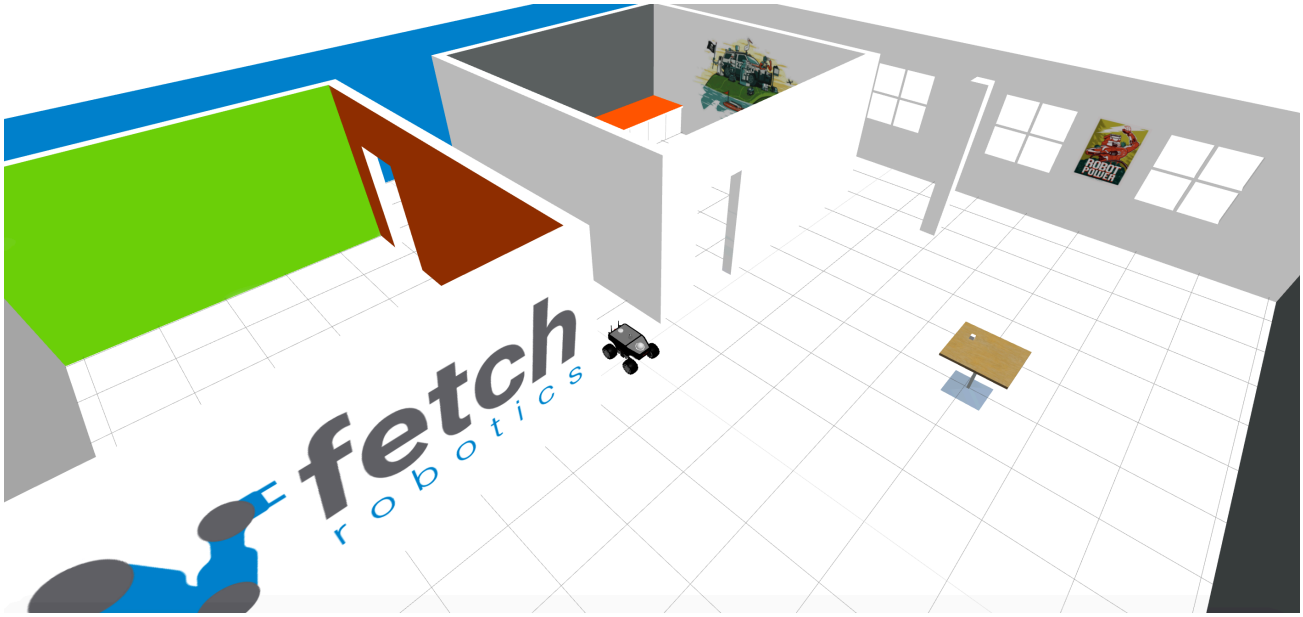
ROS Navigation

Course Project



SUMMARY

Estimated time to completion: **1'5 hours**

What will you learn with this unit?

- Practice everything you've learned through the course
- Put together everything that you learned into a big project
- Create a launch file that launches each part of the Navigation Stack
-

END OF SUMMARY

Navigate the Summit Robot!

In this project, you will have to make a Summit robot (http://www.robotnik.eu/mobile-robots/summit-xl/) navigate around a room autonomously.

For this goal, you will have to apply all of the things that you are learning along the course. It's really important that you complete it because all of the structures that you create for this project will be asked about in our **official exam**.

Note: Our offical exam is only available, at present, for on-site or virtual classes.

Basically, in this project, you will have to:

- Apply all of the theory given in the course
- Follow the steps provided below without looking at the provided solutions (unless you get really stuck)
- Make as many tests as required in the simulation environment until it works

To finish this project successfully, we provide the 5 steps you should follow with clear instructions, and even solutions, below.

Also, remember to:

- Create your packages and code in the simulation environment as you have been doing throughout the course.
- Use the consoles to gather information about the status of the simulated robot.
- Use the IDE to create your files and execute them through the consoles, observing the results on the simulation screen. You can use other consoles to watch calls to topics, services, or action servers.
- Everything that you create in this unit will be automatically saved in your space. You can come back to this unit at any time and continue with your work from the point where you left off.
- Every time you need to reset the position of the robot, just press the restart button in the simulation window.
- Use the debugging tools to try to find what is not working and why (for instance, the Rviz tool is very useful for this purpose).

What does Summit provide for programming?

Sensors

- **Laser sensor**: Summit has a Hokuyo Laser, (https://en.wikipedia.org/wiki/Inertial_measurement_unit) which provides information about the objects that it detects in its range. The value of the sensor is provided through the topic */hokuyo_base/scan*
- **Odometry**: The odometry of the robot can be accessed through the */odom* topic

Actuators

- **Speed**: You can send speed commands to move the robot through the topic */summit_xl_control/cmd_vel*.

Steps you should cover

These are the steps that you should follow throughout the duration of the project. These steps will assure you that you have practised and created all of the structures asked about in the final exam of this course. If you perform all of the steps mentioned here, you will find the exam passable.

NOTE 1: We do provide the solution for each step. Do not watch unless you are really stuck.

Step 1: Generate a Map of the Environment

As you've learned in the first 2 chapters of the course, the first thing you need in order to navigate autonomously with a robot is a map of the environment. Where else would this project start?

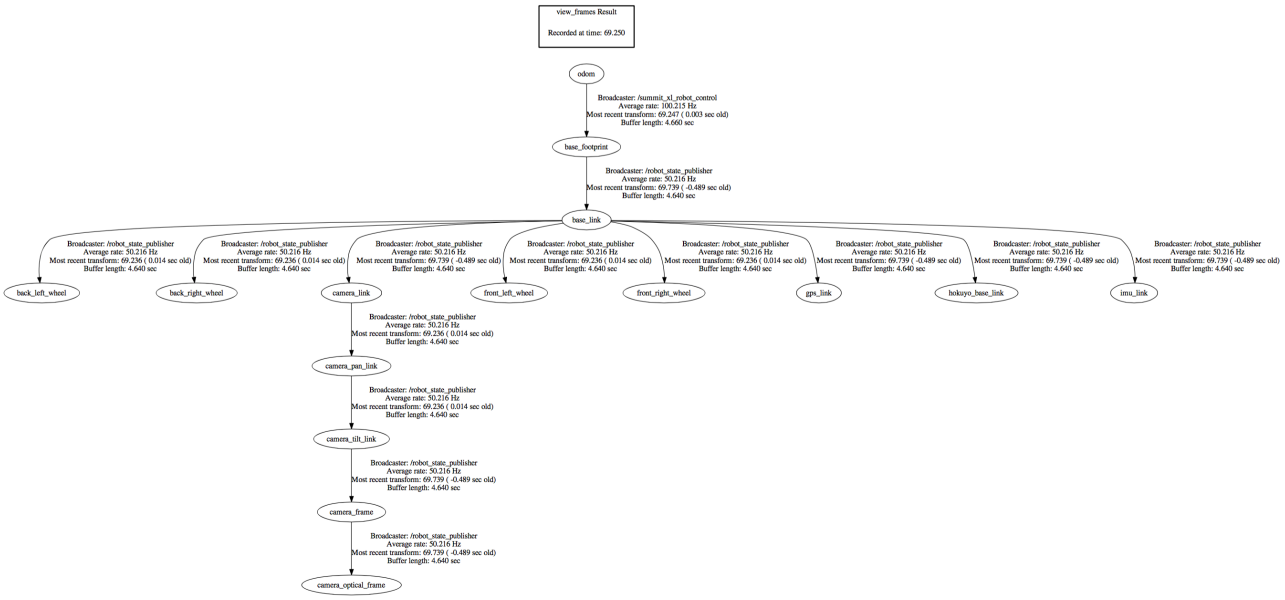
Therefore, the first step would not be anything other than creating a map of the environment that we want to navigate. In order to achieve this, we've divided this first step into 7 sub-steps:

- Make sure that the Summit robot is publishing it's transform data correctly.
- Create a package called **my_summit_mapping** that will contain all of the files related to mapping.
- Create a **launch file** that will launch the slam_gmapping node and add the necessary parameters in order to properly configure the slam_gmapping node.
- Launch the slam_gmapping node and create a map of the simulated environment.
- Save the map that you've just created.
- Create a script that automatically saves maps.
- Create a **launch file** that will provide the created map to other nodes.

So, let's start doing them!

1. Generate the necessary files in order to visualize the frames tree.

You should get a file like this one:



2. Create a package called *my_summit_mapping.*

Create the package, adding **rospy** as the only dependency.

3. Create the launch file for the gmapping node.

In the mapping section (Chapater 2) of this course, you've seen how to create a launch file for the slam_gmapping node. You've also seen some of the most important parameters to set. So, in this step, you'll have to create a launch file for the slam_gmapping node and add the parameters that you think you need to set.

Here you can see a full list of parameters that you can set to configure the slam_gmapping node: http://docs.ros.org/hydro/api/gmapping/html/ (http://docs.ros.org/hydro/api/gmapping/html/)

Here you have an example launch file for the gmapping node:

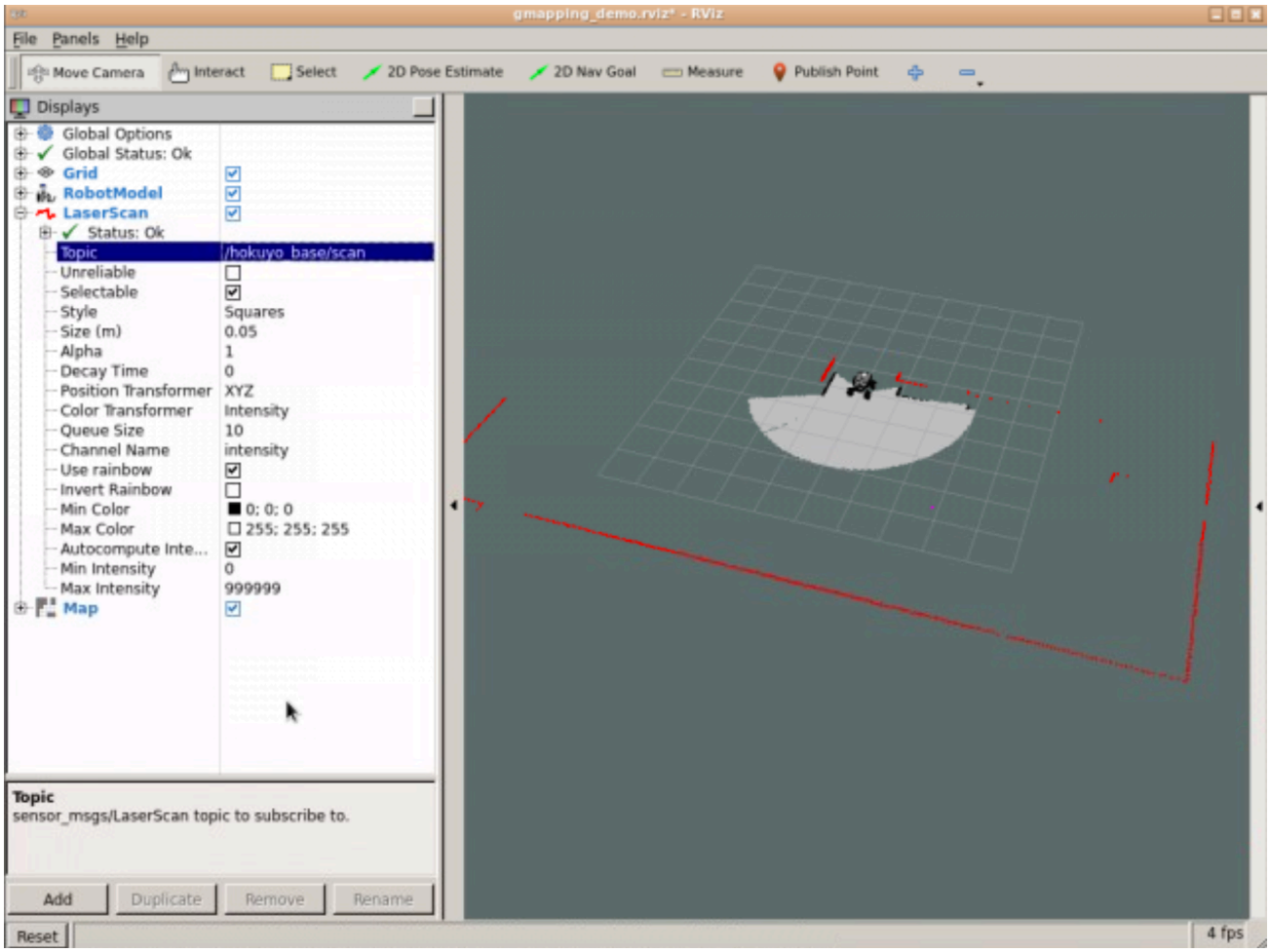
In [ ]:
```
<launch>
    <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping" output
      <!-- simulation remap from="scan" to="/hokuyo_laser_topic"/ -->
      <!-- real -->
      <!-- remap from="scan" to="/scan_filtered"/ -->
      <remap from="scan" to ="/hokuyo_base/scan"/>
      <!-- param name="map_udpate_interval" value="5.0"/ -->
      <param name="base_frame" value="base_footprint"/>
      <param name="odom_frame" value="odom"/>
      <param name="map_udpate_interval" value="5.0"/>
      <param name="maxUrange" value="2.0"/>
      <param name="sigma" value="0.05"/>
      <param name="kernelSize" value="1"/>
      <param name="lstep" value="0.05"/>
      <param name="astep" value="0.05"/>
      <param name="iterations" value="5"/>
      <param name="lsigma" value="0.075"/>
      <param name="ogain" value="3.0"/>
      <param name="lskip" value="0"/>
      <param name="srr" value="0.1"/>
      <param name="srt" value="0.2"/>
      <param name="str" value="0.1"/>
      <param name="stt" value="0.2"/>
      <param name="linearUpdate" value="0.2"/>
      <param name="angularUpdate" value="0.1"/>
      <param name="temporalUpdate" value="3.0"/>
      <param name="resampleThreshold" value="0.5"/>
      <param name="particles" value="100"/>
      <param name="xmin" value="-50.0"/>
      <param name="ymin" value="-50.0"/>
      <param name="xmax" value="50.0"/>
      <param name="ymax" value="50.0"/>
      <param name="delta" value="0.05"/>
      <param name="llsamplerange" value="0.01"/>
      <param name="llsamplestep" value="0.01"/>
      <param name="lasamplerange" value="0.005"/>
      <param name="lasamplestep" value="0.005"/>
    </node>
</launch>
```

4. Launch the node using the launch file that you've just created, and create a map of the environment.

In order to move the robot around the environment, you can use the keyboard teleop. To launch the keyboard teleop, just execute the following command:

In [ ]:
```
roslaunch summit_xl_gazebo keyboard_teleop.launch
```

Also, remember to launch Rviz and add the proper displays in order to visualize the map you are generating. You should get something like this in RViz:



Now, start moving the robot around the room to create the map.

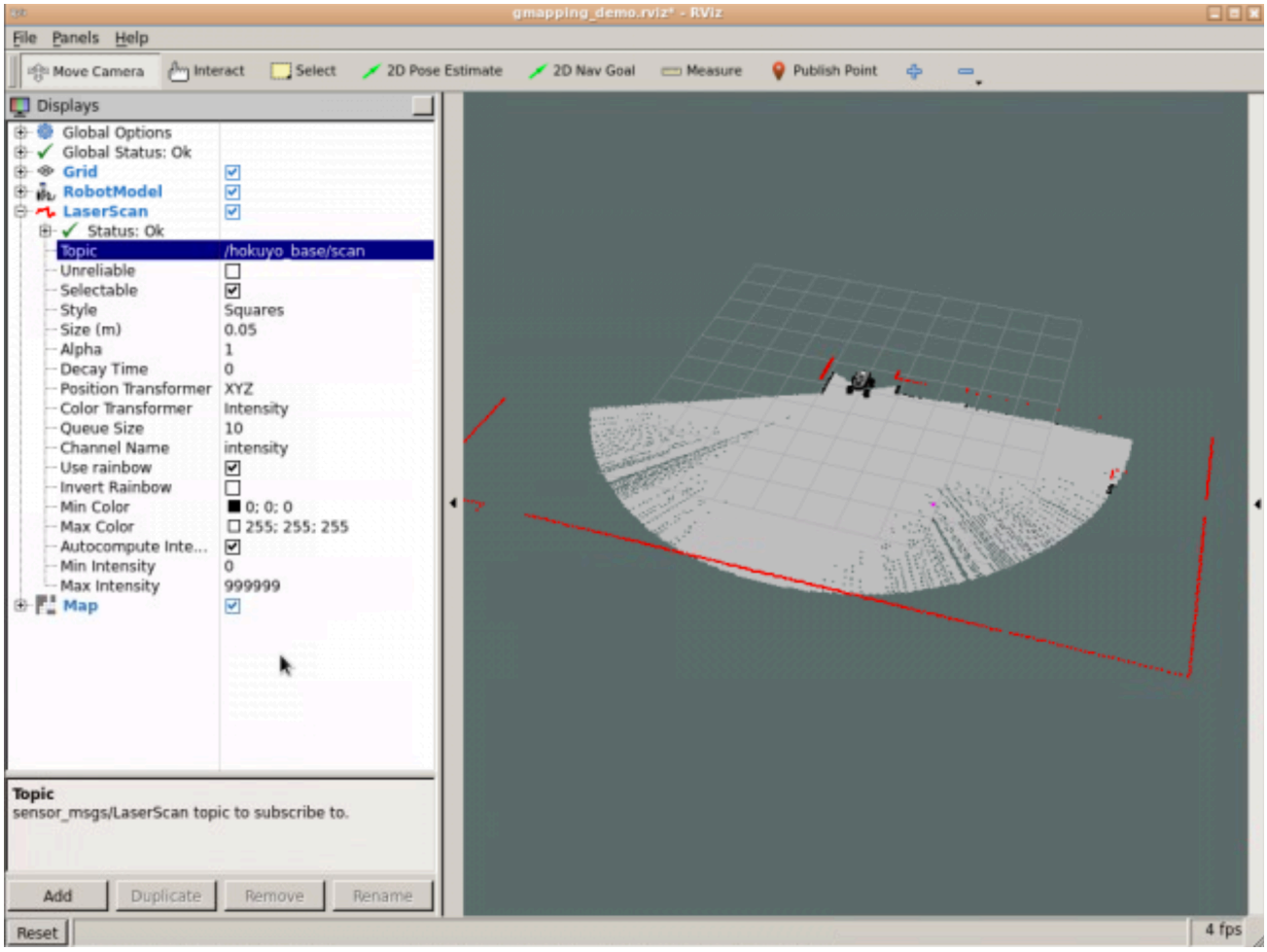
What's happening? Are you having difficulty generating a proper map? Is there anything strange in the mapping process? Do you know what could be the reason behind it?

What's happening here is the following:

- Your maxUrange parameter in the launch file of the slam_gmapping node is set to 2. This parameter sets the maximum range of the laser scanner that is used for map building. Since this is a small value, the robot can't get enough data to know where it is, so it may get lost. This will cause some strange issues during the mapping process, such as the robot readjusting its position incorrectly.

- In order to solve it, you should set the maxUrange parameter to some higher value, for instance, 7. This way, the robot will be able to get more data from the environemnt, so that it won't get lost. Now, you'll be able to finalize the mapping process correctly.

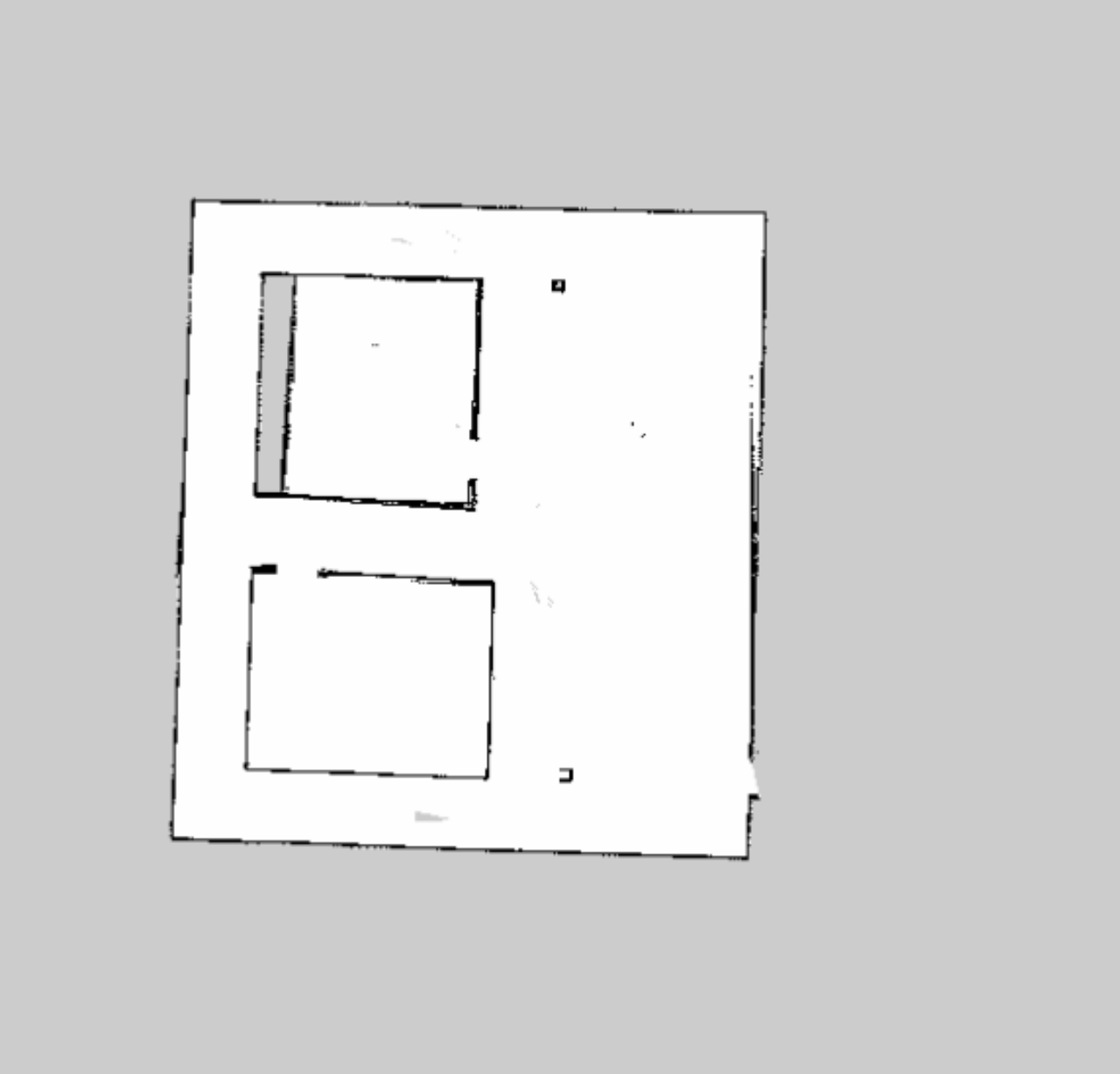
Once you modify this parameter and relaunch the slam_gmapping node, you should get something like this in RViz:



Now, you will be able to map the environment much more easily.

5. Save the map.

Create a directory in your package named **maps**, and save the map files there. Your map image file should look something similar to this:



6. Create a launch file that launches the map_server node.

As you've also seen in the course, the map you've just created will be used by other nodes (this means, it'll be used in further steps) in order to perform navigation.

Therefore, you'll need to create a launch file in order to provide the map. As you know, this is done through the map_server node.

Finally, launch this file and check that it's really providing the map.

Step 2: Make the Robot Localize itself in the Map

Step 2: Make the Robot Localize Itself in the Map

This step has 4 actions for you to do:

- Create a package called **my_summit_localization** that will contain all of the files related to localization.
- Create a **launch file** that will launch the amcl node and add the necessary parameters in order to properly configure the amcl node.
- Launch the node and check that the robot properly localizes itself in the environment.
- Create a table with three different spots.
- Create a ROS Service to save the different spots into a file.

1. Create a package called my_summit_localization.

Create the package adding **rospy** as the only dependency.

2. Create the launch file for the amcl node.

In the localization section (Chapater 3) of this course, you saw how to create a launch file for the amcl node. You've also seen some of the most important parameters to set. So, in this step, you'll have to create a launch file for the amcl node and add the parameters you think you need to set.

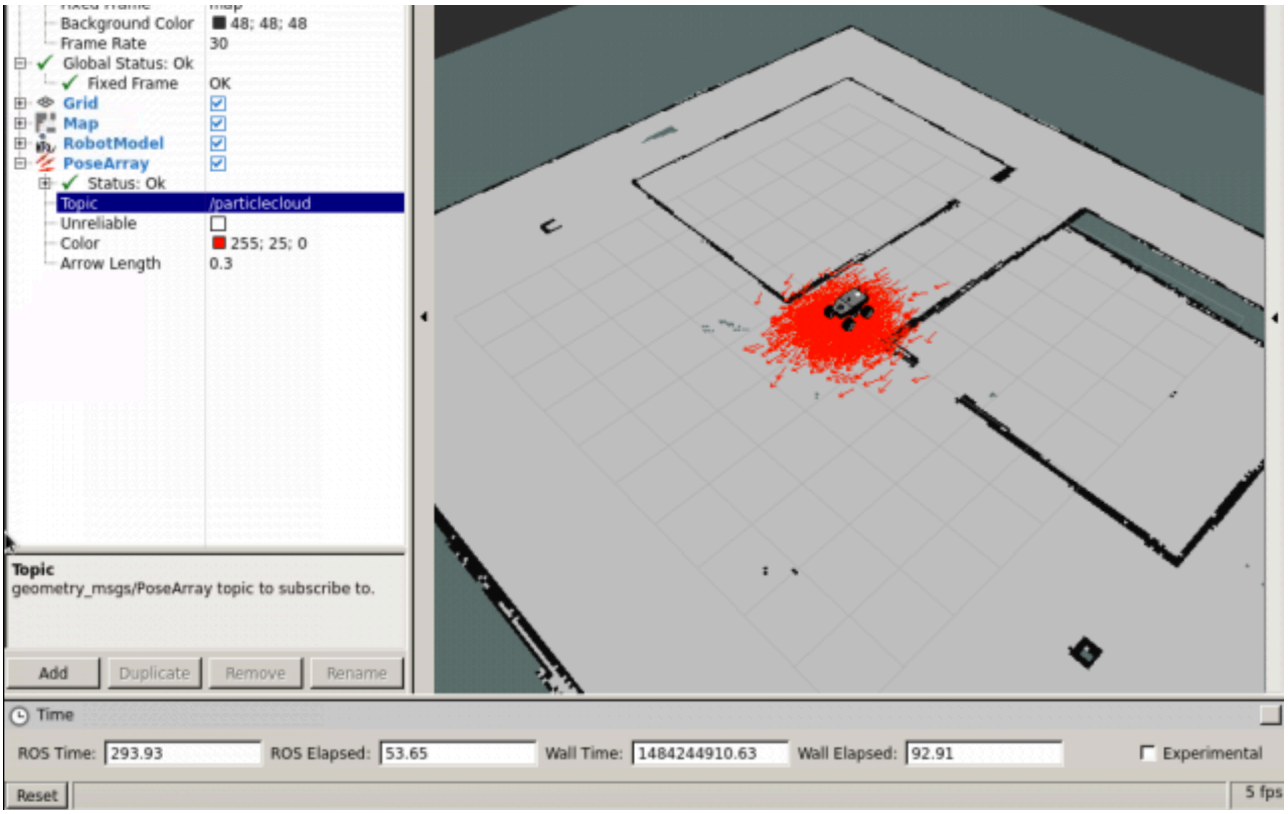
Here you can see a full list of parameters that you can set to configure the amcl node: http://wiki.ros.org/amcl (http://wiki.ros.org/amcl)

Remember that before setting the amcl node parameters, you'll need to load the map created in Step 1. For that, you'll just need to include the launch you've created in Step 1 in the amcl node launch file in order to provide the map to other nodes.

3. Launch the node and check that the Summit robot correctly localizes itself in the map.

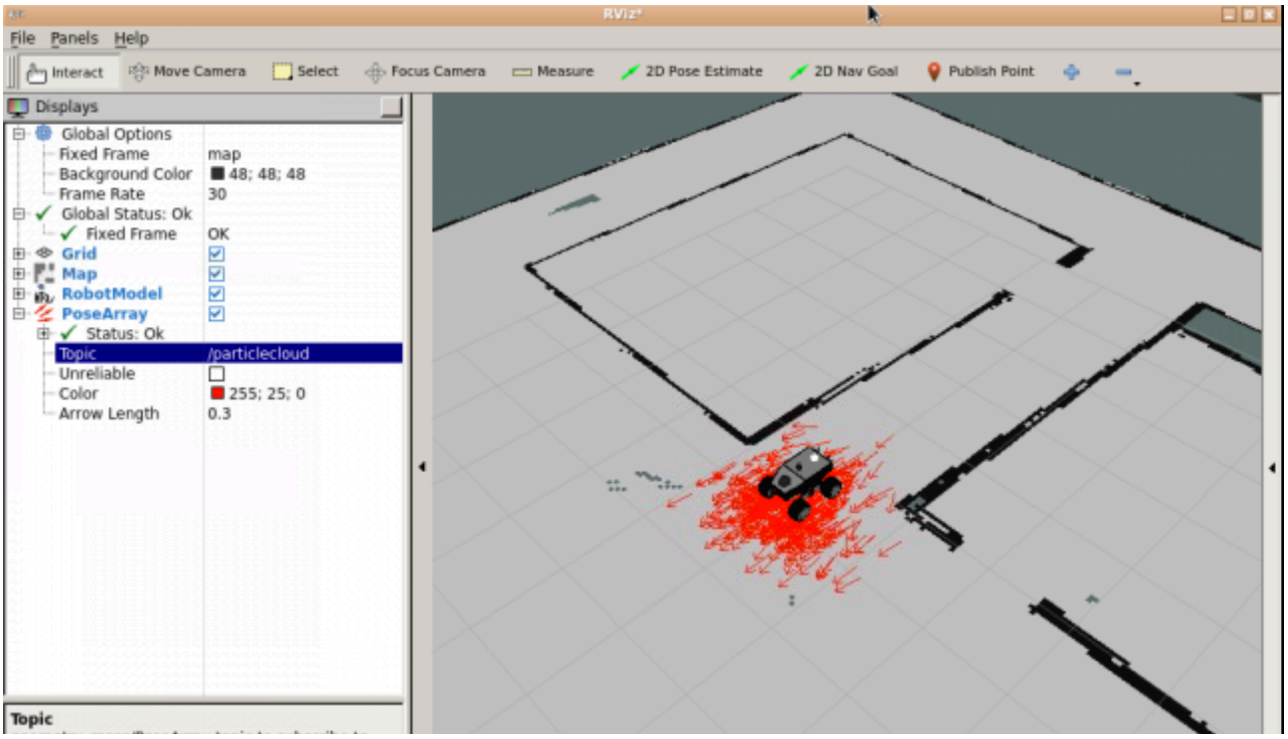
Launch the node and check in RViz that the Summit robot correctly localizes itseld in the map.

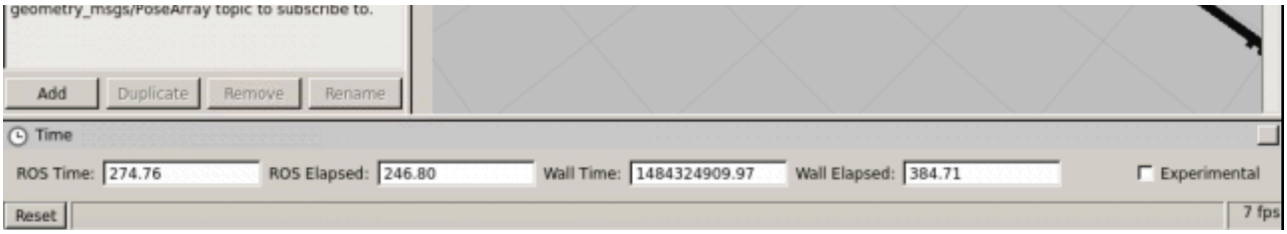
If you've done all of the previous steps correctly, you should see something like this in Rviz:



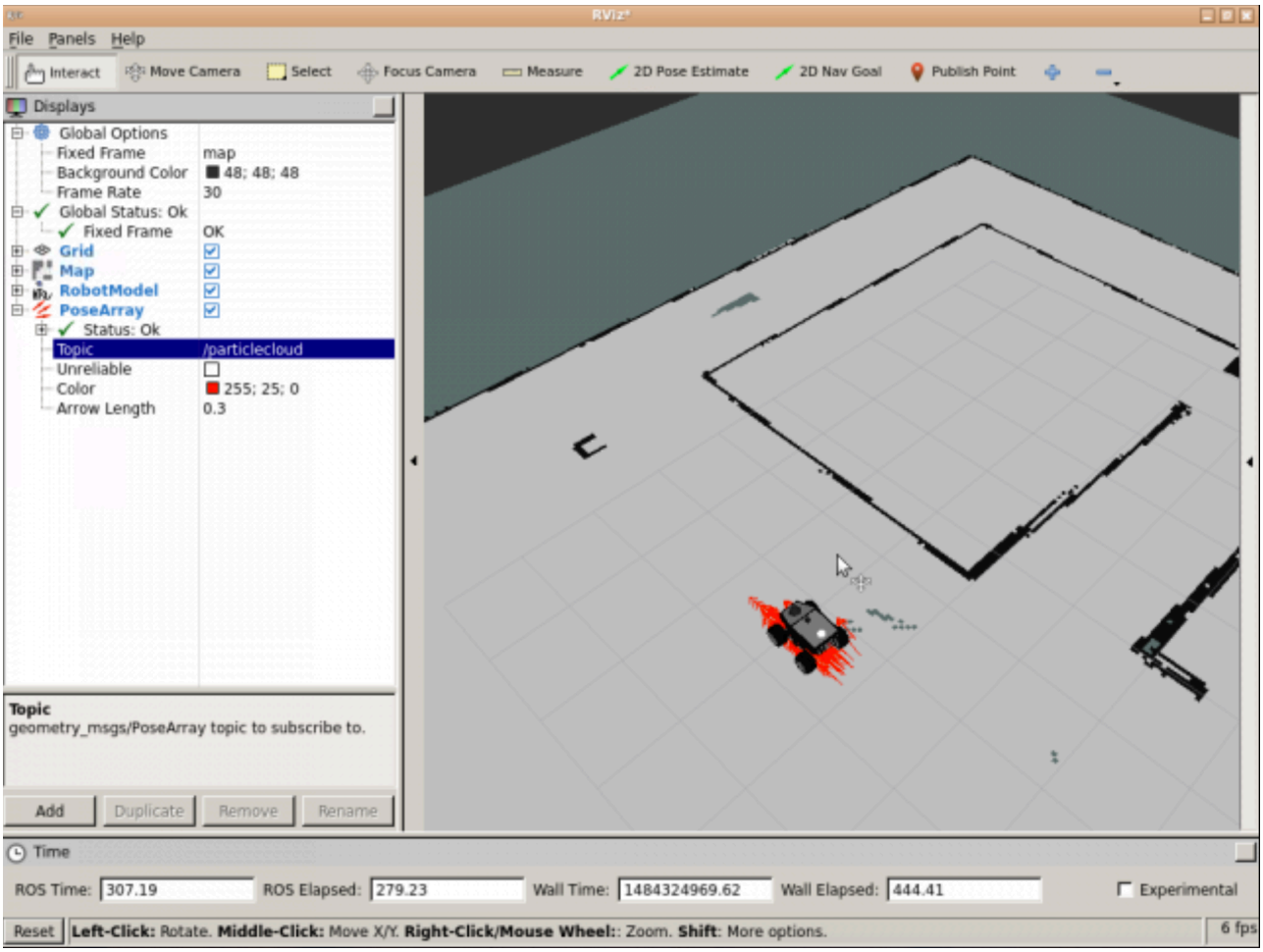
In order to check that the localization works fine, move the robot around the environment and check that the particle cloud keeps getting smaller the more you move the robot.

Dispersed particles:





Closer particles:

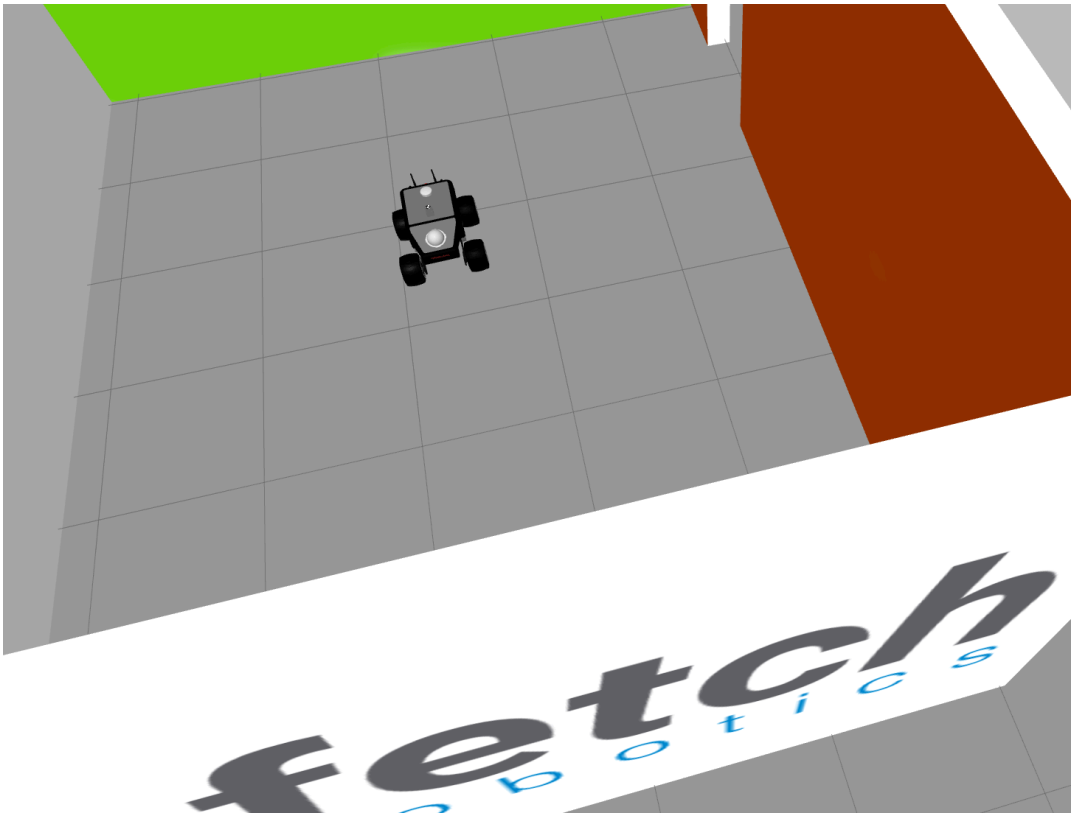


4. Create the spots table.

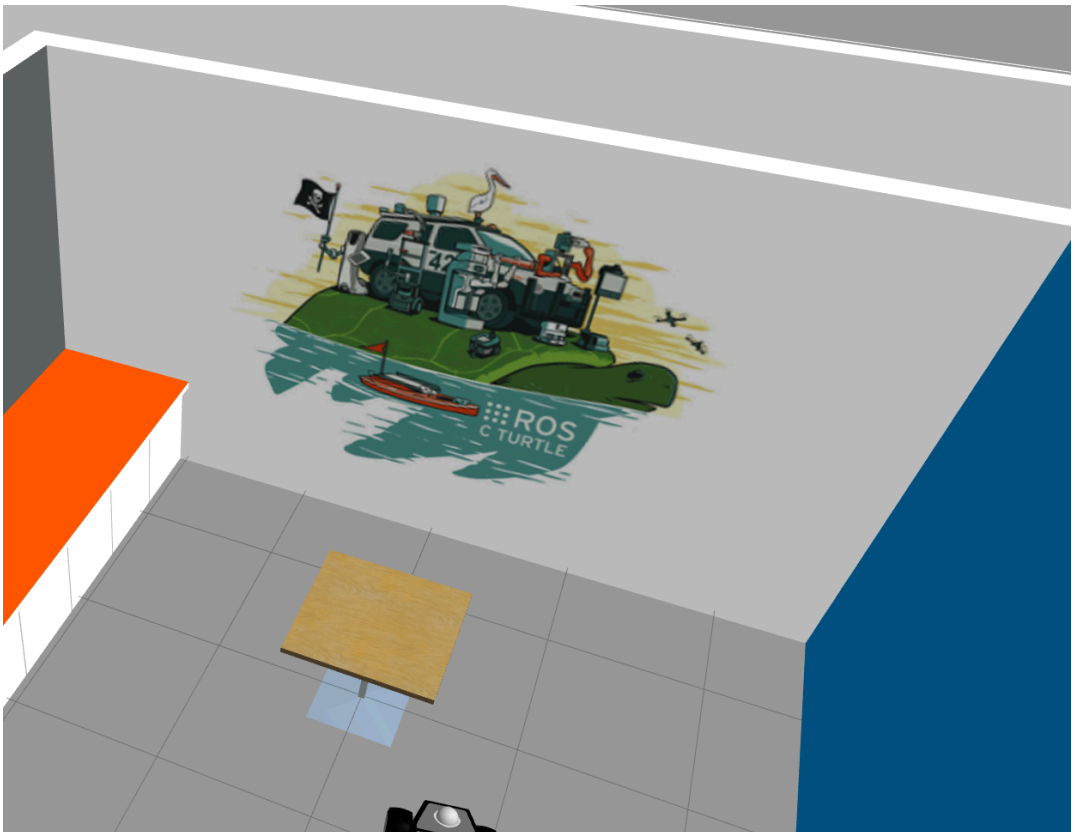
Once you've checked that localization is working fine, you'll need to create a table with 3 different spots in the environemnt. For each point, assign a tag (with the name of the spot) alongside its coordinates in the map.

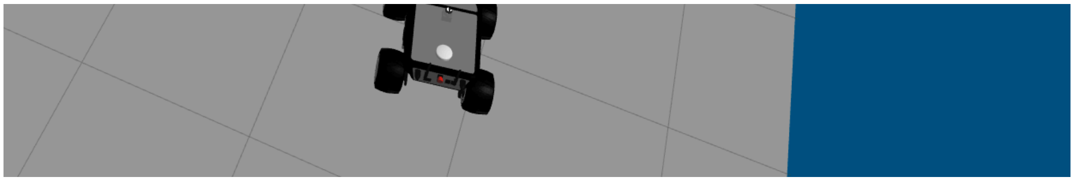
These are the 3 spots that you'll have to register into the table:

Fetch Room Center (**label: room**):

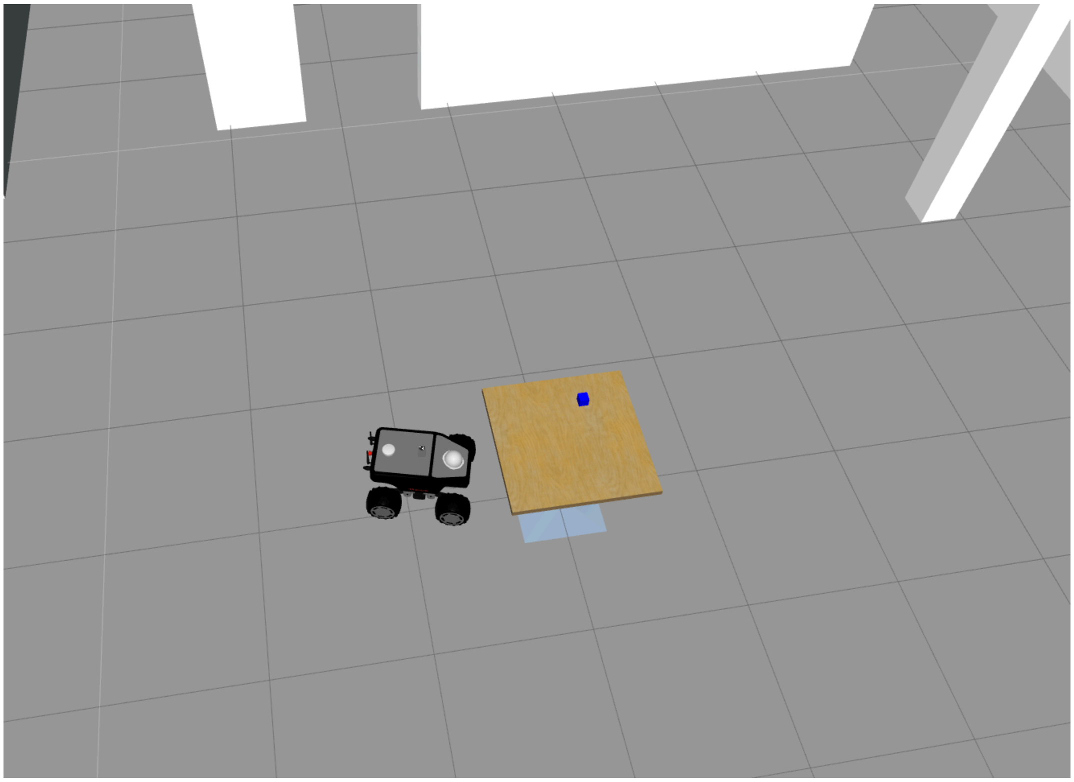


Facing the Turtle's Poster (**label: turtle**):





Besides the Table (**label: table**):



You can access the coordinates of each position by looking into the topics that the amcl node publishes on. The only data that you really need to save are the position and orientation.

Next, you have a screenshot of the */amcl_pose* topic:

```
ubuntu@ip-172-31-44-229:/home/user/catkin_ws/src/summit/summit_xl_common/summit_xl_navigation/launch$ rostopic echo -n1 /amcl_pose
header:
  seq: 0
  stamp:
    secs: 232
    nsecs: 502000000
  frame_id: map
pose:
  pose:
    position:
      x: -0.023827886945
      y: -0.0305145982103
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: -0.00486628383324
      w: 0.999988159571
  covariance: [0.22456029407813477, -1.3061872939890162e-05, 0.0, 0.0, 0.0, 0.0, -1.3061872939890162e-05, 0.17435311007245433, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.06808637427489805]
---
```

Create a file named **spots.yaml** and write in the pose data that you've gotten from the 3 spots.

5. Create a ROS Service that saves the different spots into a file.

In the previous section, you created a file with the coordinates of the 3 spots. But, the whole process was too cumbersome, don't you think? Why don't we create a ROS Service that does the work for us? Sounds like a good idea, right? Then, let's do it!

So, now you'll create a ROS program that will do the following:

- It will launch a node named **spot_recorder**.
- This node will contain a service server named **/save_spot** that will take a string as input.
- When this service is called, it will store the current coordinates of the robot (position and orientation values) with a label that will be the string provided on the service call.
- When the *end* string is provided in the service call, the node will write all of the values stored into a file named **spots.txt**.
- Finally, the service will return a message indicating if the file has been saved correctly.

In order to achieve this, let's divide it into smaller parts.

1 Service Message

First of all, you'll have to determine the kind of data you need for your service message.

- Determine what input data you need (**request**)
- Determine what data you want the service to return (**response**)

Next, see if there is an already-built message in the system that suits your needs. If there isn't, then you'll have to create your own custom message with the data you want.

If this is the case, then do the following. Inside the pacakge you just created, create a new directory named **srv**. Inside this directory, create a file named **MyServiceMessage.srv** that will contain the definition of your service message.

This file could be something like this:

In [ ]:
```
# request
string label
---
#response
bool navigation_successfull
string message
```

Now, you'll have to modify the **package.xml** and **CMakeLists.txt** files of your package in order to compile the new message. Assuming that the only dependency you added to your package was **rospy**, you should do the following modifications to these files:

In CMakeLists.txt

You will have to edit/uncomment four functions inside this file:

- find_package()
- add_service_files()
- generate_messages()
- catkin_package()

In [ ]:
```
find_package(catkin REQUIRED COMPONENTS
  rospy
  std_msgs
  message_generation
)
```

In [ ]:
```
add_service_files(
  FILES
  MyCustomServiceMessage.srv
)
```

In [ ]:
```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

In [ ]:
```
catkin_package(
      CATKIN_DEPENDS
      rospy
)
```

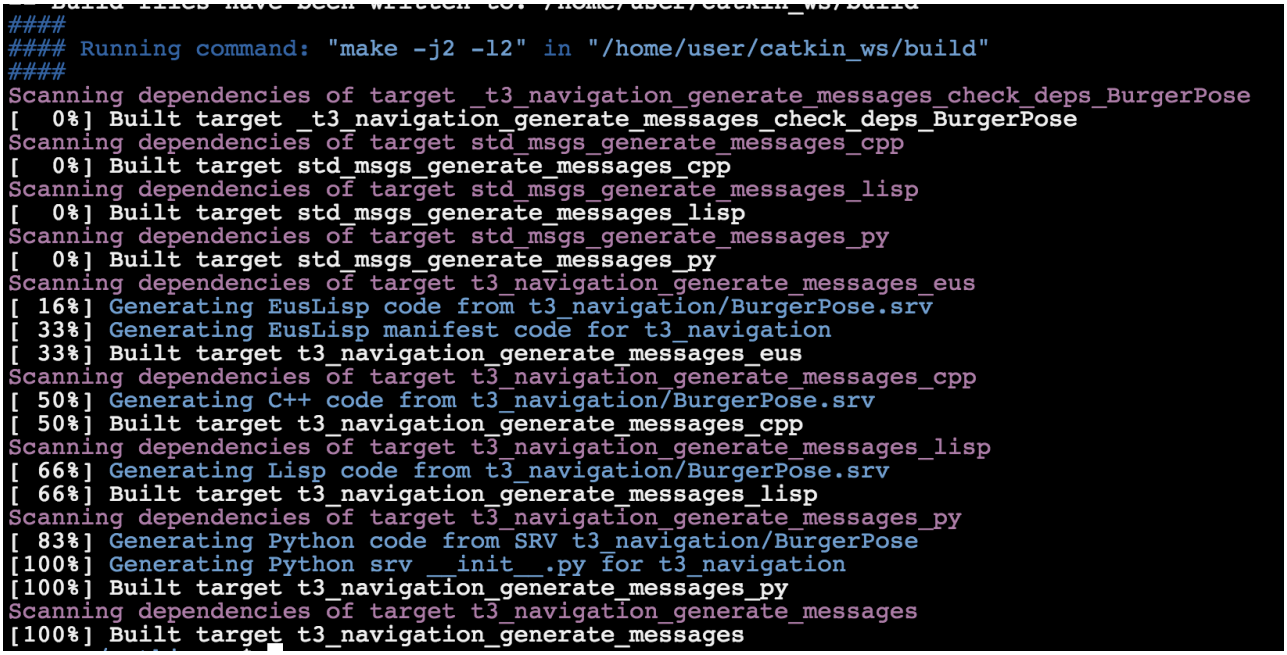
In package.xml:

Add these 2 lines to the file:

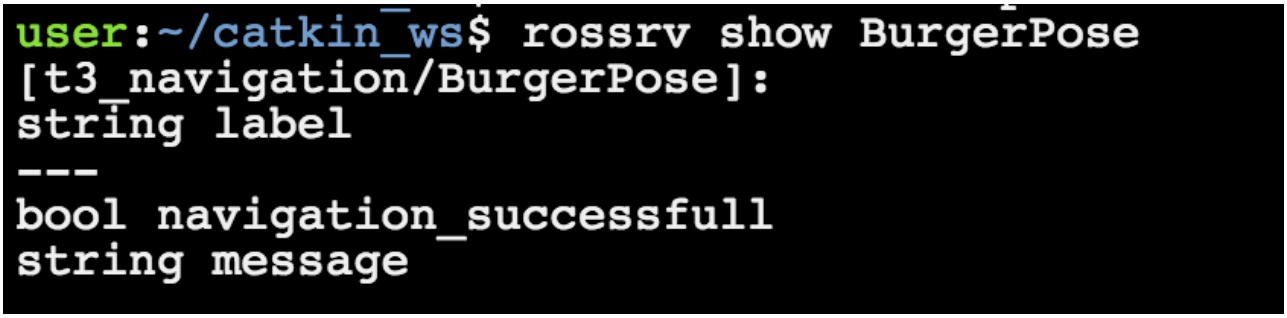
In [ ]:
```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

When you've finshed with the modifications, compile your package and check that your message has been correctly compiled and is ready to use. In order to check this, you can use the command ***rossrv show MyServiceMessage***.

catkin_make output:



rossrv show output:



2 Service Code

Once your message is created, you're ready to use it in your service! So, let's write the code for

Once your message is created, you're ready to use it in your service. So, let's write the code for our service. Inside the src directory of your package, create a file named **spots_to_file.py**. Inside this file, write the code necessary for you service.

3 Launch file

Create a launch file for the node you've just created. You can also launch this node in the same launch file you've created to launch the slam_gmapping node. It's up to you.

4 Test it

Using the keyboard teleop, move the robot to the 3 different spots shown above. At each one of these spots, perform a service call to the service you've just created. In the service call, provide the string with the name that you want to give to each spot.

For example, when you are at the center of the Fetch Room, call your service like this:

In [ ]:
```
rosservice call /record_spot "label: fetch_room"
```

Repeat this for each of the 3 spots. When finished, do one last service call, providing *end* as the string in order to create the file.

Step 3: Set Up the Path Planning System

This step has 5 actions for you to take:

- Create a package called **my_summit_path_planning** that will contain all of the files related to Path Planning.
- Create a **launch file** that will launch the move_base node.
- Create the necessary parameter files in order to properly configure the move_base node.
- Create the necessary parameter files in order to properly configure the global and local costmaps.
- Create the necessary parameter files in order to properly configure the global and local planners.
- Launch the node and check that everything works fine.

1. Create a package named my_summit_path_planning.

2. Create the launch file for the move_base node.

In the Path Planning section (Chapter 4) of this course, you saw how to create a launch file for the move_base node. So, in this step, you'll have to create a launch file for the move_base node and add the parameters and the parameter files required.

Here you have an example of the move_base.launch file:

In [ ]:
```
<launch>
  <!--- Run AMCL -->
  <include file="$(find my_summit_localization)/launch/amcl.launch" />
  <remap from="cmd_vel" to="/move_base/cmd_vel" />

  <!-- Run move_base -->
  <node pkg="move_base" type="move_base" respawn="false" name="move_base"
    <rosparam file="$(find my_summit_path_planning)/config/move_base_para
    <rosparam file="$(find my_summit_path_planning)/config/costmap_common
    <rosparam file="$(find my_summit_path_planning)/config/costmap_common
    <rosparam file="$(find my_summit_path_planning)/config/local_costmap_
    <rosparam file="$(find my_summit_path_planning)/config/global_costmap
    <rosparam file="$(find my_summit_path_planning)/config/dwa_local_plan
    <rosparam file="$(find my_summit_path_planning)/config/global_planner

    <param name="base_global_planner" value="navfn/NavfnROS" />
    <param name="base_local_planner" value="dwa_local_planner/DWAPlannerR
    <param name="controller_frequency" value="5.0" />
    <param name="controller_patience" value="15.0" />
  </node>

</launch>
```

3. Create the move_base_params.yaml file

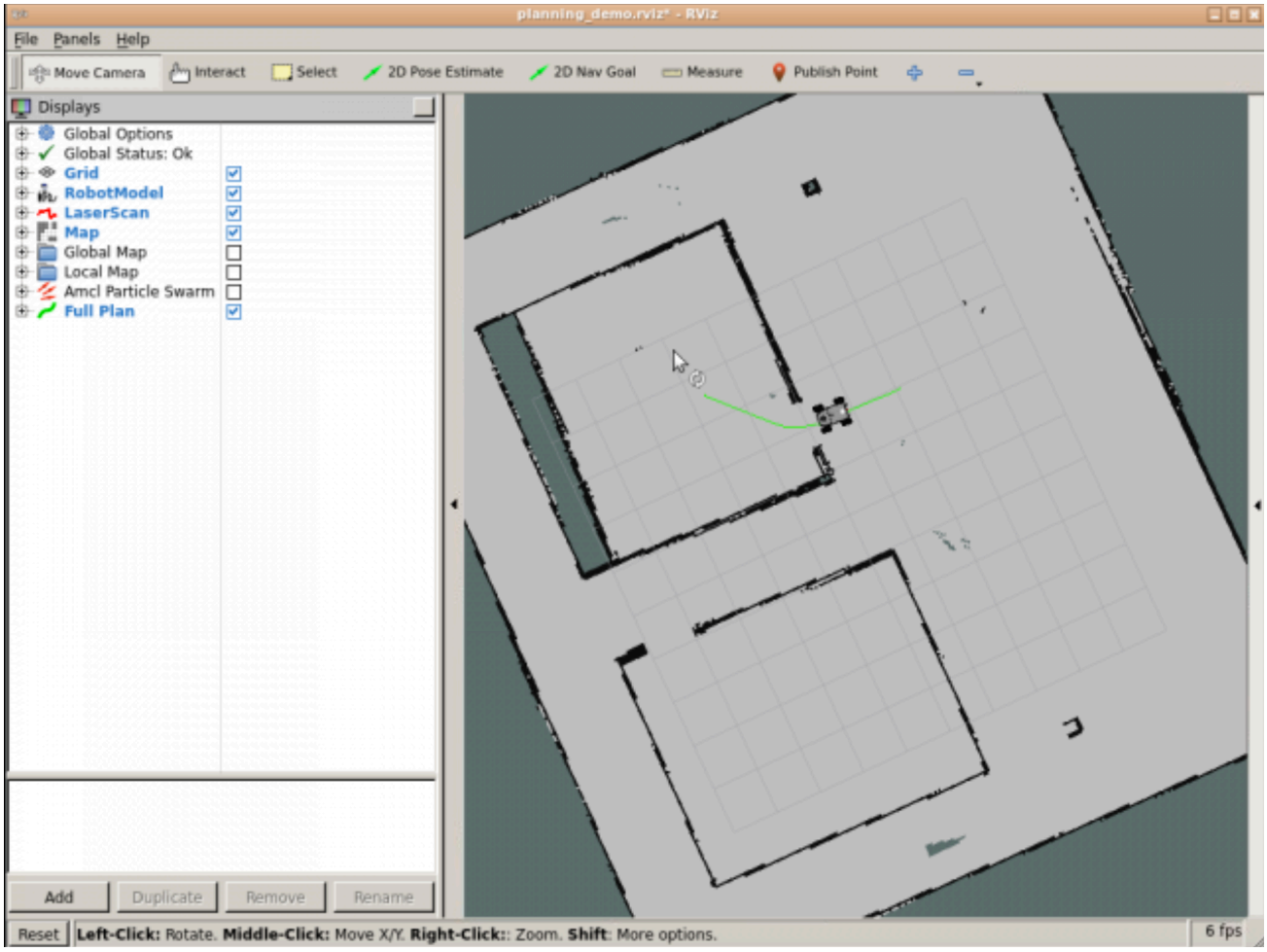
4. Create the costmap_common_params.yaml, the global_costmap_params.yaml, and the local_costmap_params.yaml files

5. Create the navfn_global_planner_params.yaml and dwa_local_planner_params.yaml files

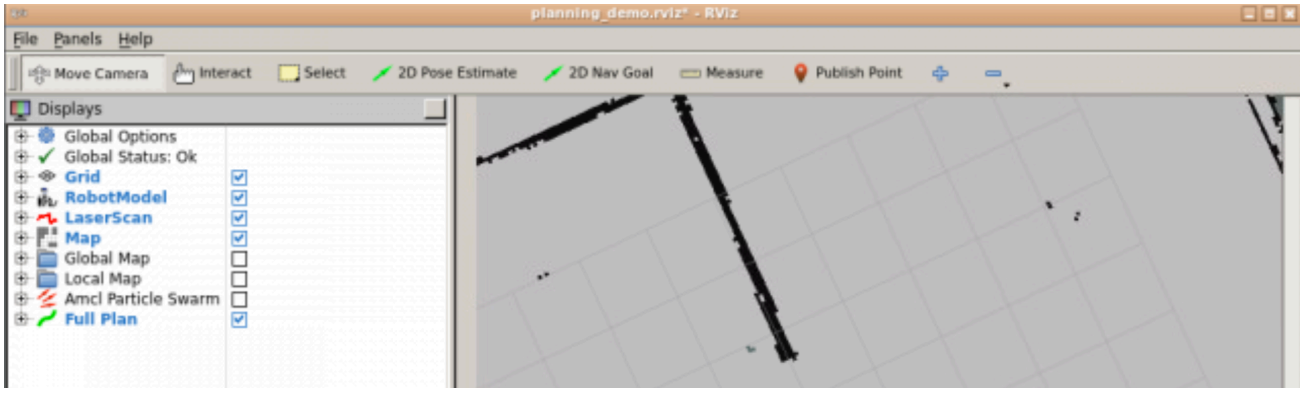
6. Launch the node and check that everything works fine

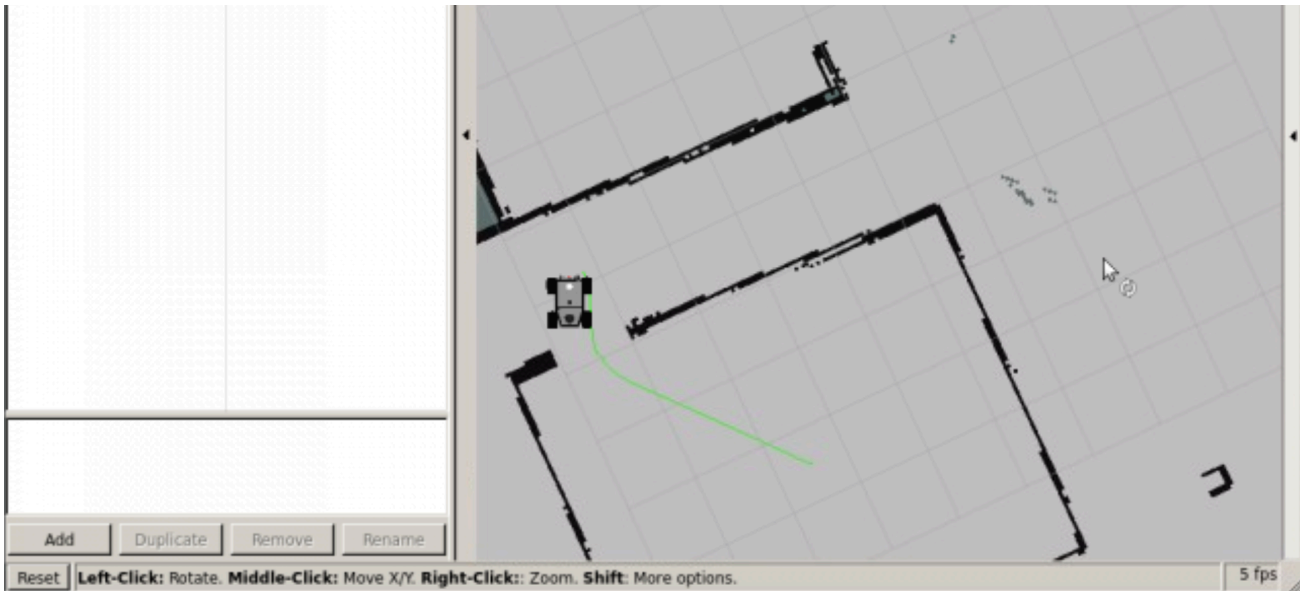
Once you think your parameter files are correct, launch the node and test that the path planning works fine. Check that the robot can plan trajectories and navigate to different points in the environment.

Room 1:

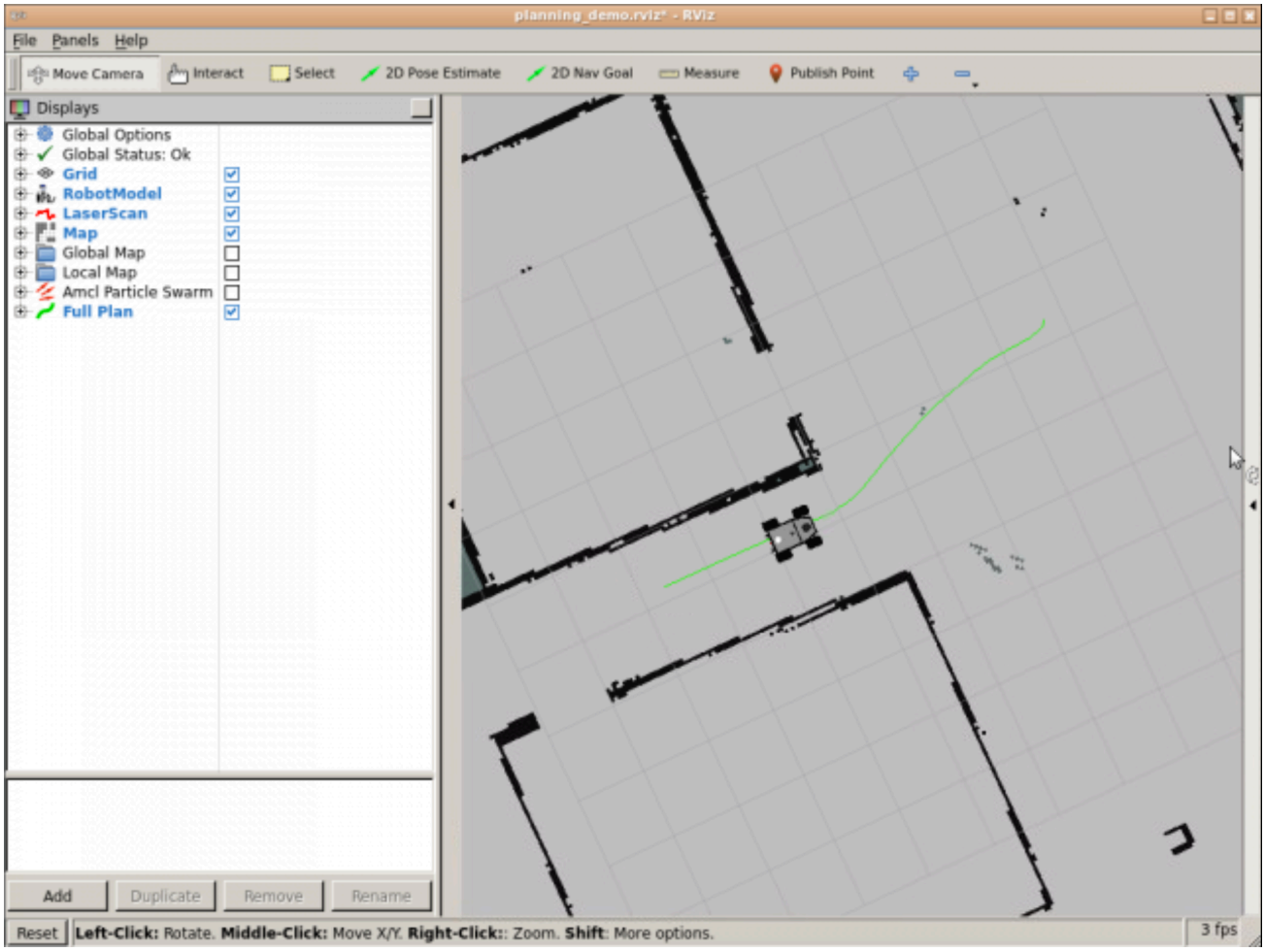


Room 2:





Main Hall:



Step 4: Create a ROS program that interacts with the Navigation

Stack

In Step 2 of this project, you created a table with 3 different spots, each one associated with a label. You may have been wondering why you were asked to do this, right? Well, now you'll have the answer!

In this final step, you will have to create a ROS node that will interact with the navigation stack. This node will contain a service that will take a string as input. This string will indicate one of the labels that you selected in Step 2 for the different spots in the simulation. When this service receives a call with the label, it will get the coordinates (position and orientation) associated with that label, and will send these coordinates to the move_base action server. If it finished correctly, the service will return an "OK" message.

Summarizing, you will have to create a node that contains:

- A service that will take a string as input. Those strings will be the labels you selected in Step 2 for each one of the spots.
- An action client that will send goals to the move_base action server.

In order to achieve this, follow the next steps:

1. Create the package

Create the package called *my_summit_navigation* that will contain all of the files related to this project. Remember to specify the **rospy** dependency.

2. Create the service message

First of all, you'll have to determine the kind of data you need for your message.

- Determine what input data you need (**request**)
- Determine what data you want the service to return (**response**)

Next, see if there is an already-built message in the system that suits your needs. If there isn't, then you'll have to create your own custom message with the data you want.

If this is the case, then do the following. Inside the package that you just created, create a new directory named **srv**. Inside this directory, create a file named **MyServiceMessage.srv** that will contain the definition of your service message.

This file could be something like this:

In [ ]:
```
# request
string label
---
#response
bool navigation_successfull
string message
```

Now, you'll have to modify the **package.xml** and **CMakeLists.txt** files of your package in order to compile the new message. Assuming the only dependency you added to your package was **rospy**, you should do the following modifications to these files:

In CMakeLists.txt

You will have to edit/uncomment four functions inside this file:

- find_package()
- add_service_files()
- generate_messages()
- catkin_package()

In [ ]:
```
find_package(catkin REQUIRED COMPONENTS
  rospy
  std_msgs
  message_generation
)
```

add_service_files(FILES MyCustomServiceMessage.srv)

generate_messages(DEPENDENCIES std_msgs)

In [ ]:
```
catkin_package(
      CATKIN_DEPENDS
      rospy
)
```

In package.xml:

Add these 2 lines to the file:

In [ ]:
```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

When you've finshed with the modifications, compile your package and check that your message has been correctly compiled and is ready to use. In order to check this, you can use the command ***rossrv show MyServiceMessage***.

3. Create the Service Server.

Once your message is created, you're ready to use it in your service! So, let's write the code for our service. Inside the src directory of your package, create a file named **get_coordinates_service_server.py**. Inside this file, write the code necessary for your service.

4. Create the Action Client

Inside the src directory, create a file named **send_coordinates_action_client.py** that will contain the code of an action client.

Once you've finished writting your code, it's time to test it! Call your service by providing one of the labels, and check that the Summit robot navigates to that spot. For example, if your label is "table," type the following into a WebShell:

In [ ]:
```
rosservice call /get_coordinates "label: 'table'"
```

5. Create a launch file that manages everything.

As you may have noticed in the previous step, you need to have the **move_base** node running in order to make everything work. So, let's create a launch file that starts the **move_base** node alongside with the Service Server you have just created above.

Solutions

Please Try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

Follow this link to open the solutions for the Navigation Project:Navigation Project Solutions (extra_files/nav_project_solutions.ipynb)