Probabilistic Robotics

Albert Clerigues Garcia
Hassan Zaal
Di Meng
Abdelrahman Abubakr

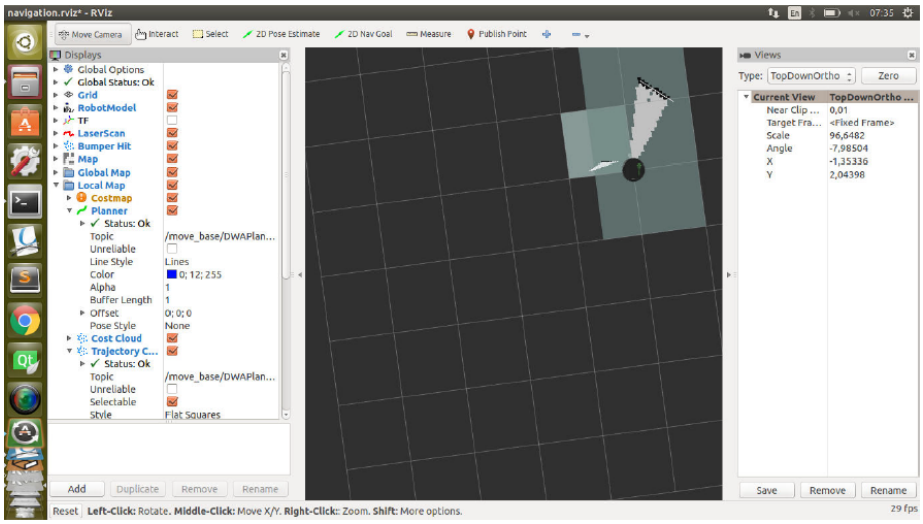
March 2017

1 Configuration

We set up ROS MASTER URI and ROS HOSTNAME, then we checked if the connection is properly done. We faced many problems in this step to connect to turtlebot, but we finally solved them. We launched the graphical interface to check the robot's status, and started the Kinect on the robot but it crashed.

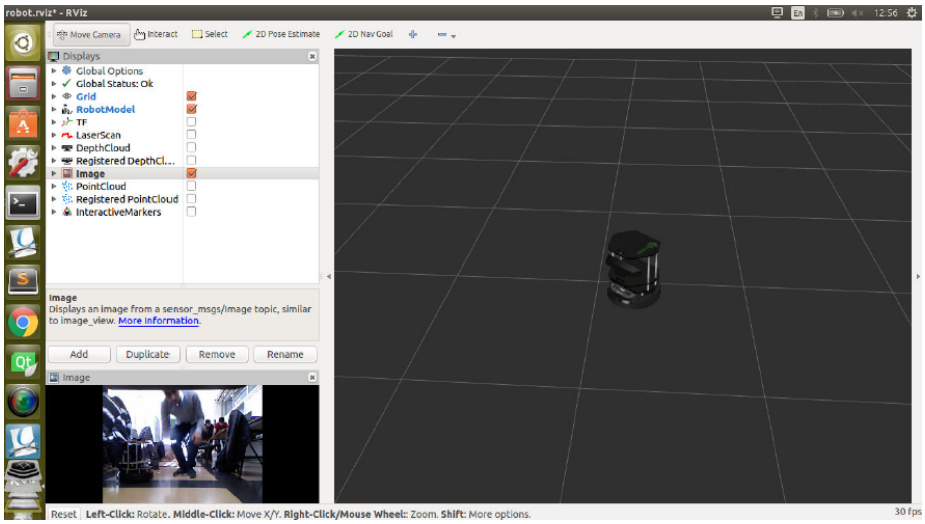
2 Navigation and mapping

We ran keyboard teleoperation and controlled the robot by keyboard commands. We drove the robot around to build a small map and visualized the map and the robot's position. Next figure shows this map.



3 Follower

We ran follower package and moved slowly in front of the robot. We wanted to run the panorama package, but it crashed. However, we managed to ran the camera as shown in next figure.



4 Turtlebot driver

First we find the topics to subscribe to obtain the position at *'/gazebo/model_states'* which uses ModelStates message type. Then we find the topic to publish and send the velocity commands *'/mobile_base/commands/'*.

We add the correspondent imports for the message object types and create the appropriate subscriber and publisher instances. We then modified the function read_position to read and process the ModelStates object messages. The ModelStates object has the following structure:

```
1
2  string[ ] name
3  geometry\_msgs/Pose[ ] pose
4  geometry\_msgs/Point position
5  float64 x
6  float64 y
7  float64 z
8
9  geometry\_msgs/Quaternion orientation
10 float64 x
11 float64 y
12 float64 z
```

```
float64 w
geometry_msgs/Twist[] twist
geometry_msgs/Vector3 linear
float64 x
float64 y
float64 z

geometry_msgs/Vector3 angular
float64 x
float64 y
float64 z
```

Where the information of a particular object is referenced by its index. The turtlebot is referred to as "*mobile_base*" inside the gazebo simulator, so we look for its index in the 'name' array to be able to obtain its pose.

Once we have the Pose object of the turtlebot, we extract the position.x and position.y. We also get the quaternion that defines its rotation, which we convert to Euler angles using the function Euler_from_quaternion provided by tf.transformations, obtaining the rotation around the z-axis. The function compute_velocity was also implemented using proportional control to avoid as maximum as possible overshooting and sloppy movement. If far from the goal we assign a clipped linear speed proportional to the remaining distance and an angular velocity inverse of the distance in radians to face the goal. Finally, if we are over the goal we rotate to face the desired angle also using proportional control to the remaining radians.

5 multi-goal [optional part]

Finally we implemented the multi-goal optional part. We edited the launch file to include a new parameter "*goals_file*" that we then open from the function load_goals, appending the x, y, theta values of each row to the goals array. The function next_goal() has been modified to increase the variable self.active_goal and checking whether or not the final goal has been reached, finishing the execution if necessary.