobotHW
28 {
29     hardware_interface::JointStateInterface jnt_state_interface_;
30     hardware_interface::VelocityJointInterface jnt_vel_interface_;
31
32     EPOS2Controller *eposR_, *eposL_;
33
34 public:
35     enum joints
36     {
37         LEFT_MOTOR=0,
38         RIGHT_MOTOR=1
39     };
40
41     BaseMotors();
42     virtual ~BaseMotors();
43     void initialize();
44     void read();
45     void write();
46
47     double cmd_[2];
48     double pos_[2];
49     double vel_[2];
50     double eff_[2];
51 }; //class BaseMotors
52 }; //namespace remix
53
54
55 #endif // BASEMOTORS_H

```

FIGURE 26 – Fin du contenu du fichier baseMotors.h

Dans cette espace, on définit la classe. Elle s'appelle Basemotors, et possède l'ensemble des méthodes publiques associées à la classe hardware_interface : :RobotHW.

On indique que l'on souhaite posséder un joint_state_interface (obligatoire pour ros control) et que l'on désire une commande en vitesse. On crée également deux pointeurs vers des variables du type EPOS2Controller, défini dans le répertoire Cybedroid.

Dans la partie public :, on a d'abord défini des valeurs avec enum joints. Cette méthode est optionnelle mais est très utile si on a plusieurs moteurs. On gère ainsi directement le nom, et non la valeur.

On définit ensuite toutes les méthodes qui seront utilisées, et qui seront décrites dans BaseMotors.cpp. Les deux premières méthodes sont le créateur et le destructeur de classe. On a ensuite l'initialisation, qui est une méthode optionnelle en général, mais obligatoire dans notre cas. Les méthodes read() et write() sont obligatoires car ce sont elles qui seront utilisées par le controller manager. On définit ensuite les variables qui seront utilisées pour caractériser l'état et le contrôle des deux moteurs. On a fait des arrays, comme ça on peut accéder aux caractéristiques du premier moteur avec par cmd_[0] ou cmd_[LEFT_MOTOR].

7.5 BaseMotors.cpp

Nous allons voir le programme BaseMotors.cpp par morceaux successifs. Ce programme est bien entendu à adapter en fonction du cas présent.

7.5.1 Les includes

```
1 #include "BaseMotors.h"
2 #include "controller_manager/controller_manager.h"
3 #include "hardware/devices/Customs/MDP/EPOS2Controller.h"
4 #include <ros/callback_queue.h>
5 #include "ros/ros.h"
6
7 using namespace remix;
8
```

FIGURE 27 – Les includes du fichier baseMotors.cpp

7.5.2 Le créateur

```
BaseMotors::BaseMotors()
{
    cmd_[LEFT_MOTOR] = 0.0;
    cmd_[RIGHT_MOTOR] = 0.0;
    pos_[LEFT_MOTOR] = 0.0;
    pos_[RIGHT_MOTOR] = 0.0;
    vel_[LEFT_MOTOR] = 0.0;
    vel_[RIGHT_MOTOR] = 0.0;
    eff_[LEFT_MOTOR] = 0.0;
    eff_[RIGHT_MOTOR] = 0.0;

    // Put 0 for left wheel and 1 for right wheel : Very important !
    eposL_ = new EPOS2Controller(1, "epos_left");
    eposR_ = new EPOS2Controller(2, "epos_right");

    // connect and register the joint state interface
    hardware_interface::JointStateHandle state_handle_eposL("left_wheel_joint", &pos_[LEFT_MOTOR], &vel_[LEFT_MOTOR], &eff_[LEFT_MOTOR]);
    jnt_state_interface_.registerHandle(state_handle_eposL);
    hardware_interface::JointStateHandle state_handle_eposR("right_wheel_joint", &pos_[RIGHT_MOTOR], &vel_[RIGHT_MOTOR], &eff_[RIGHT_MOTOR]);
    jnt_state_interface_.registerHandle(state_handle_eposR);

    registerInterface(&jnt_state_interface_);

    // connect and register the joint position interface
    hardware_interface::JointHandle pos_handle_eposL(jnt_state_interface_.getHandle("left_wheel_joint"), &cmd_[LEFT_MOTOR]);
    jnt_vel_interface_.registerHandle(pos_handle_eposL);
    hardware_interface::JointHandle pos_handle_eposR(jnt_state_interface_.getHandle("right_wheel_joint"), &cmd_[RIGHT_MOTOR]);
    jnt_vel_interface_.registerHandle(pos_handle_eposR);

    registerInterface(&jnt_vel_interface_);
};
```

FIGURE 28 – Le créateur de la classe BaseMotors

On commence par initialiser toutes les variables à 0. On vient également créer deux pointeurs vers les variateurs.

On connecte et enregistre chaque joint state interface, en renseignant le nom de la liaison (ils doivent correspondre aux noms des joints de l'URDF), ainsi que chaque élément correspondant aux éléments du JointStateInterface, soit la position, la vitesse et l'effort. C'est de cette manière que le controller manager connaît les variables à utiliser pour établir la commande. Il ne reste plus qu'à enregistrer cette interface afin de pouvoir l'utiliser par la suite.

Ensuite, il faut dire de quelle manière le controller manager va agir sur notre système, c'est-à-dire qu'elle va être la sortie du controller manager qui arrivera sur l'hardware interface afin de déplacer le moteur. Les entrées et les sorties peuvent varier en fonction du type de controller manager que l'on souhaite utiliser. Comme dans le cas précédent, on vient enregistrer cette interface afin de pouvoir l'utiliser par la suite. On a de cette manière là créé les deux interfaces permettant au DiffDriveController de "contacter" notre système.

7.5.3 Le destructeur de la classe BaseMotors

```
BaseMotors::~BaseMotors(){  
    eposL_>SetVelocityTargetValue(0.0);  
    eposR_>SetVelocityTargetValue(0.0);  
  
    eposL_>Shutdown();  
    eposR_>Shutdown();  
  
    eposL_>Close();  
    eposR_>Close();  
};
```

FIGURE 29 – Le destructeur de la classe BaseMotors

Cette méthode permet d'arrêter correctement la classe en cas d'interruption (souvent un Ctrl+C). On s'assure alors de stopper le déplacement.

7.5.4 La fonction initialize()

```
void  
BaseMotors::initialize()  
{  
    eposL_>Setup("/dev/serial/by-id/usb-FTDI_USB-COM232_Plus2_FT0FK865-if00-port0", 115200);  
    eposR_>Setup("/dev/serial/by-id/usb-FTDI_USB-COM232_Plus2_FT0FK865-if01-port0", 115200);  
  
    if( ( eposL_>Open() == 1 ) && ( eposR_>Open() == 1 ) ){  
        ROS_INFO("EPOS motor controllers ready and set in velocity control mode.");  
        eposL_>EnableVelocityMode();  
        eposR_>EnableVelocityMode();  
    }else{  
        throw runtime_error("Failed to set up EPOS Controllers");  
    }  
}
```

FIGURE 30 – La fonction initialize()

Cette fonction permet d'initialiser la connexion avec les deux variateurs. Elle est à adapter en fonction de ce dont vous avez besoin, si vous en avez besoin.

7.5.5 La fonction read()

```
void
BaseMotors::read()
{
    pos_[0] = RES_MOTOR * eposL_ -> GetIncrementalEncoderPosition();
    vel_[0] = eposL_ -> GetVelocityAveragedValue(); //// Unités???

    pos_[1] = -RES_MOTOR * eposR_ -> GetIncrementalEncoderPosition();
    vel_[1] = eposR_ -> GetVelocityAveragedValue();
}
```

FIGURE 31 – La fonction read()

Cette fonction sera appelée périodiquement et permettra de renseigner dans les variables définies l'état de notre système. Ici, on vient mettre à jour la position et la vitesse de chaque roue. Si on avait accès aux efforts de chacune des roues, on pourrait également les renseigner. Comme on avait renseigné en créant les interfaces que la position incluse dans le JointStateInterface était pos_[0], le controller manager pourra directement modifier l'état connu du système en lisant ces valeurs.

7.5.6 La fonction write()

```
void
BaseMotors::write()
{
    eposL_ -> SetVelocityTargetValue(cmd_[LEFT_MOTOR]);
    eposR_ -> SetVelocityTargetValue(-cmd_[RIGHT_MOTOR]);
}
```

FIGURE 32 – La fonction read()

Cette fonction sera également appelée périodiquement et permettra de contrôler les moteurs avec les valeurs que le DiffDriveController (mobile_base_controller) aura écrites dans cmd_[].

7.5.7 La fonction main()

```
int main(int argc, char **argv)
{
    ros::init(argc, argv, "base_hardware_interface");
    ros::NodeHandle nh;
    ros::CallbackQueue queue;
    nh.setCallbackQueue(&queue);

    BaseMotors base_motors= BaseMotors();
    controller_manager::ControllerManager cm(&base_motors, nh);

    ros::Rate rate(50.0);
    ros::Duration elapsed_time;
    struct timespec current_time, last_time;
    clock_gettime(CLOCK_MONOTONIC, &last_time);
    static const double BILLION = 1000000000.0;

    ros::AsyncSpinner spinner(4, &queue);
    spinner.start();

    try {
        base_motors.initialize();
        while(ros::ok()){
            // Get change in time
            clock_gettime(CLOCK_MONOTONIC, &current_time);
            elapsed_time = ros::Duration(current_time.tv_sec -
                                         last_time.tv_sec +
                                         (current_time.tv_nsec - last_time.tv_nsec) / BILLION);

            last_time = current_time;

            ros::Time now = ros::Time::now();

            base_motors.read();
            cm.update(now, elapsed_time);
            base_motors.write();
            rate.sleep();
        }
    } catch (runtime_error & e) {
        ROS_ERROR(e.what());
    }

    spinner.stop();

    return 0;
}
```

FIGURE 33 – La fonction main()

Les 4 premières lignes du main permettent de lancer ROS. On vient ensuite utiliser le créateur pour créer notre variable `base_motors`. On crée également notre controller manager que l'on nomme `cm`.

On définit 2 éléments qui serviront à ROS, et d'autres éléments utiles dans notre cas.

On initialise le `AsyncSpinner` `spinner` avec 4 en paramètre. Cela permettra d'autoriser la node que l'on crée à recevoir des infos d'un publisher en passant par un topic, sans pour autant mettre en pause notre programme comme la fonction `ros::spin()` ou `ros::spinOnce()` l'auraient fait.

La boucle `try` permet de ne rien commander en cas d'erreur. On vient tout d'abord initialiser `base_motors` avec la fonction que nous avons défini. On rentre alors dans la boucle principale de commande, qui tournera tant que `ros` fonctionne correctement.

On vient tout d'abord calculer le temps qui s'est écoulé entre la dernière fois que la boucle a été parcourue et cette fois-ci. Cela permettra au controller manager `cm` de pouvoir connaître le temps nécessaire à l'exécution d'un PID, au calcul de l'accélération, etc.

Il ne reste plus qu'à indiquer à la classe de lire le contenu avec la fonction que nous avons écrite, faire un update du controller manager qui lira les valeurs obtenues par le read(), et remplira cmd_[] afin les mouvements nécessaires. On fait ensuite la fonction write() qui applique les mouvements à chacune des roues.

Rate.sleep() permet d'attendre une période de la fréquence renseignée lors de la déclaration de rate, soit 50Hz.

Cette boucle tourne ensuite à l'infini jusqu'à ce que ROS soit fermée et que le spinner soit stoppé avant de sortir de la fonction main().

7.6 diff_drive_controller_params.yaml

Ce fichier contient la configuration du contrôleur que nous sommes en train de créer.

```
1 mobile_base_controller:
2   type      : "diff_drive_controller/DiffDriveController"
3   left_wheel : 'left_wheel_joint'
4   right_wheel : 'right_wheel_joint'
5   publish_rate: 50.0           # default: 50
6   pose_covariance_diagonal : [0.001, 0.001, 1000000.0, 1000000.0, 1000000.0, 1000.0]
7   twist_covariance_diagonal: [0.001, 0.001, 1000000.0, 1000000.0, 1000000.0, 1000.0]
8
9   # Wheel separation and diameter. These are both optional.
10  # diff_drive_controller will attempt to read either one or both from the
11  # URDF if not specified as a parameter
12  # wheel_separation : 0.402
13  # wheel_radius : 0.072
14
15  # Wheel separation and radius multipliers
16  wheel_separation_multiplier: 1.0 # default: 1.0
17  wheel_radius_multiplier   : 1.0 # default: 1.0
18
19  # Velocity commands timeout [s], default 0.5
20  cmd_vel_timeout: 0.5
21
22  # allow multiple cmd vel publishers
23  allow_multiple_cmd_vel_publishers: False
```

FIGURE 34 – Début du fichier diff_drive_controller_params.yaml

On définit tout d'abord le namespace mobile_base_controller, qui correspond au nom de notre contrôleur. On choisit ensuite le type que l'on souhaite. Ici, il s'agit d'un DiffDriveController. Certains paramètres détaillés ensuite sont optionnels et dépendent du cas d'utilisation, il faut donc se renseigner de la manière d'implémenter un contrôleur en fonction de son type, et de l'utilisation qu'on souhaite en faire.

Pour information, la suite de la définition de mobile_base_controller est :

```
25 #Publish_cmd
26 publish_cmd: True
27
28 # Base frame_id
29 base_frame_id: base_footprint #default: base_link
30 enable_odom_tf: true
31 odom_frame_id: odom
32
33 # Velocity and acceleration limits
34 # Whenever a min_* is unspecified, default to -max_*
35 linear:
36   x:
37     has_velocity_limits    : true
38     max_velocity           : 1.0 # m/s
39     min_velocity           : -0.5 # m/s
40     has_acceleration_limits: true
41     max_acceleration       : 0.25 # m/s^2
42     min_acceleration       : -0.2 # m/s^2
43     has_jerk_limits        : false
44     max_jerk               : 5.0 # m/s^3
45   angular:
46     z:
47       has_velocity_limits    : true
48       max_velocity           : 1.7 # rad/s
49       has_acceleration_limits: true
50       max_acceleration       : 1.5 # rad/s^2
51       has_jerk_limits        : false
52       max_jerk               : 2.5 # rad/s^3
```

FIGURE 35 – Suite du fichier diff_drive_controller_params.yaml

Pour terminer, il faut également renseigner les paramètres du JointStateController.

```
54 joint_state_controller:
55   type: joint_state_controller/JointStateController
56   publish_rate: 50
```

FIGURE 36 – Fin du fichier diff_drive_controller_params.yaml

7.7 CMakeLists

```
1 cmake_minimum_required(VERSION 2.8.3)
2 project(leenby)
3
4 add_compile_options(-std=c++11 -lpthread)
5
6 find_package(catkin REQUIRED COMPONENTS
7   roscpp
8   std_msgs
9   sensor_msgs
10  controller_manager
11  tf)
12
13 find_package(Boost REQUIRED COMPONENTS system)
14
15 catkin_package(
16   INCLUDE_DIRS
17     include
18
19   CATKIN_DEPENDS
20     roscpp
21     std_msgs
22     sensor_msgs
23     controller_manager
24     tf
25 )
26
27 include_directories(
28   include
29   cyberdroid/include
30   ${catkin_INCLUDE_DIRS}
31 )
32
33 ## Declare Cyberdroid Framework as a C++ library
34 file(GLOB_RECURSE CYBERDROID_SOURCES RELATIVE ${CMAKE_CURRENT_SOURCE_DIR} "cyberdroid/*.cpp")
35 add_library(cyberdroid_framework ${CYBERDROID_SOURCES})
36
37 add_executable(base_hw_ctrl src/BaseMotors.cpp)
38 #add_executable(base_hw_ctrl src/Leenby.cpp)
39 target_link_libraries(base_hw_ctrl
40   ${catkin_LIBRARIES}
41   cyberdroid_framework
42 )
```

FIGURE 37 – Contenu du fichier CMakeLists.txt

C'est dans ce fichier qu'on peut voir comment l'exécutable a été renommé `base_hw_ctrl`. Il faut bien voir également que les librairies et les dépendances ont été ajoutées également dans ce fichier.

8 ROS Network

Cette section explique comment faire pour mettre en place un ROS Network, c'est-à-dire de faire en sorte que plusieurs PC se trouvent sous le même ROS Master, en passant par un port SSH. Il suffit donc de connecter les deux PC sur le même réseau Wifi privé, et ils peuvent être reliés en connaissant les adresses IP.

Il est également possible d'utiliser un câble Ethernet. Il vous faudra pour cela activer le DHCP automatique dans les paramètres de la connexion.

Afin de tester une connexion entre deux ordinateurs, il est possible de faire un test de ping avec la commande : `ping ip_de_l'autre_réseau`. Si aucune information de temps de réponse n'apparaît, c'est que la connexion n'est pas possible avec les paramètres actuels.

8.1 Sur le PC principal - RosMaster

Il faut dans un premier temps connecter le PC principal qui hébergera le RosMaster au réseau Wifi. En tapant "ifconfig" dans la console, on accède à l'adresse IP du PC sur le réseau. On la voit dans la case wl., entre inet et netmask. Après avoir noté l'adresse IP de ce PC, on tape dans la console "nano /.bashrc". On descend jusqu'à trouver la case export ROS_MASTER_URI. On renseigne l'adresse IP du Master (celle lue précédemment) :

- `export ROS_MASTER_URI=http://192.168.x.xxx:11311` (IP du master)
- `export ROS_HOSTNAME=192.168.x.xxx` (IP du master)

Il ne reste plus qu'à lancer le roscore, et les fonctions que l'on souhaite sur Ros. Il faut penser à sourcer bashrc avec la commande : `source /.bashrc`.

8.2 Sur le PC secondaire

On vient en tout premier lieu également se connecter au réseau Wifi, et on trouve l'adresse IP de la même manière. On ouvre bashrc, et on renseigne cette fois-ci l'adresse IP du Master (celle de la section précédente) dans ROS_MASTER_URI et l'adresse IP du PC secondaire dans ROS_HOSTNAME.

- `export ROS_MASTER_URI=http://192.168.x.xxx:11311` (IP du master)
- `export ROS_HOSTNAME=192.168.x.xxx` (IP du PC secondaire)

On vient également sourcer bashrc sur le PC secondaire avec `source /.bashrc`. Il faut alors ouvrir le port SSH avec la commande : "SSH IP_du_Master". On a dès lors accès au ROS du PC principale depuis le PC secondaire. Il faudra toutefois avoir l'ensemble des packages que l'on souhaite utiliser sur ce PC.