Lecture 7 (19/10) - Autonomous Guided Vehicles

Starts 19 October, 2020 1:00 PM

Autonomous Guided Vehicles

In this lecture, we will learn about how Autonomous Guided Vehicles (AGV) —like the one in your final project— work.

We meet on Zoom at 13:00. Link: https://dtudk.zoom.us/j/63569471741?pwd=emNGMEQxNitXdGFhc3hSZXhqdVFNQT09

Reading Material

Programming Robots with ROS: A Practical Introduction to the Robot Operating System (link)
Quigley, Morgan · Gerkey, Brian · Smart, William D.
Chapters: 7,8.9,10

Ros navigation tuning guide (link)
Zheng, Kaiyu. 2017

Time Plan

- 13:00 Lecture:
  - Types of Locomotion
  - Differential Drive Kinematics – (One Pager)
  - AGV System as a whole!
  - Localization & Mapping
  - Navigation – Path Planning
- 1500 - Exercises

0 % 0 of 1 topics complete

31391-sffas-4-Autonomous-Guided-Vehicles-separate

PDF document

Setting Up Your System

Requirements:

You should execute the following commands before coming to the lecture:

```
sudo apt-get update
sudo apt-get install ros-melodic-turtlebot3*
```

```
sudo apt-get install ros-melodic-map-server
sudo apt-get install ros-melodic-teleop-twist-keyboard
sudo apt-get install ros-melodic-gmapping
sudo apt-get install ros-melodic-dwa-local-planner
```

```
echo "export TURTLEBOT3_MODEL=burger" >> ~/.bashrc
```

```
rospack profile
```

You may have to restart the terminal afterwards in order for everything to work.

Exercise 7.1: Simple Commanding of a mobile robot

NOTE:

We define notes with a red vertical line,

terminal commands with a black one,

and code with a blue one

We would like to command our robot to move form python (similarly to what we did with turtlesim)

```
roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

Let's make a file named **red_light_green_light**.py and save it to **~/catkin_ws/src/hello_ros/scripts/red_light_green_light.py:**

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist

cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
rospy.init_node('red_light_green_light')

red_light_twist = Twist()
green_light_twist = Twist()
green_light_twist.linear.x = 0.5

driving_forward = False
light_change_time = rospy.Time.now()
rate = rospy.Rate(10)

while not rospy.is_shutdown():
  if driving_forward:
    cmd_vel_pub.publish(green_light_twist)
  else:
    cmd_vel_pub.publish(red_light_twist)
  # BEGIN PART_1
  if rospy.Time.now() > light_change_time:
    driving_forward = not driving_forward
    light_change_time = rospy.Time.now() + rospy.Duration(3)
  # END PART_1
  rate.sleep()
# END ALL
```

Now make the file executable and run the node.

Mini exercise: The robot is bored while the light is red. Let it turn while waiting for the green light.

Exercise 7.2: Our first ever sensor, laser scanner

NOTE:

We define notes with a red vertical line,
terminal commands with a black one,
and code with a blue one

We would like to read out from the sensors of our robot from python. Let's look at the laser scanner.

Let's make a file named **turtlebot3_world.launch** and save it to **~/catkin_ws/src/hello_ros/launch/turtlebot3_world.launch:**

```
<launch>
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model ty
  <arg name="x_pos" default="-2.0"/>
  <arg name="y_pos" default="-0.5"/>
  <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/turtlel
    <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

  <node pkg="gazebo_ros" type="spawn_model" name="spawn_urdf"  args="-
```

```
  <node pkg="robot_state_publisher" type="robot_state_publisher" name="rc
    <param name="publish_frequency" type="double" value="30.0" />
  </node>

</launch>
```

Close the empty gazebo world and launch instead the new world with obstacles.

```
roslaunch hello_ros turtlebot3_world.launch
```

Let's see some raw robot data.

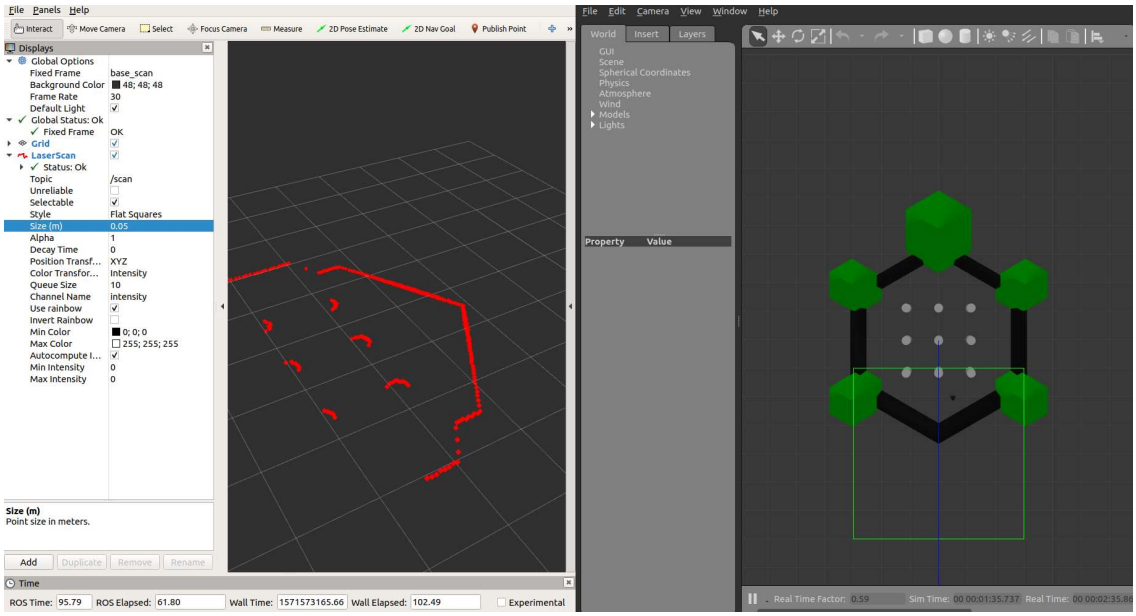
```
rostopic echo /scan
```

It's so much better to see these things through RVIZ.

```
rosrun rviz rviz
```

In RVIZ, add the topic LaserScan.

Remember to change the fixed frame to "base_scan".



Ok how to use this on python?

Let's make a file named range_ahead.py and save it to **~/catkin_ws/src/hello_ros/scripts/range_ahead.py:**

```
#!/usr/bin/env python
# BEGIN ALL
import rospy
from sensor_msgs.msg import LaserScan

# BEGIN MEASUREMENT
```

```
def scan_callback(msg):
  range_ahead = msg.ranges[0]
  tmp=[msg.ranges[0]]
  for i in range(1,21):
    tmp.append(msg.ranges[i])
  for i in range(len(msg.ranges)-21,len(msg.ranges)):
    tmp.append(msg.ranges[i])
  print "range ahead: %0.1f" % min(tmp)
  # END MEASUREMENT

rospy.init_node('range_ahead')
scan_sub = rospy.Subscriber('scan', LaserScan, scan_callback)
rospy.spin()
# END ALL
```

Make the python file executable and run the node. It prints the minimal distance ahead of the robot. Verify the correctness by moving the robot around with the keys i,k,l,j.

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

Exercise 7.3 Lets make our robot drive around by itself

NOTE:

We define notes with a red vertical line,
terminal commands with a black one,
and code with a blue one

We would like to make our robot do things in this world!!!

```
roslaunch hello_ros turtlebot3_world.launch
```

It's so much better to see these things through RVIZ.

```
rosrun rviz rviz
```

Let's make a file named wander.py and save it to

~/catkin_ws/src/hello_ros/scripts/wander.py:

```
#!/usr/bin/env python
# BEGIN ALL
import rospy
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan

def scan_callback(msg):
  global g_range_ahead
  tmp=[msg.ranges[0]]
  for i in range(1,21):
    tmp.append(msg.ranges[i])
  for i in range(len(msg.ranges)-21,len(msg.ranges)):
    tmp.append(msg.ranges[i])
  g_range_ahead = min(tmp)


g_range_ahead = 1 # anything to start
scan_sub = rospy.Subscriber('scan', LaserScan, scan_callback)
cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
rospy.init_node('wander')
state_change_time = rospy.Time.now() + rospy.Duration(1)
driving_forward = True
rate = rospy.Rate(60)

while not rospy.is_shutdown():
  print g_range_ahead
  if g_range_ahead < 0.8:
    # TURN
    driving_forward = False
    print "Turn"

  else: # we're not driving_forward
    driving_forward = True # we're done spinning, time to go forward!
    #DRIVE
    print "Drive"

  twist = Twist()
  if driving_forward:
```

```
    twist.linear.x = 0.4
    twist.angular.z = 0.0
  else:
    twist.linear.x = 0.0
    twist.angular.z = 0.4
  cmd_vel_pub.publish(twist)

  rate.sleep()
# END ALL
```

Make the file executable and run the node!

What happens? Can you reverse it? Let the robot drive "backwards" in the same manner.

Exercise 7.4 Teleoperating our robot using the keyboard

NOTE:

We define notes with a red vertical line,
terminal commands with a black one,
and code with a blue one

In this tutorial we will teleoperate our robot using our custom keyboard definitions!

We would like to make our robot do things in this world!!!

roslaunch hello_ros turtlebot_world.launch

It's so much better to see these things throught RVIZ

rosrun rviz rviz

1. Registering keyboard inputs

The first thing we want to do is to continuously input keyboard commands in python.

Let's make a file named key_publisher.py and save it to

~/catkin_ws/src/hello_ros/scripts/key_publisher.py:

```
#!/usr/bin/env python
# BEGIN ALL
import sys, select, tty, termios
import rospy
from std_msgs.msg import String

if __name__ == '__main__':
  key_pub = rospy.Publisher('keys', String, queue_size=1)
  rospy.init_node("keyboard_driver")
  rate = rospy.Rate(100)
  # BEGIN TERMIOS
  old_attr = termios.tcgetattr(sys.stdin)
  tty.setcbreak(sys.stdin.fileno())
  # END TERMIOS
  print "Publishing keystrokes. Press Ctrl-C to exit..."
  while not rospy.is_shutdown():
    # BEGIN SELECT
    if select.select([sys.stdin], [], [], 0)[0] == [sys.stdin]:
      key_pub.publish(sys.stdin.read(1))
    rate.sleep()
    # END SELECT
  # BEGIN TERMIOS_END
  termios.tcsetattr(sys.stdin, termios.TCSADRAIN, old_attr)
  # END TERMIOS_END
# END ALL
```

(**~/catkin_ws/src/hello_ros/**):

```
cd scripts
chmod +x key_publisher.py
```

Now we can also run the thing

```
rosrun hello_ros key_publisher.py
```

Open another terminal to echo it:

```
rostopic echo /keys
```

2. Converting keyboard inputs to velocity commands

The first thing we want to do is to continuously input keyboard commands in python..

Let's make a file named keys_to_twist.py and save it to

~/catkin_ws/src/hello_ros/scripts/keys_to_twist.py:

```
#!/usr/bin/env python
# BEGIN ALL
import rospy
from std_msgs.msg import String
from geometry_msgs.msg import Twist

# BEGIN KEYMAP
key_mapping = { 'w': [ 0, 0.2], 'x': [0, -0.2],
                'd': [-1, 0], 'a': [1,  0],
                's': [ 0, 0] }
# END KEYMAP

def keys_cb(msg, twist_pub):
  # BEGIN CB
  if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):
    return # unknown key.
  vels = key_mapping[msg.data[0]]
  # END CB
  t = Twist()
  t.angular.z = vels[0]
  t.linear.x  = vels[1]
  twist_pub.publish(t)

if __name__ == '__main__':
  rospy.init_node('keys_to_twist')
  twist_pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
  rospy.Subscriber('keys', String, keys_cb, twist_pub)
  rospy.spin()
# END ALL
```

(**~/catkin_ws/src/hello_ros/**):

```
cd scripts
chmod +x keys_to_twist.py
```

Let's run both nodes and control the robot using the defined keys.

```
rosrun hello_ros keys_to_twist.py
rosrun hello_ros key_publisher.py
```

Change the script in such a way, that backward movements are not possible.

Exercise 7.5: Mapping an area with your robot

NOTE:

We define notes with a red vertical line,
terminal commands with a black one,
and code with a blue one

There are two main components in navigation:

- Mapping an unknown area and,
- Navigating in a pre-mapped area

In this tutorial we will learn how to use our robot's sensors to map an area.

First thing is to run the simulator:

```
roslaunch hello_ros turtlebot3_world.launch
```

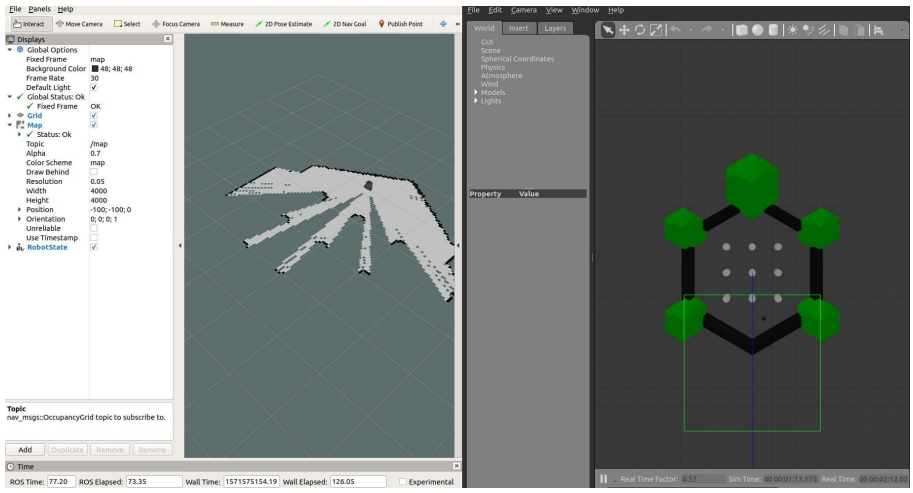
We will use rviz throughout this tutorial, so on another terminal:

```
rosrun rviz rviz
```

Add the "map" topic in RViz.

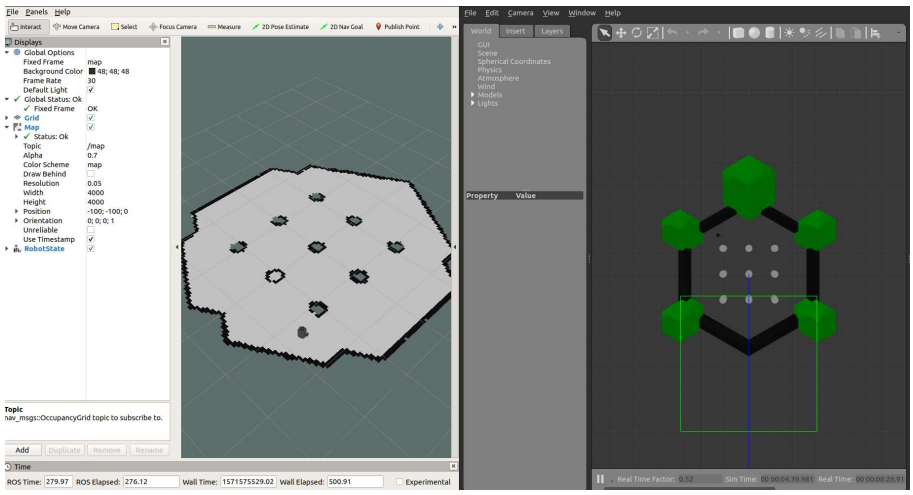
Open another terminal and type:

```
rosrun gmapping slam_gmapping _xmin:=-5 _xmax:=5 _ymin:=-5 _ymax:=5
```



The last thing we need is the teleoperation to move the robot around to cover as mach area. We will again use the official ROS-node this time.

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```



As soon as you feel satisfied with your map:

```
roscd hello_ros
mkdir maps
cd maps
rosrun map_server map_saver
```

Do as follows:

0:00

Exercise 7.7: Commanding our robot to move to a location

NOTE:

We define notes with a red vertical line,
terminal commands with a black one,
and code with a blue one

We would like to send goals to our robot through our code.

First launch the simulation

```
roslaunch turtlebot_gazebo turtlebot3_world.launch
```

It's so much better to see these things through RVIZ

```
rosrun rviz rviz
```

Lets run the navigatiojn onb the existing map

```
roslaunch hello_ros navigation.launch
```

Let's make a file named **patrol**.py and save it to

~/catkin_ws/src/hello_ros/scripts/patrol.py:

```
#!/usr/bin/env python

import rospy
import actionlib

from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal


waypoints = [
    [(1.13, -1.6, 0.0), (0.0, 0.0, -0.16547, -0.986213798314)],
    [(0.13, 1.93, 0.0), (0.0, 0.0, -0.64003024, -0.76812292098)]
]


def goal_pose(pose):
    goal_pose = MoveBaseGoal()
    goal_pose.target_pose.header.frame_id = 'map'
    goal_pose.target_pose.pose.position.x = pose[0][0]
    goal_pose.target_pose.pose.position.y = pose[0][1]
    goal_pose.target_pose.pose.position.z = pose[0][2]
    goal_pose.target_pose.pose.orientation.x = pose[1][0]
    goal_pose.target_pose.pose.orientation.y = pose[1][1]
    goal_pose.target_pose.pose.orientation.z = pose[1][2]
    goal_pose.target_pose.pose.orientation.w = pose[1][3]

    return goal_pose


if __name__ == '__main__':
    rospy.init_node('patrol')

    client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
    client.wait_for_server()

    while True:
        for pose in waypoints:
            goal = goal_pose(pose)
            client.send_goal(goal)
            client.wait_for_result()
            rospy.sleep(3)
```

Now we need allow the python file to be executable

(**~/catkin_ws/src/hello_ros/**):

```
cd scripts
chmod +x patrol.py
```

We need to

Now we can also run the thing:

```
rosrun hello_ros patrol.py
```

Exercise 7.6: Navigate in a pre-mapped area

NOTE:

We define notes with a red vertical line,
terminal commands with a black one,
and code with a blue one

We would like to make our robot do things in this world!!!

roslaunch hello_ros turtlebot3_world.launch

It's so much better to see these things throught RVIZ

rosrun rviz rviz

Now that we have a map we can use it to navigate in the area.

First let's install some prerequisites:

```
sudo apt-get install ros-melodic-dwa-local-planner
```

You can now see both the saved map

Let's make a file named move_base.launch.xml and save it to

~/catkin_ws/src/hello_ros/launch/move_base.launch.xml:

```
<launch>
  <!-- Arguments -->
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model
type [burger, waffle, waffle_pi]"/>
  <arg name="cmd_vel_topic" default="/cmd_vel" />
  <arg name="odom_topic" default="odom" />
  <arg name="move_forward_only" default="false"/>

  <!-- move_base -->
  <node pkg="move_base" type="move_base" respawn="false"
name="move_base" output="screen">
    <param name="base_local_planner"
value="dwa_local_planner/DWAPlannerROS" />
    <rosparam file="$(find
turtlebot3_navigation)/param/costmap_common_params_$(arg model).yaml"
command="load" ns="global_costmap" />
    <rosparam file="$(find
turtlebot3_navigation)/param/costmap_common_params_$(arg model).yaml"
command="load" ns="local_costmap" />
    <rosparam file="$(find
turtlebot3_navigation)/param/local_costmap_params.yaml" command="load"
/>
    <rosparam file="$(find
turtlebot3_navigation)/param/global_costmap_params.yaml"
command="load" />
    <rosparam file="$(find
turtlebot3_navigation)/param/move_base_params.yaml" command="load" />
    <rosparam file="$(find
turtlebot3_navigation)/param/dwa_local_planner_params_$(arg model).yaml"
command="load" />
    <remap from="cmd_vel" to="$(arg cmd_vel_topic)"/>
    <remap from="odom" to="$(arg odom_topic)"/>
    <param name="DWAPlannerROS/min_vel_x" value="0.0" if="$(arg
move_forward_only)" />
  </node>
</launch>
```

Also lets make the main file:

~/catkin_ws/src/hello_ros/launch/navigation.launch:

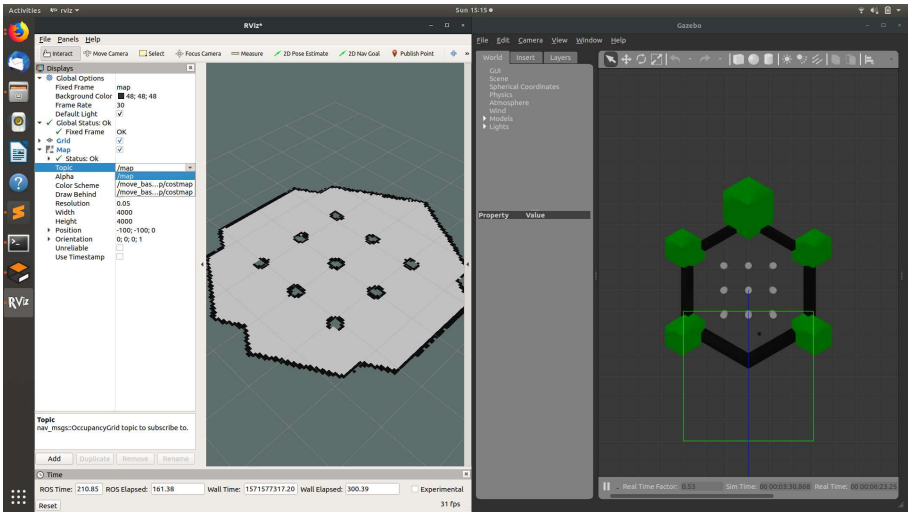
```
<launch>
```

```
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model
type [burger, waffle, waffle_pi]"/>
  <arg name="move_forward_only" default="false"/>


  <arg name="initial_pose_x" default="-2.0"/>
  <arg name="initial_pose_y" default="-0.5"/>
  <arg name="initial_pose_a" default="0.0"/>

  <param name="/use_sim_time" value="true"/>

  <!--  ****** Maps *****  -->
  <arg name="map_file" default="$(find hello_ros)/maps/map.yaml"/>
  <node name="map_server" pkg="map_server" type="map_server"
args="$(arg map_file)" />

  <!-- AMCL -->
  <include file="$(find turtlebot3_navigation)/launch/amcl.launch">
    <arg name="initial_pose_x" value="$(arg initial_pose_x)" />
    <arg name="initial_pose_y" value="$(arg initial_pose_y)" />
    <arg name="initial_pose_a" value="$(arg initial_pose_a)" />
  </include>

   <!--  ************** Navigation  ***************  -->
  <include file="$(find hello_ros)/launch/move_base.launch.xml">
    <arg name="model" value="$(arg model)" />
    <arg name="move_forward_only" value="$(arg move_forward_only)"/>
  </include>
</launch>
```

Now we can also run the main launch file:

```
roslaunch hello_ros navigation.launch
```

You can now see both the saved map from previous exercise along with two costmaps in RViz:

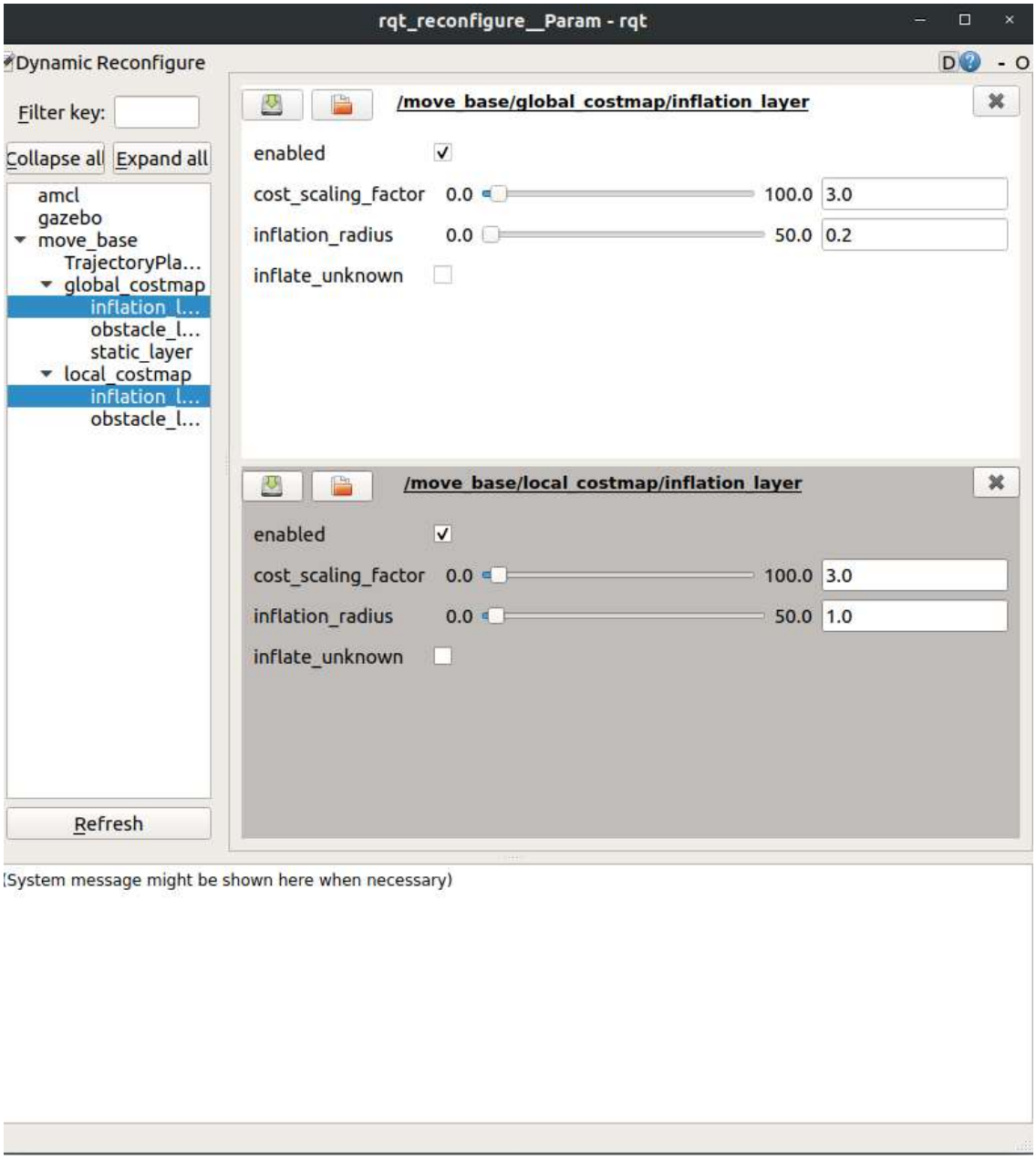


We might need to change a bit in the configuration in order to use the maps. It is also very useful to understand the parameters of the path planner, path execution and localization.

To do so run the following command:

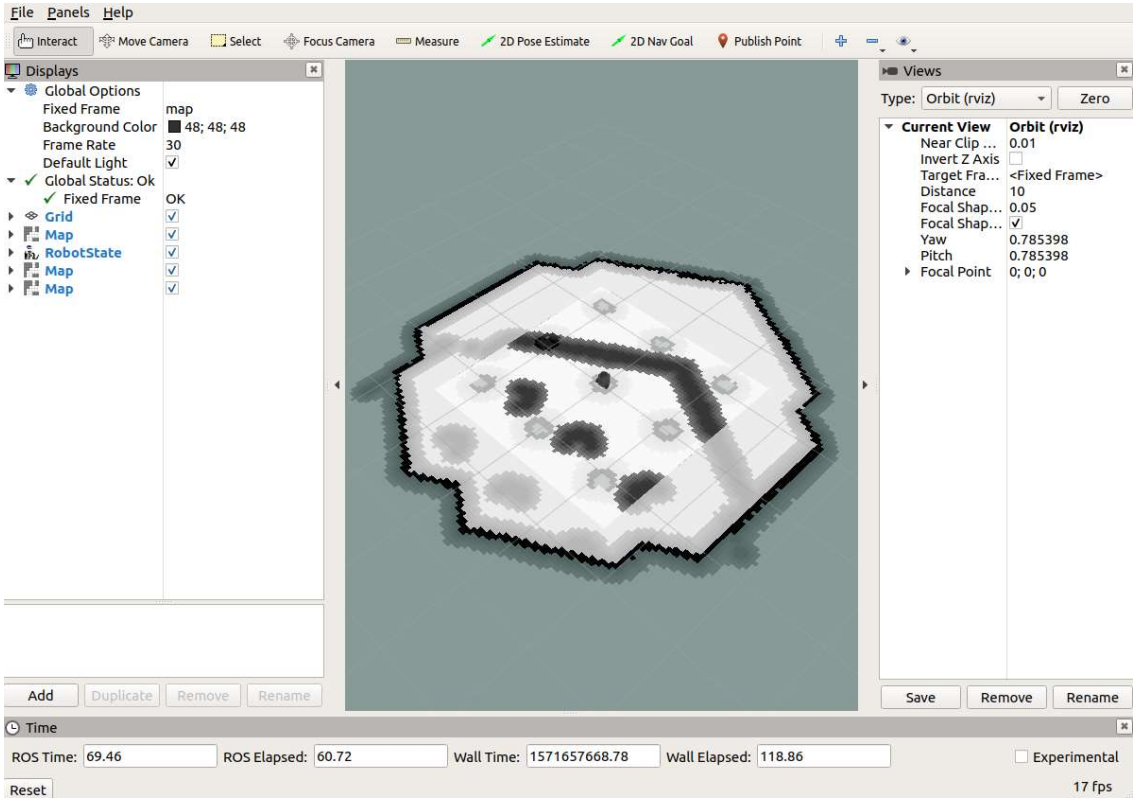
```
rosrun rqt_reconfigure rqt_reconfigure
```

This should open a new window:



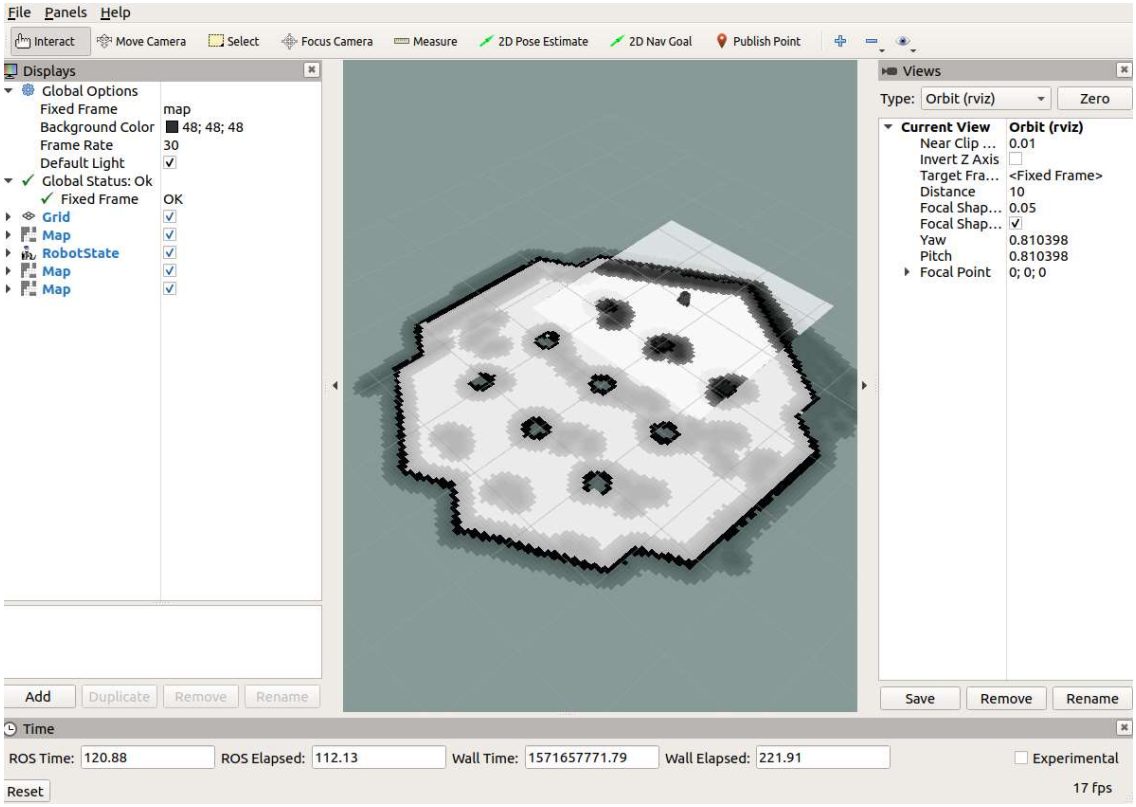
For example: Under "move_base" -> "global_costmap" -> "inflation_layer" you can set the "inflation_radius" to 0.2. Do the same for "local_costmap".

In RViz add the map and the costmaps. It will probably look a bit like this:



This is because the robot don't know where it is with respect to the map. We will give an initial estimate by looking where the robot is in Gazebo, then press "**2D Pose Estimate**" in RViz and then press the location of the robot in the RViz map.

This should make the costmap look a bit better:



Now we should be able to have the robot follow a motion plan towards a goal. We can test this by pressing the "**2D Nav Goal**" button in RViz and then press

somewhere on the map. This should make the robot move toward the goal in both RViz and Gazebo while taking the map into account.

See the video below:

0:00 / 8:42

Mini exercise: Wouldn't it be interesting to monitor the plans of the navigation system? Try to visualize the local an global plan of the robot! (Hint: Add the right modules and topics in rviz)

Exercise 7.8: Lets fake the localization

NOTE:

We define notes with a red vertical line,

terminal commands with a black one,

and code with a blue one

In this tutorial we would like to use fake localization rather than the estimated one. This can be useful, if you want to test your setup excluding possible localization errors.

roslaunch turtlebot_gazebo turtlebot_world.launch

It's so much better to see these things throught RVIZ

rosrun rviz rviz

Lets alter the navigation file:

We don't need the localization node anymore (AMCL).

~/catkin_ws/src/hello_ros/scripts/navigation_no_localization.launch:

```
<launch>

  <arg name="initial_pose_x" default="0.0"/>
  <arg name="initial_pose_y" default="0.0"/>
  <arg name="initial_pose_a" default="0.0"/>

  <param name="/use_sim_time" value="true"/>

   <!--  ************** Navigation  ***************  -->
  <include file="$(find hello_ros)/launch/move_base.launch.xml"/>

  <!--  ****** Maps *****  -->
  <arg name="map_file" default="$(find hello_ros)/maps/map.yaml"/>
  <node name="map_server" pkg="map_server" type="map_server"
args="$(arg map_file)" />

</launch>
```

Now we have to fake the localization. By publishing a 0-transform between odom and map we get a perfect localization estimate:

```
rosrun tf static_transform_publisher 0 0 0 0 0 0 odom map 0.001
```

Now we can also run the main launch file:

```
roslaunch hello_ros navigation_no_localization.launch
```