Excercise - Lecture 4

May 2021

Course: Introduction to Robot Operating System (ROS)

Lecturer:
Jon Arrizabalaga (jon.arrizabalaga@tum.de)
Course responsible:
Prof. Markus Ryll

This document proposes two exercises to assist students in understanding the content presented in Lecture 4 of "Introduction to ROS". If completed, it is guaranteed that the students have acquired the necessary knowledge for following the upcoming lectures. Since this exercice is self-evaluated and will not be graded, the lecture slides, excercise answers, and a template with gaps to be filled, have been uploaded. Seeking a high learning outcome, students should try to complete the exercises without looking into the provided answers. In contrast to the exercises of last week, students are encouraged to use the templates. In case of difficulties, either the information in the wiki / forums or other students might be of great help. In other words, for your best interest, avoid looking into the solutions. Other than that, if students feel that the mentioned resources are not enough and wish to gain a more in-depth understanding, please do not hesitate to contact me by sending an email to jon.arrizabalaga@tum.de .

Notice1: Due to the computational cost of running the video and recording the screen simultaneously, the simulation in the video is slow (Real-Time factor 0.4). Do not be surprised if your robot is faster!

1 Environment description

1.1 R2D2 robot

The robot we will use in this excercise is R2D2, a beloved droid that has conquered the heart of Star Wars fans. Before starting, clone the original repo from `https://github.com/ros/urdf_sim_tutorial.git` to a ROS workspace and launch the robot by

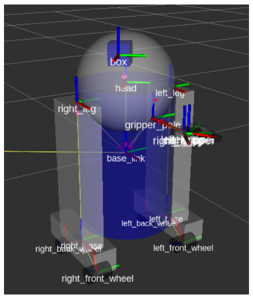
```
$ cd
$ mkdir robot_ws
$ cd robot_ws
$ mkdir src
$ catkin build
$ source devel/setup.bash
$ roslaunch urdf_sim_tutorial 13-diffdrive.launch
```

Do not get surprised with the long output printed when running the command. After waiting a little (Gazebo requires some time to start, depends on your machine), three different windows should come out: RViz, Gazebo and a small GUI. See Figure 1.

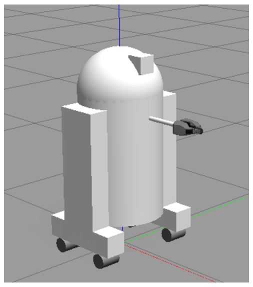
If you are running Ubuntu in VMWare, there are problems related to 3D graphics which will crash the Gazebo. To account for this, run `$ export LIBGL_ALWAYS_SOFTWARE=1`[1] . In order to avoid running it in every new command window, you can add it to the end of the bash file with

```
$ echo "export LIBGL_ALWAYS_SOFTWARE=1" >> ~/.bashrc && source ~/.bashrc
```

[1] Original thread is `https://github.com/uuvsimulator/uuv_simulator/issues/59`



(a) RViz window


(b) Gazebo window

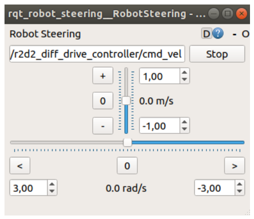

rqt_robot_steering__RobotSteering - ...
Robot Steering
/r2d2_diff_drive_controller/cmd_vel
Stop
+
1,00
0
0.0 m/s
-
-1,00
<
0
>
3,00
0.0 rad/s
-3,00
(c) Gui Window

Figure 1: Windows opened when running the command given above.

During the lecture we have already explained the difference between RViz and Gazebo. RViz is just a visualization tool, while Gazebo is a simulation environment, which includes physics, such as gravity, collisions, etc. The GUI is a remote control that allows us to control the differential drive, by sending longitudinal and angular velocity commands. Before continuing, play around with the GUI, the models in Gazebo and RViz should moving according to this commands.

In order to understand the robot's layout better, lets look into the rqt graph, and tf tree. For the first run `$ rqt_graph` and select Nodes/Topics (active) at the top left of the window, while for the second open the pdf generated by running `$ rosrun tf view_frames`.

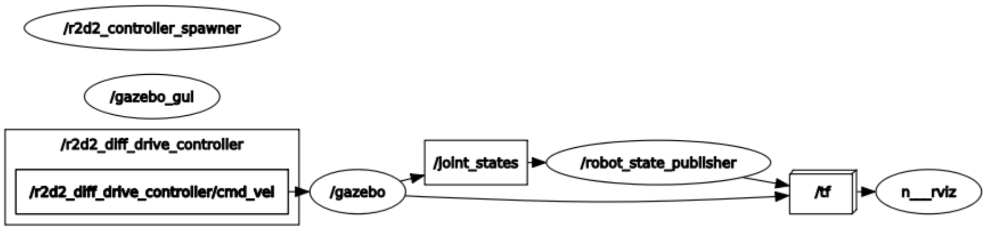


Figure 2: rqt graph.

By looking into figure 2 we can understand what happens when sending commands from the GUI. If we run `$ rosnode list`, the GUI can be identified with node `rqt_qui_steerging`. This node publishes velocity commands to topic `/r2d2_diff_drive_controller/cmd_vel`. Since Gazebo is a node that is subscribed to this topic, it receives these commands. When this happens, it simulates the physics and calculates the resultant position of the joints. As mentioned in the lecture, the TF poses respective to the joint states are directly published by node `/robot_state_publisher`. RViz is a node that subscribes to all the information.

In other words, Gazebo is a virtual environment that replaces not only the hardware, but also the world. In the background, Gazebo runs a physics engine which is defined by a shared library (.so) located at the xacro files that describe the robot. This file (.so) is generated when building the workspace and, if needed, can be modified. For the time being, we can assume that it is a black box with velocity commands as inputs and the position of all the joints as output.

Apart from the wheels, R2D2 can also move its head and gripper. For spinning the head, run

```
$ rostopic pub /r2d2_head_controller/command std_msgs/Float64 "data: -0.707"
```

For the gripper to be opened and out, run

```
$ rostopic pub  /r2d2_gripper_controller/command std_msgs/Float64MultiArray "layout:
dim:
- label: ''
size: 3
stride: 1
data_offset: 0
data: [0, 0.5, 0.5]"
```

In order to close and retract it, run the same command as above with `data: [-0.4, 0, 0]`. See Figure 3 to see how it should look like.

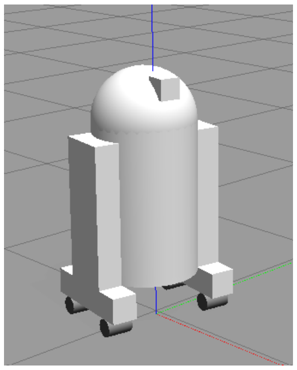


Figure 3: Closed and retracted gripper. Compare to Figure 1b.

2 Desription of excercise

You will be writing the algorithm that will send R2D2 to pick up a lost object and bring it back to its initial location. Ready for the challenge?

The mission is divided into four steps:

1. Place the object in a desired location.
2. Send R2D2 to pick up the object.
3. Pick up the object.
4. Bring the robot and the object back to its initial position.

Notice: Watching the video might help to understand the four steps mentioned above!

2.1 Structure of the workspace

From the provided files, clone the workspace `/templates/exercise_r2d2_ws`. The workspace has already been prepared, so that you do not have to spend time with the CMakeLists.txt and package.xml files. The workspace has two packages:

1. **urdf_sim_tutorial:** This package is cloned from a tutorial called "Using a URDF in Gazebo"[1].It contains the description of the robot (URDF, xacros) and the required files (joint controllers, physics, etc.) to fully simulate the robot in Gazebo.
2. **mission_r2d2** This is where we are going to develop our algorithm. It contains two nodes, `lost_object.cpp`, referring to the lost object that the robot has to rescue, and `mission_r2d2.cpp`, containing the control algorithm for the desired actions of the robot.

Notice: Solutions can be found in another workspace `/solutions/exercise_r2d2_ws`. Running the commands in this workspace should make the robot move in the same way as in the video.

[1]Tutorial available in `http://wiki.ros.org/urdf/Tutorials/Using%20a%20URDF%20in%20Gazebo`

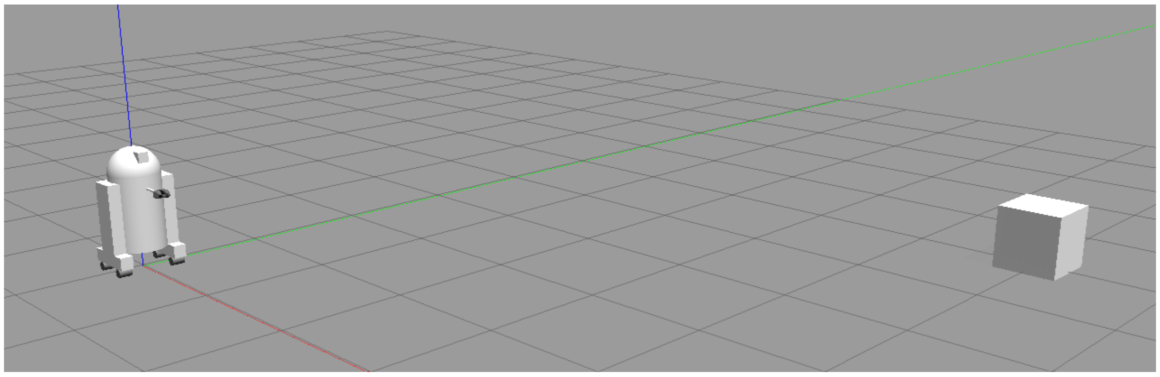


Figure 4: Default layout of the world. The lost object, represented by a box, is placed in (x=5, y=5).

2.2 Format of templates

For the sake of simplicity, the entire workspace has already been prepared to ensure that you focus on filling the gaps of the files that run the nodes: `lost_object.cpp` and `mission_r2d2.cpp`. Following the exercise of last week, both scripts have been written in an object-oriented manner, where the node is wrapped in a class, enhancing accessibility to variables. In fact, both classes contain a ROS timer [2] as a member variable, which is an infinite loop that ensures the lines of code within it run at a given frequency. In other words, the actions of the node will be defined according to the lines we write inside the loop.

Notice: Before starting to fill the gaps, make sure that you understand which is the workflow of each script, i.e, you must identify the order in which every function is called.

3 Excercise 1 - Node for lost object

Prior to sending the robot to find the lost object, it must be positioned in a certain location. Therefore, in this exercise, you will write a node that assigns a TF coordinate to the object and moves it in Gazebo.

A launch file, that opens R2D2 alongside the lost obstacle, represented by a box, has already been created. You can run this with `$roslaunch mission_r2d2 r2d2_and_lostObject.launch`. As shown in Figure 4, by default the lost object is located in position (x=5,y=5). On top of that, if you check the TF tree, you will notice that it does not have a tf assigned to it. Therefore, you will complete the file `lost_object.cpp`. This node has the following characteristics:

- A **TF broadcaster** that broadcasts the location of the lost object.
- A **service** that updates the location of the box in the gazebo world. The service is of type `gazebo_msgs::SetModelState`, the respective topic being `"/gazebo/set_model_state"`. Remember that services are similar to publishers and subscribers, but, instead of sharing data repetitively at a given frequency, they do it once [2].
- The object will stay still in the coordinates given by the user. However, once the robot reaches the location and picks it up, the object will move with the robot. To account for this, the node **subscribes** to topic `/r2d2/pickup_alert` with message type `std_msgs::Boolean`.

Notice: Frame /odom is considerd to be fixed, while /based_link is attached to the robot. You will have to take this into account when working with TF and using RViz.

Fill in the gaps of the template `lost_object.cpp`. You will identify the gaps by `...` , accompanied by some hints. Remember that you will have to build every time you do a modification. For testing, open two different command windows and run

```
$ roslaunch mission_r2d2 r2d2_and_lostObject.launch
$ rosrun mission_r2d2 lost_object _x0:=1 _y0:=10
```

[2]More information available in `http://wiki.ros.org/roscpp/Overview/Timers`

[2]More information available in: `http://wiki.ros.org/roscpp/Overview/Services`

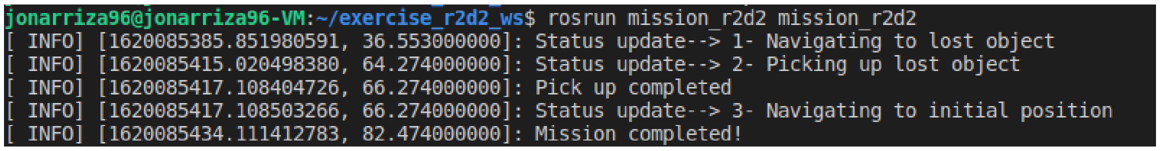

```
jonarriza96@jonarriza96-VM:~/exercise_r2d2_ws$ rosrun mission_r2d2 mission_r2d2
[ INFO] [1620085385.851980591, 36.553000000]: Status update--> 1- Navigating to lost object
[ INFO] [1620085415.020498380, 64.274000000]: Status update--> 2- Picking up lost object
[ INFO] [1620085417.108404726, 66.274000000]: Pick up completed
[ INFO] [1620085417.108503266, 66.274000000]: Status update--> 3- Navigating to initial position
[ INFO] [1620085434.111412783, 82.474000000]: Mission completed!
```

Figure 5: Output of command window when fulfilling mission sucessfully.

The first command launches the robot and the box, as shown in Figure 4, while the second places the object in the desired location, for example 1 and 10 in the given command. When launching the second command, if the TF in RViz (1) and the box in Gazebo (2) move to the specified location, you have succeeded! Notice that you will not be able to test the subscriber until you complete the second exercise.

4 Excercise 2 - Node for R2D2

File `mission_r2d2.cpp` contains a very similar structure to the previous one. However, its goal is to guide the robot to accomplish the mission. For this purpose, the actions have been grouped in three stages:

1. **Navigate to lost object:** The robot locates the object and drives towards it. For this purpose, it starts rotating until its orientation is aligned with respect to the object, and then, it drives forward. As soon as it is closer to 1m of tolerance, it goes into the next stage.
2. **Pick up object:** It closes and retracts the gripper, as shown in Figure 3. As soon as this happens, it publishes a message that confirms this action.
3. **Return with lost object:** By applying a negative velocity command, the robot returns with the object attached to it. In a similar manner to stage 1, when the robot is closer than 1m from the starting location, the mission is considered to be a success.

To fulfill these actions, it contains

- A **publisher** that publishes velocity commands to the robot's drive train under the topic `/r2d2_diff_drive_controller/cmd_vel` and message type `geometry_msgs::Twist`.
- A **publisher** that publishes motion commands to the gripper under the topic `/r2d2_gripper_controller/command` and message type `std_msgs::Float64MultiArray`.
- A **publisher** that publishes a boolean message `std_msgs::Boolean` under the topic `/r2d2_gripper_controller/command`, when the robot finds and picks up the object.

Notice: The calculation of the velocity commands are a coarse approximation: The robot rotates until its orientation is close to the obstacle's. Once this occurs, it only drives forward. This accounts for the lack of accuracy if the object is far away. If interested, feel free to improve it!

Fill in the gaps of `mission_r2d2.cpp`. You can test the entire setup by running these commands in three separate windows:

```
$ roslaunch mission_r2d2 r2d2_and_lostObject.launch
$ rosrun mission_r2d2 lost_object _x0:=5 _y0:=5
$ rosrun mission_r2d2 mission_r2d2
```

If the robot behaves in a similar manner to the video and your command window looks like Fig. 5, you have succeeded!

Notice: You can place the robot at any location. However, the approximated velocity controller cannot guarantee that the robot will find the object if the location is very far away or a significant rotation is required. To be on the safe side, while testing we recommend to place the object in (x=5, y=5). Once it works for these coordinates, you can place it in other locations and see what happens.