ROS and experimental robotics.
Part 1: an introduction to ROS.

1 ROS tutorial

In addition to the first introduction courses (lasting 4h) you have, it is mandatory for you to carefully follow the ROS tutorial indicated on Moodle as it will explain practically how ROS works, its structure, its terminology, its basic commands and how to exploit `rospy` to actually program your own ROS node.

This section constitutes thus a **mandatory homework**, and you have about **10 days** to complete the entire tutorial (the precise dates will be given during the first presentation course of the teaching unit). In the meantime, a discussion forum is made available on Moodle and all your teachers will try respond as fast as possible to the questions you might have. Practically, we recommend you to open the ROS tutorial on Firefox in the virtual machine, so that it will be easier to copy/paste commands from the web page to your terminal. **Be careful: following the tutorial is not about copying and pasting commands, but rather about understanding the core concepts and understanding all the proposed linux/ROS commands.**

In the end of the tutorial period, your knowledge and understanding of ROS will be assessed through multiple choice questions on Moodle.

The tutorial is available at `https://moodle-sciences.upmc.fr/moodle-2021/mod/page/view.php?id=130145`. Note that the creation of a ROS workspace mentioned in some tutorial (e.g. `Installing and configuring your ROS environment`) **is already done on the provided virtual machine**. So, your ROS workspace is located at `/home/turtle/catkin_ws`.

For the record, we will use the **`ROS 1 Noetic Ninjemys`** distribution in this teaching unit.

At the end of this tutorial, you are supposed to precisely know:

- what are the root ROS concepts : nodes, messages, publishing and receiving messages, services, parameters;
- what is the architecture supporting these concepts: data structures, ROS master and all the servers running with it (parameter server, logging facilities, etc.), `roscore` , etc.;
- what is a ROS package, what it is made off, what is the package build system used in ROS and how to use it;
- master the core shell commands that allow to investigate a ROS architecture (`rosnode` , `rosmsg` , `rostopic` , `rosservice` , `roslaunch` , `rosrun` , etc.);
- and all the concepts highlighted during the first two courses (client/server applications, callback, etc.)

The two first practical sessions are devoted to the writing of your very first own ROS node. We will gradually complexify the way it works, in order for you to deal with subscribing and publishing to topics, exploiting services and the parameter server. At first, you will write a basic teleoperation node which will capture the arrow keys of your keyboard so as to make the turtle from the `tutlesim` package move on the screen. Then, we will complexify this example by changing the color of the line following the turtle as a function of its position on the screen. Finally, we will also change the color of the screen itself when the turtle is approaching each side of the screen thanks to a parameter available in the parameter server.

One of the advantage of ROS is reusability. It means that the teleoperation node you will write to move the turtle on your screen can also be used to move a real robot from your keyboard. Thanks to the use of parameters and launch files, we will illustrate this by controlling a TurtleBot 3 burger (see Figure 1) from the very same teleoperation node.

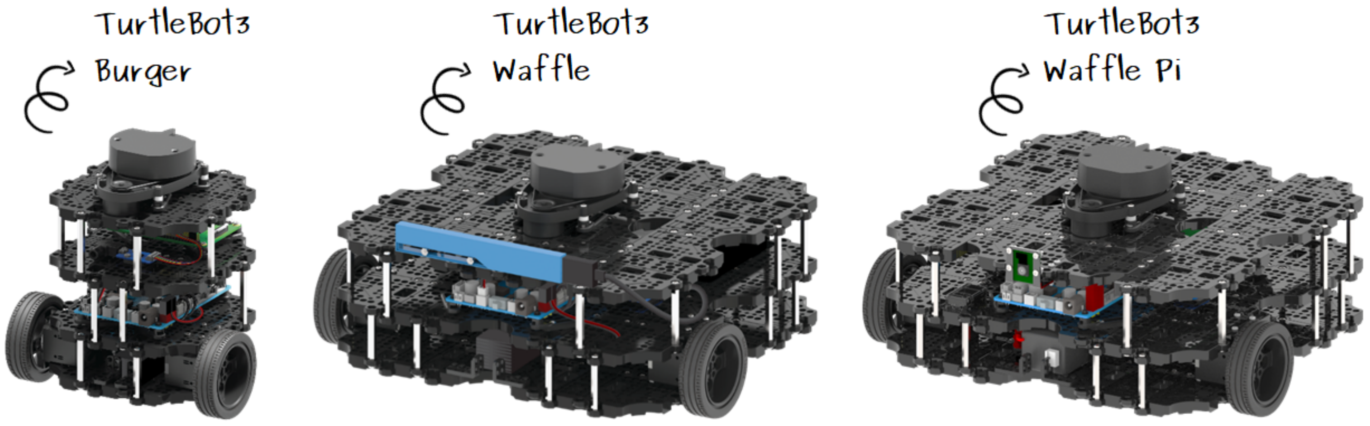

Figure 1: The entire TurtleBot 3 family. In this teaching unit, you will work with the small one, the TurtleBot3 Burger.

It is clear that all these tasks (working with topics, services, parameters and launch files) will require some times to be mastered. Nevertheless, most of these points will be reused in the Part 3 of this teaching unit. So if you do not have the time to complete everything by the end of Part 1, you will have the opportunity to work on these concepts later.

2.1 Part A: a teleoperation node

In this first part, we will write a teleoperation node that aims at controlling the movement of the turtle in the `turtlesim` package. All the forthcoming development will be based on `Python`, and we will thus use the ROS client library `rospy` to interface with ROS.

It is expected you already master Python syntax (*it should be the case thanks to the Python teaching unit from the first semester!*) as well as classical scientific libraries like `numpy` or `scipy`. If not, please refer to the documents provided during the first semester so that you will no transform this ROS class into a Python class.

The `rospy` code API documentation is available at `http://docs.ros.org/api/rospy/html/`. A short tutoriel on `rospy` can be found at `http://wiki.ros.org/rospy/Overview`.

2.1.1 A look inside the `turtlesim_node`

To begin with, we will run the `turtlesim_node` from the `turtlesim` package, and analyze its functioning before sending data to make the turtle move. Of course, all the commands, code, etc. you will develop must be written *inside* the virtual machine (i.e. the remote PC) where ROS is already installed. So please start it up before every practical sessions.

#2.1.1. From a Terminal, run the `turtlesim_node` from the `turtlesim` package.

Solution:

```
$ rosrun turtlesim turtlesim_node
```

#2.1.2. List all the currently running nodes, and display information about the node `/turtlesim`. What kind of information is available?

Solution:

```
$ rosnode list
$ rosnode info /turtlesim
```

#2.1.3. Now, list all of the currently active topics (i.e. topics where data are or can be sent or received). Indicate what topics are exposed by the `turtlesim_node`, and who is subscribed or publishing to them.

Solution:

```
$ rostopic list
$ rostopic info /turtle1/cmd_vel
```

#2.1.4. The turtle can be moved by sending appropriate messages to the topic `/turtle1/cmd_vel`. Find out what kind of message (i.e. data type) is expected on this topic.

Solution:

From the output of the previous command, one reads:

```
Type:  geometry_msgs/Twist
```

#2.1.5. Now that you know the expected message on `/turtle1/cmd_vel`, display the message description to understand precisely what it is made of.

Solution:

`$ rosmsg info geometry_msgs/Twist`, or
`$ rosmsg show geometry_msgs/Twist`

2.1.2 Creation of the `mybot_teleop` package

#2.1.6. Place yourself in the `src` folder of your ROS workspace, and create a ROS package called `mybot_teleop` with dependencies to `std_msgs` and `rospy`.

Solution:

```
$ cd  /catkin_ws/src
$ catkin_create_pkg mybot_teleop std_msgs rospy
```

#2.1.7. Next go the newly created folder, and edit the file `package.xml` to enter your name as a developer. Finally, compile the new (empty) package. Do not forget, after compilation, to source the generated setup file `~/catkin_ws/devel/setup.bash` (it has to be done only once, after the initial compilation, in all the currently opened terminal sessions).

Solution:
```
$ cd  /catkin_ws
$ catkin_make
$ .   /catkin_ws/devel/setup.bash
```

#2.1.8. Download from Moodle the file `mybot_teleop.py`, and place it inside a `scripts` folder inside the `mybot_teleop` package. On the currently opened terminal, go to the `scripts` directory and make the Python script executable with the following commands:

```
$ cd ~/catkin_ws/src/mybot_teleop/scripts
$ chmod +x mybot_teleop.py
```

#2.1.9. Open the Python script `mybot_teleop.py` in your favorite editor to visualize the code. You are now ready to write your first ROS node!

2.1.3 Creation of the `mybot_teleop` node

In this subsection, we will write a ROS node `/mybot_teleop` which will talk to the `/turtlesim` node through the `/turtle1/cmd_vel` topic. Basically, `/mybot_teleop` is thus a very simple publisher node, similar to the one you already worked on during the ROS tutorial in §1.

#2.1.10. Writing a simple publisher node requires the following steps. For each of them, you have to find the correct syntax in Python, that you will have to report in the `mybot_teleop.py` script. Please keep in parallel open the ROS tutorial webpage dedicated to the "*writing (of) a simple publisher and subscriber (Python)*".

- import the data type expected for the topic `/turtle1/cmd_vel`;

Solution:
```
from geometry_msgs.msg import Twist
```

- define a publisher on the `/turtle1/cmd_vel` topic with the imported data type on the step before;

Solution:
```
pub = rospy.Publisher('turtle1/cmd_vel', Twist, queue_size=10)
```

- initialize a new node with name `mybot_teleop`;

Solution:
```
rospy.init_node('mybot_teleop', anonymous=False)
```

- define a rate at which you would like the data to be published;

Solution:
```
r = rospy.Rate(10) #10Hz
```

- as long as the ROS node is not shut down, and according to the key pressed on the keyboard, build an appropriate message and publish it on `/turtle1/cmd_vel` to make the turtle move.

Solution:
The entire code `mybot_teleop_corr1.py` is available for download from the Moodle page of the teaching unit. Access is restricted to Teachers profiles. Two solutions are possible and correct:

- the message is only sent each time a key is pressed (this is the proposed solution). In this case, the turtle will move for 1s and then stops (this delay is hardcoded in the turtle_sim node), waiting next for a new velocity command;
- or, the message is constantly sent at a given time rate, making the turtle always moving except if a given key is pressed and then sends a null velocity command.

Milestone

Call your teacher at the end of this subsection to demonstrate the use of your teleoperation node. You will also be questioned on the basic shell commands exploited above to investigate the ROS architecture.

2.2 Part B: using services

In this subsection, we will write a second ROS node which will possibly run in parallel to the previous teleoperation node. This time, this new node will subscribe to the `/turtle1/pose` topic (which returns the position of the turtle on the screen) and modify the color of the line behind it thanks to a call to the ROS service `/turtle1/set_pen` made available by the `turtlesim_node`. Of course, you will have to run first the `turtlesim_node` if it is not already running.

#2.2.1. Like before, create a new package `mybot_color` with the same dependencies as before. Edit the file `package.xml`, compile the new package, and source again the bash configuration in the `devel` folder of your `catkin` workspace.

Solution:

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg mybot_color std_msgs rospy
```
Edit `package.xml` to add the student name
```
$ cd ~/catkin_ws
$ cd catkin_make
$ .   /catkin_ws/devel/setup.bash
```

#2.2.2. Create a `scripts` folder inside the new package, download the file `mybot_color.py` from Moodle and place it inside the `scripts` folder. Do not forget to make the downloaded script executable! (see #2.1.8)

#2.2.3. List again of the available topics and study the expected message that should be received on `/turtle1/pose`.

Solution:

```
$ rostopic list
$ rostopic info /turtle1/pose
$ rosmsg info turtlesim/Pose
```
It appears the message is made of 5 `float32` values dedicated to `x`, `y`, `theta` (position and orientation of the turtle on the screen) and `linear_veolicity` and `angular_velocity` (current linear and angular velocity of the turtle).

#2.2.4. Writing a simple subscriber node requires the following steps. For each of them, you have to find the correct syntax in Python, that you will have to report in the `mybot_color.py` script. Please keep in parallel open the ROS tutorial webpage dedicated to the "*writing (of) a simple publisher and subscriber (Python)*".

- initialise a new ROS node with name `mybot_color`;

Solution:

```
rospy.init_node('mybot_color', anonymous=False,log_level=rospy.INFO, disable_signals=False)
```

All of the method options are not mandatory, please see the documentation.

- import the data type expected for the topic `/turtle1/pose`;

Solution:

```
from turtlesim.msg import Pose
```

- define a subscriber on the `/turtle1/pose` topic with the imported data type on the step before together with a callback function which will be run each time a new data is present on the subscribed topic;

Solution:

```
sub = rospy.Subscriber('/turtle1/pose', Pose, read_pose_callback)
```

- and write the callback function for whatever you would like to do with the available data. For now, just print in the Terminal the `x` and `y` position of the turtle.

Solution:

```
def read_pose_callback(msg):
    rospy.loginfo("Current position:  x=%.2f; y=%.2f" % (msg.x, msg.y))
```

The turtle position is here directly printed on the terminal where the node is running (thanks to `rospy.loginfo`), but also sent to the ROS master. Another solution is to use directly the `print` command.
The entire code is available on Moodle (see `mybot_color_corr1.py` script).

Your are now able to get the `x` and `y` position of the turtle thanks to the continuous reading of the `/turtle1/pose` topic. We want now to change the color of the line following the turtle as a function of this position. As an example, we want to change the line to red when approaching the edge of the screen. To do that, the `turtlesim_node` exposes a ROS service named `turtle1/set_pen`.

#2.2.5. First, list all the services exposed by the `/turtlesim` node, and display information about the service `turtle1/set_pen`.

Solution:

```
$ rosnode info /turtlesim
```

and then

```
$ rosservice info /turtle1/set_pen
```

#2.2.6. Then display what data type is expected to be sent to the service `turtle1/set_pen` and what kind of data will be returned.

Solution:

```
$ rossrv info turtlesim/SetPen
```

The service uses as in input a set of `uint8` data `r`, `g`, `b`, `width` and `off`. This service does not return anything (as no data type are mentioned after the three dashes in the above command output.)

#2.2.7. Writing a simple service client in a ROS node requires the following steps. For each of them, you have to find the correct syntax in Python, that you will have to add to the `mybot_color.py` script. Please keep in parallel open the ROS tutorial webpage dedicated to the "*writing (of) a Simple Service and Client (Python)*".

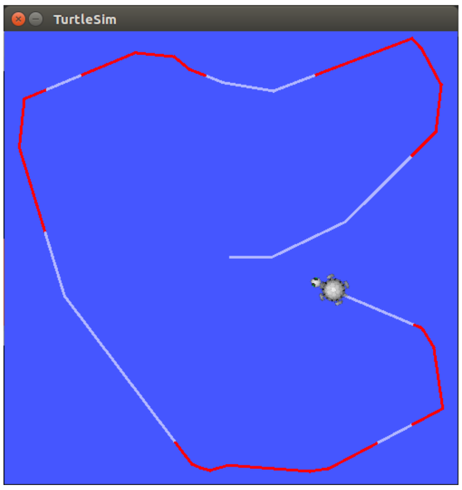

Figure 2: A capture of the turtlesim view, where the line turns red when approaching the screen corners.

- first, import the data type expected to be sent to call the `turtle1/set_pen` service.

Solution:
```
from turtlesim.srv import SetPen
```

- then use the `wait_for_service` method from `rospy` to wait for the service to be ready, and create a handle for calling the `/turtle1/set_pen` service.

Solution:
```
rospy.wait_for_service('/turtle1/set_pen')
change_pen_color = rospy.ServiceProxy('/turtle1/set_pen', SetPen)
```
The name of the handle (`change_pen_color`) can be freely chosen.

- and finally, use your handle wherever you need in the script, with the correct arguments, to call the service. For the record, the default pen values used in the turtlesim node are `r`=179, `g`=184, `b`=255, `width`=3, `off`=False. In this exercise, please try to make a minimal number of calls to the service (i.e. only when entering or leaving the corners of the screen).

Solution:
```
change_pen_color(255, 0, 0, 3, False) #RED color
```
The call must be done as a function of the turtle position, so this call must be added in the reading position callback function.
The entire code is availble on Moode (see `mybot_color_corr2.py`).

At the end, you should have 4 terminals respectively occupied by `roscore`, the `turtlesim_node`, the `mybot_teleop` node and the `mybot_color` node. A capture of one example of the expected outcome is shown in Figure 2.

#2.2.8. Please validate your architecture with a `rqt_graph` shown to your teacher, as well as a figure analog to Figure 2.

Milestone

Call your teacher at the end of this subsection to demonstrate the use of your `mybot_color` node. You will also be questioned on the basic shell commands exploited above to investigate the ROS architecture and your knowledge on ROS services.

2.3 Part C: launchfile and parameters to teleoperate the Turtlebot 3 Burger

At the end of the previous part, you had to launch *multiple nodes* by hand, each of them in their own terminal. Launch files allow you to specify in one place all the nodes you would like to run together. Additionnaly, this is a convenient place to write the values of some parameters of your nodes, i.e. values in you code that can be changed at run time.

We will use launch files in the following to make the `mybot_teleop` node generic, so that you will be able to teleoperate a real robot with the very same ROS node than the one teleoperating the turtlesim. Of course, the speed commands between the real robot and the turtlesim may differ: this is where we will exploit parameters to change the value of the command accordingly.

2.3.1 Using launch files to launch multiple nodes

A launch file is a specific ROS file written using the XML format and specifying which nodes must be run, with which arguments or parameters, eventually on which distant machine, etc. It is a very powerful way to deal with very large scale ROS architectures, including a lot of ROS nodes with their own parameters.

#2.3.1. First, download from Moodle the minimal launch file `turtlesim.launch` (see Listing 3 on page 14), and place it inside a new `launch` folder inside the `mybot_teleop` package. Open it with your favorite code editor, and comment its content (see `http://wiki.ros.org/roslaunch/XML` for understanding the main options for the `<node>` XML tag).

Solution:

```
<node pkg="mybot_teleop" name="mybot_teleop" type="mybot_teleop.py"
     output="screen" required="true">
```

- `pkg`: points to the package associated with the node that is to be launched;
- `name`: overwrite the name of the node with the name argument, this will take priority over the name that is given to the node in the code;
- `type`: refers to the name of the node executable file;
- `output`: stdout/stderr from the node will be sent to the screen;
- `required`: if the node is shut down or dies, kill entire roslaunch.

#2.3.2. Run *by hand* the `turtlesim_node` in one terminal, and then use the previous launch file to run with `roslaunch` the `mybot_teleop` node in a second terminal. You should able to teleoperate the turtlesim like in §2.1.

Solution:

```
$ rosrun turtlesim turtlesim_node
$ roslaunch mybot_teleop turtlesim.launch
```

#2.3.3. Modify the launch file to automatically run the `turtlesim_node` *and* the `mybot_teleop` node. Now, you can run both nodes with only one `roslaunch` command.

Solution:

It is sufficient to add:

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

Then the following command launches both nodes:

```
$ roslaunch mybot_teleop turtlesim.launch
```

2.3.2 Using launchfiles to set parameters

In the previous `mybot_teleop` node, nothing can be tuned to adapt the node to the teleoperation of anything different from the original turtlesim. For instance, if one wants to make tunable the speed command sent to the robot when pressing a key on the keyboard, it is necessary to modify the node code itself. Even if in this case this should not be a problem, we will benefit from the possibility to define parameters in the launch file to make the `mybot_teleop` node a bit more generic.

A parameter server is launched together with the ROS master when running the `roscore` command. It stores all the parameters of the currently running nodes, and a change of a parameter value is made by sending a request to this parameter server.

#2.3.4. First, after running the `turtlesim_node` and the `mybot_teleop` node, list all available parameters.

Solution:

```
$ rosparam list
```

#2.3.5. Next, thanks to the `<param>` tag, add two parameters with respective name `linear_scale` and `angular_scale` and values `1.0` to your `turtlesim.launch` file (see `http://wiki.ros.org/roslaunch/XML/param` for the exact syntax).

Solution:

It is sufficient to add the two following lines to the root of the XML file (under the `<launch>` tag):

```
<param name="linear_scale" value="1.0" />
<param name="angular_scale" value="1.0" />
```

#2.3.6. After launching again the launch file, check if the two parameters are now correctly recorded in the parameters server.

Solution:

```
$ rosparam list
```

The two parameters are now listed.

The question is now: how can you get these two parameters values in your node code? The client API `rospy` proposes one method `get_param()` to get a parameter value from its name.

#2.3.7. Adapt the script `mybot_teleop.py` to get the parameters value from the parameters server, and check if everything is working as before (it should be the case!). You can then change the parameters values to higher or lower values and verify the effect on the turtle movement on the screen.

Solution:

```
# Get the linear_scale parameter
if rospy.has_param('linear_scale'):
    linear_scale = rospy.get_param('linear_scale')
else:
    # If not set, the default value is set to 2.0
    linear_scale = 2.0
```

Checking first if the parameter exists in the parameters server is considered good practice. Next, the students have to think about setting a default value (see above). Of course, the python variable `linear_scale` has to be used later in the code to send the actual speed command, for instance :

```
message.linear.x = linear_scale * linear
pub.publish(message)
```

The script can be download on Moodle (see `mybot_teleop_corr2.py`).

Milestone

Call your teacher at the end of this subsection to demonstrate the use of your launch file. You will also be questioned on how you handled parameters in your ROS node.

2.3.3 Teleoperating the (simulated) Turtlebot3 burger

We will now use the `mybot_teleop` node to teleoperate a simulated Turtlebot 3 burger robot. Working with a simulated realistic robot (or even a real robot!) is not different from listening and subscribing to topics, using services, etc. like before, except that the node you are working with is actually driving a simulated robot inside a physical simulator. The simulator we will be using in all the following is `gazebo`. `gazebo` provides a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios. Interestingly, `gazebo` works well with ROS and can be launched and controlled directly from it. You will learn how to exploit `gazebo` together with a robot (physical) description and sensors in the next chapter. In the meantime, we will use it here only as a convenient visualization tool allowing you to simply see the robot movement in an empty world.

#2.3.8. To begin, launch the `turtlebot3_empty_world` launch file from the `turtlebot3_gazebo` package. `gazebo` should now be running (the first launch might be long), and a simulated TurtleBot 3 Burger should appear at the center of the simulated (empty) world.

Solution:

```
$ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

#2.3.9. List all currently running nodes and available topics. What is the name of the topic on which must be send the velocity command to make the robot move?

Solution:

```
$rosnode list
$rostopic list
```

The command must be sent to the topic `/cmd_vel`.

#2.3.10. From the previous step, one can see that the topic name used to send the velocity command is now different from the one used to command the turtlesim. Modify your code to account for this change by using an additional parameter set in a new launchfile `turtlebot.launch` you have to write (get inspiration from the previous one!). Then change the teleoperation node code to exploit this new parameter.

Solution:

Inside the node, you can add:

```
# First, get the name of the topic to publish to
if rospy.has_param('cmd_vel_topic_name'):
    topic_name = rospy.get_param('cmd_vel_topic_name')
else:
    # If not set, the default topic is cmd_vel
    topic_name = 'cmd_vel'
print("Command sent to " + topic_name)
```

#2.3.11. Set the two parameters `linear_scale` and `angular_scale` to the appropriate values (to be determined!) in the `turtlebot.launch` launch file, and run the `mybot_teleop` node. You should now be able to teleoperate the turtlebot!

#2.3.12. Finally, inspired by the `turtlebot3_empty_world.launch` file from the `turtlebot3_gazebo` package, write your own `turtlebot3_empty_world.launch` file in charge of launching both `gazebo` *and* your teleoperation node.

Solution:

```
<launch>

  <!-- Parameters -->
  <param name="cmd_vel_topic_name" value="cmd_vel" />
  <param name="linear_scale" value="0.05" />
  <param name="angular_scale" value="0.4" />

  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle, waffle_pi]"/>
  <arg name="x_pos" default="0.0"/>
  <arg name="y_pos" default="0.0"/>
  <arg name="z_pos" default="0.0"/>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find turtlebot3_gazebo)/worlds/empty.world"/>
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="gui" value="true"/>
    <arg name="headless" value="false"/>
    <arg name="debug" value="false"/>
  </include>

  <param name="robot_description" command="$(find xacro)/xacro --inorder $(find turtlebot3_description)/urdf/turtlebot3_$(arg mod
    <node pkg="gazebo_ros" type="spawn_model" name="spawn_urdf" args="-urdf -model turtlebot3_$(arg model) -
x $(arg x_pos) -y $(arg y_pos) -z $(arg z_pos) -param robot_description" />

  <!-- Run one mybot_teleop node -->
  <node pkg="mybot_teleop" name="teleop" type="mybot_teleop_corr2.py"
        output="screen"/>

</launch>
```

Milestone

Call your teacher at the end of this subsection to demonstrate the teleoperation of the simulated Turtlebot 3 Burger.

2.3.4 Teleoperating the (real) Turtlebot3 burger

#2.3.13. Read carefully the “How-to operate the Turtlebot 3 burger” document on Moodle, giving you all the steps required to setup and connect to the real robot. Ask your teacher a robot and try now to teleoperate it from the exact same `mybot_teleop` node . . . but you might again have to adapt the parameters to the robot (and this is actually all the point)!

Scripts to be completed

Listing 1: mybot_teleop.py script

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import rospy
import sys, termios, tty
import click

# Arrow keys codes
keys = {'\x1b[A':'up', '\x1b[B':'down', '\x1b[C':'right', '\x1b[D':'left', 's':'stop', 'q':'quit'}

if __name__ == '__main__':

    try:
        # Get character from console
        mykey = click.getchar()
        if mykey in keys.keys():
            char=keys[mykey]

        if char == 'up':    # UP key
            # Do something
        if char == 'down':  # DOWN key
            # Do something
        if char == 'left':  # RIGHT key
            # Do something
        if char == 'right': # LEFT
            # Do something
        if char == "quit":  # QUIT
            # Do something

    except rospy.ROSInterruptException:
        pass
```

Listing 2: mybot_color.py script

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import rospy

# Callback function for reading turtlesim node output
def read_pose_callback(msg):

    # First step: display read position on the terminal
    # TO BE COMPLETED


if __name__ == '__main__':
    try:

        # CODE TO BE COMPLETED

        # And then ... wait for the node to be terminated
        rospy.spin()

    except rospy.ROSInterruptException:
        pass
```

Listing 3: minimal turtlesim.launch script

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>

  <!-- Run one mybot_teleop node -->
  <node pkg="mybot_teleop" name="teleop" type="mybot_teleop.py"
        output="screen" required="true">
  </node>

</launch>
```