ROS Notes

Release 0.9.0

2019-06-07 13:14:43

Getting started

1 ROS Tutorials and wiki 3

2 Preparing 5

3 ROS Overview 9

4 Publisher-Subscriber 11

5 Launch files 13

6 Simulators 15

7 Unified Robot Description Format URDF 17

8 Xacro 29

9 Mobile robot 35

10 Gazebo 41

11 Navigation 45

12 Robot arm model 49

13 Industrial robots 51

14 Service and Client 53

15 Parameters server 55

16 ActionServer and ActionClient 57

17 Plugin 59

18 Nodelet 61

19 Rqt plugins 63

20 Rviz plugins 65

21 Gazebo plugins 67

22 Arduino 69

23 TIVA C 71

24 STM32 73

25 Raspberry pi 75

26 Cameras 77

27 OpenCV 79

28 Point Cloud Library (PCL) 81

29 Visual Servoing Platform library ViSP 85

30 The CImg Library 87

Warning: Work in progress

2019-06-07 13:14:43

CHAPTER 1

ROS Tutorials and wiki

Ros-Tutorials, Roscpp, Rospy, Roscpp_overview:

```
git clone https://github.com/ros/ros_tutorials.git
```

Roslaunch and tutorials

```
https://github.com/ros/ros_comm.git
```

Rqt tools and rqt_tutorials:

```
https://github.com/ros-visualization/rqt.git
```

URDF_

```
git clone https://github.com/ros/urdf.git
```

TF

```
git clone  https://github.com/ros/geometry_tutorials.git
```

Gazebo

```
git clone
```

Rviz and Rviz_Plugins:

MoveIt

```
git clone https://github.com/ros-planning/moveit_tutorials.git
```

ROS-I

```
git clone https://github.com/ros-industrial/industrial_training.git
```

(continues on next page)

(continued from previous page)

```
http://wiki.ros.org/Industrial/Tutorials

https://github.com/ros-industrial-consortium/godel

https://github.com/ros-industrial-consortium/packml

https://github.com/Jmeyer1292/robot_cal_tools

https://github.com/ros-industrial-consortium/ipa_seminar
```

1.1 ROS books:

CHAPTER 2

Preparing

2.1 Installation

Refer to the official website for more information. The instructions below are valid for almost all releases. In this tutorial `kinetic` and `ubuntu` 16.04 are used.

Open a terminal:

```
sudo apt-get install python-rosinstall python-rosinstall-generator python-wstool␣
↪build-essential

sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /
↪etc/apt/sources.list.d/ros-latest.list'

// key for kinectic
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key␣
↪421C365BD9FF1F717815A3895523BAEEB01FA116
sudo apt-get update
sudo apt-get install ros-kinetic-desktop-full
sudo rosdep init
rosdep update
```

After installation you need to tell ubunut where ROS is installed

```
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Or you open the hidden file `.bashrc` with a text editor and add

```
source /opt/ros/kinetic/setup.bash
```

To check the installation in a terminal type:

```
roscore
```

2.2 Catkin workspace

Like Arduino, Eclispe and other IDE, to develop in ROS you need to create a workspace. A workspace mainly contain the source code folder and the compiled code and eventually other folders and files.

In order to create a workspace in ROS, `catkin` tools are used. These tools automate some job in `Cmake`. There are 3 ways to use catkin. The first one is the most used. The second one `catkin_tools` is under development. The third one `catkin_simple` can be used in order to simplify the `CMakeLists.txt` file.

Before using the catkin commands, you need to create a workspace directory and a `src` subdirectory. You can call the workspace as you want. The directories can be created using the `terminal` or using the desktop utilities. Usually we will use the terminal.

If you want to use an IDE, `RoboWare Studio` and `Qt` plugin `ros_qtc_plugin` are good choices.

2.2.1 catkin

We will create a workspace called `catkin_ws` and a subdirectory `src` in the home directory of linux. The following code will create a folder called `src`. The `-p` option will create the folder `catkin_ws` if it doesn't exist

```
mkdir -p ~/catkin_ws/src
```

Navigate to the `src` folder and initialize the workspace. The command `catkin_init_workspace` should be executed within the folder `src`

```
cd ~/catkin_ws/src
catkin_init_workspace
```

Go back to the workspace directory and type `` `catkin_make` `` in order to build the source code

```
cd ~/catkin_ws/
catkin_make
```

In order to build the worksapce after any modification of the source code, the `catkin_make` command should be run from the workspace directory.

2.2.2 catkin tools

This is a python package that can be used to create and build a ROS workspace. It is not installed by default. Run the following commnad to install it

```
sudo apt-get install python-catkin-tools
```

The documentation of this package can be found:

https://github.com/catkin/catkin_tools

http://catkin-tools.readthedocs.org/

Once the workspace and src folders are created, to initialize the workspace run the command `catkin init` from the workspace directory not from the `src`:

```
cd ~/catkin_ws/
catkin init
```

To build the workspace run

```
catkin build
```

The command `catkin build` will initialize the workspace if it was not initialized by the `catkin init` command. The `catkin build` command can be run from any sub-directory of the workspace, contrary to `catkin_init_workspace`.

2.2.3 catkin simple

2.2.4 Workspace sub-directory

Once is build the workspace will contain the `src`, `build` and `devel` folders. These folders are create by all the previous methods.

2.2.5 Environment variable

When you create the worksapce the first time, after the compilation run the command:

```
// this will be valid only for the opened terminal
      source devel/setup.bash
```

To make it permanent do the following:

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

Make sure `ROS_PACKAGE_PATH` environment variable includes the worksapce in use:

```
echo $ROS_PACKAGE_PATH
```

2.2.6 Illustration

Remember you can put the workspace in any directory and give any name.

??????????????? make a gif

2.3 Creating a ROS Package

Simply a package is a collection of programs.

2.3.1 Catkin

When creating a package, a name should be given and all dependencies:

```
catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

For example we create `beginner_tutorials` package that depend on `std_msgs`, `rospy`, `roscpp`

```
cd ~/catkin_ws/src

// create a package
catkin_create_pkg beginner_tutorials std_msgs rospy roscpp

// build all packages in the workspace
cd ~/catkin_ws
catkin_make
```

??????????????? make a gif

2.3.2 catkin_tools

2.3.3 catkin_simple

CHAPTER 3

ROS Overview

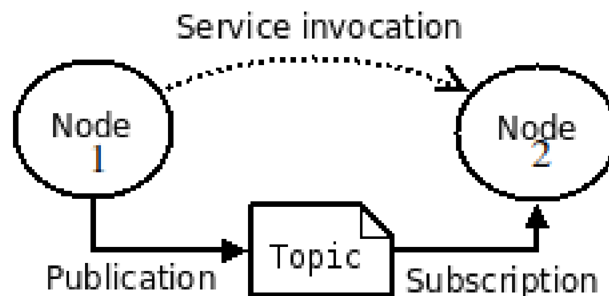
3.1 Overview

Nodes are programs, a package is a collection of programs. ROS is based on nodes, that are processes that communicate with each others, send messages via topics. A topic is the bus where a message is sent.

The master, that can be run by `` `roscore` ``, provides naming and registration services to the rest of nodes in the ROS system. When a node publish an information, using a message structure, it publish it on a topic. A node must subscribe to a topic in order to read that information. The master track publishers and subscribers.

Messages are data types or data structures, one can use ROS messages, `std_msgs`, or create new one. User defined messages are stored in files with .msg extensions in the msg folder in the src folder of the workspace.

Topics are only one way buses, it mean if a node publish a topic, it doesn't wait an answer. If a node need to receive a reply from another node, services should be used.



As you can see node1 publish a message on a topic. Node2 receive the message on that topic. It is similar to the concepts of newsletters. If you are subscribed to it, you receive emails otherwise no. And evry topic should have a name.

3.2 Turtle sim

In order to illustrate these concepts we will use the preinstalled package `turtlesim`. Open 4 terminal or use a terminal multiplexer and run:

```
// Terminal 1
roscore

// terminal 2
rosrun turtlesim turtlesim_node

// teminal 3
rosrun turtlesim turtle_teleop_key

// terminal 4
rqt_graph
```

In the graph we can see that the node (process) `teleop_turtle` is publishing on a topic called `cmd_vel`. The node `turtlesim` is receiving from the topic `cmd_vel`.

In another terminal run:

```
rostopic list

rostopic info /turtle1/cmd_vel
```

`rostopic list` show all list of topics that are active. `rostopic info` show the type of message that is published on the topic. We can see that on `cmd_vel` is being published a message of type `geometry_msgs/Twist`.

Run the follwing command:

```
rosmsg show geometry_msgs/Twist
```

you will get the data structure of the `Twist` message. `A Twist` is message that contain 2 variables `linear` and `angular` of type `geometry_msgs/Vector3`. A `geometry_msgs/Vector3` type contain 3 variables `x`, `y`, `z` of type `float64`.

Simply a `geometry_msgs/Twist` is a data structure that can be used to write linear and angular speed. So on topic `cmd_vel` the `teleop_turtle` node is sending the desired speed to `turtlesim` node.

The speed can be sent to the turtle from any other node that publish on the `/turtle1/cmd_vel` topic. For example if you open `rqt`

```
rqt
```

Then from plugin, Robot tools, Robot steering, we can control the speed of the turtle.

CHAPTER 4

Publisher-Subscriber

In ROS processes that are called nodes communicate with each others using topics. The are other ways to communicate that we will see in other chapters. Nodes sends messages over topics. The node that send a messages is called publisher. The one that is receiving is called subscriber.

In the workspace create a package called

```
catkin_create_pkg first_steps roscpp rospy std_msgs
cd first_steps
mkdir scripts
```

4.1 Simple Publisher-Subscriber ROS program

In this section we will make a package that contain 2 nodes. A publisher node (talker.cpp) and a subscriber node (listener.cpp). These two source files should be created in the src folder of the package.

Append to the CMakeLists.txt of the package the following:

```
add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})

add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
```

This will create two executables, talker and listener, which by default will go into package directory of your devel space, located by default at ~/catkin_ws/devel/lib/<package name>.

4.1.1 Publisher node

4.1.2 Subscriber Node

4.1.3 Building nodes

Packages should be stored in workspace, if no work space is present, one should be created. The following steps should be done:

- Create and build a workspace
- Source the package environment variable
- Create a package
- Copy or create node source file into the src folder of the package
- Eventually create new messages and services
- Add nodes, messages and services to the CMakeLists.txt of the package
- Build the workspace

Listing.ref{lstquickROS} show steps necessary to create in the home directory a workspace named catkin_ws, create a package called first_tutorial, and create two nodes in that package then build the workspace.

`label=lstquickROS listing/tutorial/quickROS`

This will create two executables, talker and listener, which by default will go into package directory of your devel space, located by default at ~/catkin_ws/devel/lib/<package name>.

4.1.4 Run nodes

```
roscore

rosrun beginner_tutorials talker      (C++)
rosrun beginner_tutorials talker.py   (Python)
```

```
rosrun beginner_tutorials listener     (C++)
rosrun beginner_tutorials listener.py  (Python)
```

4.1.5 Useful ROS commands

rosnode rostopic rosmsg

CHAPTER 5

Launch files

5.1 Basic launch file

Ros launch `talker_listener.launch` file that execute 2 nodes:

Listing 1: talker_listener.launch

```
<launch>
  <node name="listener" pkg="roscpp_tutorials" type="listener_node" output="screen"/>
  <node name="talker" pkg="roscpp_tutorials" type="talker_node" output="screen"/>
</launch>
```

The previous launch file execute one instance of the `listener_node` with a the name `listener`, and one instance of the executable `talker_node` with the name `talker`, that are part of the package `roscpp_tutorials`. So the `type` id the name of the compiled node. The `name` is the name of the instance of the node (process). The tag `output` have value `screen`, it tell ROS to show the outputs of the nodes in the terminal. If `roscore` is not running, `roslaunch` execute it.

Open a terminal and run the roslaunch command:

```
roslaunch roscpp_tutorials talker_listener.launch
```

Of course the package `roscpp_tutorials` should be in your active workspace or installed on your computer. Or you substitute the package name and node names with your own.

Clone the repository in your workspace

```
git clone https://github.com/ros/ros_tutorials.git
```

Or install it

```
sudo apt-get install ros-kinetic-roscpp-tutorials
```

5.2 Parameters

You can also set parameters on the Parameter Server. These parameters will be stored on the Parameter Server before any nodes are launched.

```
<launch>
  <param name="somestring1" value="bar" />
  <!-- force to string instead of integer -->
  <param name="somestring2" value="10" type="str" />

  <param name="someinteger1" value="1" type="int" />
  <param name="someinteger2" value="2" />

  <param name="somefloat1" value="3.14159" type="double" />
  <param name="somefloat2" value="3.0" />

  <!-- you can set parameters in child namespaces -->
  <param name="wg/childparam" value="a child namespace parameter" />

  <!-- upload the contents of a file to the server -->
  <param name="configfile" textfile="$(find roslaunch)/example.xml" />
  <!-- upload the contents of a file as base64 binary to the server -->
  <param name="binaryfile" binfile="$(find roslaunch)/example.xml" />
</launch>
```

5.3 Substitution args

Listing 2: launch_file.launch

```
<arg name="gui" default="true" />
<param name="foo" value="$(arg my_foo)" />

<node name="add_two_ints_server" pkg="beginner_tutorials" type="add_two_ints_server"␣
↪/>
<node name="add_two_ints_client" pkg="beginner_tutorials" type="add_two_ints_client"␣
↪args="$(arg a) $(arg b)" />
```

Execution example:

```
roslaunch beginner_tutorials launch_file.launch a:=1 b:=5
```

5.4 Namespaces

5.5 Including files

CHAPTER 6

Simulators

6.1 Rviz

To run rviz

```
rosrun rviz rviz
```

6.2 Gazebo

Gazebo is a Robot simulator with physics engine. Fro more information about it check the website.

Gazebo can be interfaced with ROS. The follwing package should be installed:

```
sudo apt-get install ros-kinetic-gazebo-ros-pkgs ros-kinetic-gazebo-msgs ros-kinetic-
↪gazebo-plugins ros-kinetic-gazebo-ros-control
```

To run Gazebo:

```
roscore
rosrun gazebo_ros gazebo
```

6.2.1 Gazebo launch

Try `gazebo_ros` package:

```
roslaunch gazebo_ros empty_world.launch
roslaunch gazebo_ros empty_world.launch paused:=true use_sim_time:=false gui:=true␣
↪throttled:=false recording:=false debug:=true verbose:=true
roslaunch gazebo_ros willowgarage_world.launch
roslaunch gazebo_ros mud_world.launch
```

(continues on next page)

(continued from previous page)

```
roslaunch gazebo_ros shapes_world.launch
roslaunch gazebo_ros rubble_world.launch
```

CHAPTER 7

Unified Robot Description Format URDF

Note: Refer to URDF-Tutorials and clone the URDF-Tutorials-repo if you follow the official tutorial

```
git clone https://github.com/ros/urdf_tutorial.git
```

In this chapter we will create a model of Epson SCARA Robot.

In the workspace in the `src` directory create a folder called Robots, where we will put all robot description packages.

```
// navigate to src
mkdir Robots

catkin_create_pkg epson_g3_description geometry_msgs urdf rviz xacro

cd epson_g3_description

mkdir urdf scripts rviz
```

Back to the workspace and compile the packages.

7.1 Launch file

The following launch file is taken from URDF-Tutorials. It have different parameters that allow it to execute different robot models.

Listing 1: display.launch from urdf_tutorial

```
<launch>
  <arg name="model" default="$(find urdf_tutorial)/urdf/01-myfirst.urdf"/>
  <arg name="gui" default="true" />
  <arg name="rvizconfig" default="$(find urdf_tutorial)/rviz/urdf.rviz" />
```

(continues on next page)

(continued from previous page)

```
  <param name="robot_description" command="$(find xacro)/xacro --inorder $(arg model)
↪" />
  <param name="use_gui" value="$(arg gui)"/>

  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_
↪publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher
↪" />
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)" required="true
↪" />
</launch>
```

It execute 3 nodes: rviz, joint_state_publisher, state_publisher. In the launch file are present 2 parameters `robot_description` and `use_gui` that are need by the two nodes‘‘joint_state_publisher‘‘ and `state_publisher`. There are also 3 arguments with default values, one of them is full name of the urdf file.

The node `joint_state_publisher` read the urdf file from the parameter `robot_description` finds all of the non-fixed joints and publishes a `sensor_msgs/JointState` message with all those joints defined on the topic `/joint_states`. We set by default the parameter `use_gui` to true, so when the node is executed it will show a window/widget with sliders that let us control the robot joints.

The node `state_publisher` uses the URDF specified by the parameter `robot_description` and the joint positions from the topic `/joint_states` to calculate the forward kinematics of the robot and publish the results via `tf`.

Launch the `display.launch` file

```
roslaunch urdf_tutorial display.launch model:=urdf/01-myfirst.urdf
```

Or independently from the working direcory

```
roslaunch urdf_tutorial display.launch model:='$(find urdf_tutorial)/urdf/01-myfirst.
↪urdf'
```

In this case the file name is the same of the default value. So in this case it can be omitted

```
roslaunch urdf_tutorial display.launch
```

Rviz files can be deleted from the launch file if you don’t have them. They can be created from rviz later. If there is no rviz file, Rviz will not show the frames neither the robot neither select the right `Fixed Frame`.

We will modify launch file from the tutorial in the followingway:

Listing 2: display.launch

```
<launch>

  <arg name="model" default="$(find epson_g3_description)/urdf/scara.urdf"/>
  <arg name="gui" default="true" />

  <param name="robot_description" textfile="$(find epson_g3_description)/urdf/scara.
↪urdf" />
  <param name="use_gui" value="$(arg gui)"/>

  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_
↪publisher" />
```

(continues on next page)

(continued from previous page)

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

</launch>
```

Download display.launch

Basically I change the package name, the urdf default name and delete the reference for rviz configuration file. We will back later to rviz configuration files.

I create also a scripts folder where I will create some bash file to help me executing nodes. Simply in the bash script launch.sh there is:

```
roslaunch epson_g3_description display.launch model:='$(find epson_g3_description)/
↪urdf/scara.urdf'
```

7.2 URDF basics

URDF is an xml file that describe the geometry of a robot. URDF is a tree structure with one root link. The measuring units are meters and radians.

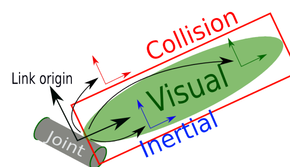
7.2.1 Robot

A robot is composed mainly from links, and joints.

```
<robot name="robot_name">
  <link>  </link>
  <link>  </link>

  <joint>  </joint>
  <joint>  </joint>
</robot>
```

7.2.2 Link

The link element describes a rigid body with an inertia, visual features, and collision properties.



The main components of `link` tag are as follow:

```
<link name="link_name">

  <visual>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
```

(continues on next page)

(continued from previous page)

```
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
  </visual>

  <collision>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
          <cylinder length="0.6" radius="0.2"/>
      </geometry>
  </collision>

  <inertial>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <mass value="1"/>
      <inertia
          ixx="1.0" ixy="0.0" ixz="0.0"
          iyy="1.0" iyz="0.0"
          izz="1.0"/>
  </inertial>
</link>
```

The `visual` tag specifies the shape of the object (`box`, `cylinder`, `sphere`, `mesh`, etc.) for visualization purposes. Its `origin` is the reference frame of the visual element with respect to the reference frame of the link (The reference frame of the link is its joint).

The `collision` can be the same as visual, or its geometry a little bit bigger. Its `origin` is the reference frame of the collision element, relative to the reference frame of the link.

The `inertial` tag is need if the model is loaded in a simulator with physics engine. Its `origin` is the pose of the inertial reference frame, relative to the link reference frame. The origin of the inertial reference frame needs to be at the center of gravity. The axes of the inertial reference frame do not need to be aligned with the principal axes of the inertia.

7.2.3 Joint

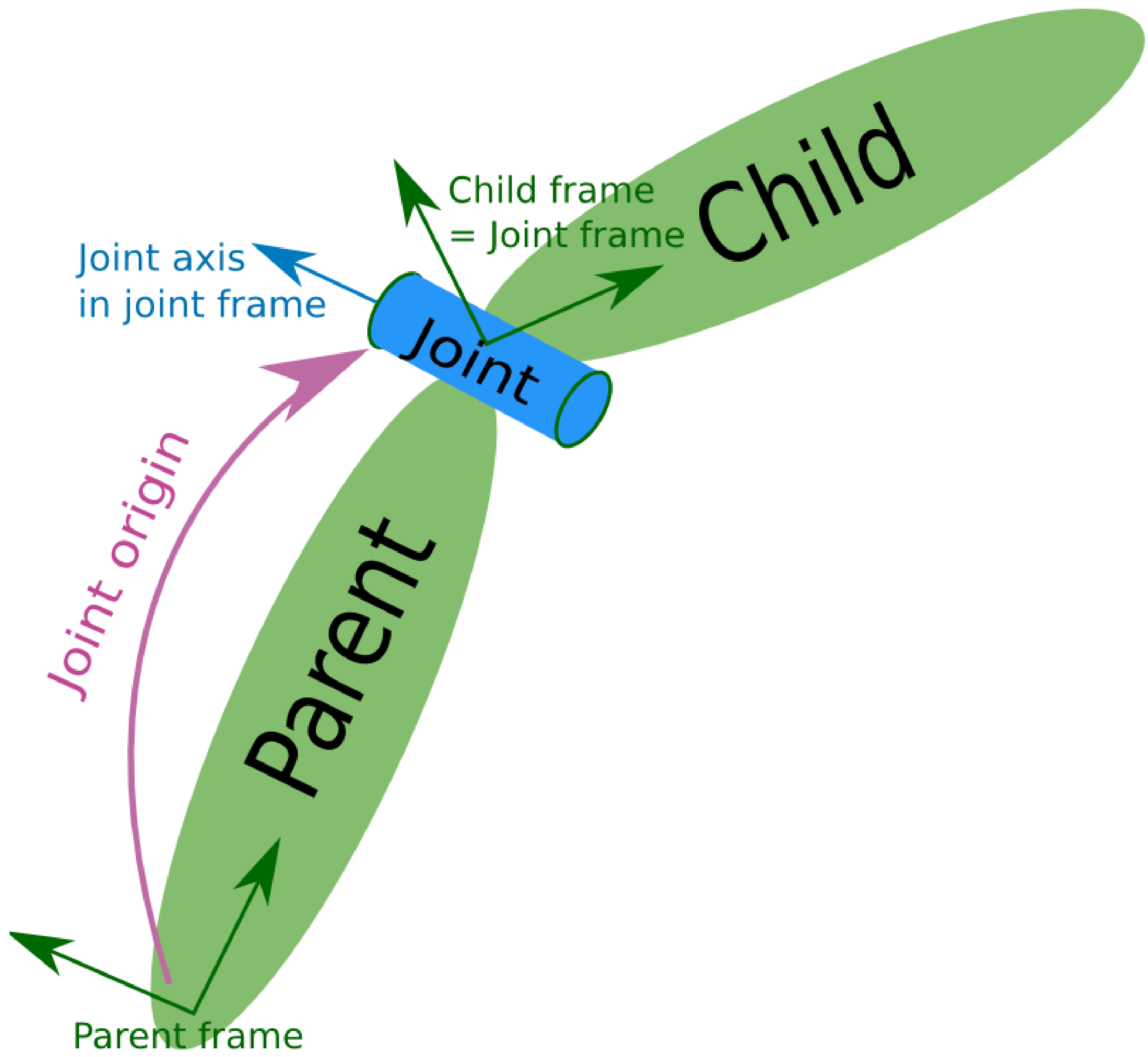
The joint describe the relative motion between two links. It can be `revolute`, `continuous`, `prismatic`, `fixed`, `floating`, `planar`.

Basic properties of a joint tag:

```
<joint name="joint_name" type="continuous">
  <parent link="link1"/>
  <child link="link2"/>

  <origin xyz="0 0 0" rpy="0 0 0"/>
  <axis xyz="1 0 0"/>

</joint>
```

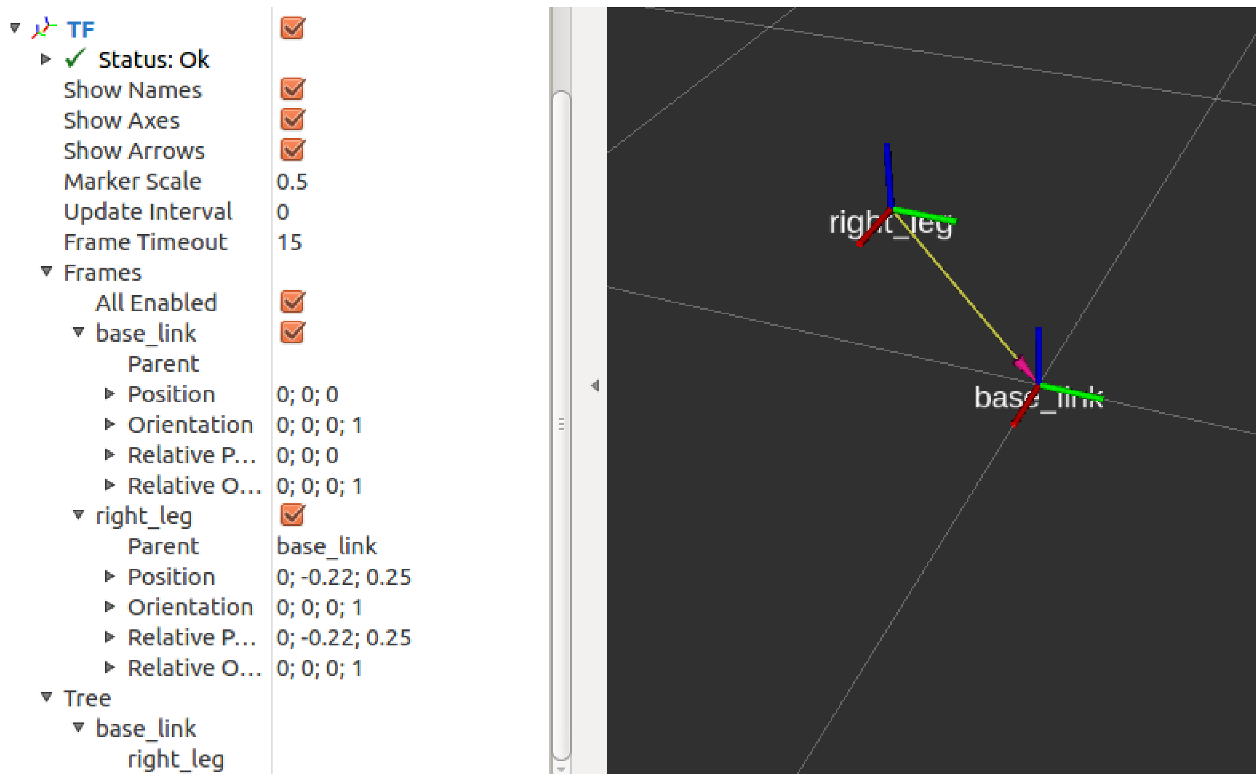
The `origin` is the transform from the parent link to the child link. The joint is located at the origin of the child link. So the origin is the relative position of the child frame respect to the parent frame.

The joint `axis` specified in the joint frame. This is the axis of rotation for revolute joints, the axis of translation for prismatic joints, and the surface normal for planar joints. The `axis` is specified in the joint frame of reference. Fixed and floating joints do not use the `axis` field.


Child frame
= Joint frame
Child
Joint axis
in joint frame
Joint
Joint origin
Parent
Parent frame

The `joint` have other properties as `dynamics`, `limit`, etc. `Limits` are in radians for revolute joints and meters for prismatic joints and are omitted if the joint is continuous or fixed.

The following image show the relationship between two joints.



The previous image is produced by the following URDF model. Note that there is no visual aspect of the links. Only joints are defined.

```
<?xml version="1.0"?>
<robot name="origins">

  <link name="base_link">
  </link>

  <link name="right_leg">
  </link>

  <joint name="base_to_right_leg" type="fixed">
    <parent link="base_link"/>
    <child link="right_leg"/>
    <origin xyz="0 -0.22 0.25"/>
  </joint>

</robot>
```

7.2.4 Tansmission

Transmissions link actuators to joints and represents their mechanical coupling. The transmission element is an extension to the URDF robot description model that is used to describe the relationship between an actuator and a joint. This

allows one to model concepts such as gear ratios and parallel linkages. A transmission transforms efforts/flow variables such that their product - power - remains constant. Multiple actuators may be linked to multiple joints through complex transmission.

```
<transmission name="tran1">
  <type>transmission_interface/SimpleTransmission</type>

  <joint name="joint1">
    <hardwareInterface>EffortJointInterface</hardwareInterface>
  </joint>

  <actuator name="motor1">
    <hardwareInterface>EffortJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>

</transmission>
```

7.2.5 Gazebo

Gazebo can be add to different elements. Refer to the URDF-Tutorials and Gazebo tutorials. In order to simulate the model correctly in Gazebo at least he `inertia` and `transmission` tags for all links should be defined.

7.2.6 Other properties

Refer to URDF-XML and URDF-Tutorials for more information.

7.3 Complete Robot model example

7.3.1 Joint and frame definition

We will define the 3D model of Epson Scara robot G3-251s. This robot have 4 links and 3 joints. On link is fixed, `base_link`. Two links rotate, we will call them link1 and link2. And the last link, link3, has translation motion.

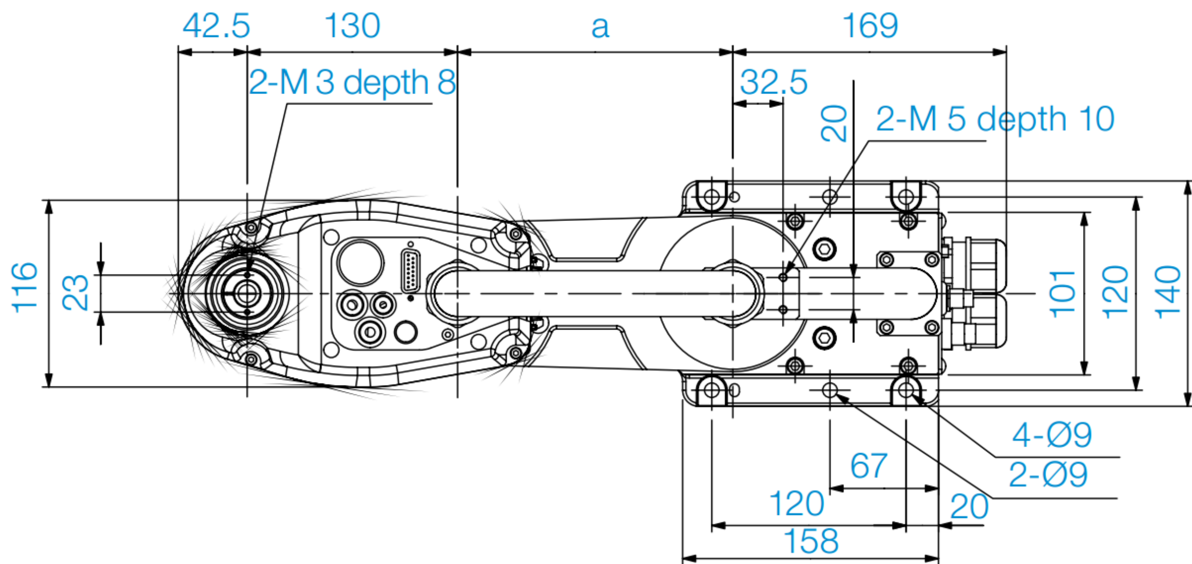
The mechanical drawing is shown below:

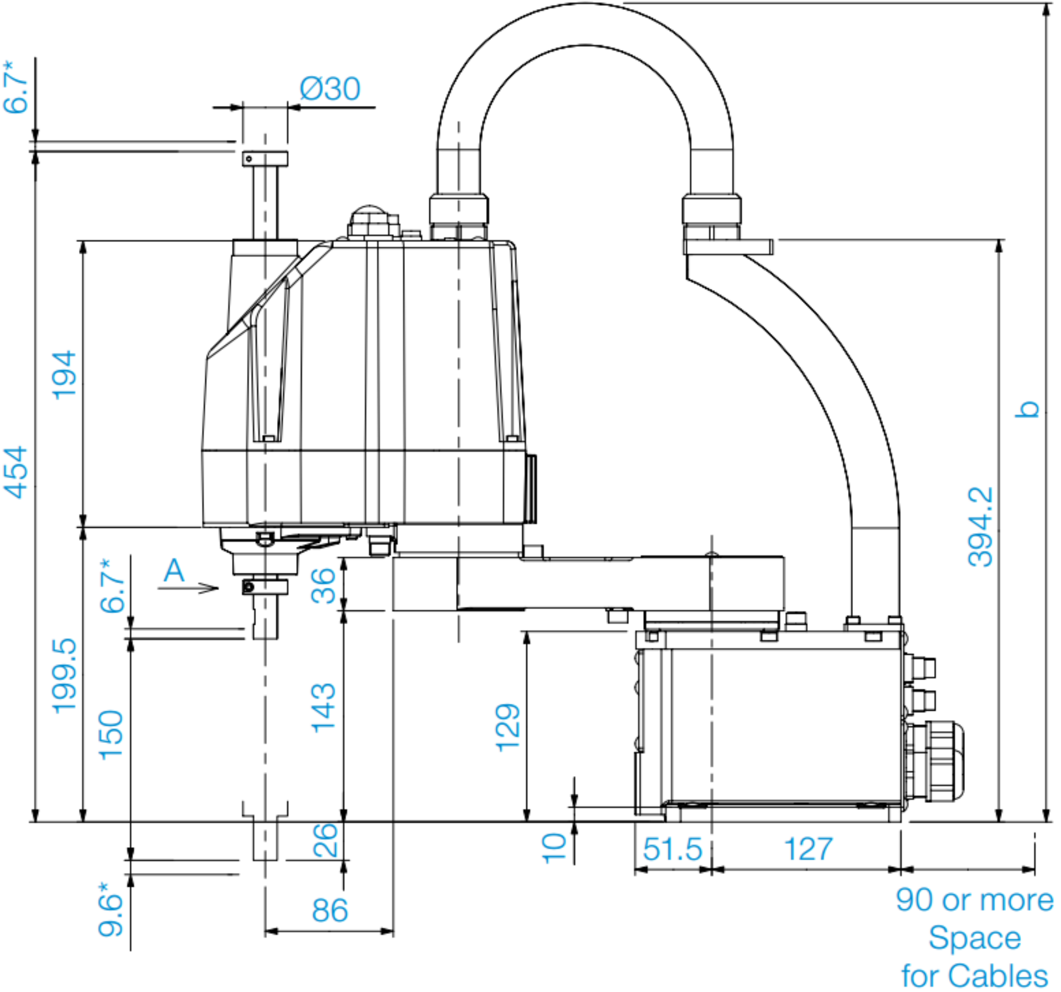
We will define first the links without visual aspect. Then we define the joints. The relative positions of the joints are taken from the previous images. As we can see, we have 2 `revolute` joints and one `prismatic`.

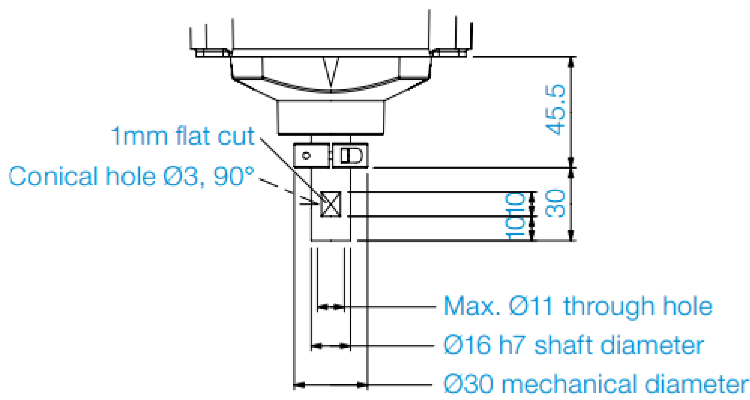
First we define the links, without the visual aspect:

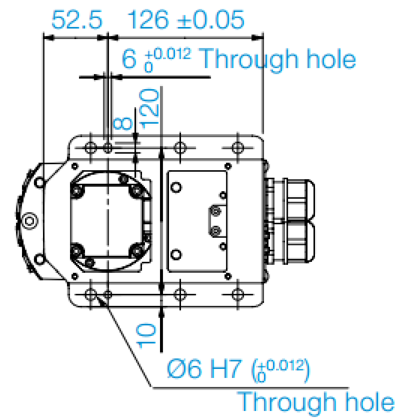
```
<link name="base_link">
</link>

<link name="link_1">
</link>

<link name="link_2">
</link>

<link name="link_3">
</link>
```

EPSON


42.5
130
a
169
2-M 3 depth 8
32.5
20
2-M 5 depth 10
116
23
101
120
140
4-Ø9
2-Ø9
67
120
20
158


6.7*
Ø30
194
454
A
6.7*
36
199.5
150
143
129
b
394.2
26
10
51.5
127
9.6*
86
90 or more
Space
for Cables
* indicates the stroke margin by mechanical stop.





Detail of "A"
(Calibration point position of Joints #3 and #4)

Reference through hole
(View from the bottom of the base)

	G3_251S	G3_301S	G3_351S
a	120	170	220
b	Max. 545	Max. 575	Max. 595

We define the `joint_1` between the base and link1. We can define it where we want along the rotate axis. But we will define it in the contact between link1 and the base. In zero position, all links open, the robot lie on the x axis of the base link. Joint1 has on offset in z of 129mm, so 0.129 m . The joint is define on the child link. Link1 rotate arround the z axis of joint1.

```
<joint name="link1_to_base" type="revolute">
  <parent link="base_link"/>
  <child link="link_1"/>
  <origin xyz="0 0 0.129"/>
  <axis xyz="0 0 1" />
  <limit effort="300" velocity="0.1" lower="-3.14" upper="3.14"/>
</joint>
```

In the same way we define joint2. As you can see the dimension `a` in the image differ from model to model. We will see later how we cam parameterize the robot model. For now we take the value for the model G3-251S, that is 120 mm.

```
<joint name="link2_to_link1" type="revolute">
  <parent link="link_1"/>
  <child link="link_2"/>
  <origin xyz="0.120 0 0"/>
  <axis xyz="0 0 1" />
  <limit effort="300" velocity="0.1" lower="-3.14" upper="3.14"/>
</joint>
```

The last joint, is prismatic. Link_3 slide along axis z. The stroke is 150mm.

```
<joint name="link3_to_link2" type="prismatic">
  <parent link="link_2"/>
  <child link="link_3"/>
  <origin xyz="0.130 0 0"/>
  <axis xyz="0 0 1" />
  <limit effort="300" velocity="0.1" lower="0" upper="0.150"/>
</joint>
```

We can use the command `check_urdf` to check if the urdf file have errors:

```
check_urdf scara.urdf
```

Run the launch file or run the script.

```
roslaunch epson_g3_description display.launch
```

`Download scara.urdf`

7.3.2 Check URDF model

Navigato to urdf direcory then:

```
check_urdf scara.urdf
```

If there are no errors, you will see the parents childs tree that define the robot.

The command

```
urdf_to_graphiz scara.urdf
```

will create 2 files: `scara.gv` and `scara.pdf`.

7.3.3 Visual aspect and mesh

Espson provide CAD files in step and solidworks format. Stp can be opend on linux using FreeCad. We need to select every link, convert it individually in stp. Pay attention to change the measuring unit. If you convert it in mm, you will see a model 1000 tmes bigger than the real robt in rviz. URDF use meter as unit. So in FreeCad change the unit in meter then export to stp. After that open the stp file and convert it to dae. In order to convert to dae the library `pycollada` should be installed.

Another way to convert to `dae`, is to convert the links in `vrml` using FreeCad, then open the `wrml` in MeshLab then convert to `dae`.

CHAPTER 8

Xacro

Xacro is an Xml macro that introduce the use of macros in an urdf file. It allow the use of variables, math and macros. And let us divide the robot model in different files.

8.1 Xacro syntax

8.1.1 Property

Instead of defining constant values, we can use variables, called property in xacro.

```
 <xacro:property name="width" value="0.2" />
 <xacro:property name="bodylen" value="0.6" />

 <link name="base_link">
     <visual>
         <geometry>
             <cylinder radius="${width}" length="${bodylen}"/>
         </geometry>
     </visual>
 </link>

In the previous snippet we create two properties with default values: ::

 <xacro:property name="width" value="0.2" />
 <xacro:property name="bodylen" value="0.6" />
```

In the `geometry` tag, instead of using constant values, we assign the properties just defined to the radius and the length. Note the use of `${}`.

8.1.2 Math

```
<cylinder radius="${wheeldiam/2}" length="0.1"/>
<origin xyz="${reflect*(width+.02)} 0 0.25" />
```

8.1.3 Macro

Marco are piece of code that can be used as functions.

```
<xacro:macro name="default_origin">
  <origin xyz="0 0 0" rpy="0 0 0"/>
</xacro:macro>
```

When need the macro can be called by name:

```
<xacro:default_origin />
```

When called, the macro content will placed where is called.

Macros can accept also input parameters:

```
<xacro:macro name="default_inertial" params="mass">
    <inertial>
            <mass value="${mass}" />
            <inertia ixx="1.0" ixy="0.0" ixz="0.0"
                 iyy="1.0" iyz="0.0"
                 izz="1.0" />
    </inertial>
</xacro:macro>
```

Can be called in this way:

```
<xacro:default_inertial mass="10"/>
```

Macros input parameters can be also blocks or macros. When we need to pass a block as a parameter to another macro, we precede the parameter name with `*`.

```
<xacro:macro name="blue_shape" params="name *shape">
    <link name="${name}">
        <visual>
            <geometry>
                <xacro:insert_block name="shape" />
            </geometry>
            <material name="blue"/>
        </visual>
        <collision>
            <geometry>
                <xacro:insert_block name="shape" />
            </geometry>
        </collision>
    </link>
</xacro:macro>
```

In this snippet, we have 2 parameters. The first parameter `name` is a string, the second one `shape` is a block.

A block is defined as follow:

```
<xacro:blue_shape name="base_link">
  <cylinder radius=".42" length=".01" />
</xacro:blue_shape>
```

The macro can be called ?????????????????:

```
<xacro:blue_shape mass="10" shape="blue_shape"/>
```

8.1.4 Example

Here is defined a macro called leg. As the model have two legs, in order to avoid duplicated code, we define a macro with two paramters, `prefix` that is part of the link and joint names. And `reflect` to define the position of the leg respect to the axis.

```
<xacro:macro name="leg" params="prefix reflect">

    <link name="${prefix}_leg">
        <visual>
            <geometry>
                <box size="${leglen} 0.1 0.2"/>
            </geometry>
            <origin xyz="0 0 -${leglen/2}" rpy="0 ${pi/2} 0"/>
            <material name="white"/>
        </visual>

        <!-- call to default_inertial macro -->
        <xacro:default_inertial mass="10"/>
    </link>

    <joint name="base_to_${prefix}_leg" type="fixed">
        <parent link="base_link"/>
        <child link="${prefix}_leg"/>
        <origin xyz="0 ${reflect*(width+.02)} 0.25" />
    </joint>

</xacro:macro>

<!-- call to legl macro -->
<xacro:leg prefix="right" reflect="1" />
<xacro:leg prefix="left" reflect="-1" />
```

8.2 Xacro to urdf

Once an xacro file is defined it must be converted to urdf file in order to be used.

```
rosrun xacro xacro scara.xacro --inorder > scara_generated.urdf
```

The command can be integrated inside a launch file.

```
<param name="robot_description" command="$(find xacro)/xacro --inorder $(find epson_
↪g3_description)/urdf/scara.xacro" />
```

Or better it can be parameterized:

```
<param name="robot_description" command="$(find xacro)/xacro --inorder $(arg model)" /
↪>
```

We will defined a new launch file:

```
<launch>
<arg name="pkg" default="epson_g3_description"/>

  <arg name="model" default="$(find epson_g3_description)/urdf/scara.xacro"/>
  <arg name="rvizconfig" default="$(find epson_g3_description)/rviz/urdf.rviz" />

  <arg name="gui" default="true" />

  <param name="robot_description" command="$(find xacro)/xacro --inorder $(arg model)
↪" />
  <param name="use_gui" value="$(arg gui)"/>

  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_
↪publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher
↪" />
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)" required="true
↪" />
</launch>
```

`Download display3.launch`

8.3 Scara robot with xacro

`Download scara.xacro`

8.4 Include files

When the robot model is complex, it is convenient to divide the definition in different files. Suppose we have a modile robot. We can create at least 2 files. One file to define the wheel and the other one the robot, where the wheel file will be included.

Let' create a file called `wheel.xacro`, where we define different marcos and properties.

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

  <!-- Wheels -->
  <xacro:property name="wheel_radius" value="0.04" />
  <xacro:property name="wheel_height" value="0.02" />
  <xacro:property name="wheel_mass" value="2.5" /> <!-- in kg-->

  <xacro:macro name="cylinder_inertia" params="m r h">
    <inertia  ixx="${m*(3*r*r+h*h)/12}" ixy = "0" ixz = "0"
              iyy="${m*(3*r*r+h*h)/12}" iyz = "0"
              izz="${m*r*r/2}" />
  </xacro:macro>

```

(continues on next page)

(continued from previous page)

```
   <xacro:macro name="wheel" params="fb lr parent translateX translateY flipY">
    <!-- other stuff -->
  </xacro:macro>

</robot>
```

This file can be included in other xacro file:

```
<xacro:include filename="$(find package_name)/urdf/wheel.xacro" />
```

And the macro `wheel` defined in the file `wheel.xarco` can be called:

```
<!-- Wheel Definitions -->
<wheel fb="front" lr="right" parent="base_link" translateX="0" translateY="0.5" flipY=
↪"1"/>
<wheel fb="front" lr="left" parent="base_link" translateX="0" translateY="-0.5" flipY=
↪"1"/>
```

CHAPTER 9

Mobile robot

In this chapter we will create a mobile robot model and simulate it with gazebo. create a package called `mobile_car_description`:

```
catkin_create_pkg mobile_car_description geometry_msgs urdf rviz xacro

cd mobile_car_description

mkdir launch urdf meshes rviz scripts
```

The robot that we will use is shown in the following images. It is bought from taobao.com.

The mobile robot or small car, have:

- 2x dc motors with quadrature encoders
- 2x Wheels
- 1x castor wheel
- 2x plates

9.1 Preparation

Writing urdf or xacro file is time consuming. For this reason we will define different folders and files.

```
mkdir motor wheel laser chassis wheel_castor
```

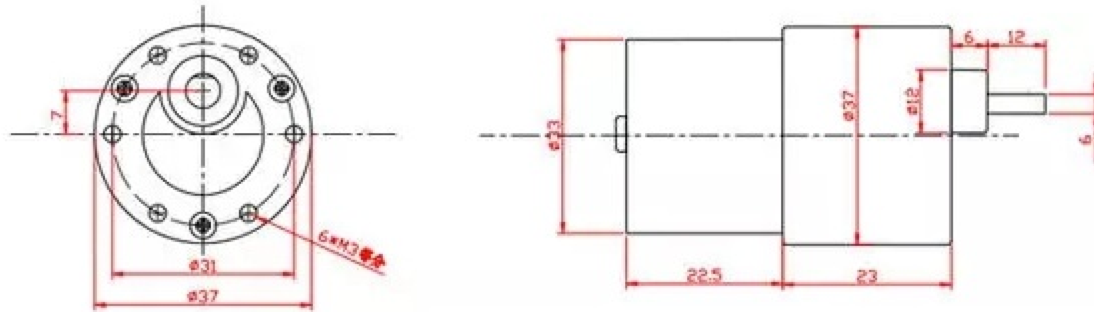
Every folder will contain macro definition and test xacro files. If the macro definition is long, it will contain a folder called steps where every step of the model creation is saved.

In the `urdf` directory we will create a file called `define.xacro` where we define constants, colors and any thing else in order to avoid code duplication. We will create also a `config.xacro` file where we will put the mechanical dimensions of the robot.

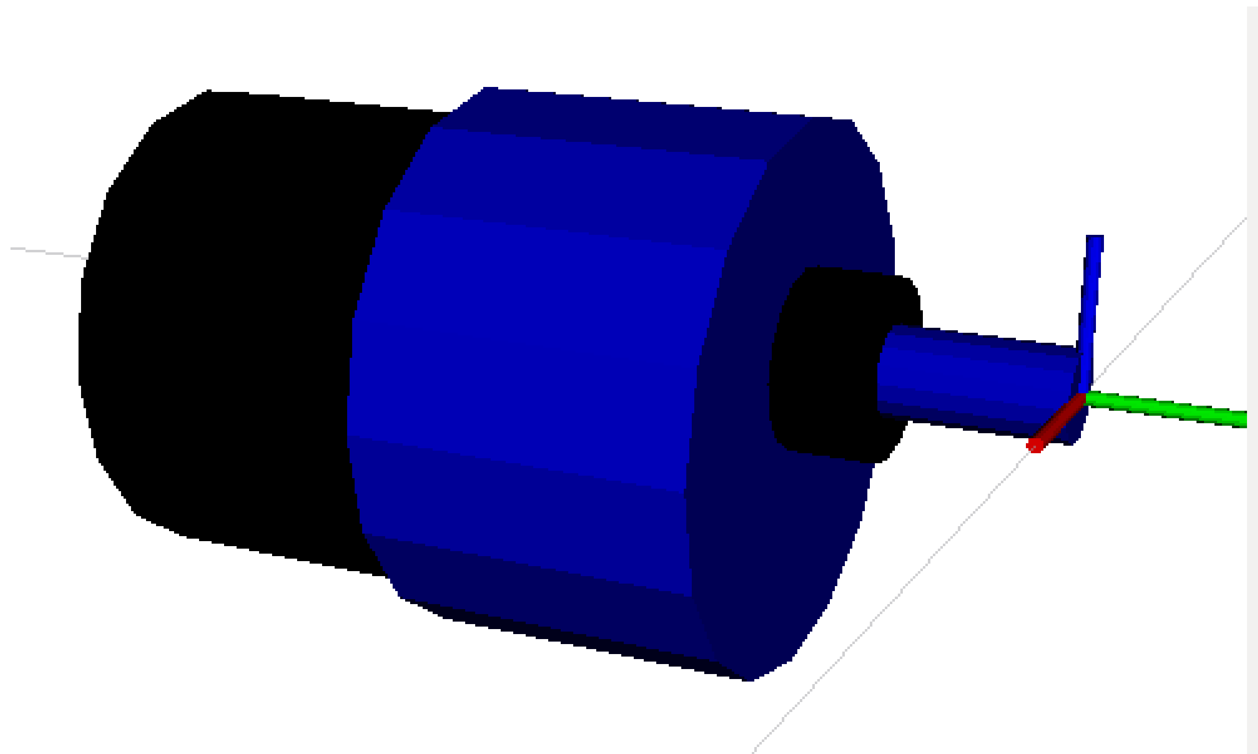
Different script will be written to speed up the testing.

9.2 Motors

As exercise we can write the model of the motor where mechanical dimensions are shown in the following image:



The final model will be like this:



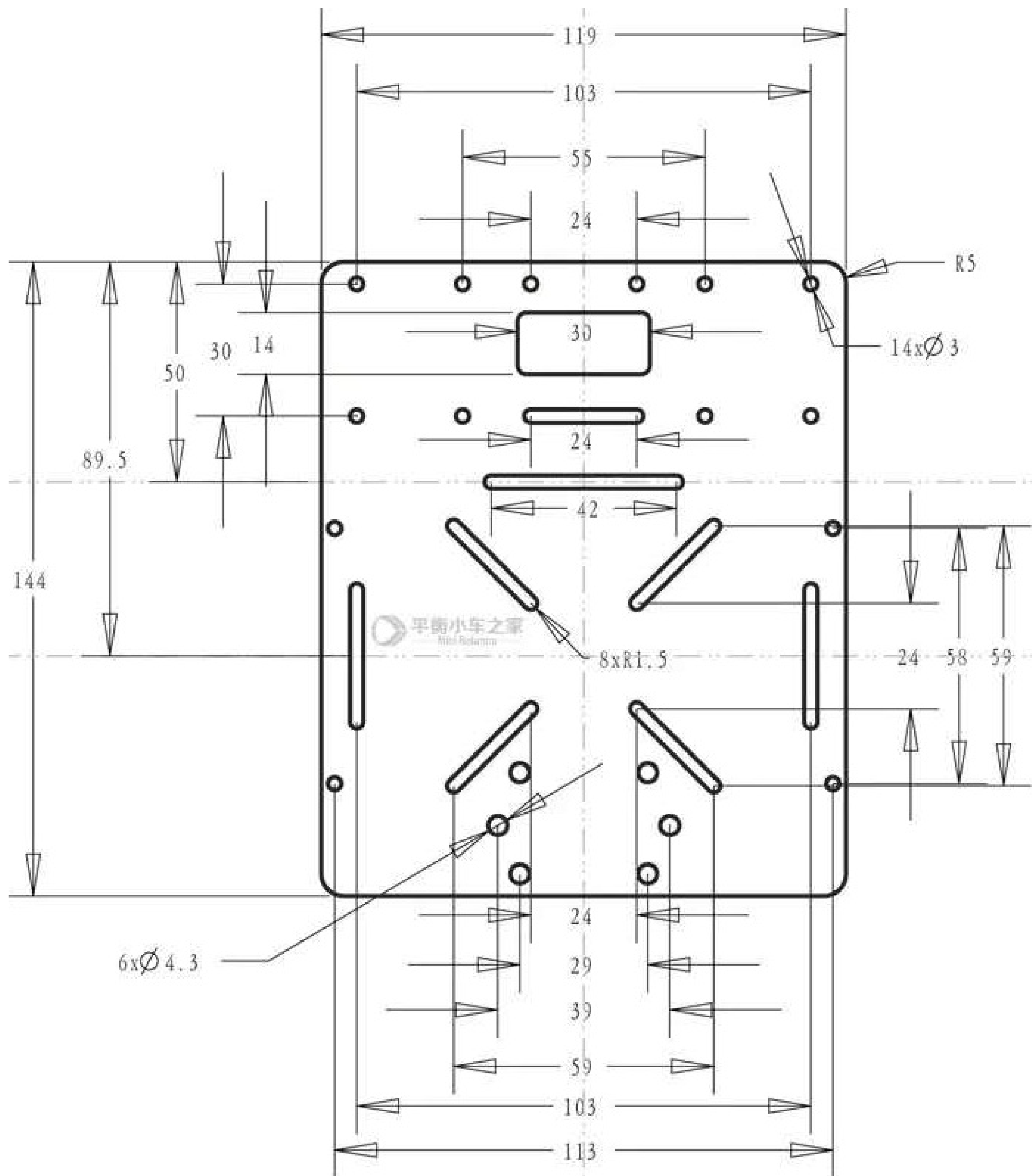
We will not discuss the creation of this model because we will not use it. It is better to import a mesh file of the motor.

`Download motor.xacro`

9.3 Wheels

9.4 Body

Mechanical dimensions are shown in the following image:


119
103
55
24
R5
30
14
30
50
14xØ3
24
89.5
42
144
8xR1.5
24
58
59
6xØ4.3
24
29
39
59
103
113

So mainly the body is a box. We will define also 2 brackets where the wheels are mounted.

9.5 Complete robot

In the following code we see how to include the definition files of the mobile robot components. And the how we call the macros to define those components.

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

  <!-- ==================================================================================================== -->
  <!-- INCLUDE FILES -->
  <xacro:include filename="$(find mobile_car_description)/urdf/define.xacro" />
  <xacro:include filename="$(find mobile_car_description)/urdf/config.xacro" />

  <xacro:include filename="$(find mobile_car_description)/urdf/wheel/wheel.xacro" />
  <xacro:include filename="$(find mobile_car_description)/urdf/wheel_castor/wheel_castor.xacro" />
  <xacro:include filename="$(find mobile_car_description)/urdf/chassis/chassis.xacro" />

  <!-- ==================================================================================================== -->
  <!-- BASE-LINK -->
  <link name="base_link">
  </link>

  <!-- ==================================================================================================== -->
  <!-- Body -->
  <chassis name="body" parent="base_link" length="${base_length}" width="${base_width}" height="${base_height}" x="0" y="0" z="0" mass="${base_mass}"/>

  <!-- ==================================================================================================== -->
  <!-- Wheels -->
  <wheel name="left" parent="body_link" radius="${wheel_radius}" height="${wheel_height}" x="${wheel_x}" y="${wheel_y}" z="${wheel_z}" mass="${wheel_mass}"/>
  <wheel name="right" parent="body_link" radius="${wheel_radius}" height="${wheel_height}" x="${wheel_x}" y="${-wheel_y}" z="${wheel_z}" mass="${wheel_mass}"/>

  <wheel_castor name="front" parent="body_link" radius="${caster_radius}" x="${caster_x}" y="${caster_y}" z="${caster_z}" mass="${caster_mass}"/>

</robot>
```

Dividing the model creation in different files make it easier to debug any error.

Open the robot with rviz:

```
roslaunch mobile_car_description display.launch model:='$(find mobile_car_
↪description)/urdf/mobile_robot.xacro'
```

9.6 Meshes

CHAPTER 10

Gazebo

In order to simulate and control the robot with gazebo we need to add some tags to the urdf files.

Create a package that depend on the robot model, `mobile_car_description` called `mobile_car_gazebo`:

```
catkin_create_pkg mobile_car_gazebo gazebo_msgs gazebo_plugins gazebo_ros gazebo_ros_
↪control mobile_car_description
mkdir launch
```

10.1 Launch file

In order to simulate the robot in Gazebo we need to open a world file. In this example we open the `empty_world.world` already defined in the package `gazebo_ros` with default values.

```
<!-- We resume the logic in empty_world.launch -->
<include file="$(find gazebo_ros)/launch/empty_world.launch">
  <arg name="world_name" value="worlds/empty_world.world"/> <!-- Note: the world_name␣
↪is with respect to GAZEBO_RESOURCE_PATH environmental variable -->
  <arg name="paused" value="false"/>
  <arg name="use_sim_time" value="true"/>
  <arg name="gui" value="true"/>
  <arg name="recording" value="false"/>
  <arg name="debug" value="false"/>
</include>
```

World files are `SDF` files (xml files).

10.1.1 Spawn robots

There are two ways to launch (`spawn`) your URDF-based robot into Gazebo using roslaunch:

- ROS Service Call Spawn Method:

The first method keeps your robot's ROS packages more portable between computers and repository check outs. It allows you to keep your robot's location relative to a ROS package path, but also requires you to make a ROS service call using a small (python) script.

- Model Database Method:

  The second method allows you to include your robot within the .world file, which seems cleaner and more convenient but requires you to add your robot to the Gazebo model database by setting an environment variable.

Service call

This method uses a small python script called spawn_model to make a service call request to the gazebo_ros ROS node (named simply "gazebo" in the rostopic namespace) to add a custom URDF into Gazebo. The spawn_model script is located within the gazebo_ros package. You can use this script in the following way:

```
rosrun gazebo_ros spawn_model -file `rospack find MYROBOT_description`/urdf/MYROBOT.
↪urdf -urdf -x 0 -y 0 -z 1 -model MYROBOT
```

To see all of the available arguments for spawn_model including namespaces, trimesh properties, joint positions and RPY orientation run:

```
rosrun gazebo_ros spawn_model -h
```

In launch file:

```
<!-- Spawn a robot into Gazebo -->
<node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-file $(find baxter_
↪description)/urdf/baxter.urdf -urdf -z 1 -model baxter" />
```

Or using xarco model:

```
<!-- Convert an xacro and put on parameter server -->
<param name="robot_description" command="$(find xacro)/xacro --inorder $(find pr2_
↪description)/robots/pr2.urdf.xacro" />

<!-- Spawn a robot into Gazebo -->
<node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-param robot_
↪description -urdf -model pr2" />
```

Model Database Robot Spawn Method

10.1.2 Complete launch file

Create a launch file in the package `mobile_car_gazebo` called `mobile_car_gazebo.launch`:

```
<launch>
  <arg name="paused" default="false"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="true"/>
  <arg name="recording" default="false"/>
  <arg name="debug" default="false"/>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="debug" value="$(arg debug)" />
```

(continues on next page)

(continued from previous page)

```
    <arg name="gui" value="$(arg gui)" />
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="recording" value="$(arg recording)"/>
  </include>

  <!-- Convert an xacro and put on parameter server -->
  <param name="robot_description" command="$(find xacro)/xacro --inorder '$(find mobile_car_description)/urdf/mobile_robot_with_laser.xacro'" />

  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher"></node>

  <node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher" output="screen">
    <param name="publish_frequency" type="double" value="50.0" />
  </node>

  <!-- Spawn a robot into Gazebo by running a python script called spawn_model-->
  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false" output="screen" args="-urdf -model mobile_robot -param robot_description"/>

</launch>
```

Run the launh file:

```
roslaunch mobile_car_gazebo mobile_car_gazebo.launch
```

10.2 gazebo_ros_control

In order to simulate the robot we need to add a controller.The plugin `gazebo_ros_control` plugin can load different libraries. In the following code, `libgazebo_ros_control.so` is loaded:

```
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/MYROBOT</robotNamespace>
    <robotSimType>gazebo_ros_control/DefaultRobotHWSim</robotSimType>
  </plugin>
</gazebo>
```

The gazebo_ros_control <plugin> tag also has the following optional child elements:

- `<robotNamespace>` The ROS namespace to be used for this instance of the plugin, defaults to robot name in URDF/SDF
- `<controlPeriod>` The period of the controller update (in seconds), defaults to Gazebo's period
- `<robotParam>` The location of the robot_description (URDF) on the parameter server, defaults to '/robot_description'
- `<robotSimType>` The pluginlib name of a custom robot sim interface to be used (see below for more details), defaults to 'DefaultRobotHWSim'. The default behavior provides the following `ros_control` interfaces:
  - hardware_interface::JointStateInterface
  - hardware_interface::EffortJointInterface

  - hardware_interface::VelocityJointInterface - not fully implemented

10.3 Modifying robot model

Add the follwing plugin to the robot model in `mobile_robot.xacro` in the package `mobile_car_description`:

```
<!-- Differential drive controller  -->
<gazebo>
  <plugin name="differential_drive_controller" filename="libgazebo_ros_diff_drive.so">

    <rosDebugLevel>Debug</rosDebugLevel>
    <publishWheelTF>false</publishWheelTF>
    <robotNamespace>/</robotNamespace>
    <publishTf>1</publishTf>
    <publishWheelJointState>false</publishWheelJointState>
    <alwaysOn>true</alwaysOn>
    <updateRate>100.0</updateRate>

    <leftJoint>left_wheel_joint</leftJoint>
    <rightJoint>right_wheel_joint</rightJoint>

    <wheelSeparation>${base_width}</wheelSeparation>
    <wheelDiameter>${2*wheel_radius}</wheelDiameter>
    <broadcastTF>1</broadcastTF>
    <wheelTorque>30</wheelTorque>
    <wheelAcceleration>1.8</wheelAcceleration>
    <commandTopic>cmd_vel</commandTopic>
    <odometryFrame>odom</odometryFrame>
    <odometryTopic>odom</odometryTopic>
    <robotBaseFrame>base_link</robotBaseFrame>

  </plugin>

</gazebo>
```

Different parameters have to be add to the plugin. The most important is `commandTopic` where we tell the controller to listen to `cmd_vel` topic.

10.4 Teleoperation

10.4.1 Rqt

Open rqt node, then use the plugin robot steering to publish the robot speed on `cmd_vel` topic.

10.4.2 teleop_twist_keyboard package

CHAPTER 11

Navigation

11.1 Hokuyo laser scanner

We will add a laser scanner to the mobile robot. The sensor behavior is already defined in gazebo as a plugin. The plugin that we will use is called `libgazebo_ros_laser.so`. We will create a new xacro file called `hokuyo_laser.xacro`, we define a link, a joint and add the gazebo plugin:

```
<gazebo reference="hokuyo_link">
<material>Gazebo/Blue</material>
<turnGravityOff>false</turnGravityOff>
<sensor type="ray" name="head_hokuyo_sensor">
  <pose>${length/2} 0 0 0 0 0</pose>
  <visualize>false</visualize>
  <update_rate>40</update_rate>
  <ray>
    <scan>
      <horizontal>
        <samples>720</samples>
        <resolution>1</resolution>
        <min_angle>-1.570796</min_angle>
        <max_angle>1.570796</max_angle>
      </horizontal>
    </scan>
    <range>
      <min>0.10</min>
      <max>10.0</max>
      <resolution>0.001</resolution>
    </range>
  </ray>
  <plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_ros_laser.so">
    <topicName>/scan</topicName>
    <frameName>hokuyo_link</frameName>
  </plugin>
```

(continues on next page)

(continued from previous page)

```
</sensor>
</gazebo>
```

Create a new xacro file called `mobile_robot_with_laser.xacro`, where we include the mobile robot definition and the laser macro:

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

  <!--␣
↪==================================================================================
↪-->
  <!-- INCLUDE FILES -->
  <xacro:include filename="$(find mobile_car_description)/urdf/mobile_robot.xacro" />
  <xacro:include filename="$(find mobile_car_description)/urdf/laser/hokuyo_laser.
↪xacro" />

  <!--␣
↪==================================================================================
↪-->
   <!-- laser -->
  <hokuyo_laser name="hokuyo_link" parent="body_link" length="${hokuyo_length}" width=
↪"${hokuyo_width}" height="${hokuyo_height}" x="${hokuyo_x}" y="${hokuyo_y}" z="$
↪{hokuyo_z}" mass="${hokuyo_mass}"/>

  <!--␣
↪==================================================================================
↪-->

</robot>
```

Open the robot in gazebo:

```
roslaunch mobile_car_gazebo mobile_car_gazebo.launch model:='$(find mobile_car_
↪description)/urdf/mobile_robot_with_laser.xacro'
```

Using `rostopic list` we can see the topic called `/scan` is being published. `rostopic echo /scan` to see the laser scanner value.

11.2 Add camera

```
<gazebo reference="camera_link">
    <sensor type="camera" name="camera1">
      <update_rate>30.0</update_rate>
      <camera name="head">
        <horizontal_fov>1.3962634</horizontal_fov>
        <image>
          <width>800</width>
          <height>800</height>
          <format>R8G8B8</format>
        </image>
        <clip>
          <near>0.02</near>
```

(continues on next page)

(continued from previous page)

```
        <far>300</far>
      </clip>
      <noise>
        <type>gaussian</type>
        <!-- Noise is sampled independently per pixel on each frame.
             That pixel's noise value is added to each of its color
             channels, which at that point lie in the range [0,1]. -->
        <mean>0.0</mean>
        <stddev>0.007</stddev>
      </noise>
    </camera>
    <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
      <alwaysOn>true</alwaysOn>
      <updateRate>0.0</updateRate>
      <cameraName>robot/camera1</cameraName>
      <imageTopicName>image_raw</imageTopicName>
      <cameraInfoTopicName>camera_info</cameraInfoTopicName>
      <frameName>camera_link</frameName>
      <hackBaseline>0.07</hackBaseline>
      <distortionK1>0.0</distortionK1>
      <distortionK2>0.0</distortionK2>
      <distortionK3>0.0</distortionK3>
      <distortionT1>0.0</distortionT1>
      <distortionT2>0.0</distortionT2>
    </plugin>
  </sensor>
</gazebo>
```

Show image:

```
rosrun image_view image_view image:=/robot/camera1/image_raw
```

11.3 Loading maps in gazebo

11.4 Teleoperate the robot

CHAPTER 12

Robot arm model

In this chapter a Robot arm model will be defined. Refer to Gazebo chapter for simulation. We will continue the model started in the introductory chapter on urdf. We will create a new package called `scara_g3_description`.

```
cd workspace/src/Robots

catkin_create_pkg scara_g3_description geometry_msgs urdf rviz xacro

cd scara_g3_description

mkdir launch urdf meshes rviz scripts
```

12.1 Joints

12.2 Links

12.3 Gripper

12.4 Meshes

12.5 Transmissions

A transmission macro is created:

```
<xacro:macro name="transmission_block" params="joint_name">
  <transmission name="tran1">

     <type>transmission_interface/SimpleTransmission</type>
```

(continues on next page)

(continued from previous page)

```
    <joint name="${joint_name}">
                <hardwareInterface>hardware_interface/PositionJointInterface</
↪hardwareInterface>
    </joint>

    <actuator name="motor1">
     <mechanicalReduction>1</mechanicalReduction>
    </actuator>

  </transmission>
</xacro:macro>

<xacro:transmission_block joint_name="joint_name1"/>
<xacro:transmission_block joint_name="joint_name2"/>
<xacro:transmission_block joint_name="joint_name3"/>
```

`transmission_interface/SimpleTransmission` is the only interface supported. The `hardwareInterface` could be position, velocity, or effort interfaces. In this case we choose `PositionJointInterface`. The hardware interface will be loaded by `gazebo_ros_control` plugin. Refer to Gazebo chapter for more information about this plugin.

12.6 Gazebo

Refer to Gazebo chapter.

CHAPTER 13

Industrial robots

13.1 ABB

```
git clone https://github.com/ros-industrial/abb.git
```

13.2 Universal robot

```
git clone https://github.com/ros-industrial/universal_robot.git
```

13.3 Yaskawa Motoman

CHAPTER 14

Service and Client

14.1 Simple Service and Client

14.2 User messages

14.3 Services

14.3.1 Service node

```
listing/tutorial/add_two_ints_server.cpp
```

14.3.2 Client node

```
listing/tutorial/add_two_ints_client.cpp
```

14.3.3 Build the package

```
caption=CmakeLists with new messages and services, label=lstCMakeLists
listing/tutorial/CMakeLists.txt
```

CHAPTER 15

Parameters server

CHAPTER 16

ActionServer and ActionClient

16.1 Simple Action server and Client

16.2 User Action

16.3 Action server

16.4 Action Client

16.5 Build the package

```
caption=CmakeLists with new messages and services, label=lstCMakeLists
listing/tutorial/CMakeLists.txt
```

CHAPTER 17

Plugin

17.1 Simple Service and Client

17.2 User messages

17.3 Services

17.3.1 Service node

```
listing/tutorial/add_two_ints_server.cpp
```

17.3.2 Client node

```
listing/tutorial/add_two_ints_client.cpp
```

17.3.3 Build the package

```
caption=CmakeLists with new messages and services, label=lstCMakeLists
listing/tutorial/CMakeLists.txt
```

CHAPTER 18

Nodelet

CHAPTER 19

Rqt plugins

CHAPTER 20

Rviz plugins

CHAPTER 21

Gazebo plugins

CHAPTER 22

Arduino

22.1 Installation

Installing the rosserial metapackage:

```
sudo apt-get install ros-kinetic-rosserial
```

Install the rosserial-arduino client package:

```
sudo apt-get install ros-kinetic-rosserial-arduino
```

Clone

```
git clone https://github.com/ros-drivers/rosserial.git
```

Download and install Arduino IDE.

To use the serial port without root permissions:

```
ls -l /dev/ttyACM*
```

or

```
ls -l /dev/ttyUSB*
//
sudo usermod -a -G dialout <username>
```

In Arduino IDE set the Sketchbook location to `/home/robot/arduino`. Arduino should create a folder called libraries inside it.

Open the terminal and navigate to

```
cd /home/robot/arduino/libraries
```

Don't forget the dot `.` at the end of the following commands, it indicate current directory

```
rosrun rosserial_arduino make_libraries.py .
```

After these steps, you should find the ros_lib voice in the examples of Arduino IDE.

22.1.1 Test the installation

Open the example string_test from Arduino IDE:

```
#include <ros.h>
#include <std_msgs/String.h>

ros::NodeHandle  nh;

std_msgs::String str_msg;
ros::Publisher chatter("chatter", &str_msg);

char hello[19] = "chapter8_tutorials";

void setup()
{
  nh.initNode();
  nh.advertise(chatter);
}

void loop()
{
  str_msg.data = hello;
  chatter.publish( &str_msg );
  nh.spinOnce();
delay(1000);
}
```

Upload the sketch to the arduino board.

Open 3 different terminals and launch:

```
roscore

rosrun rosserial_python serial_node.py /dev/ttyACM0

rostopic echo chatter
```

Download string_test.ino

CHAPTER 23

TIVA C

23.1 Installation

Download and install Energia IDE following the instruction and the website.

Clone the 2 repositories:

```
git clone https://github.com/vmatos/rosserial_tivac_tutorials.git
git clone https://github.com/vmatos/rosserial_tivac.git

git clone https://github.com/vmatos/rosserial_tivac_socket_tutorials.git
git clone https://github.com/vmatos/rosserial_tivac_socket.git

git clone https://github.com/vmatos/tiva-c-projects.git
```

CHAPTER 24

STM32

Clone

```
git clone https://github.com/bosch-ros-pkg/stm32.git

git clone https://github.com/weifengdq/ROS-STM32.git

git clone https://github.com/vmatos/stm32-projects.git
```

CHAPTER 25

Raspberry pi

CHAPTER 26

Cameras

26.1 USB

26.2 Kinect

26.3 Basler

CHAPTER 27

OpenCV

27.1 Vision

27.1.1 Camera

27.1.2 Opencv

27.1.3 Object detection

sudo apt-get install ros-kinetic-find-object-2d

CHAPTER 28

Point Cloud Library (PCL)

Note: References: Ros Industrial training. PCL_Tools.

28.1 PCL tools

Publish point cloud on the topic `topic_cloud_pcd`:

```
roscore
rosrun pcl_ros pcd_to_pointcloud table.pcd 0.1 _frame_id:=map cloud_pcd:=topic_cloud_
↪pcd
rosrun rviz rviz
```

In order to see the point cloud data, add `PointCloud2` display and select the topic `topic_cloud_pcd`.

Install `pcl-tools`

```
sudo apt install pcl-tools
```

Point cloud data can be viewed using the command line `pcl_viewer`, this command is not part of `ROS`, so no need to run `roscore`.

```
pcl_viewer table.pcd
```

28.1.1 Downsample the point cloud using the `pcl_voxel_grid`

Downsample the original point cloud using a voxel grid with a grid size of (0.05,0.05,0.05). In a voxel grid, all points in a single grid cube are replaced with a single point at the center of the voxel. This is a common method to simplify overly complex/detailed sensor data, to speed up processing steps.

```
pcl_voxel_grid table.pcd table_downsampled.pcd -leaf 0.05,0.05,0.05
pcl_viewer table_downsampled.pcd
```

28.1.2 Segmentation `pcl_sac_segmentation_plane`

Extract the table surface, find the largest plane and extract points that belong to that plane (within a given threshold).

```
pcl_sac_segmentation_plane table_downsampled.pcd only_table.pcd -thresh 0.01
pcl_viewer only_table.pcd
```

Extract the largest point-cluster not belonging to the table.

```
pcl_sac_segmentation_plane table.pcd object_on_table.pcd -thresh 0.01 -neg 1
pcl_viewer object_on_table.pcd
```

28.1.3 Remove outliers `pcl_outlier_removal`

For this example, a statistical method will be used for removing outliers. This is useful to clean up noisy sensor data, removing false artifacts before further processing.

```
pcl_outlier_removal table.pcd table_outlier_removal.pcd -method statistical
pcl_viewer table_outlier_removal.pcd
```

28.1.4 Compute the normals `pcl_normal_estimation`

This example estimates the local surface normal (perpendicular) vectors at each point. For each point, the algorithm uses nearby points (within the specified radius) to fit a plane and calculate the normal vector. Zoom in to view the normal vectors in more detail.

```
pcl_normal_estimation only_table.pcd table_normals.pcd -radius 0.1
pcl_viewer table_normals.pcd -normals 10
```

28.1.5 Mesh a point cloud `pcl_marching_cubes_reconstruction`

Point cloud data is often unstructured, but sometimes processing algorithms need to operate on a more structured surface mesh. This example uses the "marching cubes" algorithm to construct a surface mesh that approximates the point cloud data.

```
pcl_marching_cubes_reconstruction table_normals.pcd table_mesh.vtk -grid_res 20
pcl_viewer table_mesh.vtk
```

28.2 PLC basics

28.2.1 Data type

28.3 PCL ROS API

```
catkin_create_pkg pcl_test_pkg pcl_conversions pcl_ros pcl_msgs sensor_msgs
```

CHAPTER 29

Visual Servoing Platform library ViSP

CHAPTER 30

The CImg Library