



The Construct

ROS NAVIGATION IN 5 DAYS

Téllez | Ezquerro | Rodríguez

ROS NAVIGATION IN 5 DAYS

www.theconstructsim.com

ENTIRELY PRACTICAL ROBOT OPERATING SYSTEM TRAINING

ROS NAVIGATION IN 5 DAYS

Ricardo Téllez

Alberto Ezquerro

Miguel Angel Rodríguez



The Construct Sim, Gran Vía 608, 3-D 08007 Barcelona SPAIN,
+34 687 672 123, info@theconstructsim.com www.theconstruct.ai

Written by Miguel Angel Rodríguez, Alberto Ezquerro and Ricardo Téllez

Edited by Yuhong Lin and Ricardo Téllez

Cover design by Lorena Guevara

Learning platform implementation by Ruben Alves

Version 3.0

Copyright © 2019 by The Construct Sim Ltd.

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law. For permission requests, write to the publisher, addressed "Attention: Permissions Coordinator," at the address below.

The Construct Sim, Gran Vía 608, 3-D 08007 Barcelona SPAIN,
+34 687 672 123, info@theconstructsim.com www.theconstruct.ai

Index

Introduction	1
ROSjects	4
Basic Concepts	11
Mapping	31
Robot localization	61
Path planning part 1	79
Path planning part 2. Obstacle avoidance	103
Course Project	134
Final Recommendations	154

Introduction

Introduction

Welcome to the second volume of the **In 5 Days** series of books for learning ROS.

If the first volume was dedicated to learning the basics of ROS (ROS Basics in 5 days), this second volume is dedicated to teach you **how to use ROS for making a robot navigate in an environment**. Navigation for a robot means, being able to move around a dynamic environment where humans and other dynamic obstacles may be present. In this book you are going to learn how to program a ROS based robot for creating maps of the environment, how to localize itself in that map (that is, how to identify its current position in the environment with a location in the map). You will also learn how to do a plan to go from the current location of the robot to a desired location, and move the robot along that path while avoiding all the obstacles that appear in the middle.

Navigation is the most basic skill for a service robot since it allows it to move around and reach places where to help. Because of that, the OSRF (the creators of ROS) have put a special attention at providing a system that allows navigation for any wheeled based robot from moment one. The series of ROS packages created to make possible navigation on a robot is called the **ROS navigation stack**. This book is about it, **how to make it work on a wheeled robot equipped with a laser scan**.



One of the environments you will use in the course

There are other ways of navigation based in cameras, beacons, GPS, etc. However those methods are not going to be explained here. In this book we concentrate on laser based navigation, which is the method used in the navigation stack, and by hence the more straightforward to use.

Even if we only concentrate in explaining the navigation stack, the task is not simple and it will require you 5 days of full work to understand and make it work for your robot.

- On day one, you will learn some basic concepts about navigation and quickly move to learn how to make your robot build a map of the environment using odometry and laser lectures. This is called **mapping**.
- On day two, you will make use of the map created on the previous day to make the robot identify in which part of the map corresponds to its location in the real space. This is called **localization**.
- Then, on day three, four and five you will learn how to make the robot move from one location to another while avoiding obstacles. This is by far the most complicated part of the course and that is why you will need three days to get it right. This part, builds on the previous two and add a ton of new concepts and configurations. You will use the previous map created on day 1 and you will have to make the robot localize in that map as learnt in day 2. On top of that, you will have to configure the **move_base** system that is the one that actually makes the robot move.

As you will see in Chapter 0, the whole book is structured on different chapters with a suggested time of completion. Follow the suggestions of that chapter if you want to maximize your learning experience. But do not stress too much if you cannot follow that rhythm. After all is just a suggestion.

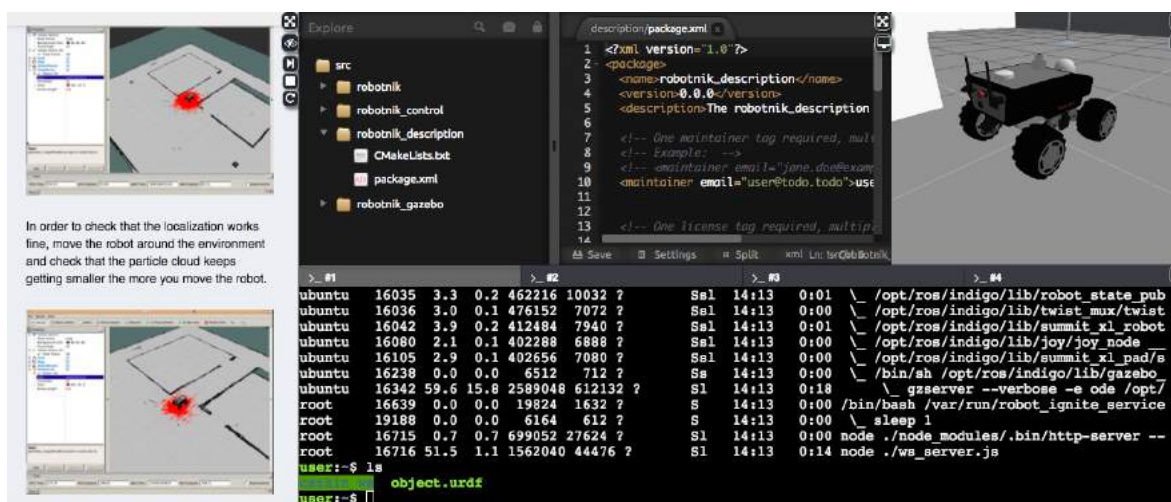
Simulations

You are going to learn all the material using robot simulations. Robot simulations allow you to practice and see the results of your programming in real time, while you are learning and doing your tests. Simulations are also very convenient because those allow us to work with any robot of the world.

So for each chapter of the book you are going to need to launch a simulation and do your exercises with it. For that purpose we have prepared an online platform called the ROS Development Studio (ROSDS): <http://rosds.online>.

Using the ROSDS, you can develop all the exercises of the book and launch the simulations without having to install anything in your computer, being the only requirement to have a web browser. Since you will be developing online with the browser, you can use any type of computer to program ROS and launch simulations.

So go now to the ROSDS and create a free account.



The ROSDS Development Environment

Once you have an account on ROSDS, you will have to open the **ROSjects** that we will provide to you for each Chapter. In the next Chapter, you have a complete guide about **ROSjects** and how to launch them.

We recommend that you perform all the exercises in the ROSDS using the free account, so you can see the results in the simulated robot with minimum hassle.

Regarding the debate about simulations versus real robots, our suggestion is that, once you understand a perception concept, get a real robot and try to apply to it (if you have the chance). Simulations are very good for learning and testing, but nothing replaces the experience of using a real robot.

Now is your turn to start learning. Go for it. Remember that we are here to answer your questions and doubts (contact us at feedback@theconstructsim.com). And remember that you can **become a ROS master** by studying the rest of subjects that can be found in our other books:

- ROS Basics in 5 days
- ROS Perception in 5 days
- ROS Manipulation in 5 days

Keep pushing your ROS learning!

ROSjects

ROSjects

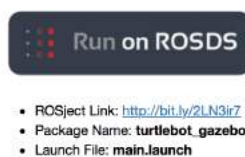
Throughout the whole book, you're going to **find a ROSject at the beginning of each Chapter**. With these ROSjects, you are going to be able to easily have access to all the material you'll need for each Chapter.

What is a ROSject?

A ROSject is, basically, a ROS project in the ROS Development Studio (ROSDS). ROSjects can easily be shared using a link. By clicking on the link, or copying it to the URL of your web browser, you will have a copy of the specific ROSject in your ROSDS workspace. This means you will have instant access to the ROSject. Also, you will be able to modify it as you wish.

How to open a ROSject?

At the beginning of each Chapter, you will see a section like this one:



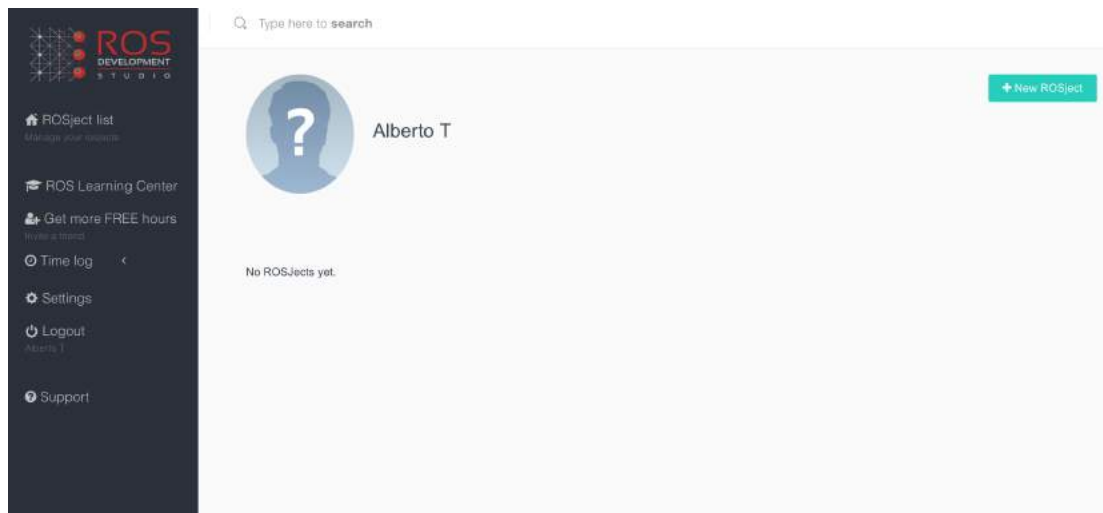
ROSject section

As you can see, it contains 3 things:

- **ROSject Link:** Link to get the ROSject
- **Package Name:** Name of the ROS package that contains the launch file to start the Gazebo simulation.
- **Launch File:** Name of the launch file that will start the Gazebo simulation related to the ROSject.

Step 1

Log into the ROSDS platform at <http://rosds.online> . If you don't have an account, you can create one for free. Once you log in, you will see an screen like the below one.

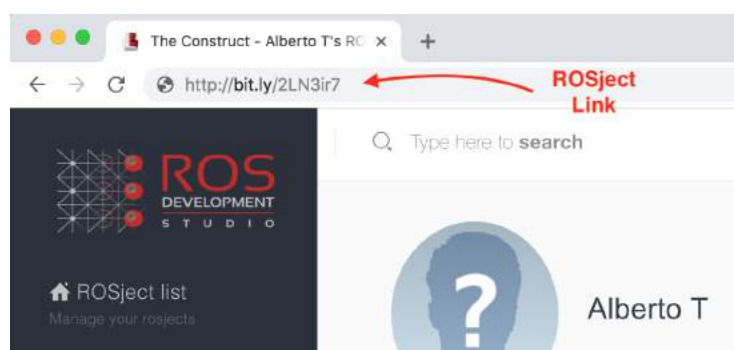


ROSjects Empty List

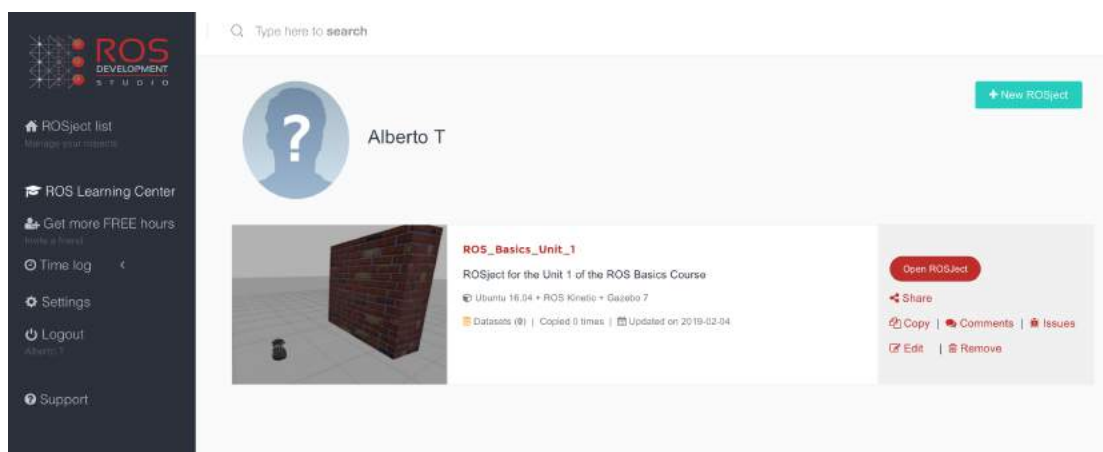
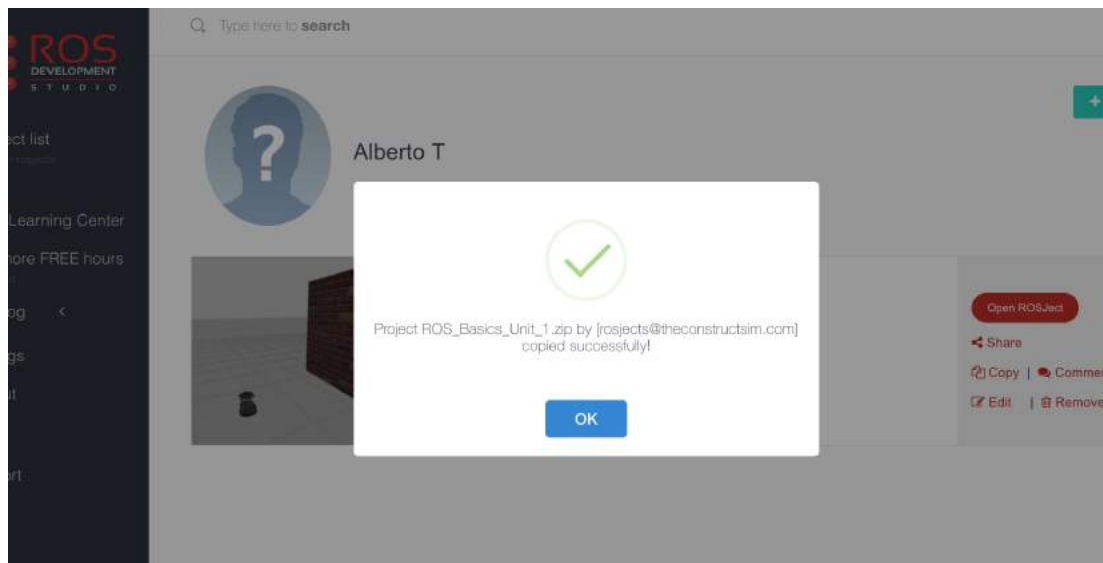
At this point you don't have any ROSject yet. So let's get one!

Step 2

Copy the ROSject Link to your web browser. Once you have copied the URL to your web browser, you will automatically have that ROSject available in your workspace.



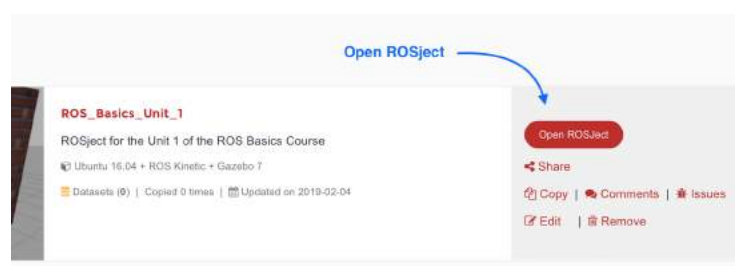
ROSject Link



ROSject in ROSDS workspace

Step 3

Open the ROSject. You can open the ROSject by clicking on the **Open ROSject** button.

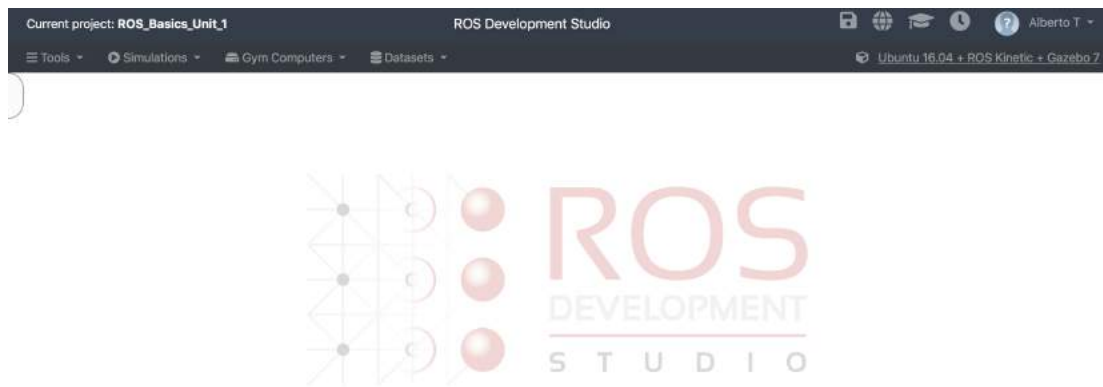


Open ROSject

You will then go to a loading screen like the below one.



After a few seconds, you will get an environment like the below one.



ROSDS Environment

Contents of the ROSjects

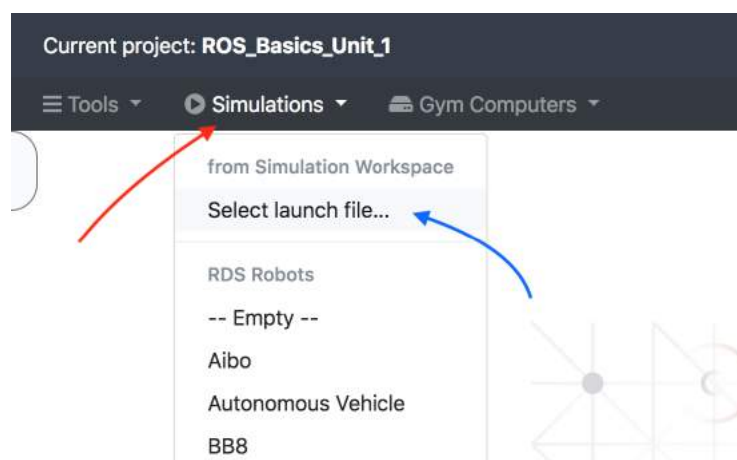
The ROSjects that are available for each chapter of the book will basically contain 2 things:

- The Gazebo simulation used for the Chapter
- All the scripts and files used for the Chapter

In order to open the simulation, follow the next steps:

Step 1

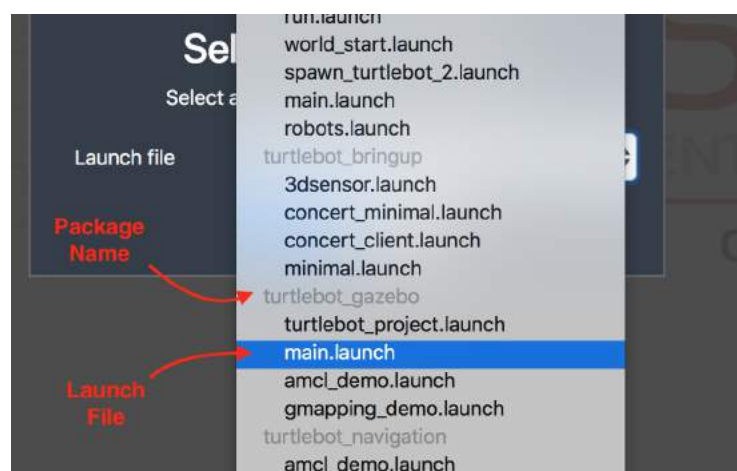
In the **Simulations** menu, click on the **Select launch file...** option. A list with all the packages and launch files available will appear.



Simulations menu

Step 2

Within the list, select the **package name** and **launch file** specified at the ROSject section of the chapter.

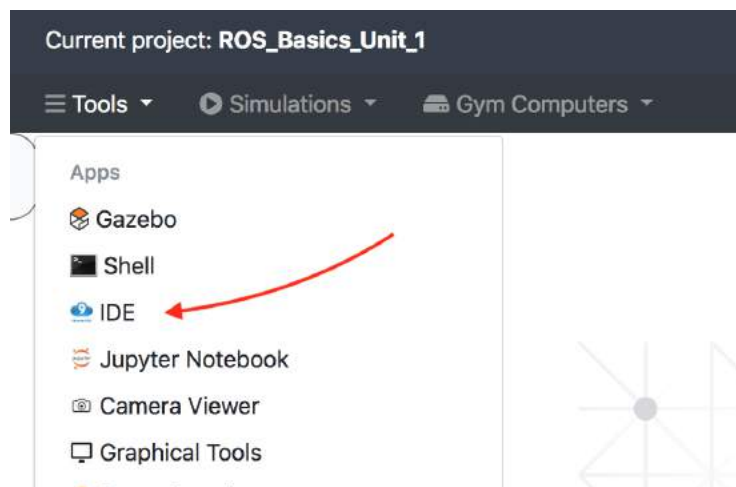


Launch Files list

In order to see all the files related to the chapter, follow the next steps:

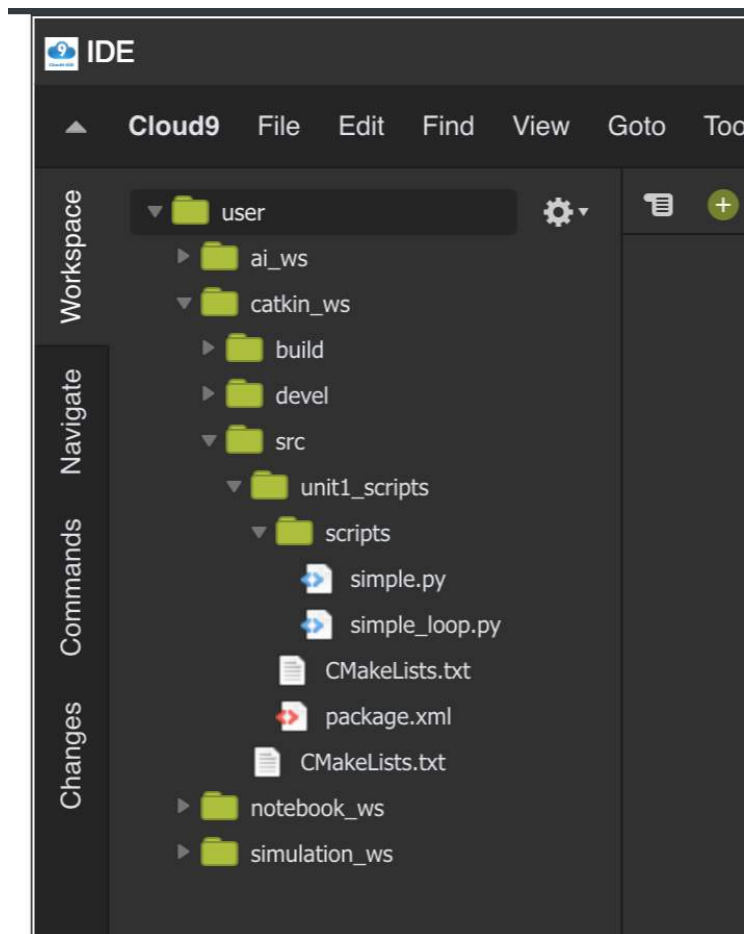
Step 1

In the **Tools** menu, click on the **IDE** option. An IDE will appear in your workspace.



Tools menu

2- You will find all the files inside the **catkin_ws** workspace. There, you have a ROS package containing all the files related to the chapter. This package will always follow the following name convention: ****<unitX>_scripts****



IDE workspace tree

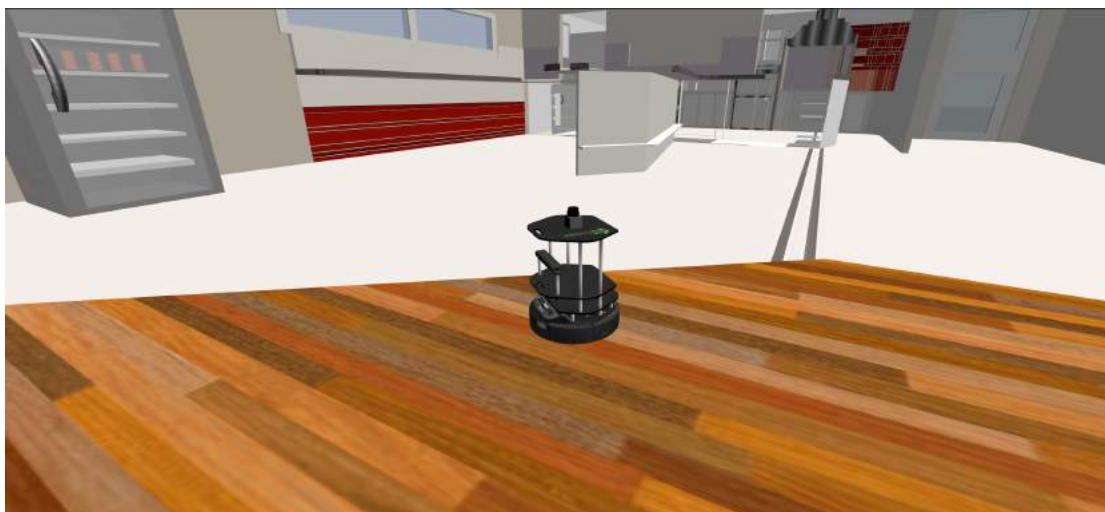
And that's it! Now just enjoy ROSDS and push your ROS learning.

Also, you can find more information and videotutorials about ROSDS here:
<https://www.youtube.com/watch?v=ELfRmuqgxns&list=PLK0b4e05LnzYGvX6EJN1gOQEI6aa3uyKS>
If you have any doubt regarding ROSDS, don't hesitate in contacting us to:
info@theconstructsim.com

Unit 1. Basic Concepts

ROS Navigation in 5 Days

Unit 1: Basic Concepts



Navigation Course Image



- ROSJect Link: <http://bit.ly/2LPJcMX>
- Package Name: **turtlebot_navigation_gazebo**
- Launch File: **main.launch**

SUMMARY

Estimated time of completion: **2 hours**What you will learn with this unit?

- What is the ROS Navigation Stack?
- What do I need to work with the Navigation Stack?
- What is the move_base node and why it is so important?
- Which parts take place in the move_base node?

Let's imagine the next scenario. You're chilling in your lab, listening to your favourite 80s music remix while carrying out a detailed investigation about the last kitten videos that have been uploaded to Youtube... when suddenly, your boss comes into the room and says the magic words: ¡Hey (Insert Your Name Here)! I have a new project for you. I need you to make this robot navigate autonomously with ROS, and I need it for yesterday. Probably you'll start to sweat, while asking yourself questions like: ¿Navigate? ¿For yesterday? ¿With ROS? ¿What the hell do I need? ¿Where do I start? ¿How does BB-8 climb stairs? Don't panic! Keep Calm... Robot Ignite Academy comes to rescue you!!

What do I need to perform robot navigation with ROS?

First, you need a map

The very first thing you need in order to perform Navigation is... of course, a **Map**. But not a treasure Map, or a Map of the state of Wisconsin... No!! You need a Map of the environment where you want your robot to navigate at. Sounds pretty obvious, right? But how could possibly any Robot perform autonomous navigation without providing it with a map of the environment? It just can't. So first thing I need to do is to get a Map of the environment. Great!! But wait... How do I get a this map? Where do I get it from? You use the robot to create it, of course! A Map is just a representation of an environment created from the sensor readings of the robot (for example, from the laser, among others). So just by moving the robot around the environment, you can create an awesome **Map** of it! In terms of ROS Navigation, this is known as **Mapping**. Would you like to see an example of how this works? Then let's do it!

Wait a second! How rude by my part! I didn't even make the proper introductions yet... In the top right corner of your screen you have the simulation window. In this window you will meet different robots during the course (depending on the Chapter you are) that will help you in the process of learning. In this first Chapter (Basic Concepts), you'll be working with the lovely Kobuki located in a cafeteria. I'm sure you'll get along well! So now we've made the proper introductions... let's start building a map with the Kobuki robot!

Example 1.1

First, you need to launch the map builder program. Execute the following command in the WebShell number #1 in order to launch the Mapping demo.

Execute in WebShell #1

```
[ ]: roslaunch turtlebot_navigation_gazebo gmapping_demo.launch
```

Next, you need a program to manually move the robot around, so that you can show the environment to the robot. Execute the following command in the WebShell number #2 in order to launch the keyboard teleoperation program. You will use this program to move the robot by pressing keys in the keyboard, so that you can move the robot around.

Execute in WebShell #2

```
[ ]: roslaunch turtlebot_teleop keyboard_teleop.launch
```

Hit the icon with a screen in the top-right corner of the IDE window



Graphic Interface icon

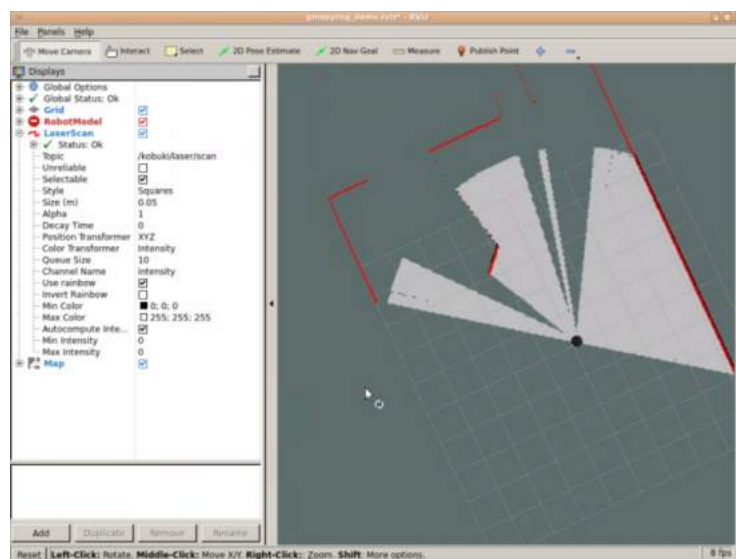
in order to open the Graphic Interface.

Now, execute the following command in the WebShell number #3 in order to start RViz with a predefined configuration.

Execute in WebShell #3

```
[ ]: roslaunch turtlebot_rviz_launchers view_mapping.launch
```

A new window should appear on the Graphic Interface tab containing the Rviz application.



Turtlebot laser scans for map building in Rviz

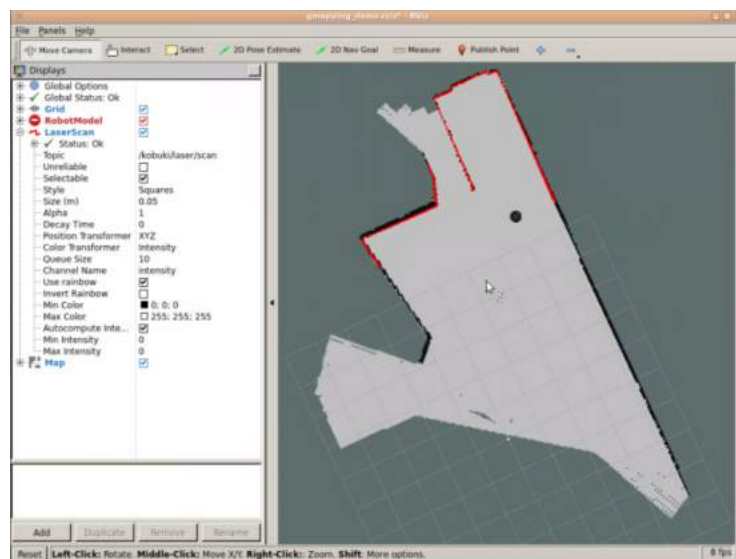
Start moving the Robot around the cafeteria using the keyboard teleop program. Remember that in order to move the robot you need to have the focus of the browser on the WebShell #2 (the one that is executing the keyboard teleop program). Once you start moving the robot, you will see how the map is created on the Graphic Interface tab.

The basic keys in order to move the robot with the keyboard are the following:

i	Move forward
.	Move backward
j	Turn left
l	Turn right
k	Stop
q	Increase / Decrease Speed
z	

Keys to control the robot

Expected Result for Example 1.1



Kobuki creating the map, in Rviz

Data for Example 1.1

Check the following Notes in order to complete the Example: Note 1: It's not necessary to map the whole room. This exercise is only a demo of the mapping process. Note 2: You can try to figure out what is happening by checking different topics. You can get an idea of the topics involved in the process by looking at the launch files code, or in RViz configuration. Use the webshells for this purpose. Note 3: You can change Rviz configuration as you want, and check how it affects the visualization of the mapping process.

Awesome, right? You'll probably have a lot of questions about how this whole process works... but remember this is just the Basic Concepts Unit. For now, you just need to get a general picture of it. You'll learn how the whole process works and how to apply this to your own robot on the **Mapping Unit (Chapter 3)**. Just be patient!

Next you need to localize the robot on that map

So now, let's move on. You've seen that you need a Map in order to Navigate autonomously with your robot, but is it enough? What do you think? As you may imagine, the answer is NO. You have a Map of the environment, yes, but this is completely useless if your robot doesn't know **WHERE it is with respect to this map**. This means, in order to perform a proper Navigation, your robot needs to know in which **position** of the Map it is located and with which **orientation** (that is, which direction the robot is facing) at every moment. In terms of ROS Navigation, this is known as **Localization**. Let's see a quick demonstration of localization.

Let's see how the Kobuki robot can localize itself in the map we previously built.

Example 1.2

IMPORTANT: Make sure to stop all the previous programs running in your Web Shells (by pressing Ctrl+C) before starting with this example.

Execute the following command in the WebShell number #1 in order to start the Localization demo.

Execute in WebShell #1

```
[ ]: roslaunch turtlebot_navigation_gazebo amcl_demo.launch
```

Execute the following command in the WebShell number #2 in order to launch the keyboard teleop program.

Execute in WebShell #2

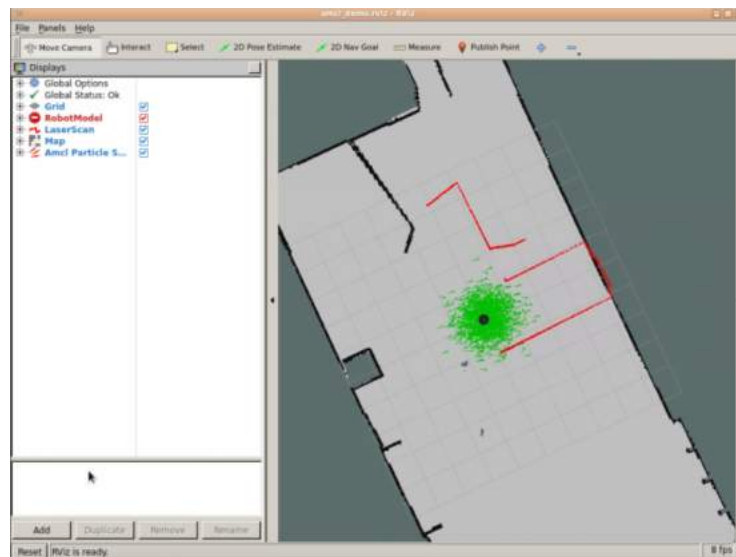
```
[ ]: roslaunch turtlebot_teleop keyboard_teleop.launch
```

Execute the following command in the WebShell number #3 in order to start RViz with a predefined configuration.

Execute in WebShell #3

```
[ ]: roslaunch turtlebot_rviz_launchers view_localization.launch
```

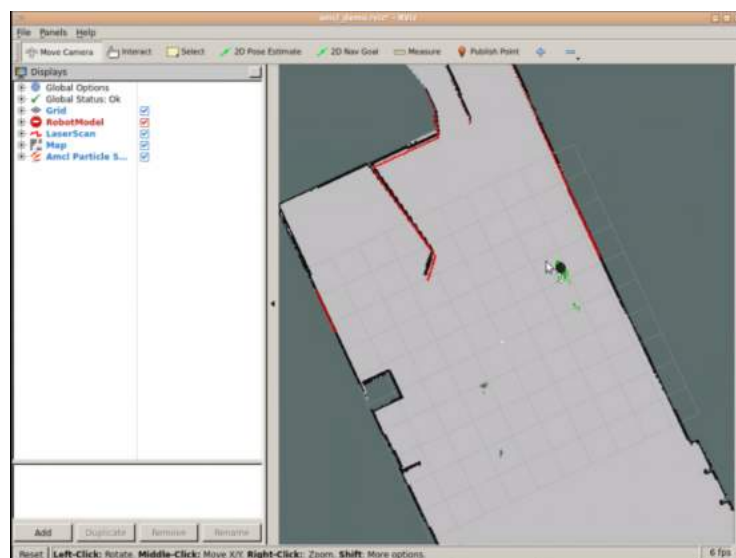
Start moving the Robot around the cafe using the keyboard teleop. Remember that in order to move the robot you need to have the focus of the browser on the WebShell #2 (the one that is executing the keyboard control program). You should see on the Rviz window the map of the cafeteria, a representation of the Kobuki on that map, and a lot of green arrows. Those green arrows represent location guesses of the robot in the map. That is, the green arrows are guesses that the localization algorithm is doing in order to figure out where in the map the robot is located. The arrows will concentrate on the most likely location when you move the robot. Let's try this.



Kobuki tries to localize on the map, in Rviz

Start moving the Robot around the cafe using the keyboard teleop program and see how the green arrows modify their position on the map, adjusting to the actual location of the robot in the map.

Expected Result for Example 1.2



Kobuki correctly localized, in Rviz

Data for Example 1.2

Check the following Notes in order to complete the Example: Note 1:: You can try to figure out what is happening by checking different topics. You can get an idea of the topics involved in the process by looking at the launch files code, or in RViz configuration.. Note 2:: You can change Rviz configuration as you want, and check how it affects the visualization of the localization process. Note 3:: You can help the robot localize by providing it its location on the map. To do that, go to

the Rviz app. Then press the button 2d Pose Estimate and go to the map and drag and drop at the location the robot is (more or less). The green arrows will be spread around that location. I bet you're getting thrilled about this whole thing, right? But I'm also sure you still have more questions. Remember, this is just the Basic Concepts Unit. In the **Localization Unit (Chapter 3)** you will learn how to configure the localization system for your robot, and how to get information from it.

Now you can send goal locations to the robot

Thought you're done? Not at all! You still need a couple of things in order to perform ROS Navigation. For now we've already covered the first block, which is building our own **Map** of the environment and being able to **Localize** the robot in it. So what's next? We should start Navigating autonomously in it, right? For this, we'll need some kind of system which tells the robot **WHERE** to go, at first, and **HOW** to go there, at last. In ROS, we call this system the **Path Planning**. The Path Planning basically takes as input the current location of the robot and the position where the robot wants to go, and gives us as an output the best and fastest path in order to reach that point. Let's see an example of how this works.

Example 1.3

IMPORTANT: Make sure to stop all the programs running in your Web Shells (by pressing Ctrl+C) before starting with this example.

Execute the following command in the WebShell number #1 in order to start the Path Planning demo.

Execute in WebShell #1

```
[ ]: roslaunch turtlebot_navigation_gazebo move_base_demo.launch
```

Execute the following command in the WebShell number #2 in order to start RViz with a predefined configuration.

Execute in WebShell #2

```
[ ]: roslaunch turtlebot_rviz_launchers view_planning.launch
```

Use the 2D Pose Estimate tool in Rviz to Localize the Robot. There are two ways of localizing the robot in the map: one is moving the robot around until it localizes itself (as we did in the previous exercise) and another one is to provide the location to the algorithm manually (because we as humans, we know where the robot is located in the map). The second approach is a lot faster and it is the one we are going to do now.

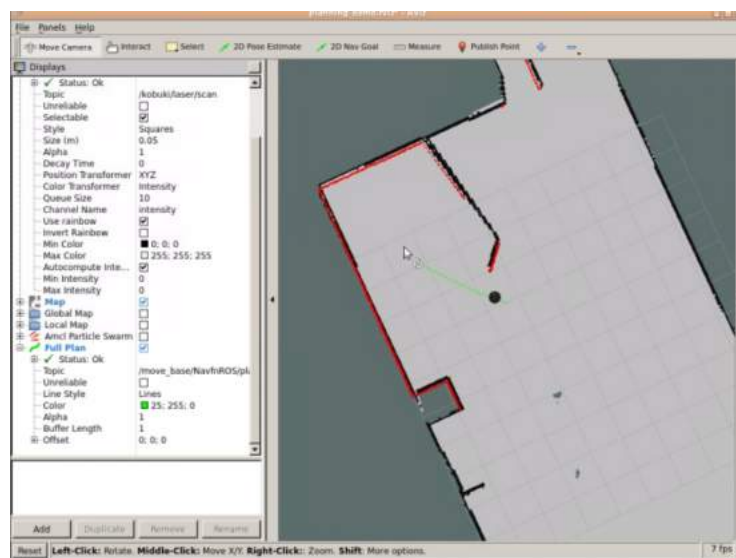
Press the "2D Pose Estimate" button at the top menu in RViz. Then, in the RViz central panel, press in the position in the map where the robot is. Indicate also the orientation of the robot, by clicking, dragging and dropping in the direction of the robot orientation.

Use the 2D Nav Goal tool in Rviz to send a Goal to the Robot.

Press the “2D Nav Goal” button at the top menu in RViz. Then, in the RViz central panel, press in the position in the map where you want your robot to go. Indicate also the orientation at the end of the movement.

At this point, you should see a green line appearing in the Rviz window, on top of the map. That is the path calculated to go from the current location to the goal location. Also, the robot will start moving towards that goal (check the simulation window).

Expected Result for Example 1.3



Kobuki plans a trajectory (in green), in Rviz

Data for Example 1.3

Check the following Notes in order to complete the Example: Note 1:: The 2D Pose Estimate allows you to tell the system what's the initial position and orientation of the robot. The robot needs to be localized in the map, in order to be able to receive a location to go (Navigation Goal). Note 2:: The 2D Nav Goal allows you to send to the robot the position in the map where you want it to navigate to.** Note 3:: In both cases, after setting the position by pressing in RViz's central panel map, you'll have to set up the orientation as well, by pointing the green arrow that appears to the correct direction (drag and drop).

I'm sure you're quite impressed right now, am I right? Sure!! But how does it know the best Path to follow? Maybe it uses the Map we created previously in order to calculate this Path? Hmmm... don't go so fast, buddy! As I told you before, we also have a Unit we're we will discuss largely about all this topics, and that's no other than the **Path Planning Part 1 (Chapter 4)**. On that unit you will learn how to set up a path planner for your own robot and how to ask to it for destinations and status using your own programs.

Finally, you need to avoid obstacles

Anyway, there is actually a question that it is relevant for the next topic we're going to introduce... HOW does ROS manage to avoid, for instance, a table in the environment? What happens if someone or something suddenly walks into the path of the robot? Will the robot know it's there? Will it be able to avoid that obstacle? All of these questions are related to the same topic, which is called **Obstacle Avoidance**. Basically, the Obstacle Avoidance system breaks the big picture (Map) into smaller pieces, which updates in real-time using the data it's getting from the sensors. This way, it assures it won't be surprised for any sudden change in the environment, or by any obstacle that appears in the way. Let's see an example of how this works.

Example 1.4

IMPORTANT: Make sure to stop all the programs running in your Web Shells (by pressing Ctrl+C) before starting with this example.

Execute the following command in the WebShell number #1 in order to start the Obstacle Avoidance demo.

Execute in WebShell #1

```
[ ]: roslaunch turtlebot_navigation_gazebo move_base_demo.launch
```

Execute the following command in the WebShell number #2 in order to copy to your workspace the URDF file of the obstacle we are going to spawn into the simulation.

Execute in WebShell #2

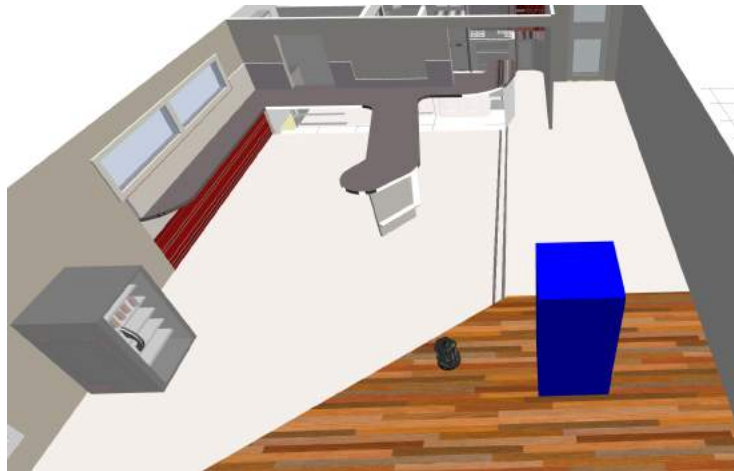
```
[ ]: cp /home/simulations/public_sim_ws/src/all/turtlebot/turtlebot_navigation_gazebo/urdf/object.urdf /home/user/catkin_ws/src
```

Execute the following command in the WebShell number #2 in order to spawn the obstacle in the room.

Execute in WebShell #2

```
[ ]: rosrun gazebo_ros spawn_model -file
    /home/user/catkin_ws/src/object.urdf -urdf -x 0 -y 0 -z 1 -model
    my_object
```

If everything goes fine, you should see a blue box spawning into the simulation. Like this:



Blue object spawned in the simulation

Execute the following command in the WebShell number #2 in order to start RViz with a predefined configuration.

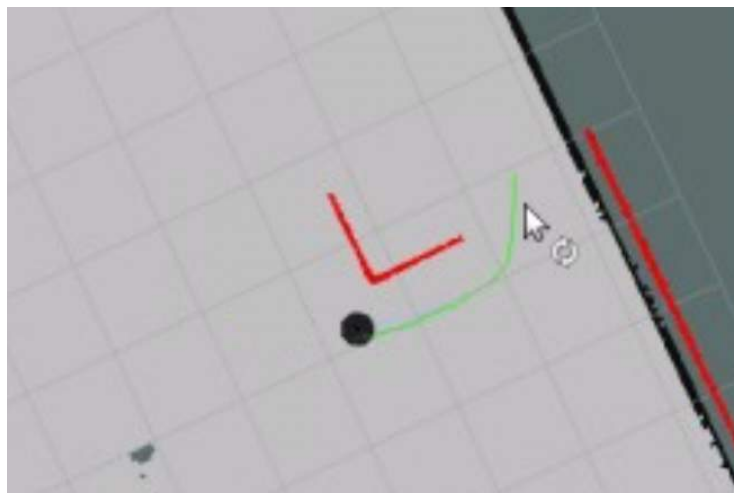
Execute in WebShell #2

```
[ ]: roslaunch turtlebot_rviz_launchers view_planning.launch
```

Use the 2D Pose Estimate and 2D Nav Goal tools in RViz (as in the previous example) in order to tell the robot WHERE it is and WHERE you want it to go. Provide to the robot a location that enforces the robot to avoid the obstacle to check how the system works. Set a Pose Goal where the robot has to somehow avoid the obstacle you've just spawned into the simulation.

Expected Result for Example 1.4

Blue box detected by the laser of the Kobuki robot:



Kobuki creates a trajectory to avoid the obstacle (in green)

Data for Example 1.4

Check the following Notes in order to complete the Example: Note 1: The command to spawn the object in the simulation is just a way to introduce a new obstacle to the simulation. It has NOTHING to do with Obstacle Avoidance. Included here for teaching purposes only. If you want to learn more about how to affect the simulation, enroll our course **Gazebo in 5 days**. Note 2: Use the 2D Nav Goal to send the robot to different places of the Caffe. Select a destination point that requires the robot to avoid the newly inserted object, so you can see how the path is being modified by the obstacle and how the robot is avoiding it and not colliding with it.

IMPORTANT: When you have finished with this exercise, make sure to remove again the object inserted by using the following command:

```
[ ]: rosservice call /gazebo/delete_model "model_name: 'my_object'"
```

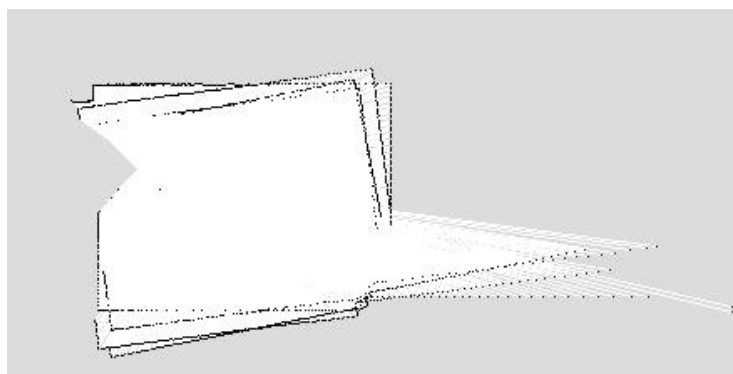
At this point we both know what you are thinking, and what I'm going to say... so let's make it easy. See you back in **Path Planning Part 2 (Chapter 5)** with more details about obstacle avoidance!

So... what do you think? By now, you've already gone through different examples of each one of the parts involved in ROS Navigation. But... Do you think this whole system would work right out of the box? For any kind of robot? The answer is **NO**. It won't.

In order to have the ROS Navigation Stack working properly, you need to provide some data to the system, depending on which is the robot you want to use and how it is built. That is, you need to have your robot properly configured.

Robot Configuration is extremely important in all the navigation modules. For instance, in the Mapping system, if you don't tell the system WHERE does your robot have the laser mounted on, which is the laser's orientation, which is the position of the wheels in the robot, etc., it won't be able to create a good and accurate Map. And as you may already know at this point, **if we don't have a good Map in ROS Navigation, we have NOTHING!**

The following image is an example of how a Map file would look like, if the laser of the robot is not properly configured:



Kobuki incorrectly creates the map

How to Configure your Robot

Robot configuration and definition is done in the **URDF** files of the robot. URDF (Unified Robot Description Format) is an XML format that describes a robot model. It defines its different parts, dimensions, kinematics, dynamics, sensors, etc. . .

Every time you see a 3D robot on ROS, a URDF file is associated with it.

For instance, let's have a look at how the laser of the Kobuki robot is defined in the URDF file of the robot:

```
[ ]: <joint name="laser_sensor_joint" type="fixed">
    <origin xyz="0.0 0.0 0.435" rpy="0 0 0"/>
    <parent link="base_link"/>
    <child link="laser_sensor_link"/>
</joint>

<link name="laser_sensor_link">
    <inertial>
        <mass value="1e-5"/>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0"
izz="1e-6"/>
    </inertial>
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <box size="0.1 0.1 0.1"/>
        </geometry>
    </collision>
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <mesh
filename="package://hokuyo/meshes/hokuyo.dae"/>
        </geometry>
    </visual>
</link>
```

IMPORTANT: The XML code shown above is just a section of the URDF file that defines the Kobuki robot. In this case, it's the section where the laser is defined.

As you can see, it's defining several things regarding the laser:

- It defines the position and orientation of the laser regarding the base ("base_link") of the robot.
- It defines inertia related values

- It defines collision related values. These values will provide the real physics of the laser.
- It defines visual values. These values will provide the visual elements of the laser. This is just for visualization purposes.

These files are usually placed into a package named `**yourrobot_description**`.

In this Course, though, you won't be covering URDF files. I've introduced you to them because it is important that you know about how robot configuration and definition is handled in ROS, but you won't go any further. If you are interested in learning more about this topic, you can have a look at the **Robot Creation with URDF** Course of the **Robot Ignite Academy**.

But wait!! Don't get too excited yet. You may not be covering this topic now, but you will still have to handle other configuration issues during this Course. I'm talking about the configuration of the many parameters that take place during the Navigation process. These parameters will allow you to modify the behavior of the different phases involved in the Navigation process (such as Mapping, Localization, etc.). But do not worry for now, you'll learn to configure the navigation system while you progress through the Chapters of this Course, as well as to use some handy configuration tools.

For now, just remember this one thing: "With great Power comes great Responsibility". Oops!! I think I'm starting to mix things... I ment: "**With great Configuration comes great Navigation**".

So... what do you think? You've already seen (overviewed) the main Basic Concepts you need to know about ROS Navigation. And you may be tempted to think you already know everything you need in order to impress your boss... but let me tell you that you're very **WRONG**. You still have a lot to learn, my young Padawan.

This was just an **introduction** to the different Units you're going to see during this course, but it's just a glimpse. You haven't even finished with the Basic Concepts Chapter!! So stop daydreaming and get back to work!

The Navigation Stack

During the previous examples, you've been launching many different ROS nodes. Each one of these nodes launched their own ROS programs in order to execute different tasks. And all of these was contained, as always happens in ROS, in packages. But... where did all of these packages came from? What are they? Are they connected between them or are they totally independent?

As you may have already figured out, you've been using what's known in ROS as **The Navigation Stack**.

The Navigation Stack is a **set of ROS nodes and algorithms** which are used to autonomously move a robot from one point to another, avoiding all obstacles the robot might find in its way. The ROS Navigation Stack comes with an implementation of several navigation related algorithms which can help you perform autonomous navigation in your mobile robots.

The Navigation Stack will **take as input** the **current location of the robot**, the **desired location the robot wants to go (goal pose)**, the **Odometry data** of the Robot (wheel encoders, IMU, GPS...) and **data from a sensor such as a Laser**. In exchange, it will **output** the necessary **velocity commands** and send them to the mobile base in order to **move the robot to the specified goal position**.

Summarizing, we can say that **the main objective of the Navigation Stack is to move a robot from a position A to a position B, assuring it won't crash against obstacles, or get lost in the process**.

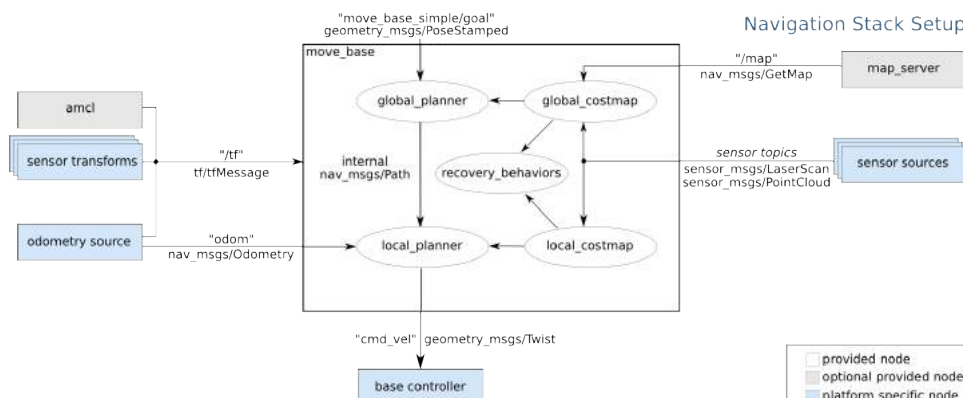
Hardware Requirements

The ROS Navigation Stack is generic. That means, it can be used with almost any type of moving robot, but there are some hardware considerations that will help the whole system to perform better, so they must be considered. These are the requirements:

- The Navigation package will work better in differential drive and holonomic robots. Also, the mobile robot should be controlled by sending velocity commands in the form: **x, y (linear velocity) z (angular velocity)**
- The robot should mount a planar laser somewhere around the robot. It is used to build the map of the environment and perform localization.
- Its performance will be better for square and circular shaped mobile bases.

Following there is a figure with the basic building blocks of the Navigational stack taken from the ROS official website (<http://wiki.ros.org/navigation/Tutorials/RobotSetup>). Maybe this doesn't make much sense to you right now, but by the end of this Course, you will be fully capable to understand this diagram and each block it forms part of it. For now, just try to get a global idea.

Fig.1.1 - Navigation Stack Setup (figure from ROS Wiki)



Navigation Stack Diagram. Image from <http://wiki.ros.org>

According to the shown diagram, we must provide some functional blocks in order to work and communicate with the Navigation stack. Following are brief explanations of all the blocks which

need to be provided as input to the ROS Navigation stack:

- **Odometry source:** Odometry data of a robot gives the robot position with respect to its starting position. Main odometry sources are wheel encoders, IMU, and 2D/3D cameras (visual odometry). The odom value should publish to the Navigation stack, which has a message type of nav_msgs/ Odometry. The odom message can hold the position and the velocity of the robot.
- **Sensor source:** Sensors are used for two tasks in navigation: one for localizing the robot in the map (using for example the laser) and the other one to detect obstacles in the path of the robot (using the laser, sonars or point clouds).
- **sensor transforms/tf:** the data captured by the different robot sensors must be referenced to a common frame of reference (usually the **base_link**) in order to be able to compare data coming from different sensors. The robot should publish the relationship between the main robot coordinate frame and the different sensors' frames using ROS transforms.
- **base_controller:** The main function of the base controller is to convert the output of the Navigation stack, which is a Twist (**geometry_msgs/Twist**) message, into corresponding motor velocities for the robot.

Exercise 1.1

Check that the system you're working in fullfils these requirements. Execute the following command in the WebShell number #1 in order to get a list of all the topics running in the system.

Execute in WebShell #1

```
[ ]: rostopic list
```

Look for the following topics:

```
/cmd_vel
```

```
/kobuki/laser/scan
```

```
/odom
```

```
/tf
```

Can you find them? Are they actives? Great! Each one of this topics has its own functionality in the Navigation process. Can you guess the purpose of each of these topics?

Try to get all the information you can about this topics, since you'll need to know them well.

Execute in WebShell #1

```
[ ]: rostopic info /cmd_vel
      rostopic info /odom
      rostopic info /kobuki/laser/scan
      rostopic info /tf
```

Get also information about the messages they use.

Execute in WebShell #1

```
[ ]: rosmmsg show geometry_msgs/Twist
      rosmmsg show nav_msgs/Odometry
      rosmmsg show sensor_msgs/LaserScan
      rosmmsg show tf2_msgs/TFMessage
```

Expected Result for Exercise 1.1

Info /cmd_vel:

```
user ~ $ rostopic info /cmd_vel
Type: geometry_msgs/Twist

Publishers: None

Subscribers:
 * /gazebo (http://ip-172-31-34-208:48232/)
```

Info cmd vel

Info /odom:

```
user ~ $ rostopic info /odom
Type: nav_msgs/Odometry

Publishers:
 * /gazebo (http://ip-172-31-34-208:48232/)

Subscribers: None
```

Info odom

Info /kobuki/laser/scan:

```
user ~ $ rostopic info /kobuki/laser/scan
Type: sensor_msgs/LaserScan

Publishers:
 * /gazebo (http://ip-172-31-34-208:48232/)

Subscribers: None
```

Info laser

Info /tf:

```
user ~ $ rostopic info /tf
Type: tf2_msgs/TFMessage

Publishers:
 * /robot_state_publisher (http://ip-172-31-34-208:48020/)
 * /gazebo (http://ip-172-31-34-208:48232/)

Subscribers: None
```

Info tf

Twist message:

```
user ~ $ rosmmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Twist Message

Odometry message:

```
user ~ $ rosmmsg show nav_msgs/Odometry
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
  float64[36] covariance
```

Odometry Message

LaserScan message:

```
user ~ $ rosmmsg show sensor_msgs/LaserScan
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

Laser Message

tfMessage message:

```
user ~ $ rosmmsg show tf/tfMessage
geometry_msgs/TransformStamped[] transforms
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/Transform transform
  geometry_msgs/Vector3 translation
    float64 x
    float64 y
    float64 z
  geometry_msgs/Quaternion rotation
    float64 x
    float64 y
    float64 z
    float64 w
```

TF Message

- **/cmd_vel**: Receives the output of the Navigation Stack and transforms the commands into motor velocities.
- **/kobuki/laser/scan**: Provides the Laser readings to the Stack.
- **/odom**: Provides the Odometry readings to the Stack.
- **/tf**: Provides the Transformations to the Stack.

The move_base node

This is the most important node of the Navigation Stack. It's where most of the “magic” happens.

The main function of the **move_base node** is to **move a robot from its current position**

to a goal position with the help of other Navigation nodes. This node links the global planner and the local planner for the path planning, connecting to the rotate recovery package if the robot is stuck in some obstacle, and connecting global costmap and local costmap for getting the map of obstacles of the environment.

Following is the list of all the packages which are linked by the move_base node:

- **global-planner**
- **local-planner**
- **rotate-recovery**
- **clear-costmap-recovery**
- **costmap-2D**

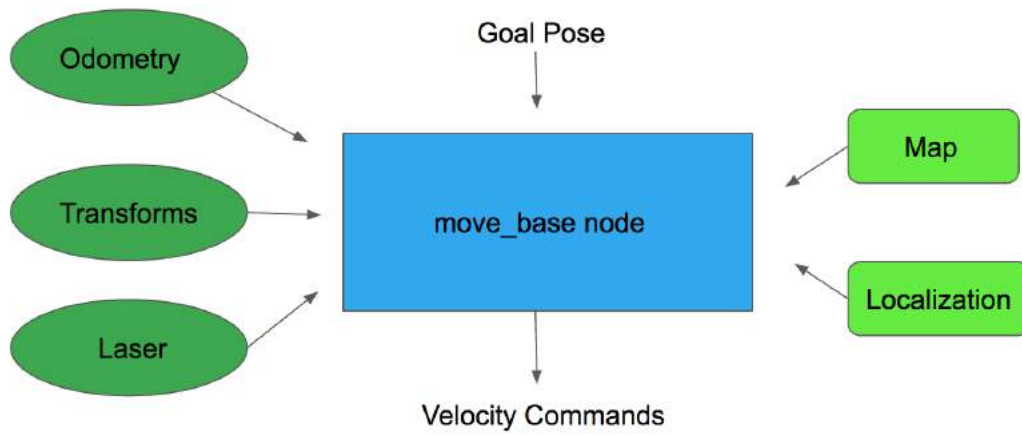
Following are the other packages which are interfaced to the move_base node:

- **map-server**
- **AMCL**
- **gmapping**

Don't panic! Remember this is just an introduction to the Basic Concepts of ROS Navigation. We'll have a deeper look at all this elements in further Units. For now, just try to get a general idea of how the ROS Navigation Stack works. Try to get familiar with some of the names you read, since you'll find out very soon that some of them are very important!

Summary

The Navigation Stack is a set of ROS nodes and algorithms, which work together in order to move a robot from a position A to a position B, avoiding the obstacles it may find in its way. In order to do this, the Navigation Stack requires many data inputs (of different kinds). In exchange, it will give as an output the necessary command velocities in order to safely move the robot to the desired position.

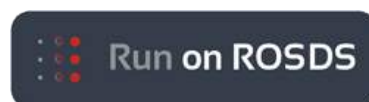


move base high level diagram

Unit 2. Mapping

ROS Navigation

Unit 2: Mapping



- ROSJect Link: <http://bit.ly/2LRTMmA>
- Package Name: **turtlebot_navigation_gazebo**
- Launch File: **main.launch**

SUMMARY

Estimated time of completion: **2 hours** What will you learn with this unit?

- What means Mapping in ROS Navigation
- How does ROS Mapping work
- How to configure ROS to make mapping work with almost any robot
- Different ways to build a Map

If you completed successfully the previous Unit (Basic Concepts), you must now know at least this ONE THING. **In order to perform autonomous Navigation, the robot MUST have a map of the environment.** The robot will use this map for many things such as planning trajectories, avoiding obstacles, etc.

You can either give the robot a prebuilt map of the environment (in the rare case that you already have one with the proper format), or you can build one by yourself. This second option is the most frequent one. So in this Unit, we will basically show you **HOW to create a map from zero.**

Before starting, though, you need to be introduced properly to a very important tool in ROS Navigation. I'm talking, of course, about **RViz**. In the 1st Chapter you already used it to visualize the whole process of ROS Navigation. In this chapter, you're going to learn how to use it in order to visualize the Mapping process.

Visualize Mapping in Rviz

As you've already seen, you can launch RViz and add displays in order to watch the Mapping progress. For Mapping, you'll basically need to use 2 displays of RViz:

- **LaserScan Display**
- **Map Display**

NOTE: There is an optional display that you may want to add to Rviz in order to make things clearer: the **Robot Model**. This display will show you the model of the robot through RViz, allowing you to see how the robot is moving around the map that it is constructing. It is also useful to detect errors.

Follow the steps in the next exercise in order to learn how to add these displays.

Exercise 2.1

a) Execute the following command in order to launch the **slam_gmapping** node. We need to have this node running in order to visualize the map.

Execute in WebShell #1

```
[ ]: roslaunch turtlebot_navigation_gazebo gmapping_demo.launch
```

b) Hit the icon with a screen in the top-right corner of the IDE window



Graphic Interface icon

in order to open the Graphic Interface.

- c) Execute the following command in order to launch Rviz.

Execute in WebShell #2

```
[ ]: rosrn rviz rviz
```

- d) Follow the next steps in order Add a **LaserScan** display to RViz.

Visualize LaserScan 1.- Click the **Add** button under Displays and choose the LaserScan display.

2.- In the Laser Scan display properties, introduce the name of the topic where the laser is publishing its data (in this case, it's **/kobuki/laser/scan**).

3.- In Global Options, change the Fixed Frame to **odom**.

4.- To see the robot in the Rviz, you can choose to also add the **RobotModel** display. This will display the current situation of the robot on the screen. You can also try to display all the reference frames of the robot by adding to Rviz the TF displays.

5.- The laser “map” that is built will disappear over time, because Rviz can only buffer a finite number of laser scans.

Visualize Map

1. Click the Add button and add the Map display.
2. In the Map display properties, set the topic to **/map**.

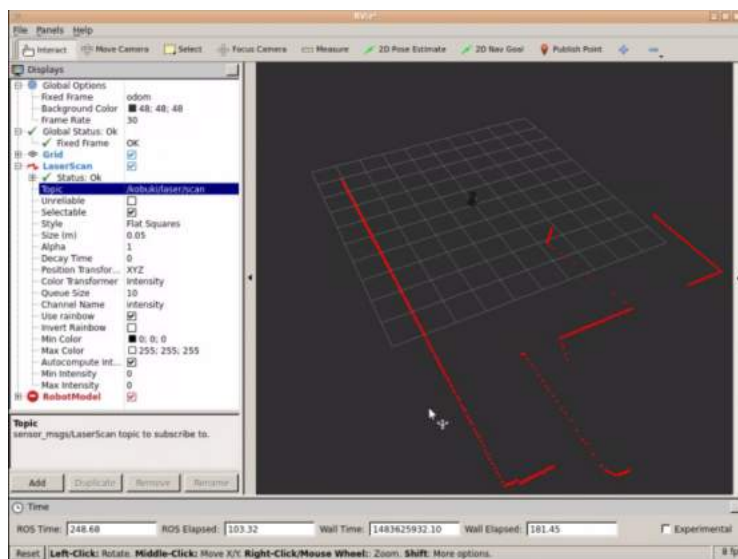
Now you will be able to properly visualize the Mapping progress through RViz

Data for Exercise 2.1

Check the following Notes in order to complete the Exercise: **Note 1:** You can change the way you visualize these elements by modifying some properties in the left panel. **Note 2:** For example, set the size of the Laser scan to 0.05 to make the laser ray thicker.

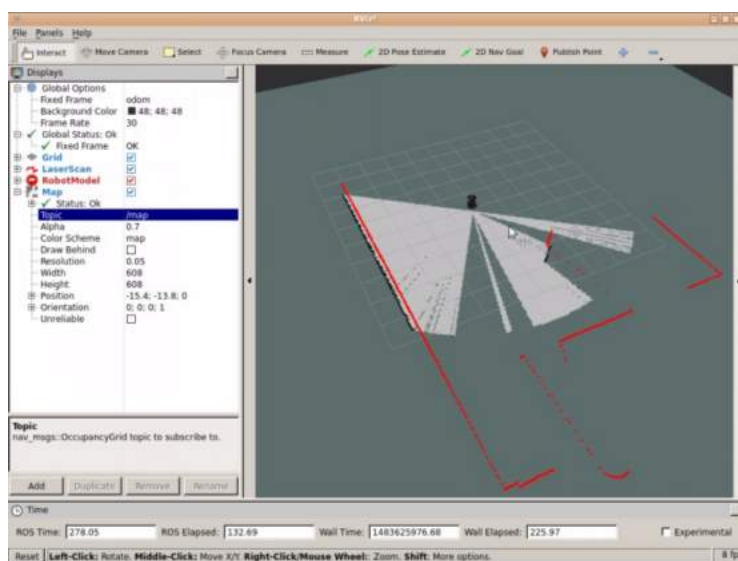
Expected Result for Exercise 2.1

RobotModel plus LaserScan visualization through RViz:



Laser visualization in RViz

RobotModel plus Map and LaserScan visualization through RViz:



Laser and Map visualization in RViz

Now that you already know how to configure RViz in order to visualize the Mapping process, you're ready to create your own Map. But first, let's see one last utility that RViz has and that will be very useful for us. In the exercise above, you've added to RViz various displays in order to be able to visualize different elements in the simulation. You also modified some things in the configuration of these displays. But... if you now close RViz, and relaunch it later, all these changes you've done will be lost, so you'll have to configure it all again. That doesn't sound good at all, right?

Saving RViz configuration

Fortunately, RViz allows you to **save your current configuration**, so you can recover it anytime you want. In order, to do so, follow the next steps:

1. Go to the top left corner of the RViz screen, and open the File menu. Select the Save Config As option.
2. Write the name you want for your configuration file, and press Save.

Now, you'll be able to load your saved configuration anytime you want by selecting the Open Config option in the File menu.

Exercise 2.2

IMPORTANT: Before starting with this Exercise, make sure your Rviz is properly configured in order to visualize the Mapping process.

- a) Execute the following command in order to start moving the robot around the environment.

Execute in WebShell #3

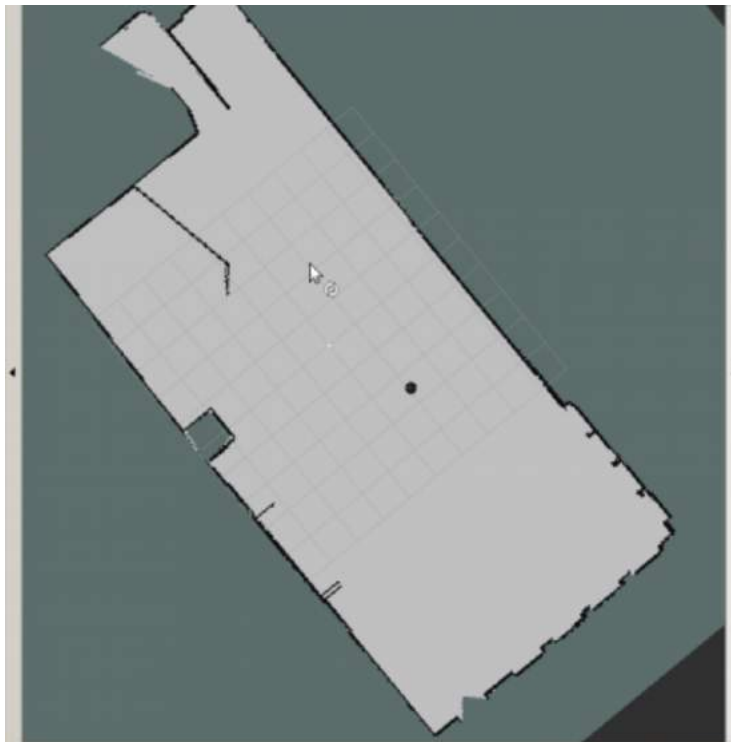
```
[ ]: roslaunch turtlebot_teleop keyboard_teleop.launch
```

- b) Move around the whole room until you have created a **FULL MAP** of the environment.

Data for Exercise 2.2

Check the following Notes in order to complete the Exercise: **Note 1:** Due to the way this simulation environment is built, you won't be able to enter the Kitchen. Don't worry about this issue and keep mapping the rest of the space. **Note 2:** Remember to keep an eye at RViz so you can see if the map is being built properly or if, otherwise, you're leaving some zones without mapping.

Expected Result for Exercise 2.2



Map Completed

Ok so... what has just happened? In order to better understand the whole process, let's first introduce two concepts.

SLAM

Simultaneous Localization and Mapping (SLAM). This is the name that defines the robotic problem of **building a map of an unknown environment while simultaneously keeping track of the robot's location on the map that is being built**. This is basically the problem that Mapping is solving. The next Unit (Localization) is also involved, but we'll get there later.

So, summarizing, **we need to do SLAM in order to create a Map for the robot**.

The gmapping package

The gmapping ROS package is an implementation of a specific SLAM algorithm called gmapping. This means that, somebody has implemented the gmapping algorithm for you to use inside ROS, without having to code it yourself. So if you use the ROS Navigation stack, you only need to know (and have to worry about) how to configure gmapping for your specific robot (which is precisely what you'll learn in this Chapter). The gmapping package contains a ROS Node called **slam_gmapping**, which allows you to create a 2D map using the laser and pose data that your

mobile robot is providing while moving around an environment. This node **basically reads data from the laser and the transforms of the robot, and turns it into an occupancy grid map (OGM)**.

So basically, what you've just done in the previous exercise was the following:

1. You used a previously created configuration launch file (**gmapping_demo.launch**) to launch the **gmapping** package with the Kobuki robot.
2. That launch file started a **slam_gmapping node** (from the gmapping package). Then you moved the robot around the room.
3. Then ,the slam_gmapping node **subscribed to the Laser (/kobuki/laser/scan) and the Transform Topics (/tf)** in order to get the data it needs, and it built a map.
4. The generated map is published during the whole process into the **/map** topic, which is the reason you could see the process of building the map with Rviz (because Rviz just visualizes topics).

The /map topic uses a message type of **nav_msgs/OccupancyGrid**, since it is an OGM. Occupancy is represented as an integer in the range {0, 100}. With 0 meaning completely free, 100 meaning completely occupied, and the special value of -1 for completely unknown.

Amazing, right? Now, you may be worrying that you only had to do a roslaunch in order to have the robot generating the map.

- What if your Kobuki does not have the laser at the center?
- What if instead of a laser, you are using a Kinect?
- What if you want to use the mapping with a different robot than Kobuki?

In order to be able to answer those questions, you still need to learn some things first.

Let's start by seeing what you can do with the Map you've just created.

Saving the map

Another of the packages available in the ROS Navigation Stack is the **map_server package**. This package provides the **map_saver node**, which allows us to access the map data from a ROS Service, and save it into a file.

When you request the map_saver to save the current map, the map data is saved into two files: one is the YAML file, which contains the map metadata and the image name, and second is the image itself, which has the encoded data of the occupancy grid map.

Command to save the map

We can save the built map at anytime by using the following command:

```
[ ]: rosrun map_server map_saver -f name_of_map
```

This command will get the map data from the map topic, and write it out into 2 files, **name_of_map.pgm** and **name_of_map.yaml**.

Exercise 2.3

- a) Save the map created in the previous exercise into a file.

Execute in WebShell #4

```
[ ]: rosrun map_server map_saver -f my_map
```

- b) Go to the IDE and look for the files created, **my_map.pgm** and **my_map.yaml**. Open both of them and check what they contain.

Data for Exercise 2.3

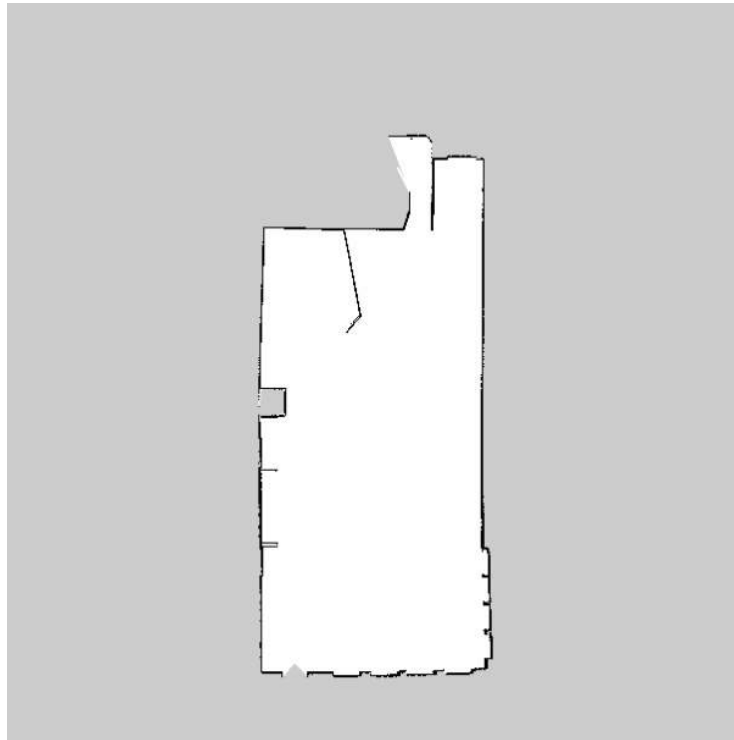
Check the following Notes in order to complete the Exercise: **Note 1:** The -f attribute allows you to give the files a custom name. By default (if you don't use the -f attribute), the names of the file would be map.pgm and map.yaml.

Note 2: Remember that, in order to be able to visualize the files generated through RViz, these files must be at the **/home/user/catkin_ws/src** directory. The files will be initially saved in the directory where you execute the command.

Note 3: You can download the files to your computer by right-clicking them, and selecting the 'Download' option.

Expected Result for Exercise 2.3

Image (PGM) File of the Map:



PGM File of the Map

YAML File of the Map.

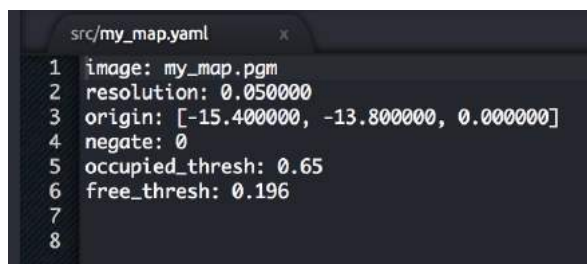
```
image: my_map.pgm
resolution: 0.050000
origin: [-15.400000, -13.800000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

YAML File of the Map

YAML File

In order to visualize the YAML file generated in the previous exercise, you can do one of the following things:

- Open it through the IDE. In order to be able to do this, the file must be in your **catwin_ws/src** directory.
- Open it through the Web Shell. You can use, for instance, the vi editor to do so typing the command **vi my_map.yaml**.
- Download the file and visualize it in your local computer with your own text editor.



```
src/my_map.yaml
1 image: my_map.pgm
2 resolution: 0.050000
3 origin: [-15.400000, -13.800000, 0.000000]
4 negate: 0
5 occupied_thresh: 0.65
6 free_thresh: 0.196
7
8
```

The YAML File generated will contain the 6 following fields:

- **image**: Name of the file containing the image of the generated Map.
- **resolution**: Resolution of the map (in meters/pixel).
- **origin**: Coordinates of the lower-left pixel in the map. This coordinates are given in 2D (x,y). The third value indicates the rotation. If there's no rotation, the value will be 0.
- **occupied_thresh**: Pixels which have a value greater than this value will be considered as a completely occupied zone.
- **free_thresh**: Pixels which have a value smaller than this value will be considered as a completely free zone.
- **negate**: Inverts the colours of the Map. By default, white means completely free and black means completely occupied.

Image File (PGM)

In order to visualize the PGM file generated in the previous exercise, you can do one of the following things:

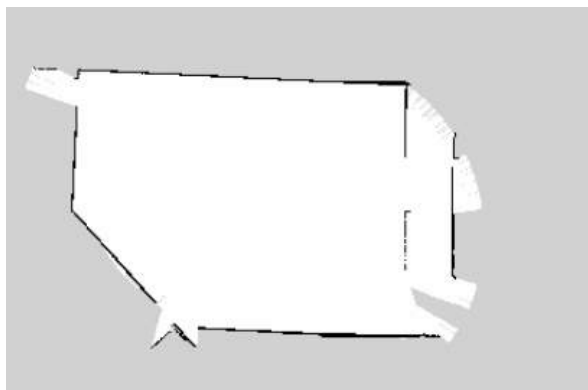
- Open it through the IDE. In order to be able to do this, the file must be in your **catwin_ws/src** directory.
- Open it through the Web Shell. You can use, for instance, the vi editor to do so typing the command **vi my_map.pgm**.
- Download the file and visualize it in your local computer with your own text editor.

What happens? Anything strange? Are you able to understand the file content? I'm sure you are not! If you try to visualize this file with a text editor, you'll get something similar to this:



```
src/my_map.pgm
1 P5
2 # CREATOR: Map_generator.cpp 0.050 m/pix
3 608 608
4 255
5 i
```

But this doesn't give us any useful information. Can you guess what's happening? This happens because this is not a text file, but a PGM (Portable Gray Map) file. A PGM is a file that represents a grayscale image. So, if you want to visualize this file properly, you should open it with an image editor. Then, you'll get something like this:



The image describes the occupancy state of each cell of the world in the color of the corresponding pixel. **Whiter pixels are free, blacker pixels are occupied, and pixels in between are unknown.** Color and grayscale images are accepted, but most maps are gray (even though they may be stored as if in color). Thresholds in the YAML file are used to divide the three categories.

When communicated via ROS messages, occupancy is represented as an integer in the range [0,100], with **0 meaning completely free and 100 meaning completely occupied, and the special value -1 for completely unknown.**

Image data is read in via `SDL_Image`; supported formats vary, depending on what `SDL_Image` provides on a specific platform. Generally speaking, most popular image formats are widely supported.

NOTE: A notable exception is that PNG is not supported on OS X.

Providing the map

Besides the `map_saver` node, the `map_server` package also provides the **map_server node**. This node reads a map file from the disk and provides the map to any other node that requests it via a ROS Service.

Many nodes request to the **map_server** the current map in which the robot is moving. This request is done, for instance, by the `move_base` node in order to get data from a map and use it to perform Path Planning, or by the localization node in order to figure out where in the map the robot is. You'll see examples of this usage in following chapters.

The service to call in order to get the map is:

- **static_map (nav_msgs/GetMap):** Provides the map occupancy data through this service.

Apart from requesting the map through the service above, there are two latched topics that you can connect in order to get a ROS message with the map. The topics at which this node writes the map data are:

- **map_metadata (nav_msgs/MapMetaData):** Provides the map metadata through this topic.
- **map (nav_msgs/OccupancyGrid):** Provides the map occupancy data through this topic.

NOTE: When a topic is latched, it means that the last message published to that topic will be saved. That means, any node that listens to this topic in the future will get this last message, even if nobody is publishing to this topic anymore. In order to specify that a topic will be latched, you just have to set the latch attribute to true when creating the topic.

Command to launch the map server node

To launch the map_server node in order to provide information of a map given a map file, use the following command:

```
[ ]: rosrn map_server map_server map_file.yaml
```

Remember: You must have created the map (the map_file.yaml file) previously with the gmapping node. You can't provide a map that you haven't created!

Exercise 2.4

IMPORTANT: Make sure to stop all the programs running in your Web Shells (by pressing Ctrl+C) before starting with this exercise.

- Launch the map_server node using the command tool shown above. Use the map file you've created in Exercise 2.3.

Execute in WebShell #1

```
[ ]: rosrn map_server map_server my_map.yaml
```

- Get information of the map by accessing the topics introduced above.

Data for Exercise 2.4

Check the following Notes in order to complete the Exercise: **Note 1:** If you launch the command from the directory where you have the map file saved, you don't have to specify the full path to the file. If you aren't in the same directory, bear in mind that you'll have to specify the full path to the file.

- a) Create a package named **provide_map**.
- b) Inside this package, create a launch file that will launch the map_server node in order to provide the map you've created.
- c) Execute your launch file, and check if the map is being provided by accessing the proper topics.

Data for Exercise 2.5

Check the following Notes in order to complete the Exercise: **Note 1:** In order to provide the map file, you'll have to add the path to your map as an argument of the <node> tag.

Expected Result for Exercise 2.5

Echo of the /map_metadata topic:

```
ubuntu@ip-172-31-44-229:~$ rostopic echo /map_metadata
map_load_time:
  secs: 1483988029
  nsecs: 462798163
resolution: 0.0500000007451
width: 608
height: 608
origin:
  position:
    x: -15.4
    y: -13.8
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
---
```

map metadata topic

Echo of the /map topic:

[illegible]

map topic

In the previous exercise you've seen the topics at which the `map_server` node writes in order to provide the map data. But as we said previously, this node also allows to get this data from a service. The service it uses to provide the map data is the following:

- **static_map (nav_msgs/GetMap):** Provides the map occupancy data through this service.

Exercise 2.6

Using a Web Shell, perform a call to the service introduced above in order to get data from the map.

Execute in WebShell #1

```
[ ]: rosservice call /static_map "{}"
```

Expected Result for Exercise 2.6

Service call to /static_map:

[illegible]

Service call to static map

Exercise 2.7

Create a Service Client that calls to this service.

- a) Create a new package named **get_map_data**. Add **rospy** as a dependency.
- b) Inside this package, create a file named **call_map_service.py**. Inside this file, write the code of your Service Client. This Service Client will call the **/static_map** service in order to get the map data, and then it will print the dimensions and resolution of the map in the screen.
- c) Create a launch file for your Service Client, and test that it works properly.

IMPORTANT REMARKS

Having shown how to create a map of an environment with a robot, you must understand the following:

- The map that you created is a **static map**. This means that the map will always stay as it was when you created it. So when you create a Map, it will capture the environment as it is at the exact moment that the mapping process is being performed. If for any reason, the environment changes in the future, these changes won't appear on the map, hence it won't be valid anymore (or it won't correspond to the actual environment).
- The map that you created is a **2D Map**. This means that, the obstacles that appear on the map don't have height. So if, for instance, you try to use this map to navigate with a drone, it won't be valid. There exist packages that allow you to generate 3D mappings, but this issue won't be covered in this Course. If you're interested in this topic, you can have a look at the following link: http://wiki.ros.org/rtabmap_ros/Tutorials/MappingAndNavigationOnTurtlebot

Hardware Requirements

Another thing you should have learnt from the previous Unit, is that **configuration** is VERY IMPORTANT in order to build a proper Map. Without a good configuration of your robot, you won't get a good Map of the environment. And without a good Map of the environment, you won't be able to Navigate properly. So, in order to build a proper Map, you need to fulfill these 2 requirements:

1. Provide Good Laser Data
2. Provide Good Odometry Data

The **slam_gmapping** node will then try to transform each incoming laser reading to the odom frame.

Exercise 2.8

- a) Make sure that your robot is publishing this data and identify the topic it is using in order to publish the data.

b) When you've identified the topics, check the structure of the messages of these topics.

Expected Result for Exercise 2.8

Topics list:

```

/camera/rgb/image_raw/theor
/clock
/cmd_vel
/cmd_vel_mux/active
/cmd_vel_mux/input/navi
/cmd_vel_mux/input/safety_c
/cmd_vel_mux/input/teleop
/cmd_vel_mux/parameter_desc
/cmd_vel_mux/parameter_upda
/gazebo/link_states
/gazebo/model_states
>_#1 /gazebo/parameter_descripti
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/joint_states
/kobuki/laser/scan
/mobile_base/commands/veloc
/mobile_base_nodelet_manage
/odom
/rosout
/rosout_agg
/tf
/tf_static

```

Topics List

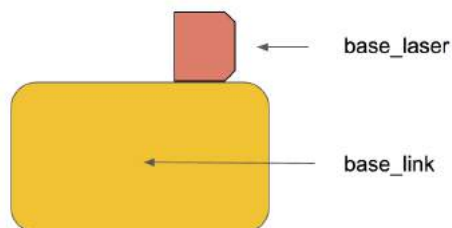
echo /odom:

```

user ~ $ rostopic echo -n1 /odom
header:
  seq: 20254
  stamp:
    secs: 675
    nsecs: 283000000
  frame_id: odom_frame
child_frame_id: base_link
pose:
  pose:
    position:
      x: 0.0
      y: 0.0
      z: 0.0927561574185
    orientation:
      x: 0.0
      y: 0.0
      z: 0.0
      w: 1.0
  covariance: [1e-05, 0.0, 0.0, 0.0,

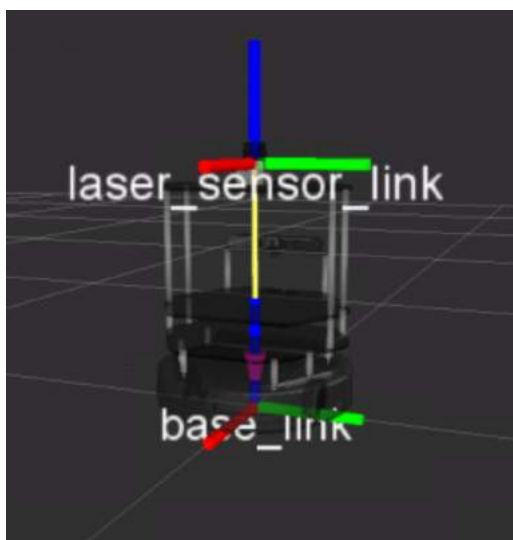
```

Odometry Message



TF Diagram

For instance, in the case of the Kobuki robot that you are using in this Chapter, the frames looks like this:



TFs in RViz

Now, we need to define a relationship (in terms of position and orientation) between the `base_laser` and the `base_link`. For instance, we know that the `base_laser` frame is at a distance of 20 cm in the y axis and 10 cm in the x axis referring the `base_link` frame. Then we'll have to provide this relationship to the robot. **This relationship between the position of the laser and the base of the robot is known in ROS as the TRANSFORM between the laser and the robot.**

For the `slam_gmapping` node to work properly, you will need to provide 2 transforms:

- **the frame attached to laser -> base_link:** Usually a fixed value, broadcast periodically by a `robot_state_publisher`, or a `tf static_transform_publisher`.
- **base_link -> odom:** Usually provided by the Odometry system

Since the robot needs to be able to access this information anytime, we will publish this information to a **transform tree**. The transform tree is like a database where we can find information about all

the transformations between the different frames (elements) of the robot.

You can visualize the transform tree of your running system anytime by using the following command:

```
[ ]: rosrn tf view_frames
```

This command will generate a pdf file containing a graph with the transform tree of your system.

Exercise 2.9

Execute the following commands in order to visualize the transform tree of your system.

Execute in WebShell #1

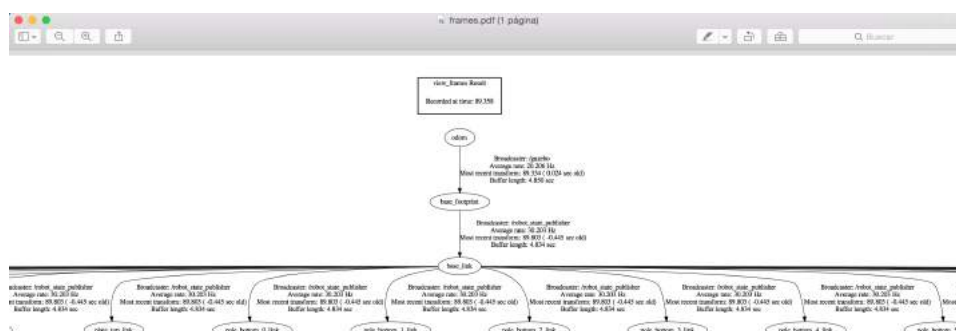
```
[ ]: roscd
    cd ..
    cd src
    rosrn tf view_frames
```

Download this file to your computer using the IDE Download option, and open it. Check if the required transforms of the slam_gmapping node are there.

Data for Exercise 2.9

Check the following Notes in order to complete the Exercise: **Note 1:** You can download the files to your computer by right-clicking them on the IDE, and selecting the **Download** option.

Expected Result for Exercise 2.9



Frames Tree PDF

Now, let's imagine you just mounted the laser on your robot, so the transform between your laser and the base of the robot is not set. What could you do? There are basically 2 ways of publishing a transform:

- Use a **static_transform_publisher**
- Use a **transform broadcaster**

In this Course, we'll use the `static_transform_publisher`, since it's the fastest way. The `static_transform_publisher` is a ready-to-use node that allows us to directly publish a transform by simply using the command line. The structure of the command is the next one:

```
[ ]: static_transform_publisher x y z yaw pitch roll frame_id
    child_frame_id period_in_ms
```

Where:

- **x, y, z** are the offsets in meters
- **yaw, pitch, roll** are the rotation in radians
- **period_in_ms** specifies how often to send the transform

You can also create a launch file that launches the command above, specifying the different values in the following way:

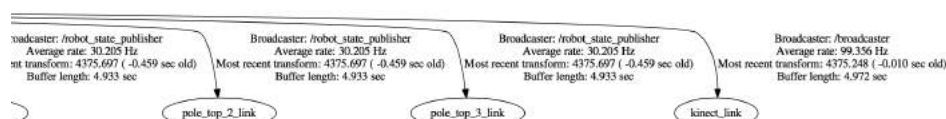
```
[ ]: <launch>
    <node pkg="tf" type="static_transform_publisher"
    name="name_of_node"
        args="x y z yaw pitch roll frame_id child_frame_id
    period_in_ms">
    </node>
</launch>
```

Exercise 2.10

Create a package and a launch file in order to launch a `static_transform_publisher` node. This node should publish the transform between the Kinect camera mounted on the robot and the base link of the robot.

Generate again the frames graph you got in the previous exercise and check if the new transform is being published.

Expected Result for Exercise 2.10

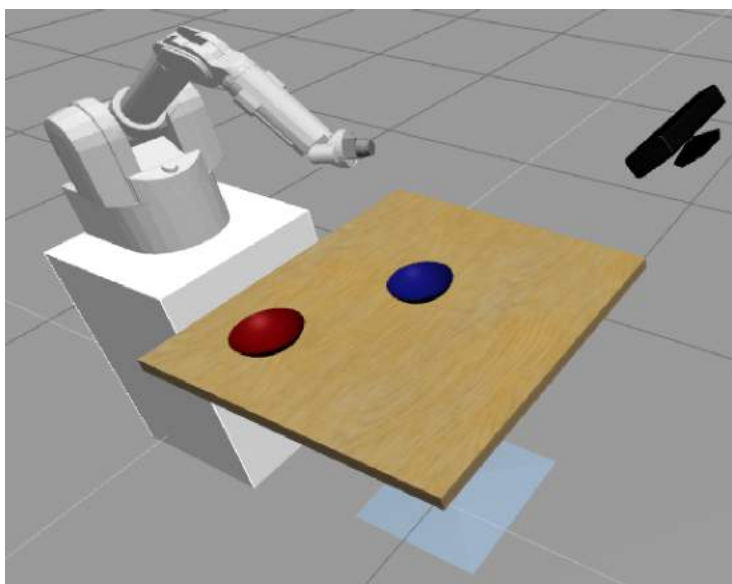


Kinect Camera Link

IMPORTANT REMARK

Having shown how to create a transform broadcaster, let me tell you a secret. This is not something that you'll usually have to do. Yeah, I'm really sorry. In the previous Chapter (Basic Concepts), you learned that the description of the robot model is made in the URDF files, right? Well, the **publication of the transforms is also handled by the URDF files**. At least, this is the common use. There exist, though, some cases where you do have to publish a transform separately from the URDF files. For instance:

- If you temporarily add a sensor to the robot. That is, you add a sensor that will be used during a few days, and then it will be removed again. In a case like this, instead of changing the URDF files of the robot (which will probably be more cumbersome), you'll just create a transform broadcaster to add the transform of this new sensor.
- You have a sensor that is external from your robot (and static). For instance, check the following scenario:



IRI Wam with external Kinect Camera

You have a robotic arm and, separated from it, you also have a Kinect camera, which provides information about the table to the robotic arm. In a case like this, you won't specify the transforms of the Kinect camera in the URDF files of the robot, since it's not a part of the robot. You'll also use a separated transform broadcaster.

Creating a launch file for the slam_gmapping node

At this point, I think you're prepared to create the launch file in order to start the **slam_gmapping** node. The main task to create this launch file, as you may imagine, is to correctly set the

parameters for the `slam_gmapping` node. This node is highly configurable and has lots of parameters you can change in order to improve the mapping performance. These parameters will be read from the ROS Parameter Server, and can be set either in the launch file itself or in a separated parameter files (YAML file). If you don't set some parameters, it will just take the default values. You can have a look at the complete list of parameters available for the `slam_gmapping` node here: <http://wiki.ros.org/gmapping> Let's now check some of the most important ones:

General Parameters

- **base_frame** (default: `"base_link"`): Indicates the name of the frame attached to the mobile base.
- **map_frame** (default: `"map"`): Indicates the name of the frame attached to the map.
- **odom_frame** (default: `"odom"`): Indicates the name of the frame attached to the odometry system.
- **map_update_interval** (default: `5.0`): Sets the time (in seconds) to wait until update the map.

Exercise 2.11

IMPORTANT: Before starting with this Exercise, make sure you've stopped the previously launched `slam_gmapping` node by pressing `Ctrl + C` on the console where you executed the command.

- In the `turtlebot_navigation_gazebo` package, search for the launch file named **`gmapping_demo.launch`**. You will see that what this file actually does is to call another launch file named **`gmapping.launch.xml`**, which is in the `turtlebot_navigation` package.
- Create a new package named **`my_mapping_launcher`**. Inside this package create a directory named `launch`, and inside this directory create a file named **`my_gmapping_launch.launch`**. Inside this file, copy the contents of the **`gmapping.launch.xml`** file.
- Modify the launch file you've just created, and set the **`map_update_interval`** parameter to 15.
- Launch the `gmapping` node using the new launch file created, and launch Rviz with your previously saved configuration.
- Move the robot around and calculate the time that takes to update the map now.

Data for Exercise 2.11

Check the following Notes in order to complete the Exercise: **Note 1:** Keep in mind that Rviz may have some delay, so the times may not be exact.

Expected Result for Exercise 2.11

The map now updates every 15 seconds.

Laser Parameters

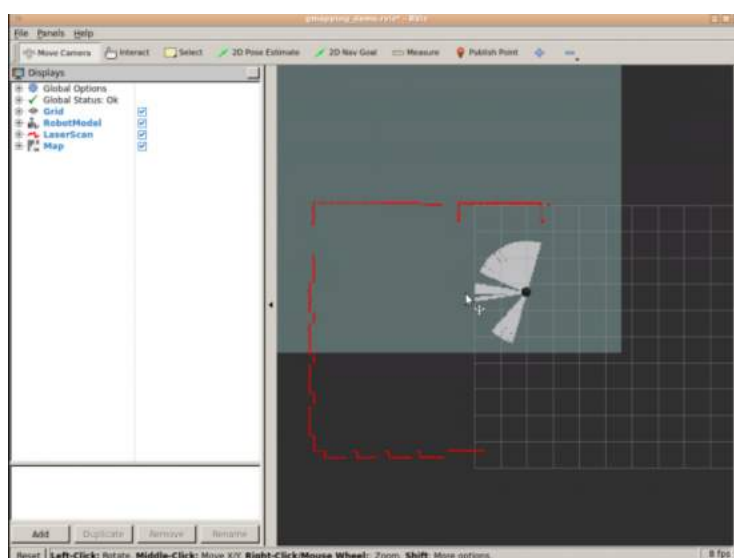
- **maxRange (float)**: Sets the maximum range of the laser. Set this value to something slightly higher than the real sensor's maximum range.
- **maxUrange (default: 80.0)**: Sets the maximum usable range of the laser. The laser beams will be cropped to this value.
- **minimumScore (default: 0.0)**: Sets the minimum score to consider a laser reading good.

Exercise 2.12

IMPORTANT: Before starting with this Exercise, make sure you've stopped the previously launched `slam_gmapping` node by pressing `Ctrl + C` on the console where you executed the command.

- Modify again this file, and set now the **maxUrange** parameter to 2.
- Launch again the node and check what happens now with the mapping area of the robot.

Expected Result for Exercise 2.12



MaxUrange parameter to 2

Initial map dimensions and resolutions

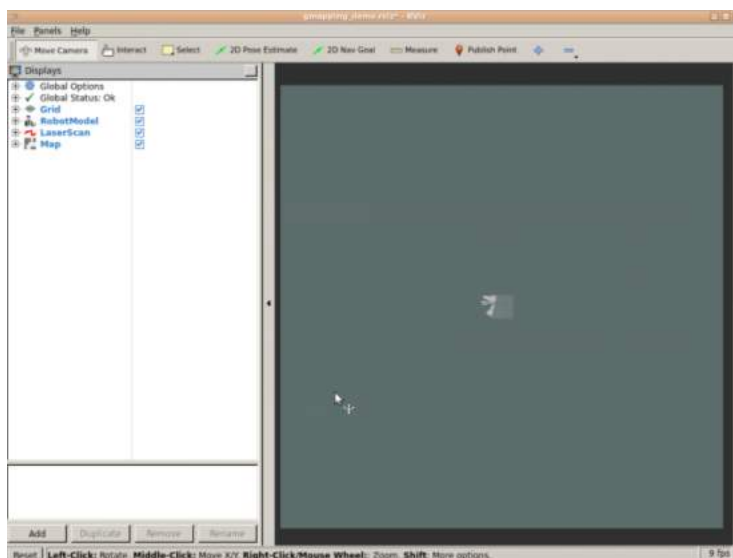
- **xmin (default: -100.0)**: Initial map size
- **ymin (default: -100.0)**: Initial map size
- **xmax (default: 100.0)**: Initial map size
- **ymax (default: 100.0)**: Initial map size
- **delta (default: 0.05)**: Sets the resolution of the map

Exercise 2.13

IMPORTANT: Before starting with this Exercise, make sure you've stopped the previously launched `slam_gmapping` node by pressing `Ctrl + C` on the console where you executed the command.

- Modify again this file, and set now the **xmin**, **ymin**, **xmax** and **ymax** parameters to 100 and -100, respectively.
- Launch again the node and check how the initial map looks now.

Expected Result for Exercise 2.13



Bigger Initial Map Size

Other Parameters

- **linearUpdate (default: 1.0)**: Sets the linear distance that the robot has to move in order to process a laser reading.
- **angularUpdate (default: 0.5)**: Sets the angular distance that the robot has to move in order to process a laser reading.
- **temporalUpdate (default: -1.0)**: Sets the time (in seconds) to wait between laser readings. If this value is set to -1.0, then this function is turned off.
- **particles (default: 30)**: Number of particles in the filter

Now you've already seen (and played with) some of the parameters that take place in the mapping process, it's time to create your own parameters file.

But first, let me explain you one thing. In the **gmapping_demo.launch** file, the parameters were loaded in the launch file itself, as you've seen. So you changed the parameters directly in the launch file. But this is not the only way you have to load parameters. In fact, parameters are usually loaded from an external file. This file that contains the parameters is usually a **YAML file**. So, you can also write all the parameters in a YAML file, and then load this file (and the parameters) in the launch file just by adding the following line inside the **<node>** tag:

```
[ ]: <rosparam file="$(find
    my_mapping_launcher)/params/gmapping_params.yaml" command="load" />
```


This will have the exact same result as if the parameters are loaded directly through the launch file. And as you will see, it is a much more clean way to do it.

Exercise 2.15

IMPORTANT: Before starting with this Exercise, make sure you've stopped the previously launched `slam_gmapping` node by pressing **Ctrl + C** on the console where you executed the command.

- a) Create a new directory named **params** inside the package created in Exercise 2.11.
- b) Create a YAML file named **gmapping_params.yaml**, and write in all the parameters you want to set.
- c) Remove all the parameters that are being loaded in the launch file, and load instead the YAML file you've just created.
- d) Launch again the node and check that everything works fine.

Data for Exercise 2.15

Check the following Notes in order to complete the Exercise: **Note 1:** Remember that in order to set a parameter in the YAML file, you have to use this structure: **name_of_paramter: value_of_parameter**

Extra content (optional)

Manually modify the map (for convenience)

Sometimes, the map that you create will contain stuff that you do not want to be there. For example:

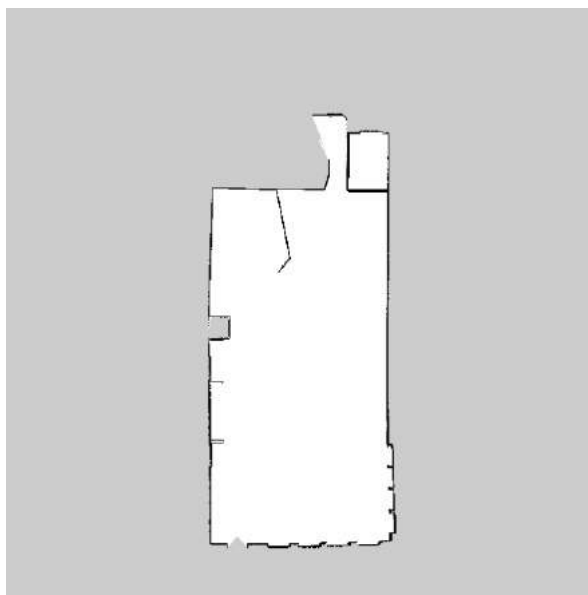
1. There could be people detected by the mapping process that has been included in the map. You don't want them on the map as black dots!
2. There could be zones where you do not want the robot to move (for instance, avoid the robot going close to down stairs area to prevent it from falling down).

For all these cases, you can take the map generated and modify it manually, just by using an image editor. You can include forbidden areas, or delete people and other stuff included in the map.

Exercise 2.16

- a) Download the map file (PGM file) generated in Exercise 2.3 to your local computer
- b) Open the map with your favourite image editor.
- c) Edit the file in order to prevent the robot to move close to the door.

Expected Result for Exercise 2.16



Edited Map

Build a Map Using Logged Data

Until now we've seen how to build a map by moving the robot in real-time. But this is not the only way of creating a map, of course! As you already know, the process of building a map is based on reading the data that is being published in the laser and the transform topics. That's why we were moving the robot around, to publish the data that the robot was getting in real-time while moving. But if we just need some data being published on those topics... doesn't it come to your mind another obvious way of creating a map? That's right! You could just use a bag file in order to publish data on those topics, and therefore build a map. In order to build a map using logged data, you will have to follow these 2 steps (which are divided into sub-steps):

1. Create the bag file

In order to create a proper bag file for Mapping, you'll need to follow the next steps:

- a) First of all, launch your keyboard teleop to start moving the robot:

```
[ ]: roslaunch pkg_name keyboard_teleop_launch_file.launch
```

- b) Make sure that the robot is publishing its laser data and the tf's.

```
[ ]: rostopic list
```

- c) Start recording scans and transforms (note that the scan topic may vary from robot to robot):

[esto no puede funcionar porque no estas grabando la odometria]

```
[ ]: rosbag record -O mylaserdata /laser_topic /tf_topic
```

This will start writing a file in the current directory called mylaserdata.bag.

d) Drive the robot around. General advice:

- Try to limit fast rotations, as they are hardest on the scan-matcher. It helps to lower the speed.
- Visualize what the robot “sees” with its laser; if the laser can’t see it, it won’t be in the map.
- Loop closure is the hardest part; when closing a loop, be sure to drive another 5-10 meters to get plenty of overlap between the start and end of the loop.

f) Kill the rosbag instance, and note the name of the file that was created.

Exercise 2.17

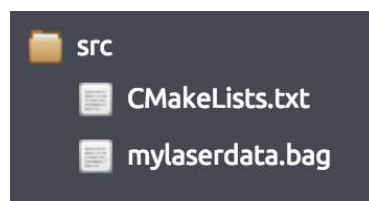
Create your own bag file by following the steps shown above.

Data for Exercise 2.17

Check the following Notes in order to complete the Exercise: **Note 1:** In order to be able to see the bag file in the IDE, you have to run the rosbag record command from the catkin_ws/src directory.

Note 2: You can check if a bag file has been created properly by using the next command: rosbag info name_of_bag_file.bag

Expected Result for Exercise 2.17



Bag File

2. Build the Map

Once we have the bag file, we’re ready to build the map! In order to do see, you’ll need to follow the next steps:

a) Start the slam_gmapping node, which will take in laser scans (in this case, on the /kobuki/laser/scan topic) and produce a map:

```
[ ]: rosrn gmapping slam_gmapping scan:=kobuki/laser/scan
```

b) Play the bag file to provide data to the slam_gmapping node:

```
[ ]: rosbag play name_of_bag_file_created_in_step_1
```

Wait for rosbag to finish and exit.

c) Save the created map using the map_saver node as shown in previous sections:

```
[ ]: rosrn map_server map_saver -f map_name
```

Now you will have 2 files (map_name.pgm and map_name.yaml) that should look the same as the ones you created in Exercise 1.3.

Exercise 2.18

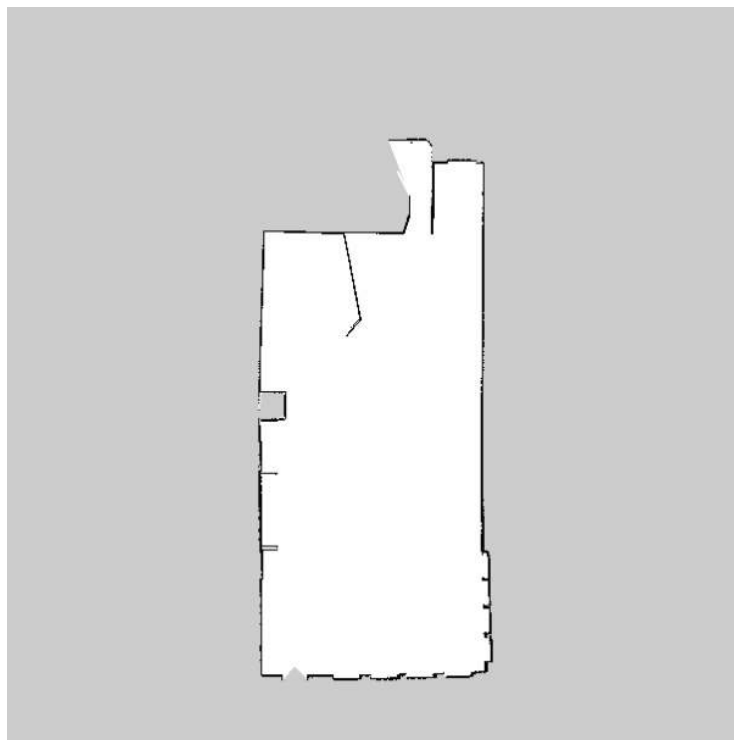
Create your own map from the bag file you've created in the previous exercise by following the steps shown above.

Data for Exercise 2.18

Check the following Notes in order to complete the Exercise: **Note 1:** Remember that the files will be saved to the directory from which you've executed the command.

Expected Result for Exercise 2.18

Image File:



Map Image

YAML File:

```
image: my_map.pgm
resolution: 0.050000
origin: [-15.400000, -13.800000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Map YAML

Summary

The very first thing you need in order to navigate is a Map of the environment. You can't navigate if you don't have a Map. Furthermore, this Map does have to be built properly, so it accurately represents the environment you want to navigate.

In order to create a Map of the environment, ROS provides the `slam_gmapping` node (of the `gmapping` package), which is an implementation of the SLAM (Simultaneous Localization and Mapping) algorithm. Basically, this node takes as input the laser and odometry readings of the robot (in order to get data from the environment), and creates a 2D Map.

This Map is an occupancy representation of the environment, so it provides information about the occupancy of each pixel in the Map. When the Map is completely built, you can save it into a file.

Also bear in mind that you only need the `slam_gmapping` node (the Mapper), when you are creating a map. Once the map has been created and saved, this node is not necessary anymore, hence it must be killed. At that point, you should launch the `map_server` node, which is the node that will provide the map that you just created to other nodes.

##

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 Mapping:[Mapping Solutions](#)

Unit 3. Robot Localization

ROS Navigation

Unit 3: Robot Localization



The localization environment



- ROSJect Link: <http://bit.ly/2LS9lp5>
- Package Name: **husky_navigation_launch**
- Launch File: **main.launch**

SUMMARY

Estimated time to completion: **2 hours** What will you learn with this unit?

- What does Localization mean in ROS Navigation?

- How does Localization work?
- How do we perform Localization in ROS?

In both the Basic Concepts Unit and in the Mapping Unit, we introduced and mentioned several times the importance of the Robot's localization in ROS Navigation. When the robot moves around a map, **it needs to know which is its POSITION within the map, and which is its ORIENTATION**. Determining its location and rotation (better known as the Pose of the robot) by using its sensor readings is known as **Robot Localization**.

In this Chapter, you will learn how to deal with the localization issue in ROS. But as we did in the previous chapter, let's first have a look at how RViz can help us with the localization process.

Visualize Localization in Rviz

As you've already seen, you can launch RViz and add displays in order to watch the localization of the robot. For this chapter, you'll basically need to use 3 elements of RViz:

- **LaserScan** Display (Shown in the previous chapter)
- **Map** Display (Shown in the previous chapter)
- **PoseArray** Display

Exercise 3.1

a) Execute the following command in order to launch the **amcl** node. We need to have this node running in order to visualize the Pose Arrays.

Execute in WebShell #1

```
[ ]: roslaunch husky_navigation amcl_demo.launch
```

Hit the icon with a screen in the top-right corner of the IDE window



Graphic Interface icon

in order to open the Graphic Interface.

b) Launch Rviz and add the necessary displays in order to visualize the localization data (not the map and the laser).

Execute in WebShell #2

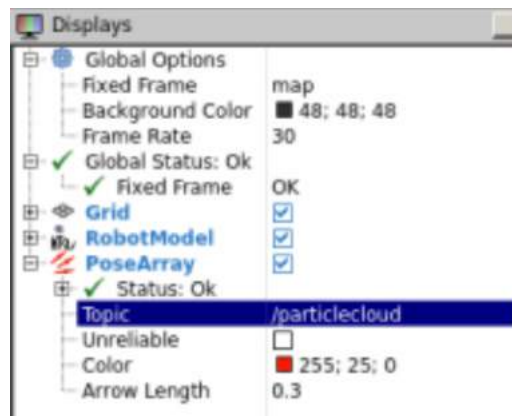
```
[ ]: rosrn rviz rviz
```

Visualize Pose Array (Particle Clouds)

- Click the Add button under displays and choose the **PoseArray** display.
- In the display properties, introduce the name of the topic where the particle cloud are being published (usually **/particlecloud**).
- To see the robot's position, you can choose to also add the RobotModel or TF displays.

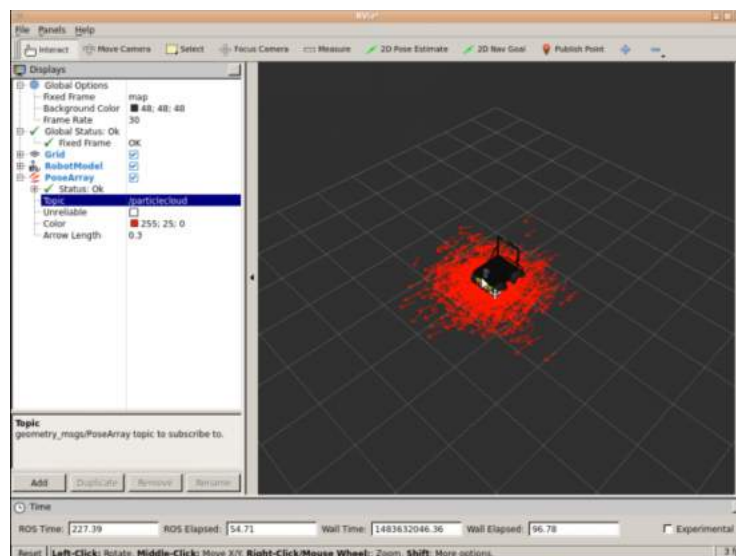
Data for Exercise 3.1

Check the following notes in order to complete the exercise: **Note 1:** You can change the arrow's length and color by modifying the "Color" and "Arrow Length" properties.



Selecting the appropriate topic for particles visualization

Expected Result for Exercise 3.1



Particles visualization in Rviz

Now that you already know how to configure RViz in order to visualize the localization of the robot, let's get to work!

REMEMBER: Remember to save your RViz configuration in order to be able to load it again

whenever you want. If you don't remember how to do it, check the previous chapter (Mapping Chapter).

Exercise 3.2

IMPORTANT: Before starting with this exercise, make sure that your Rviz is properly configured in order to visualize the localization process. Also, add the necessary elements in order to visualize the map and the laser.

In the next exercise, we'll see an example of how ROS deals with the Robot localization issue.

- a) Add the LaserScan and Map displays in order to visualize the position of the robot in the room through RViz.
- b) Using the 2D Pose Estimate tool, set an initial position and orientation in RViz for the Husky robot. It doesn't have to be exact, just take a look at the Husky's position and orientation in the simulation and try to set it in a similar way in RViz.
- c) Make the robot move around the room by using the keyboard teleop program.

Execute in WebShell #3

```
[ ]: roslaunch husky_navigation_launch keyboard_teleop.launch
```

Data for Exercise 3.2

Check the following notes in order to complete the exercise:

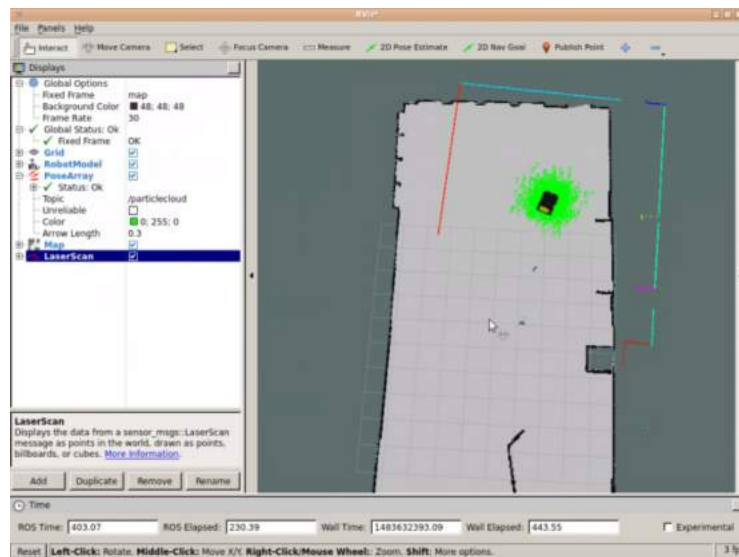
Note 1: Localization in ROS is visualized through elements called particles (we'll see more on this later on in the unit).

Note 2: The spread of the cloud represents the localization system's uncertainty about the robot's pose.

Note 3: As the robot moves around the environment, this cloud should shrink in size due to additional scan data allowing amcl to refine its estimation of the robot's position and orientation.

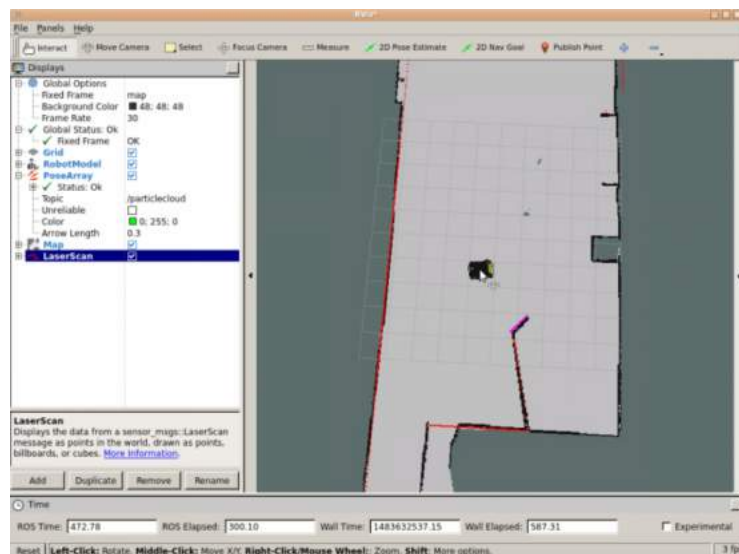
Expected Result for Exercise 3.2

Before moving the robot:



Initial localization usually has a position error

After moving the robot:



After a few movements, the robot achieves perfect localization

Ok, but... what just happened? What were those strange arrows we were visualizing in RViz? Let's first introduce some concepts in order to better understand what you've just done.

Monte Carlo Localization (MCL)

Because the robot may not always move as expected, it generates many random guesses as to where it is going to move next. These guesses are known as particles. Each particle contains

a full description of a possible future pose. When the robot observes the environment it's in (via sensor readings), it discards particles that don't match with these readings, and generates more particles close to those that look more probable. This way, in the end, most of the particles will converge in the most probable pose that the robot is in. So **the more you move, the more data you'll get from your sensors, hence the localization will be more precise**. These particles are those **arrows** that you saw in RViz in the previous exercise. Amazing, right?

This is known as the Monte Carlo Localization (MCL) algorithm, or also particle filter localization.

The AMCL package

The AMCL (Adaptive Monte Carlo Localization) package provides the **amcl node**, which uses the MCL system in order to track the localization of a robot moving in a 2D space. This node **subscribes to the data of the laser, the laser-based map, and the transformations of the robot, and publishes its estimated position in the map**. On startup, the amcl node initializes its particle filter according to the parameters provided.

NOTE: As you may have noticed, in order to name this ROS package (and node), the word **Adaptive** has been added to the Monte Carlo Localization algorithm. This is because, in this node, we will be able to configure (adapt) some of the parameters that are used in this algorithm. You'll learn more about these parameters later on in this chapter.

So, basically, what you've done in the previous exercise was the following:

- First, you launched an **amcl node** using the preconfigured **amcl_demo.launch** file.
- Second, you set up an initial pose by using the 2D Pose Estimate tool (which published that pose to the **/initialpose** topic).
- Then, you started moving the robot around the room, and the **amcl node** began reading the data published into the **laser topic (/scan)**, the **map topic (/map)**, and the **transform topic (/tf)**, and published the estimated pose where the robot was in to the **/amcl_pose** and the **/particlecloud** topics.
- Finally, via RViz, you accessed the data being published by this node into the **/particlecloud** topic, so you were able to visualize it, thanks to the cloud of "arrows," which were indicating the most probable position the robot was in, and its orientation.

Now, everything makes more sense, right?

FLASHBACK

Now you may be thinking... but where does this node get the map from? And that's a great question!

In the previous unit, you learned about the **map_server** node, which allows you to provide the data of a map from the map's file. Do you remember? If not, I suggest you go back and take a quick look in order to refresh your memory.

Back again? So basically, what we're doing here is to call that functionality in order to provide the map data to the **amcl** node. And we're doing it through the **amcl_demo.launch** file that you launched back in Exercise 3.1. If you take a look at this file, you'll see at the top of it, there is a section like this:

```
<!-- Run the map server -->
<arg name="map_file" default="$(find husky_navigation)/maps/my_map.yaml"/>
<node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" />
```

map server Call

Here, the file is launching a **map_server** node, which will take the provided map file (arg **map_file**) and turn it into map data. Then, it will publish this data to the **/map** topic, which will be used by the **amcl** node to perform localization.

Exercise 3.3

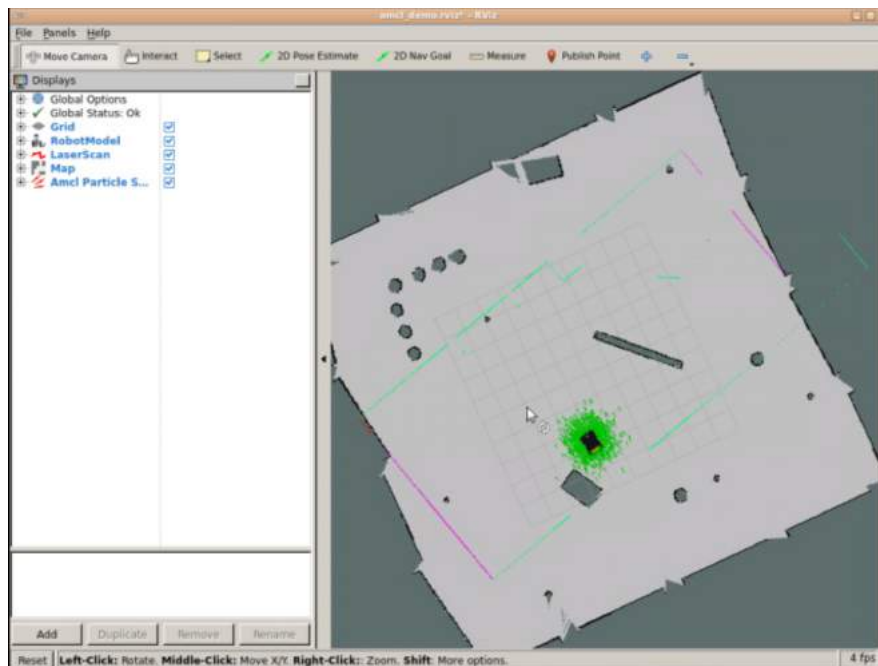
IMPORTANT: Before starting with this exercise, make sure that you've stopped the previously launched **amcl** node by pressing **Ctrl + C** on the console where you executed the command.

- Have a look at the **amcl_demo.launch** file in the **husky_navigation** package. You'll see that what it's actually doing is calling another launch file named **amcl.launch**.
- Create a new package named **my_amcl_launcher**. Inside this package, create a new directory named **launch**. And inside of this directory, create a new file named **change_map.launch**. Copy the contents of the **amcl.launch** file into this new file.
- In the launch file that you've just created, add a section where you launch the **map_server** node (as you've just seen above), and provide a different map file.
- Open **RViz** and see what happens now.

Data for Exercise 3.3

Check the following notes in order to complete the exercise: **Note 1:** The map files are located in a directory called **maps** of the **husky_navigation** package.

Expected Result for Exercise 3.3



A wrong map was used. The robot cannot localize properly

Exercise 3.4

IMPORTANT: Before starting with this exercise, make sure that you've set the correct map file into the amcl launch file again, and then launch the node.

In Exercise 3.2, you visualized localization through Rviz and the `/particlecloud` topic. But, now you've learned that this is not the only topic that the `amcl` node writes on. Get information about the estimated pose of the robot without using the `/particlecloud` topic.

Data for Exercise 3.4

Check the following notes in order to complete the exercise: **Note 1**: Keep in mind that in order to be able to visualize these topics, the amcl node must be launched.

Note 2: Move the robot through the room and check if the pose changes correctly.

Expected Result for Exercise 3.4

Output of the /amcl_pose topic:

[illegible]

Output of the topic that publishes current robot position

Exercise 3.5

Create a service server that, when called, returns the pose (position and orientation) of the robot at that exact moment.

- Create a new package named **get_pose**. Add rospy as a dependency.
- Inside this package, create a file named **get_pose_service.py**. Inside this file, write the code of your **Service Server**.
- Create a launch file in order to launch your Service.
- Using a WebShell, call your service.
- Using your Service, get the pose data from these 2 spots in the environment:



Robot is facing the fridge



Robot is facing the door

Hardware Requirements

As we saw in the previous chapter (Mapping), **configuration is also very important to properly localizing the robot in the environment.**

In order to get a proper Robot localization, we need to fulfill 3 basic requirements:

- Provide Good **Laser Data**
- Provide Good **Odometry Data**
- Provide Good Laser-Based **Map Data**

Exercise 3.6

Make sure that your robot is publishing this data, and identify both the topic and the message type that each topic is using in order to publish the data.

Have a look at one or more of the messages that were published in this topic in order to check their structure.

Data for Exercise 3.6

Check the following notes in order to complete the exercise: **Note 1:** Keep in mind that in order to be able to visualize these topics, the amcl node must be launched.

Expected Result for Exercise 3.6

```
/imu/data/yaw/parameter_updates
/initialpose
/joint_states
/joy_teleop/cmd_vel
/map
/map_metadata
/navsat/fix
/navsat/fix/position/parameter_descriptions
/navsat/fix/position/parameter_updates
/navsat/fix/status/parameter_descriptions
/navsat/fix/status/parameter_updates
/navsat/fix/velocity/parameter_descriptions
/navsat/fix/velocity/parameter_updates
/navsat/vel
/odometry/filtered
/particlecloud
/platform_control/cmd_vel
/rosout
/rosout_agg
/scan
/set_pose
/tf
/tf_static
/twist_marker_server/cmd_vel
/twist_marker_server/feedback
```

Some of the relevant topics you must have

Transforms

As we also saw in the previous chapter (Mapping), we need to be publishing a correct **transform** between the laser frame and the base of the robot's frame. This is pretty obvious, since as you've already learned, the robot **uses the laser readings in order to constantly re-calculate it's localization**.

More specifically, the `amcl` node has these 2 requirements regarding the transformations of the robot:

- `amcl` transforms incoming laser scans to the odometry frame (`~odom_frame_id`). So, there must be a path through the tf tree from the frame in which the laser scans are published to the odometry frame.
- `amcl` looks up the transform between the laser's frame and the base frame (`~base_frame_id`), and latches it forever. So **`amcl` cannot handle a laser that moves with respect to the base**.

NOTE: If you want more information regarding this issue, go back to the Mapping Chapter and review this section.

Exercise 3.7

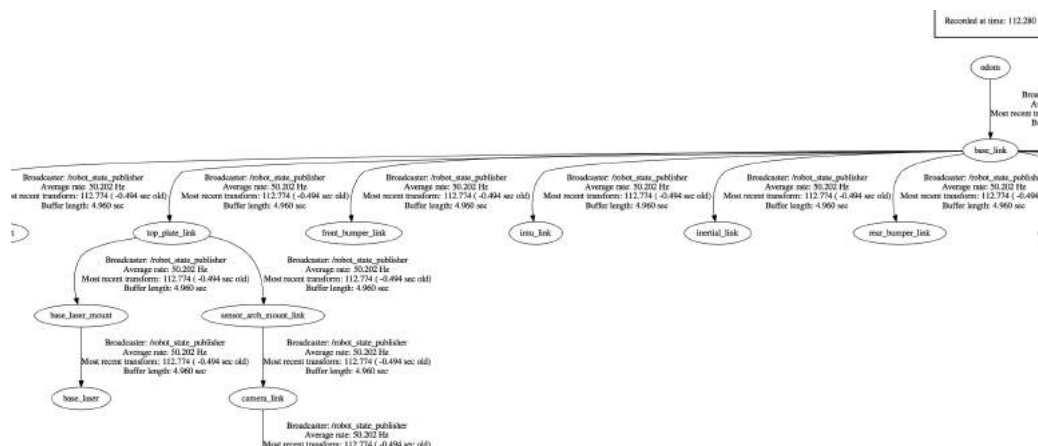
Generate the transforms tree and check if the above requirements are fulfilled.

Data for Exercise 3.7

Check the following notes in order to complete the exercise: **Note 1:** Remember that you can download a file through the IDE by right-clicking on them and selecting the "Download" option.

Note 2: Also remember that in order to be able to access the file from the IDE, it needs to be in the `catkin_ws/src` directory.

Expected Result for Exercise 3.7



A part of the transforms tree

Creating a launch file for the AMCL node

As you saw in the Mapping Chapter, you also need to have a launch file in order to start the amcl node. This node is also highly customizable and we can configure many parameters in order to improve its performance. These parameters can be set either in the launch file itself or in a separate parameters file (YAML file). You can have a look at a complete list of all of the parameters that this node has here: <http://wiki.ros.org/amcl> Let's have a look at some of the most important ones:

General Parameters

- **odom_model_type (default: "diff")**: It puts the odometry model to use. It can be "diff," "omni," "diff-corrected," or "omni-corrected."
- **odom_frame_id (default: "odom")**: Indicates the frame associated with odometry.
- **base_frame_id (default: "base_link")**: Indicates the frame associated with the robot base.
- **global_frame_id (default: "map")**: Indicates the name of the coordinate frame published by the localization system.
- **use_map_topic (default: false)**: Indicates if the node gets the map data from the topic or from a service call.

Filter Parameters These parameters will allow you to configure the way that the particle filter performs.

- **min_particles (default: 100)**: Sets the minimum allowed number of particles for the filter.
- **max_particles (default: 5000)**: Sets the maximum allowed number of particles for the filter.
- **kld_err (default: 0.01)**: Sets the maximum error allowed between the true distribution and the estimated distribution.
- **update_min_d (default: 0.2)**: Sets the linear distance (in meters) that the robot has to move in order to perform a filter update.
- **update_min_a (default: $\pi/6.0$)**: Sets the angular distance (in radians) that the robot has to move in order to perform a filter update.
- **resample_interval (default: 2)**: Sets the number of filter updates required before resampling.
- **transform_tolerance (default: 0.1)**: Time (in seconds) with which to post-date the transform that is published, to indicate that this transform is valid into the future.
- **gui_publish_rate (default: -1.0)**: Maximum rate (in Hz) at which scans and paths are published for visualization. If this value is -1.0, this function is disabled.

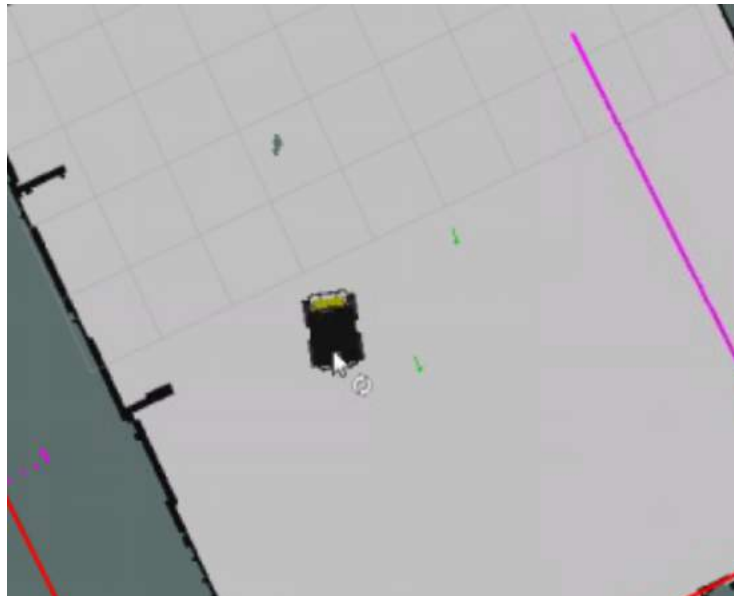
Exercise 3.8

IMPORTANT: Before starting with this exercise, make sure that you've stopped the previously launched amcl node by pressing Ctrl + C on the console where you executed the command.

- a) In the package that you created in Exercise 3.3, create a new launch file named **my_amcl_launch.launch**. Copy the contents of the **amcl_demo.launch** file to this file.

- b) Modify the **min_particles** and **max_particles** parameters. Set them to 1 and 5, respectively.
- c) Launch the node again using this new launch file, and see what happens.

Expected Result for Exercise 3.8



Can you guess what happened? Well, the number of particles (green arrows) that are used to localize the robot has drastically decreased. This means that the number of guesses (regarding the pose of the robot) that the localization algorithm will be able to do is going to be very slow. This will cause the localization of the robot to be much more imprecise.

Laser Parameters These parameters will allow you to configure the way the `amcl` node interacts with the laser.

- **laser_min_range (default: -1.0)**: Minimum scan range to be considered; -1.0 will cause the laser's reported minimum range to be used.
- **laser_max_range (default: -1.0)**: Maximum scan range to be considered; -1.0 will cause the laser's reported maximum range to be used.
- **laser_max_beams (default: 30)**: How many evenly-spaced beams in each scan to be used when updating the filter.
- **laser_z_hit (default: 0.95)**: Mixture weight for the `z_hit` part of the model.
- **laser_z_short (default: 0.1)**: Mixture weight for the `z_short` part of the model.
- **laser_z_max (default: 0.05)**: Mixture weight for the `z_max` part of the model.
- **laser_z_rand (default: 0.05)**: Mixture weight for the `z_rand` part of the model.

Exercise 3.9

IMPORTANT: Before starting with this exercise, make sure that you've stopped the previously launched `amcl` node by pressing `Ctrl + C` on the console where you executed the command.

- a) Modify the `laser_max_range` parameter, and set it to 1.
- b) Launch the node again, and see what happens.

Expected Result for Exercise 3.9

The localization of the robot will now be much more difficult. This means that the particles will remain dispersed, even after you move the robot around the environment.

Do you know why this is happening? Any clue? It's actually quite obvious.

You have modified the `laser_max_range` parameter to 1, so the range of the laser is now very low. And, as you already learned in this chapter, the robot uses the data from the laser readings in order to localize itself. So, now the laser readings won't contain much useful information, and this will cause the localization of the robot to be much more difficult.

Now that you have already seen (and played with) some of the parameters that you can configure in the `amcl` node, it's time for you to create your own launch file. But, as you learned in the previous chapter, you can also load these parameters from an external YAML file. So, now, let's create our own launch and parameter files!

Exercise 3.10

Create a launch file and a parameters file for the `amcl` node.

- a) Inside the `my_amcl_launcher` package, create a new directory named `params`. Inside this directory, create a file named `my_amcl_params.yaml`
- b) Write the parameters that you want to set for your `amcl` node into this file.
- c) Modify the launch file you created in Exercise 3.8. Remove all of the parameters that are being loaded on the launch file, and load the parameters file that you've just created instead.
- d) Execute your launch file and test that everything works fine.

Data for Exercise 3.10

Check the following notes in order to complete the exercise:

Note 1: As discussed previously, the `amcl` node needs to get data from the `/map` topic in order to work properly, so you'll need to publish map information into the proper topic.

AMCL through Services

Until now, we've seen how to deal with localization through topics. We were getting data from the laser, odometry, and map topics, and publishing it into 2 new topics that provided us with the robot's localization information. But, there are (or there can be) some services involved as well, if you like. Let's have a quick look at them:

Services Provided by the amcl node **global_localization (std_srvs/Empty)**: Initiate global localization, wherein all particles are dispersed randomly throughout the free space in the map.

Services Called by the amcl node **static_map (nav_msgs/GetMap)**: amcl calls this service to retrieve the map that is used for laser-based localization.

Summarizing, the amcl node provides a service called **global_localization** in order to restart the positions of the particles, and is able to make use of a service called **static_map** in order to get the map data it needs.

Exercise 3.11

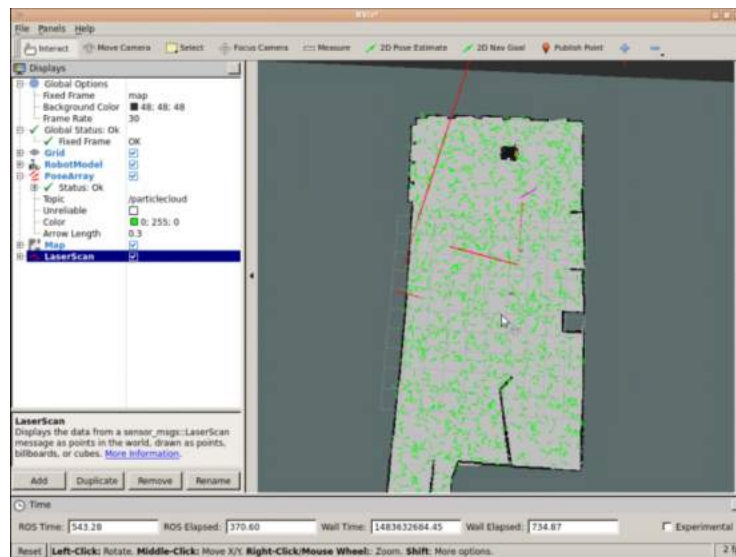
- a) Launch RViz in order to visualize the **/particlecloud** topic.
- b) Create a new package named **initialize_particles**. Add rospy as a dependency.
- c) Inside this package, create a new file named **init_particles_caller.py**. Inside this Python file, write the code for a service client. This client will perform a call to the **/global_localization** service in order to disperse the particles.

Data for Exercise 3.11

Check the following Notes in order to complete the Exercise:

Note 1: Keep in mind that in order to be able to call this service, you need to have the amcl node running.

Expected Result for Exercise 3.11



Filter particles dispersed

So I have a service that allows me to disperse the filter particles randomly all around the environment. Alright, but... why do I need this for? It is really useful for me?

That's actually a very good question. And the answer is YES! Of course! Until now, when launching the `amcl` node, you were using RViz (the 2D Pose Estimate tool) in order to provide an approximate pose of the robot. This way, you were providing a HUGE help to the `amcl` node in order to localize the robot. That's why the particles were all initialized close to the robot's position.

But let me ask you... What happens if you have no idea about where the robot actually is? Or what happens if you can't use RViz, so you can't provide an approximate pose of the robot?

If this is the case, then you will have to initialize the particles all around the environment, and start moving the robot around it. This way, the sensors will start to gather information about the environment and provide it to the robot. And as you already know, as the robot keeps getting more and more information about the environment, the particles of the filter will keep getting more and more accurate.

Let's do an exercise so you can understand better how this works.

Exercise 3.12

- Inside the package that you created in the exercise above, create a new file named **`square_move.py`**.
- Inside this file, write the code to move the robot. The robot must perform a square movement. When it finishes, it has to print the covariance of the filter particles into the screen.
- Test that your code works (the robot moves in a square).
- When you've tested that it works, add the necessary code so that your program does the following:

us. In order to achieve this, the amcl node uses the MCL (Monte Carlo Localization) algorithm.

You can't navigate without a map, as you learned in the previous chapter. But, surely, you can't navigate if your robot isn't able to localize itself in it either. So, the localization process is another key part of ROS Navigation.

Basically, the amcl node takes data from the laser and the odometry of the robot, and also from the map of the environment, and outputs an estimated pose of the robot. The more the robot moves around the environment, the more data the localization system will get, so the more precise the estimated pose it returns will be.

##

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 Localization:[Localization Solutions](#)

Unit 4. Path Planning 1

ROS Navigation

Chapter 4: Path Planning Part 1



Husky Robot Image

 Run on ROSDS

- ROSJect Link: <http://bit.ly/2nak1WG>
- Package Name: **husky_navigation_launch**
- Launch File: **main.launch**

SUMMARY

Estimated time to completion: **3 hours** What will you learn with this unit?

- What does Path Planning mean in ROS Navigation?

- How does Path Planning work?
- How does the move_base node work?
- What is a Costmap?

We're arriving at the end of the course, guys! For now, we've seen how to create a map of an environment, and how to localize the robot in it. So, at this point (and assuming everything went well), we have all that we need in order to perform Navigation. That is, we're now ready to plan trajectories in order to move the robot from pose A to pose B.

In this chapter, you'll learn how the Path Planning process works in ROS, and all of the elements that take place in it. But first, as we've been doing in previous chapters, let's have a look at our digital best friend, RViz.

Visualize Path Planning in Rviz

As you've already seen in previous chapters, you can also launch RViz and add displays in order to watch the Path Planning process of the robot. For this chapter, you'll basically need to use 3 elements of RViz:

- Map Display (Costmaps)
- Path Displays (Plans)
- 2D Tools

Exercise 4.1

a) Execute the next command in order to launch the move_base node.

Execute in WebShell #1

```
[ ]: roslaunch husky_navigation move_base_demo.launch
```

b) Open the Graphic Interface and execute the following command in order to start RViz.

Execute in WebShell #2

```
[ ]: rosrune rviz rviz
```

c) Properly configure RViz in order to visualize the necessary parts.

Visualize Costmaps

- Click the Add button under Displays and chose the Map element.
- Set the topic to **/move_base/global_costmap/costmap** in order to visualize the global costmap
- Change the topic to **/move_base/local_costmap/costmap** in order to visualize the local costmap.
- You can have 2 Map displays, one for each costmap.

Visualize Plans

- Click the Add button under Displays and chose the Path element.
- Set the topic to **/move_base/NavfnROS/plan** in order to visualize the global plan.
- Change the topic to **/move_base/DWAPlannerROS/local_plan** in order to visualize the local plan.
- You can also have 2 Path displays, one for each plan.

d) Use the 2D Pose Estimate tool in order to provide an initial pose for the robot.

e) Use the 2D Nav Goal tool in order to send a goal pose to the robot. Make sure to select an unoccupied (dark grey) or unexpected (light grey) location.

Data for Exercise 4.1

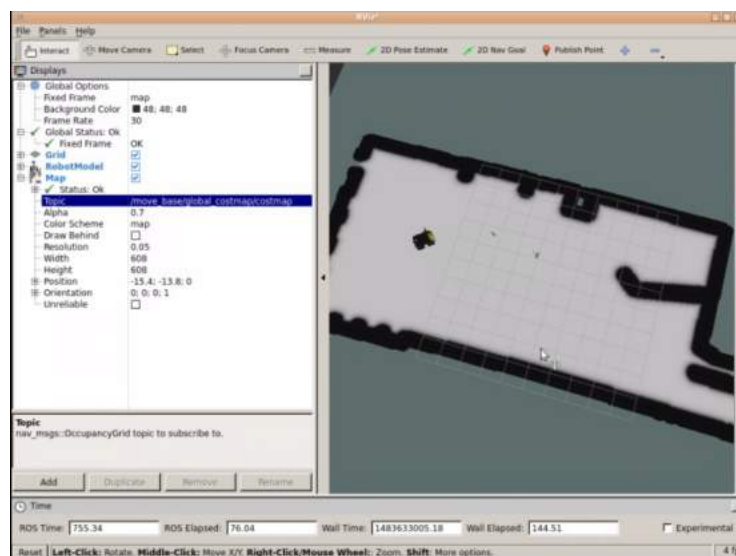
Check the following notes in order to complete the exercise:

Note 1: Bear in mind that if you don't set a 2D Nav Goal, the planning process won't start. This means that until you do, you won't be able to visualize any plan in RViz.

Note 2: In order for the 2D tools to work, the Fixed Frame at Rviz must be set to map.

Expected Result for Exercise 4.1

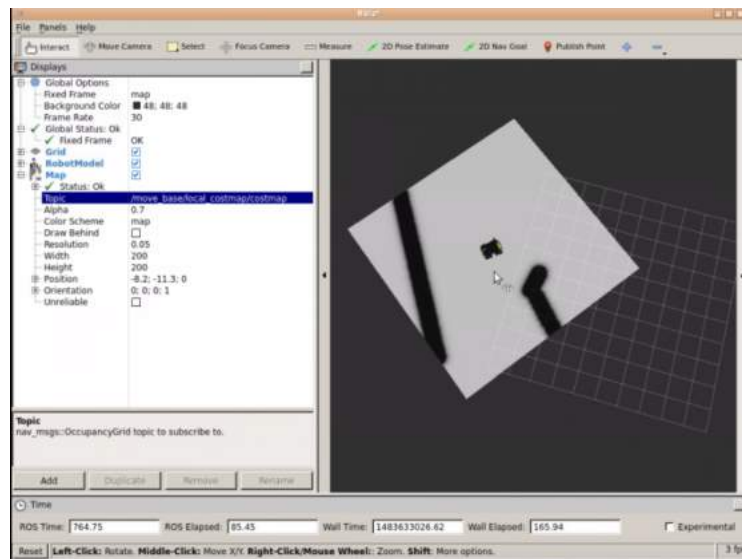
Global Costmap:



Global Costmap

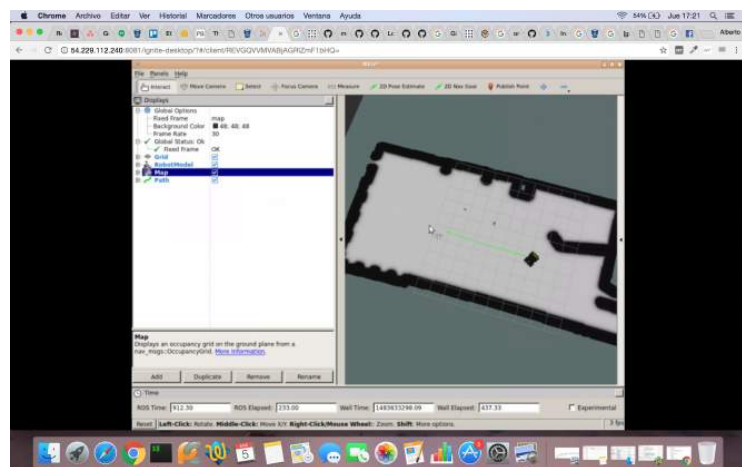
Local Costmap:

4 - Path Planning 1



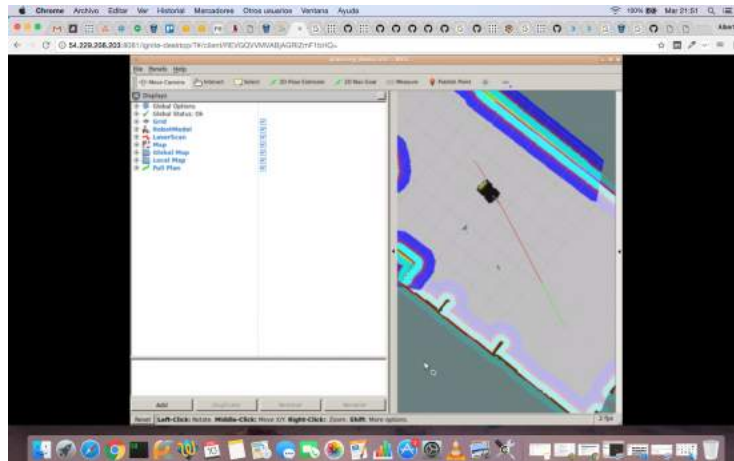
Local Costmap

Global Plan:



Global Plan

Local Plan (in red) and Global Plan (in green):



Local(red) and Global(green) Plans

REMEMBER: Remember to save your RViz configuration in order to be able to load it again whenever you want. If you don't remember how to do it, check the Mapping Chapter.

That's awesome, right? But what has just happened? What was that 2D Nav Goal tool I used in order to move the robot? And what's a Costmap? How does ROS calculate the trajectories?

Keep calm!! By the end of this chapter, you'll be able to answer all of those questions. But let's go step by step, so that you can completely understand how the whole process works.

The move_base package

The move_base package contains the **move_base node**. Doesn't that sound familiar? Well, it should, since you were introduced to it in the Basic Concepts chapter! The move_base node is one of the major elements in the ROS Navigation Stack, since it links all of the elements that take place in the Navigation process. Let's say it's like the Architect in Matrix, or the Force in Star Wars. Without this node, the ROS Navigation Stack wouldn't make any sense!

Ok! We understand that the move_base node is very important, but... what is it exactly? What does it do? Great question!

The **main function of the move_base node is to move the robot from its current position to a goal position**. Basically, this node is an implementation of a SimpleActionServer, which takes a goal pose with message type geometry_msgs/PoseStamped. Therefore, we can send position goals to this node by using a SimpleActionClient.

This Action Server provides the topic **move_base/goal**, which is the input of the Navigation Stack. This topic is then used to provide the goal pose.

Exercise 4.2

- a) In a WebShell, visualize the `move_base/goal` topic.
- b) As you did in the previous exercise, send a goal to the robot by using the 2D Nav Goal tool in RViz.
- c) Check what happens in the topic that you are listening to.

Expected Result for Exercise 4.2

```
ubuntu@ip-172-31-44-229:~$ rostopic pub /move_base/goal move_base_msgs/MoveBaseActionGoal "header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 0
  frame_id: ''
goal_id:
  stamp:
    secs: 0
    nsecs: 0
  id: ''
goal:
  target_pose:
    header:
      seq: 0
      stamp:
        secs: 0
        nsecs: 0
      frame_id: 'map'
    pose:
      position:
        x: 1.16
        y: -4.76
        z: 0.0
      orientation:
        x: 0.0
        y: 0.0
        z: 0.75
        w: 0.66"
publishing and latching message. Press ctrl-C to terminate
```

Goal Message

So, each time you set a Pose Goal using the 2D Nav Goal tool from RViz, what is really happening is that a new message is being published into the `move_base/goal` topic.

Anyway, this is not the only topic that the `move_base` Action Server provides. As every action server, it provides the following 5 topics:

- `move_base/goal` (`move_base_msgs/MoveBaseActionGoal`)
- `move_base/cancel` (`actionlib_msgs/GoalID`)
- `move_base/feedback` (`move_base_msgs/MoveBaseActionFeedback`)
- `move_base/status` (`actionlib_msgs/GoalStatusArray`)
- `move_base/result` (`move_base_msgs/MoveBaseActionResult`)

Exercise 4.3

Without using Rviz, send a pose goal to the `move_base` node.

- a) Use the command line tool in order to send this goal to the Action Server of the move_base node.
- b) Visualize through the webshells all of the topics involved in the action, and check their output while the action is taking place, and when it's done.

Data for Exercise 4.3

Check the following notes in order to complete the exercise:

Note 1: Remember that the SimpleActionServer subscribes to the /move_base_node/goal topic in order to read the pose goal.

Note 2: In order to see an example of a valid message for the /move_base/goal topic, you can listen to the topic while you send a pose goal via the 2D Nav Goal tool of RViz.

Note 3: Keep in mind that in order to be able to send goals to the Action Server, the move_base node must be launched.

Expected Result for Exercise 4.3

Sending goal:

```
ubuntu@ip-172-31-44-229:~$ rostopic pub /move_base/goal move_base_msgs/MoveBaseActionGoal "header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 0
  frame_id: ''
goal_id:
  stamp:
    secs: 0
    nsecs: 0
  id: ''
goal:
  target_pose:
    header:
      seq: 0
      stamp:
        secs: 0
        nsecs: 0
      frame_id: 'map'
    pose:
      position:
        x: 1.16
        y: -4.76
        z: 0.0
      orientation:
        x: 0.0
        y: 0.0
        z: 0.75
        w: 0.66"
publishing and latching message. Press ctrl-C to terminate
```

Goal Message

Echo feedback:

```

^Cubuntu@ip-172-31-44-229:~$ rostopic echo /move_base/feedback
WARNING: no messages received and simulated time is active.
Is /clock being published?
header:
  seq: 388
  stamp:
    secs: 1611
    nsecs: 654000000
  frame_id: ''
status:
  goal_id:
    stamp:
      secs: 1611
      nsecs: 654000000
    id: /move_base-7-1611.654000000
  status: 1
  text: This goal has been accepted by the simple action server
feedback:
  base_position:
    header:
      seq: 0
      stamp:
        secs: 1611
        nsecs: 640000000
      frame_id: map
    pose:
      position:
        x: 0.936578248096
        y: -4.23249236439
        z: 0.0
      orientation:
        x: 0.0
        y: 0.0
        z: 0.716234530584
        w: 0.697859654372
  ---

```

Echo of feedback topic

Echo status accepted:

```

---
header:
  seq: 7402
  stamp:
    secs: 1616
    nsecs: 962000000
  frame_id: ''
status_list:
-
  goal_id:
    stamp:
      secs: 1611
      nsecs: 654000000
    id: /move_base-7-1611.654000000
    status: 1
    text: This goal has been accepted by the simple action server
---

```

Goal Accepted

Echo status reached:

```

---
header:
  seq: 7868
  stamp:
    secs: 1709
    nsecs: 962000000
  frame_id: ''
status_list:
-
  goal_id:
    stamp:
      secs: 1611
      nsecs: 654000000
    id: /move_base-7-1611.654000000
    status: 3
    text: Goal reached.
---

```

Goal Reached

Echo result:


```
ubuntu@ip-172-31-44-229:~$ rostopic echo /move_base/result
WARNING: no messages received and simulated time is active.
Is /clock being published?
header:
  seq: 6
  stamp:
    secs: 1625
    nsecs: 254000000
  frame_id: ''
status:
  goal_id:
    stamp:
      secs: 1611
      nsecs: 654000000
    id: /move_base-7-1611.654000000
  status: 3
  text: Goal reached.
result:
```

Echo of result topic

Exercise 4.4

- Create a new package named **send_goals**. Add rospy as a dependency.
- Inside this package, create a file named **send_goal_client.py**. Write into this file the code for an Action Client in order to send messages to the Action Server of the move_base node.
- Using this Action Client, move the robot to three different Poses of the Map. When the robot has reached the 3 poses, start over again creating a loop, so that the robot will keep going to these 3 poses over and over.

Expected Result for Exercise 4.4

The robot moves infinitely to the 3 poses given.

So, at this point, you've checked that you can send pose goals to the move_base node by sending messages to the /move_base/goal topic of its Action Server.

When this node **receives a goal pose**, it links to components such as the global planner, local planner, recovery behaviors, and costmaps, and **generates an output, which is a velocity command** with the message type geometry_msgs/Twist, and sends it to the /cmd_vel topic in order to move the robot.

The move_base node, just as you saw with the slam_gmapping and the amcl nodes in previous chapters, also has parameters that you can modify. For instance, one of the parameters that you can modify is the frequency at which the move_base node sends these velocity commands to the base controller. Let's check it with a quick exercise.

Exercise 4.5

- a) Create a new package named **my_move_base_launcher**. Inside this package, create 2 directories, one named **launch** and the other one named **params**. Inside the launch directory, create 2 new files named **my_move_base_launch_1.launch** and **my_move_base_launch_2.launch**. Inside the params directory, create a new file named **my_move_base_params.yaml**.
- b) Have a look at the **move_base_demo.launch** and **move_base.launch** files of the **husky_navigation** package. Also, have a look at the **planner.yaml** file of the same package.
- c) Copy the contents of these files to the files that you created in the first step.
- d) Modify the **my_move_base_launch_1.launch** file so that it loads your second launch file.
- e) Modify the **my_move_base_launch_2.launch** file so that it loads your **move_base** parameters file.
- f) Modify the **my_move_base_params.yaml** file, and change the **controller_frequency** parameter.
- g) Launch the **my_move_base_launch_1.launch** file, and check what happens now.

At this point, all you know is that sending a pose goal to the **move_base** node activates some kind of process, which involves other nodes, and that results in the robot moving to that goal pose. That's very interesting, but... what is this process that is going on? How does it work? What are these other nodes that take place?

Don't worry, you'll be able to answer all of those questions by the end of this chapter. But for now, let's start by introducing one of the main parts that take place in this process: the **global planner**.

The Global Planner

When a new goal is received by the **move_base** node, this goal is immediately sent to the global planner. Then, the **global planner is in charge of calculating a safe path in order to arrive at that goal pose**. This path is calculated before the robot starts moving, so it will **not take into account the readings that the robot sensors are doing** while moving.

Each time a new path is planned by the global planner, this path is published into the **/plan** topic. Let's do an exercise to check this.

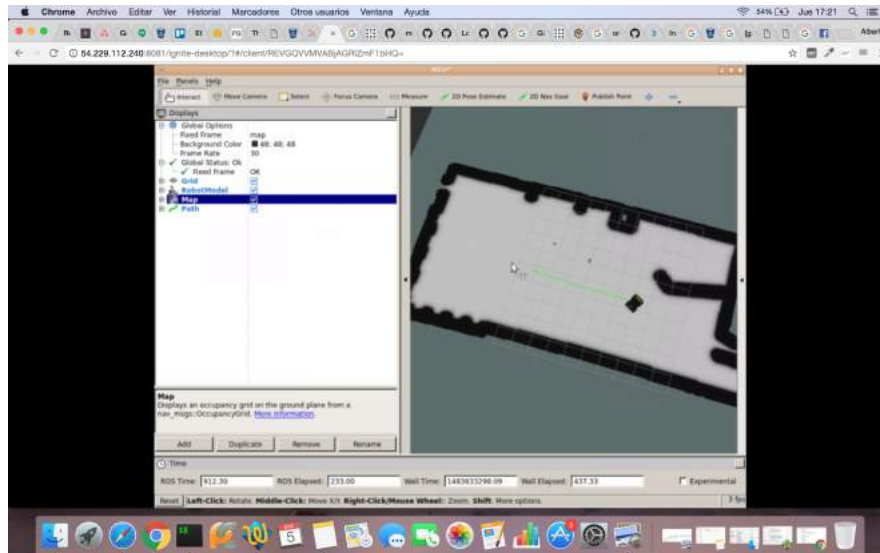
Exercise 4.6

- a) Open Rviz and add a Display in order to be able to visualize the Global Plan.
- b) Subscribe to the topic where the Global Planner publishes its planned path, and have a look at it.

c) Using the 2D Nav Goal tool, send a new goal to the move_base node.

Expected Result for Exercise 4.6

Global Plan in RViz:



Robot Planning in RViz

Global Plan topic:

```

header:
  seq: 0
  stamp:
    secs: 730
    nsecs: 2600000000
  frame_id: map
pose:
  position:
    x: -0.499851767717
    y: -2.69851744322
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
-
header:
  seq: 0
  stamp:
    secs: 730
    nsecs: 2600000000
  frame_id: map
pose:
  position:
    x: -0.510958640441
    y: -2.7209135312
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
-

```

Plan Message

You've probably noticed that when you send a goal in order to visualize the path plan made by the global planner, the robot automatically starts executing this plan. This happens because by sending this goal pose, you're starting the whole navigation process.

In some cases, you might be interested in just visualizing the global plan, but not in executing that plan. For this case, the `move_base` node provides a service named `/make_plan`. This service allows you to calculate a global plan without causing the robot to execute the path. Let's check how it works with the next exercise.

Exercise 4.7

Create a Service Client that will call one of the services introduced above in order to get the plan to a given pose, without causing the robot to move.

- Create a new package named **make_plan**. Add **rospy** as a dependency.
- Inside this package, create a file named **make_plan_caller.py**. Write the code for your Service Client into this file.

Data for Exercise 4.7

Check the following notes in order to complete the exercise:

Note 1: The type of message used by the `/make_plan` service is `nav_msgs/GetPlan`.

Note 2: When filling this message in order to call the service, you don't have to fill all of the fields of the message. Check the following message example:

```
ubuntu@ip-172-31-44-229:/opt/ros/indigo/share/husky_rviz_launchers/rviz$ rosservice call /move_base/make_plan "start:
header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 0
  frame_id: 'map'
pose:
  position:
    x: 1.16
    y: -4.76
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.75
    w: 0.66
goal:
  header:
    seq: 0
    stamp:
      secs: 0
      nsecs: 0
    frame_id: 'map'
  pose:
    position:
      x: 1.16
      y: -4.50
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.75
      w: 0.66
  tolerance: 0.0"
```

Message Example

Expected Result for Exercise 4.7

Returned Plan:

```

plan:
  header:
    seq: 0
    stamp:
      secs: 0
      nsecs: 0
    frame_id: ''
  poses:
    -
      header:
        seq: 0
        stamp:
          secs: 2151
          nsecs: 238000000
        frame_id: map
      pose:
        position:
          x: 1.15000024661
          y: -4.79999986589
          z: 0.0
        orientation:
          x: 0.0
          y: 0.0
          z: 0.0
          w: 1.0
    -
      header:
        seq: 0
        stamp:
          secs: 2151
          nsecs: 238000000
        frame_id: map
      pose:
        position:
          x: 1.15000024661
          y: -4.74999986514
          z: 0.0
        orientation:
          x: 0.0

```

Returned Plan

So, you now know that the first step of this navigation process is to calculate a safe plan so that your robot can arrive to the user-specified goal pose. But... how is this path calculated?

There exist different global planners. Depending on your setup (the robot you use, the environment it navigates, etc.), you would use one or another. Let's have a look at the most important ones.

Navfn

The Navfn planner is probably the most commonly used global planner for ROS Navigation. It uses Dijkstra's algorithm in order to calculate the shortest path between the initial pose and the goal pose. Below, you can see an animation of how this algorithm works.

Carrot Planner

The carrot planner takes the goal pose and checks if this goal is in an obstacle. Then, if it is in an obstacle, it walks back along the vector between the goal and the robot until a goal point that

is not in an obstacle is found. It, then, passes this goal point on as a plan to a local planner or controller. Therefore, this planner does not do any global path planning. It is helpful if you require your robot to move close to the given goal, even if the goal is unreachable. In complicated indoor environments, this planner is not very practical. This algorithm can be useful if, for instance, you want your robot to move as close as possible to an obstacle (a table, for instance).

Global Planner

The global planner is a more flexible replacement for the navfn planner. It allows you to change the algorithm used by navfn (Dijkstra's algorithm) to calculate paths for other algorithms. These options include support for A*, toggling quadratic approximation, and toggling grid path.

Change the Global Planner

The global planner used by the move_base node it's usually specified in the move_base parameters file. In order to do this, you will add one of the following lines to the parameters file:

```
[ ]: base_global_planner: "navfn/NavfnROS" # Sets the Navfn Planner

base_global_planner: "carrot_planner/CarrotPlanner" # Sets the Carrot
Planner

base_global_planner: "global_planner/GlobalPlanner" # Sets the Global
Planner
```

NOTE: It can also, though, be specified in the launch file, like it is our case.

Exercise 4.8

- Open Rviz and add a display in order to be able to visualize the global plan.
- Send a goal using the 2D Nav Goal tool. This goal must be "inside" an obstacle. Check what happens.
- Modify the **my_move_base_launch_2.launch** file so that it now uses the carrot planner.
- Repeat step b, and check what happens now.

Data for Exercise 4.8

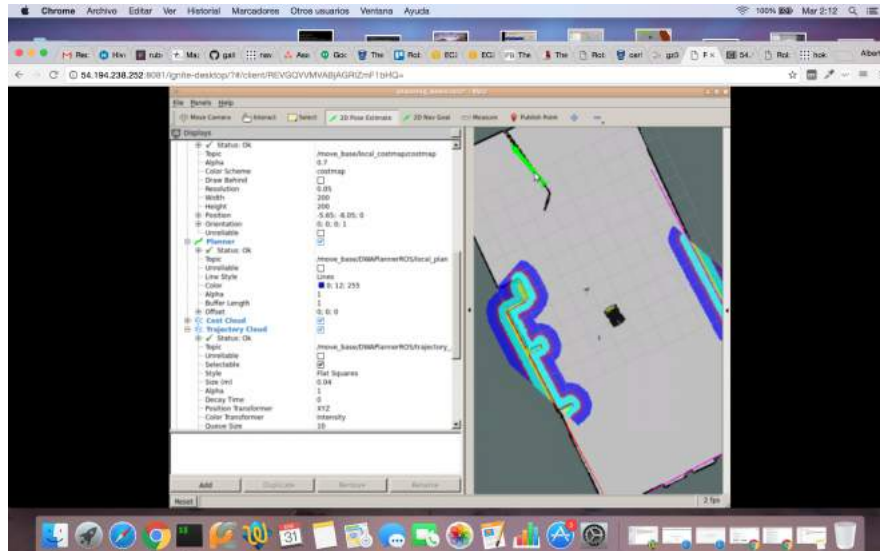
Check the following notes in order to complete the exercise:

Note 1: To make sure you've properly changed the global planner, you can use the following command:

`roscpp get /move_base/base_global_planner.`

Expected Result for Exercise 4.8

Sending goal:



Sending Goal

Navfn:

```
[WARN] [1485825088.240058940, 847.853000000]: Clearing costmap to unstuck robot (3.000000e).
```

```
[WARN] [1485825089.177535176, 848.253000000]: Rotate recovery behavior started.
```

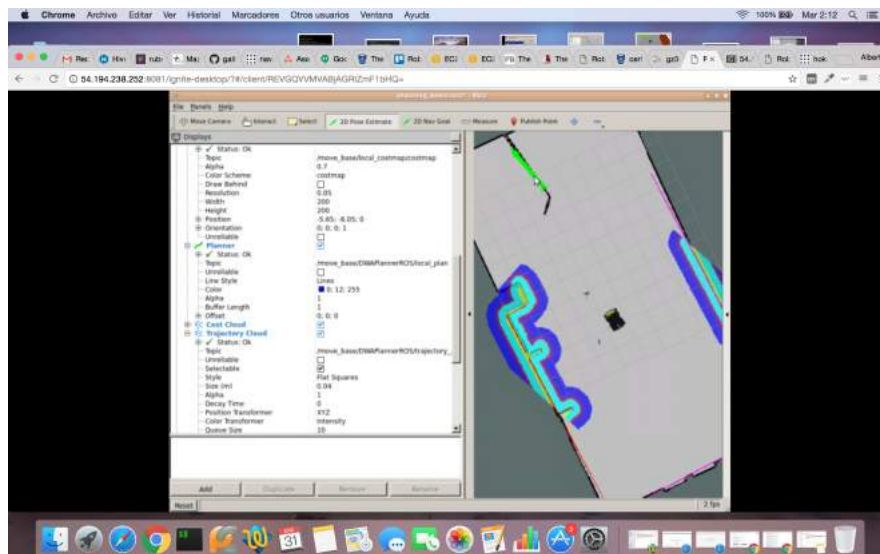
```
[WARN] [1485825100.899294797, 853.853000000]: Clearing costmap to unstuck robot (1.840000e).
```

```
[WARN] [1485825101.778941714, 854.353000000]: Rotate recovery behavior started.
```

```
[ERROR] [1485825132.992342099, 860.853000000]: Aborting because a valid plan could not be found. Even after executing all recovery behaviors.
```

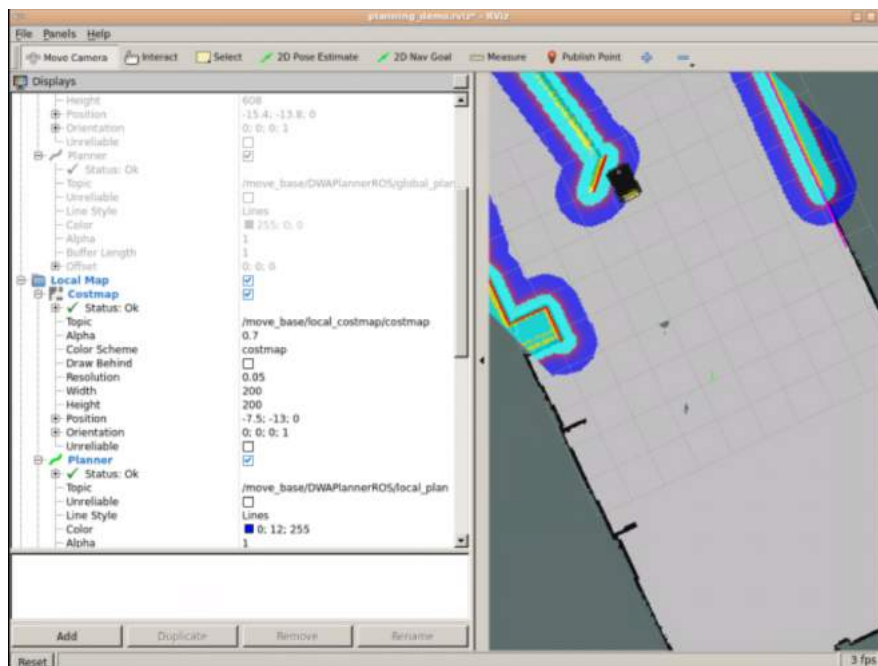
Error using Navfn

Sending goal at obstacle:



Sending Goal

Carrot Planner:



Planning Execution with Carrot Planner

The global planner also has its own parameters in order to customize its behaviour. The parameters for the global planner are also located in a YAML file. Depending on which global planner you use, the parameters to set will be different. In this course, we will have a look at the parameters for the navfn planner because it's the one that is most commonly used. If you are interested in seeing the parameters you can set for the other planners, you can have a look at them here:

carrot planner: http://wiki.ros.org/carrot_planner

global planner: http://wiki.ros.org/global_planner

Navfn Parameters

- **/allow_unknown (default: true):** Specifies whether or not to allow navfn to create plans that traverse unknown space. NOTE: if you are using a layered costmap_2d costmap with a voxel or obstacle layer, you must also set the track_unknown_space param for that layer to be true, or it will convert all of your unknown space to free space (which navfn will then happily go right through).
- **/planner_window_x (default: 0.0):** Specifies the x size of an optional window to restrict the planner to. This can be useful for restricting NavFn to work in a small window of a large costmap.
- **/planner_window_y (default: 0.0):** Specifies the y size of an optional window to restrict the planner to. This can be useful for restricting NavFn to work in a small window of a large costmap.
- **/default_tolerance (default: 0.0):** A tolerance on the goal point for the planner. NavFn will attempt to create a plan that is as close to the specified goal as possible, but no farther away than the default_tolerance.

- **cost_factor**
- **neutral_cost**
- **lethal_cost**

Here you can see an example of a global planner parameters file:

```
[ ]: NavfnROS:
    visualize_potential: false
    allow_unknown: false

    planner_window_x: 0.0
    planner_window_y: 0.0

    default_tolerance: 0.0
```

Exercise 4.9

Change the **use_dijkstra** parameter to false, and repeat Exercise 4.8. Check if something changes now.

So... summarizing:

Until now, you've seen that a global planner exists that is in charge of calculating a safe path in order to move the robot from an initial position to a goal position. You've also seen that there are different types of global planners, and that you can choose the global planner that you want to use. Finally, you've also seen that each planner has its own parameters, which modify the way the planner behaves.

But now, let me ask you a question. When you plan a trajectory, this trajectory has to be planned according to a map, right? A path without a map makes no sense. Ok, so... can you guess what map the global planner uses in order to calculate its path?

You may be tempted to think that the map that is being used is the map that you created in the Mapping Chapter (Chapter 2) of this course... but, let me tell you, that's not entirely true.

There exists another type of map: **the costmap**. Does it sound familiar? It should since you were introduced to it back in the first exercise of this chapter.

A costmap is a map that represents places that are safe for the robot to be in a grid of cells. Usually, the values in the costmap are binary, representing either free space or places where the robot would be in collision.

Each cell in a costmap has an integer value in the range {0,255}. There are some special values frequently used in this range, which work as follows:

- **255 (NO_INFORMATION)**: Reserved for cells where not enough information is known.
- **254 (LETHAL_OBSTACLE)**: Indicates that a collision-causing obstacle was sensed in this cell

- **253 (INSCRIBED_INFLATED_OBSTACLE)**: Indicates no obstacle, but moving the center of the robot to this location will result in a collision
- **0 (FREE_SPACE)**: Cells where there are no obstacles and, therefore, moving the center of the robot to this position will not result in a collision

There exist 2 types of costmaps: **global costmap** and **local costmap**. The main difference between them is, basically, the way they are built:

- The **global costmap** is created from a static map.
- The **local costmap** is created from the robot's sensor readings.

For now, we'll focus on the global costmap since it is the one used by the global planner. So, **the global planner uses the global costmap in order to calculate the path to follow.**

Let's do an exercise so that you can have a better idea of how a global costmap looks.

Exercise 4.10

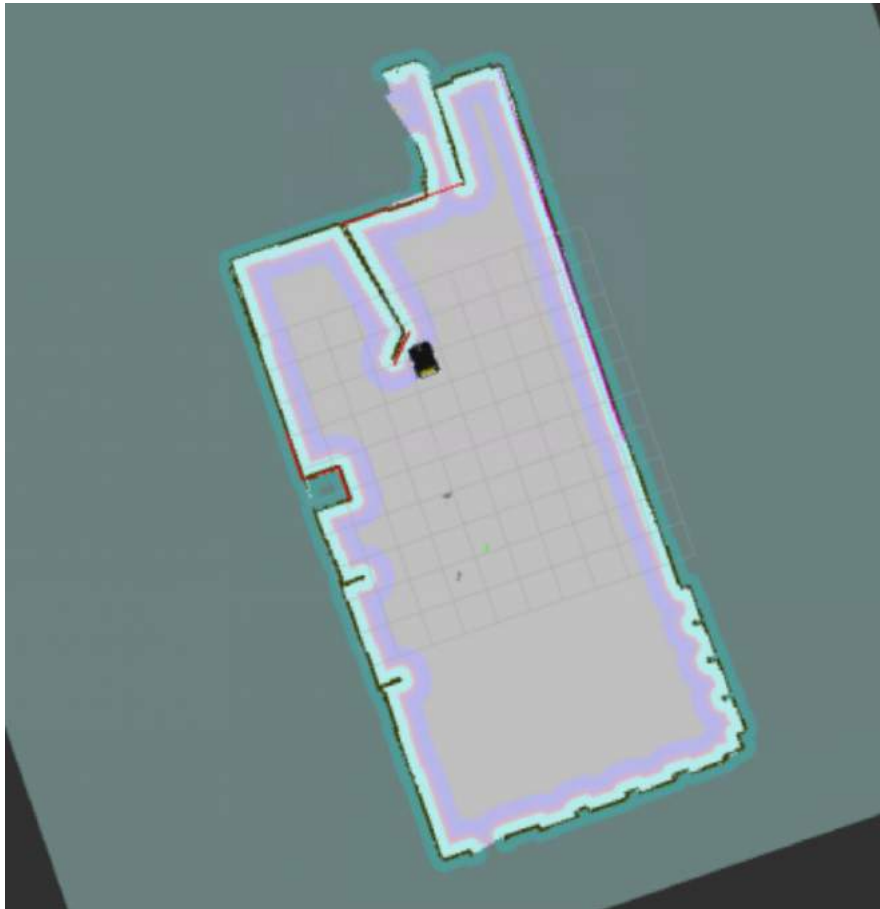
Launch Rviz and add the necessary display in order to visualize the global costmap.

Data for Exercise 4.10

Check the following notes in order to complete the exercise:

Note 1: You can change the colours used for the global costmap at the Color Scheme parameter in the RViz configuration:

Expected Result for Exercise 4.10



Global Costmap in RViz

So, now that you've seen how the global costmap looks, let's learn some more about it.

Global Costmap

The global costmap is created from a user-generated static map (as the one we created in the Mapping Chapter). In this case, the costmap is initialized to match the width, height, and obstacle information provided by the static map. This configuration is normally used in conjunction with a localization system, such as `amcl`. This is the method you'll use to initialize a **global costmap**. The global costmap also has its own parameters, which are defined in a YAML file. Next, you can see an example of a global costmap parameters file.

```
[ ]: global_frame: map
    static_map: true
    rolling_window: false

    plugins:
```

```

- {name: static,                                type: "costmap_2d::StaticLayer"}
- {name: inflation,                             type:
"costmap_2d::InflationLayer"}
- {name: obstacles,                             type: "costmap_2d::VoxelLayer"}

```

Costmap parameters are defined in 3 different files:

- A YAML file that sets the parameters for the global costmap (which is the one you've seen above). Let's name this file `global_costmap_params.yaml`.
- A YAML file that sets the parameters for the local costmap. Let's name this file `local_costmap_params.yaml`.
- A YAML file that sets the parameters for both the global and local costmaps. Let's name this file `common_costmap_params.yaml`.

Now, we'll focus on the global costmap parameters since it's the costmap that is used by the global planner.

Global Costmap Parameters

The parameters you need to know are the following:

- **global_frame (default: "/map")**: The global frame for the costmap to operate in.
- **static_map (default: true)**: Whether or not to use a static map to initialize the costmap.
- **rolling_window (default: false)**: Whether or not to use a rolling window version of the costmap. If the `static_map` parameter is set to true, this parameter must be set to false.
- **plugins**: Sequence of plugin specifications, one per layer. Each specification is a dictionary with a **name** and **type** fields. The name is used to define the parameter namespace for the plugin. This name will then be defined in the `common_costmap_parameters.yaml` file, which you will see in the next Unit. The type field actually defines the plugin (source code) that is going to be used.

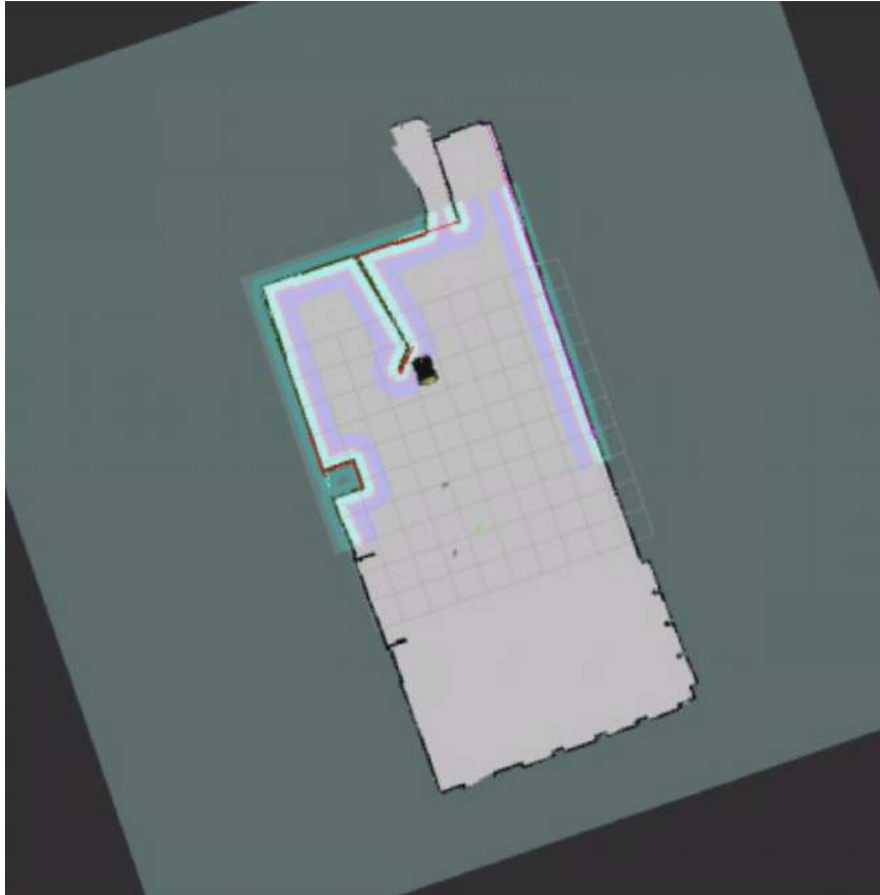
So, by setting the `static_map` parameter to true, and the `rolling_window` parameters to false, we will initialize the costmap by getting the data from a static map. This is the way you want to initialize a global costmap.

Exercise 4.11

- Add a file named `my_global_costmap_params.yaml` to the `params` directory of the package you created in Exercise 4.5.
- Copy the contents of the `costmap_global_static.yaml` file of the `husky_navigation` package into this file.
- Modify the `my_move_base_launch_2.launch` file you created in Exercise 4.5 so that it loads the global costmap parameters files you just created.

- d) Change the **rolling_window** parameter to true and launch the move_base node again.
- e) Check what changes you see in the visualization of the global costmap.

Expected Result for Exercise 4.11



Global Costmap with rolling window to True

The last parameter you need to know how to set is the plugins area. In the plugins area, we will add layers to the costmap configuration. Ok, but... what are layers?

In order to simplify (and clarify) the configuration of costmaps, ROS uses layers. Layers are like “blocks” of parameters that are related. For instance, the **static map, the sensed obstacles, and the inflation are separated into different layers**. These layers are defined in the **common_costmap_parameters.yaml** file, and then added to the **local_costmap_params.yaml** and **global_costmap_params.yaml** files.

To add a layer to a configuration file of a costmap, you will specify it in the plugins area. Have a look at the following line:

```
[ ]: plugins:
  - {name: static_map,          type: "costmap_2d::StaticLayer"}
```

Here, you're adding to your costmap configuration a layer named **static_map**, which will use the **costmap_2d::StaticLayer** plugin. You can add as many layers as you want:

```
[ ]: plugins:
  - {name: static_map,          type: "costmap_2d::StaticLayer"}
  - {name: obstacles,          type: "costmap_2d::VoxelLayer"}
  - {name: inflation,          type: "costmap_2d::InflationLayer"}
```

For instance, you can see an example on the local costmap parameters file shown above. In the case of the global costmap, you will usually use these 2 layers:

- **costmap_2d::StaticLayer**: Used to initialize the costmap from a static map.
- **costmap_2d::InflationLayer**: Used to inflate obstacles.

You may have noticed that the layers are just being added to the parameters file. That's true. Both in the global and local costmap parameters file, the layers are just added. The specific parameters of these layers are defined in the **common costmap parameters** file. We will have a look at this file later on in the chapter.

##

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 Path Planning Part 1: [Path Planning Part 1 Solutions](#)

Unit 5: Path Planning Part 2 (Obstacle Avoidance)

ROS Navigation

Unit 5: Path Planning Part 2 (Obstacle Avoidance)



Husky in house world

 Run on ROSDS

- ROSJect Link: <http://bit.ly/2nak1WG>
- Package Name: **husky_navigation_launch**
- Launch File: **main.launch**

SUMMARY

Estimated time to completion: **3 hours** What will you learn with this unit?

- What does Path Planning mean in ROS Navigation?

- How does Path Planning work?
- How does the move_base node work?
- What is a Costmap?

Until this point, you've seen how ROS plans a trajectory in order to move a robot from a starting position to a goal position. In this chapter, you will learn how ROS executes this trajectory, and avoids obstacles while doing so. You will also learn other important concepts regarding Path Planning, which we missed in the previous chapter. Finally, we will do a summary so that you can better understand the whole process. Are you up to the challenge? Let's get started then!

The Local Planner

Once the global planner has calculated the path to follow, this path is sent to the local planner. The local planner, then, will execute each segment of the global plan (let's imagine the local plan as a smaller part of the global plan). So, **given a plan to follow (provided by the global planner) and a map, the local planner will provide velocity commands in order to move the robot.**

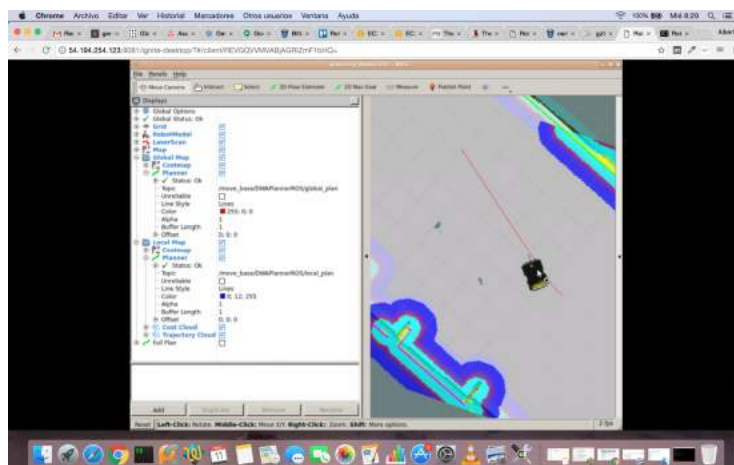
Unlike the global planner, the **local planner monitors the odometry and the laser data**, and chooses a collision-free local plan (let's imagine the local plan as a smaller part of the global plan) for the robot. So, the local planner **can recompute the robot's path on the fly** in order to keep the robot from striking objects, yet still allowing it to reach its destination.

Once the local plan is calculated, it is published into a topic named `/local_plan`. The local planner also publishes the portion of the global plan that it is attempting to follow into the topic `/global_plan`. Let's do an exercise so that you can see this better.

Exercise 5.1

- Open Rviz and add Displays in order to be able to visualize the `/global_plan` and the `/local_plan` topics of the local planner.
- Send a Goal Pose to the robot and visualize both topics.

Expected Result for Exercise 5.1



RVIZ with Husky making path planning global and local

As for the global planner, different types of local planners also exist. Depending on your setup (the robot you use, the environment it navigates, etc.) and the type of performance you want, you will use one or another. Let's have a look at the most important ones.

base_local_planner

The base local planner provides implementations of the Trajectory Rollout and the Dynamic Window Approach (DWA) algorithms in order to calculate and execute a global plan for the robot.

Summarizing, the basic idea of how this algorithms works is as follows:

- Discretely sample from the robot's control space
- For each sampled velocity, perform forward simulations from the robot's current state to predict what would happen if the sampled velocity was applied.
- Evaluate each trajectory resulting from the forward simulation.
- Discard illegal trajectories.
- Pick the highest-scoring trajectory and send the associated velocities to the mobile base.
- Rinse and Repeat.

DWA differs from Trajectory Rollout in how the robot's space is sampled. Trajectory Rollout samples are from the set of achievable velocities over the entire forward simulation period given the acceleration limits of the robot, while DWA samples are from the set of achievable velocities for just one simulation step given the acceleration limits of the robot.

DWA is a more efficient algorithm because it samples a smaller space, but may be outperformed by Trajectory Rollout for robots with low acceleration limits because DWA does not forward simulate constant accelerations. In practice, DWA and Trajectory Rollout perform similarly, so **it's recommended to use DWA because of its efficiency gains.**

The DWA algorithm of the base local planner has been improved in a new local planner separated from this one. That's the DWA local planner we'll see next.

dwa_local_planner

The DWA local planner provides an implementation of the Dynamic Window Approach algorithm. It is basically a re-write of the base local planner's DWA (Dynamic Window Approach) option, but the code is a lot cleaner and easier to understand, particularly in the way that the trajectories are simulated. So, for applications that use the DWA approach for local planning, the `dwa_local_planner` is probaly the best choice. This is the **most commonly used option.**

eband_local_planner

The eband local planner implements the Elastic Band method in order to calculate the local plan to follow.

teb_local_planner

The teb local planner implements the Timed Elastic Band method in order to calculate the local plan to follow.

Change the local planner

As for the global planner, you can also select which local planner you want to use. This is also done in the move_base node parameters file, by adding one of the following lines:

```
[ ]: base_local_planner: "base_local_planner/TrajectoryPlannerROS" # Sets
    the Trajectory Rollout algorithm from base local
    planner

    base_local_planner: "dwa_local_planner/DWAPlannerROS" # Sets the dwa
    local planner

    base_local_planner: "eband_local_planner/EBandPlannerROS" # Sets the
    eband local planner

    base_local_planner: "teb_local_planner/TebLocalPlannerROS" # Sets the
    teb local planner
```

Exercise 5.2

Change the local planner in the **my_move_base_launch_2.launch** file to use the **teb_local_planner** and check how it performs.

NOTE: Make sure to switch back to the DWAPlanner after you finish with this Exercise.

Expected Result for Exercise 5.2

```
ubuntu@ip-172-31-32-206:~$ rosparam get /move_base/base_local_planner
teb_local_planner/TebLocalPlannerROS
```

Teb local planner param

NOTE: Make sure to switch back to the DWAPlanner after you finish with the previous Exercise.

As you would expect, the local planner also has its own parameters. These parameters will be different depending on the local planner you use. In this course, we'll be focusing on the DWA local planner parameters, since it's the most common choice. Anyways, if you want to check the specific parameters for the other local planners, you can have a look at them here:

base_local_planner: http://wiki.ros.org/base_local_planner

eband_local_planner: http://wiki.ros.org/eband_local_planner

teb_local_planner: http://wiki.ros.org/teb_local_planner

dwa local planner Parameters

The parameters for the local planner are set in another YAML file. The most important parameters for the DWA local planner are the following:

Robot Configuration Parameters

- **/acc_lim_x (default: 2.5)**: The x acceleration limit of the robot in meters/sec²
- **/acc_lim_th (default: 3.2)**: The rotational acceleration limit of the robot in radians/sec²
- **/max_trans_vel (default: 0.55)**: The absolute value of the maximum translational velocity for the robot in m/s
- **/min_trans_vel (default: 0.1)**: The absolute value of the minimum translational velocity for the robot in m/s
- **/max_vel_x (default: 0.55)**: The maximum x velocity for the robot in m/s.
- **/min_vel_x (default: 0.0)**: The minimum x velocity for the robot in m/s, negative for backwards motion.
- **/max_rot_vel (default: 1.0)**: The absolute value of the maximum rotational velocity for the robot in rad/s
- **/min_rot_vel (default: 0.4)**: The absolute value of the minimum rotational velocity for the robot in rad/s

Goal Tolerance Parameters

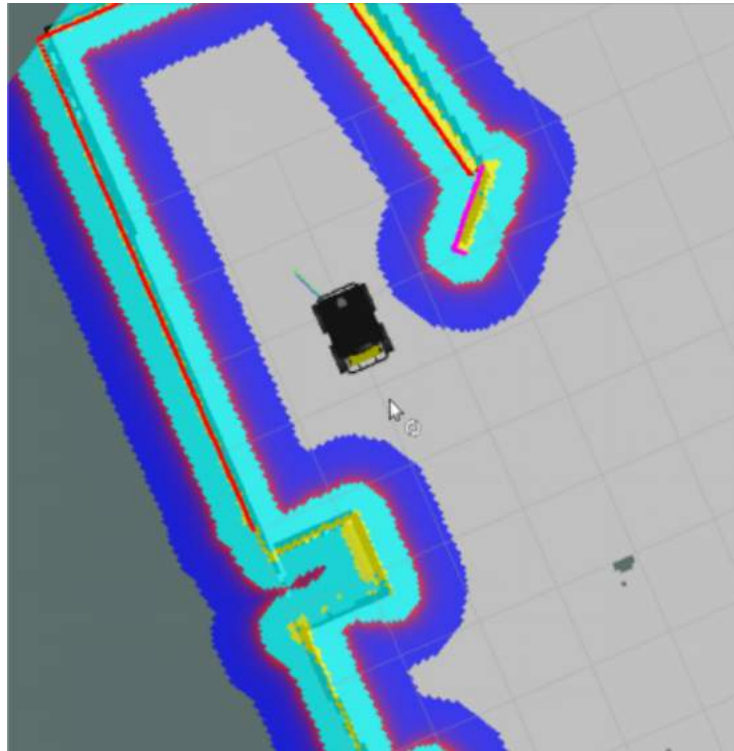
- **/yaw_goal_tolerance (double, default: 0.05)**: The tolerance, in radians, for the controller in yaw/rotation when achieving its goal
- **/xy_goal_tolerance (double, default: 0.10)**: The tolerance, in meters, for the controller in the x and y distance when achieving a goal
- **/latch_xy_goal_tolerance (bool, default: false)**: If goal tolerance is latched, if the robot ever reaches the goal xy location, it will simply rotate in place, even if it ends up outside the goal tolerance while it is doing so.

Exercise 5.3

- Open the **my_move_base_params.yaml** file you created in the previous Chapter to edit it.
- Modify the **xy_goal_tolerance** parameter of the DWAPlanner and set it to a higher value.
- Check if you notice any differences in the performance.

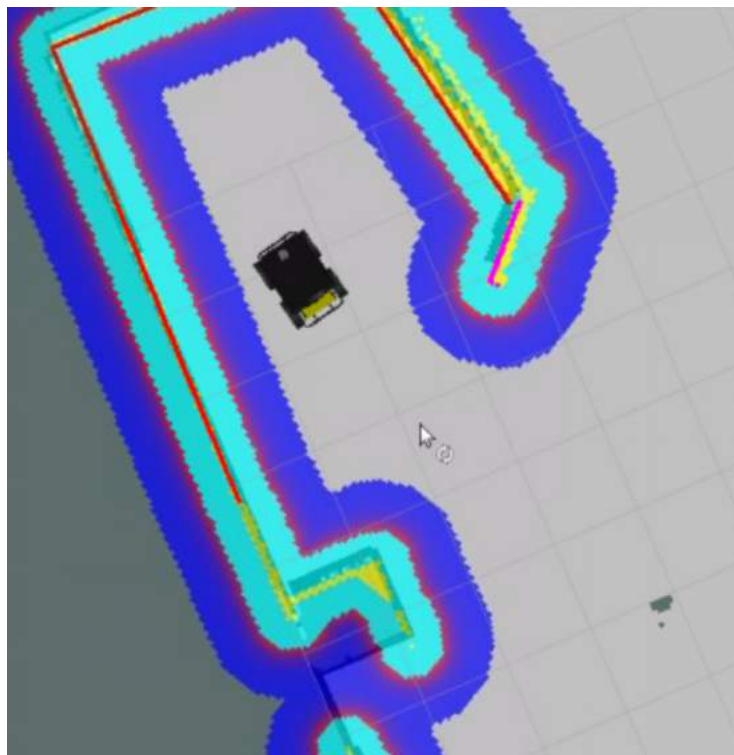
Expected Result for Exercise 5.3

High XY tolerance:



XY High tolerance values

Low XY tolerance:



XY Low tolerance values

As you've seen in the exercise, the higher you set the goal tolerances in your parameters file, the less accurate the robot will be in order to set a goal as reached.

Forward Simulation Parameters

- **/sim_time (default: 1.7)**: The amount of time to forward-simulate trajectories in seconds
- **/sim_granularity (default: 0.025)**: The step size, in meters, to take between points on a given trajectory
- **/vx_samples (default: 3)**: The number of samples to use when exploring the x velocity space
- **/vy_samples (default: 10)**: The number of samples to use when exploring the y velocity space
- **/vtheta_samples (default: 20)**: The number of samples to use when exploring the theta velocity space

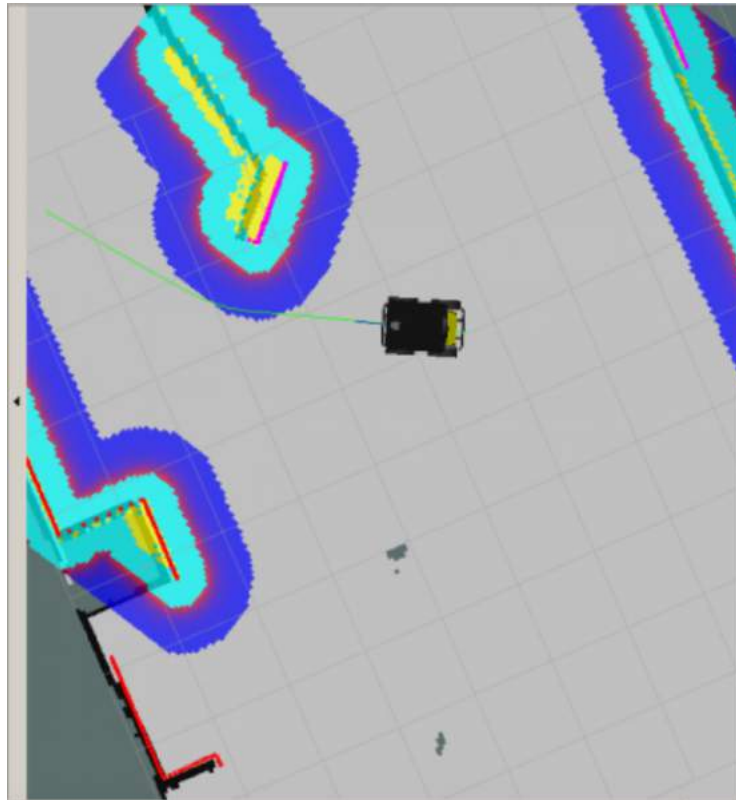
Exercise 5.4

- Modify the **sim_time** parameter in the local planner parameters file and set it to 4.0.
- Check if you notice any differences in the performance or visualization of the local planner.

Expected Result for Exercise 5.4

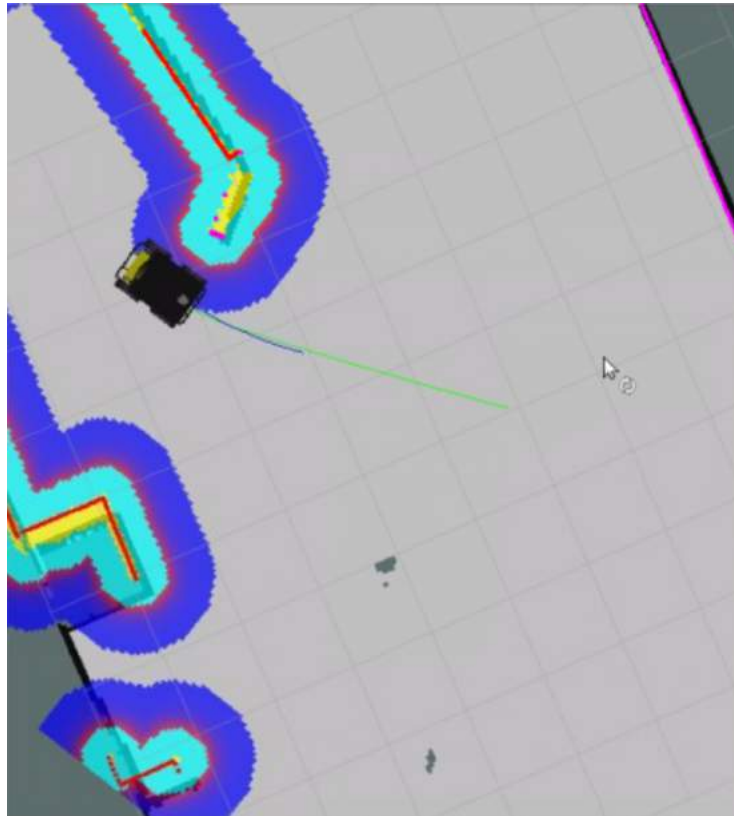
Regular sim_time:

5 - Path Planning Part 2 (Obstacle Avoidance)



Regular simulation time

High sim_time:



High simulation time

Trajectory Scoring Parameters

- **/path_distance_bias (default: 32.0)**: The weighting for how much the controller should stay close to the path it was given
- **/goal_distance_bias (default: 24.0)**: The weighting for how much the controller should attempt to reach its local goal; also controls speed
- **/occdist_scale (default: 0.01)**: The weighting for how much the controller should attempt to avoid obstacles

Here you have an example of the `dwa_local_planner_params.yaml`:

```
[ ]: DWAPlanerROS:
```

```
  # Robot Configuration Parameters - Kobuki
```

```
  max_vel_x: 0.5 # 0.55
```

```
  min_vel_x: 0.0
```

```
  max_vel_y: 0.0 # diff drive robot
```

```
  min_vel_y: 0.0 # diff drive robot
```

```
  max_trans_vel: 0.5 # choose slightly less than the base's capability
```


5 - Path Planning Part 2 (Obstacle Avoidance)

```
min_trans_vel: 0.1 # this is the min trans velocity when there is
negligible rotational velocity
trans_stopped_vel: 0.1

# Warning!
# do not set min_trans_vel to 0.0 otherwise dwa will always think
translational velocities
# are non-negligible and small in place rotational velocities will
be created.

max_rot_vel: 5.0 # choose slightly less than the base's capability
min_rot_vel: 0.4 # this is the min angular velocity when there is
negligible translational velocity
rot_stopped_vel: 0.4

acc_lim_x: 1.0 # maximum is theoretically 2.0, but we
acc_lim_theta: 2.0
acc_lim_y: 0.0 # diff drive robot

# Goal Tolerance Parameters
yaw_goal_tolerance: 0.3 # 0.05
xy_goal_tolerance: 0.15 # 0.10
# latch_xy_goal_tolerance: false

# Forward Simulation Parameters
sim_time: 1.0 # 1.7
vx_samples: 6 # 3
vy_samples: 1 # diff drive robot, there is only one sample
vtheta_samples: 20 # 20

# Trajectory Scoring Parameters
path_distance_bias: 64.0 # 32.0 - weighting for how much it
should stick to the global path plan
goal_distance_bias: 24.0 # 24.0 - weighting for how much it
should attempt to reach its goal
occdist_scale: 0.5 # 0.01 - weighting for how much the
controller should avoid obstacles
forward_point_distance: 0.325 # 0.325 - how far along to place an
additional scoring point
stop_time_buffer: 0.2 # 0.2 - amount of time a robot
must stop in before colliding for a valid traj.
scaling_speed: 0.25 # 0.25 - absolute velocity at which
to start scaling the robot's footprint
max_scaling_factor: 0.2 # 0.2 - how much to scale the
robot's footprint when at speed.

# Oscillation Prevention Parameters
oscillation_reset_dist: 0.05 # 0.05 - how far to travel before
```

```

resetting oscillation flags

# Debugging
publish_traj_pc : true
publish_cost_grid_pc: true
global_frame_id: odom

# Differential-drive robot configuration - necessary?
# holonomic_robot: false

```

Exercise 5.5

Change the **path_distance_bias** parameter in the local planner parameters file.

Check if you notice any differences in the performance.

In the global planner section, we already introduced you to costmaps, focusing on the global costmap. So, now it's time to talk a little bit about the local costmap.

Local Costmap

The first thing you need to know is that the **local planner uses the local costmap in order to calculate local plans**.

Unlike the global costmap, the local costmap is created directly from the robot's sensor readings. Given a width and a height for the costmap (which are defined by the user), it keeps the robot in the center of the costmap as it moves throughout the environment, dropping obstacle information from the map as the robot moves.

Let's do an exercise so that you can get a better idea of how the local costmap looks, and how to differentiate a local costmap from a global costmap.

Exercise 5.6

a) Open Rviz and add the proper displays in order to visualize the global and the local costmaps.

b) Execute the following command in order to spawn an obstacle in the room.

Check if you already have the **object.urdf** file in your workspace. If you don't have it yet, you'll need to execute the following command in order to move it to your workspace.

Execute in WebShell #2

```
[ ]: cp /home/simulations/public_sim_ws/src/all/turtlebot/turtlebot_navigation_gazebo/urdf/object.urdf /home/user/catkin_ws/src
```

Now, spawn the object.

Execute in WebShell #2

```
[ ]: rosrun gazebo_ros spawn_model -file  
    /home/user/catkin_ws/src/object.urdf -urdf -x 0 -y 0 -z 1 -model  
    my_object
```

c) Launch the keyboard Teleop and move close to the spawned object.

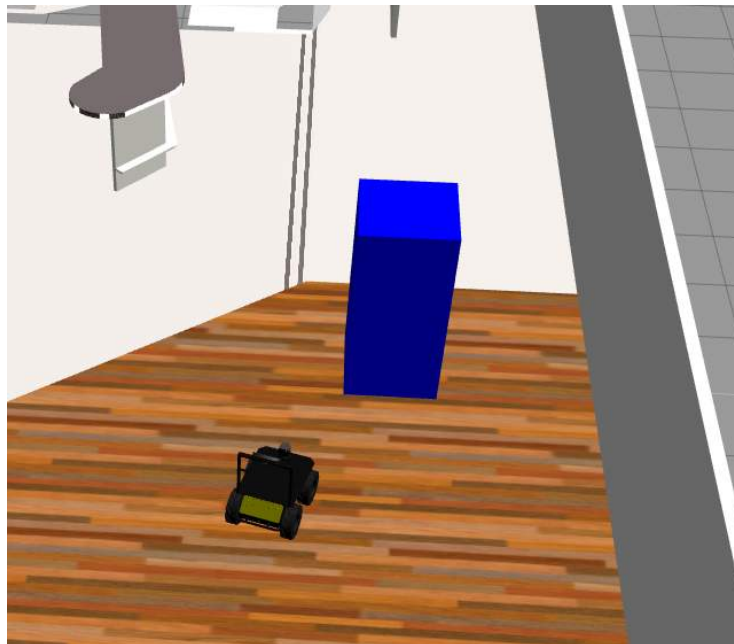
Execute in WebShell #3

```
[ ]: roslaunch husky_launch keyboard_teleop.launch
```

d) Check the differences between the global and local Costmaps.

Expected Result for Exercise 5.6

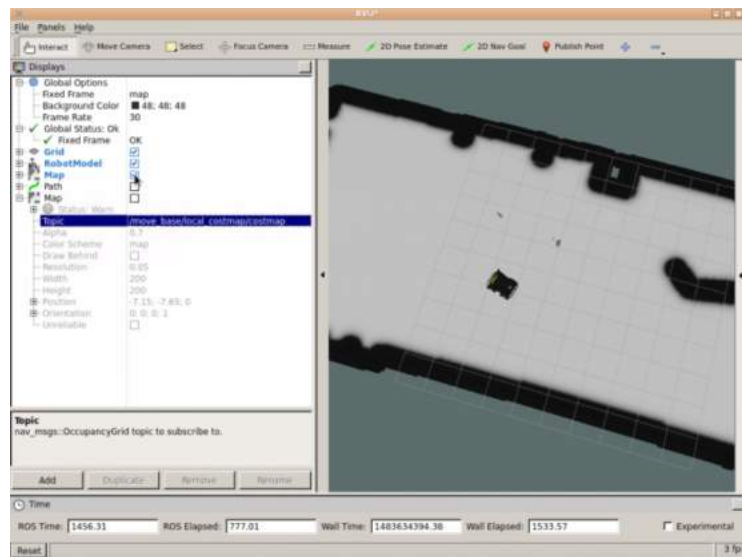
Husky facing the spawned obstacle:



Husky with obstacle in front

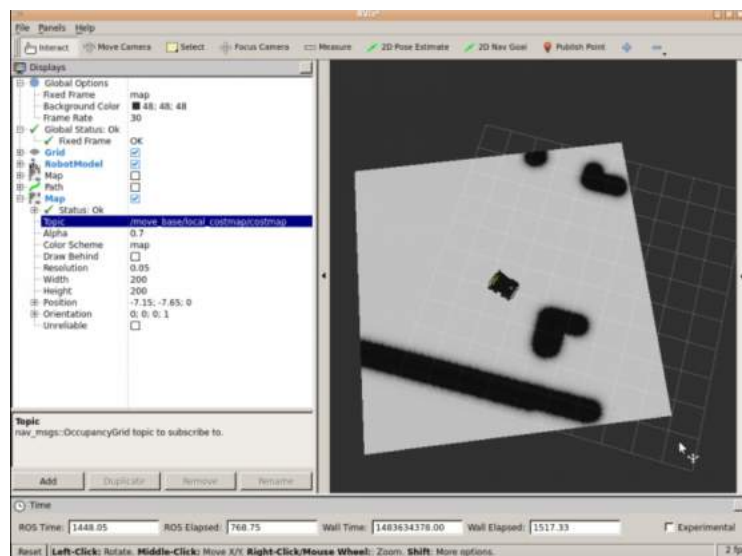
Global Costmap (Obstacle doesn't appear):

5 - Path Planning Part 2 (Obstacle Avoidance)



Obstacle doesn't appear in the Global costmap

Local Costmap (Obstacle does appear):



Obstacle appears in the Local costmap

So, as you've seen in the previous exercise, the **local costmap does detect new objects that appear in the simulation, while the global costmap doesn't**.

This happens, as you may have already deduced, because the global costmap is created from a static map file. This means that the costmap won't change, even if the environment does. The local costmap, instead, is created from the robot's sensor readings, so it will always keep updating with new readings from the sensors.

Since the global costmap and the local costmap don't have the same behavior, the parameters file must also be different. Let's have a look at the most important parameters that we need to set for the local costmap.

Local Costmap Parameters

The parameters you need to know are the following:

- **global_frame**: The global frame for the costmap to operate in. In the local costmap, this parameter has to be set to `"/odom"`.
- **static_map** : Whether or not to use a static map to initialize the costmap. In the local costmap, this parameter has to be set to `"false"`.
- **rolling_window**: Whether or not to use a rolling window version of the costmap. If the `static_map` parameter is set to true, this parameter must be set to false. In the local costmap, this parameter has to be set to `"true"`.
- **width**: The width of the costmap.
- **height**: The height of the costmap.
- **update_frequency**: The frequency in Hz for the map to be updated.
- **plugins**: Sequence of plugin specifications, one per layer. Each specification is a dictionary with a name and type fields. The name is used to define the parameter namespace for the plugin.

So, by setting the **static_map** parameter to false, and the **rolling_window** parameter to true, we are indicating that we don't want the costmap to be initialized from a static map (as we did with the global costmap), but to be built from the robot's sensor readings. Also, since we won't have any static map, the **global_frame** parameter needs to be set to **odom**. Finally, we also need to set a **width** and a **height** for the costmap, because in this case, it can't get these values from a static map.

Let's do a simple exercise so that you can modify some of these parameters.

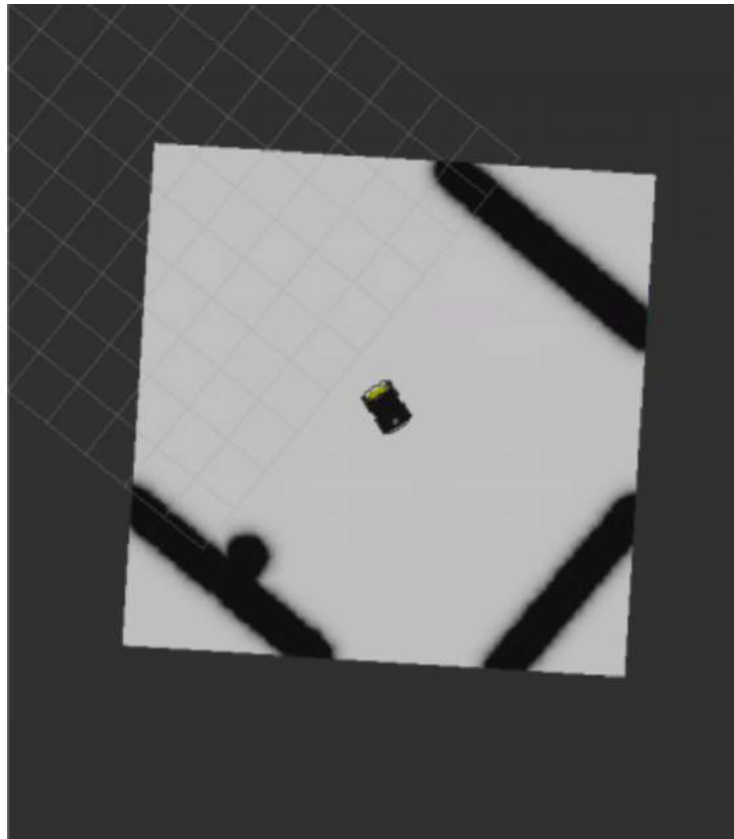
Exercise 5.7

- Add a file named **my_local_costmap_params.yaml** to the params directory of the package you created in Exercise 4.5.
- Copy the contents of the **costmap_local.yaml** file of the `husky_navigation` package into this file.
- Modify the **my_move_base_launch_2.launch** file you created in exercise 4.5 so that it loads the local costmap parameters file you just created.
- Launch Rviz and visualize the local costmap again. Visualize both the map and costmap modes.
- Modify the **width** and **height** parameters and put them to 5. Visualize the costmap again.

NOTE: Keep in mind that, for your case, the **width** and **height** parameters are being loaded directly from the launch file. So, you will have to remove from the launch file and add them to the parameters file.

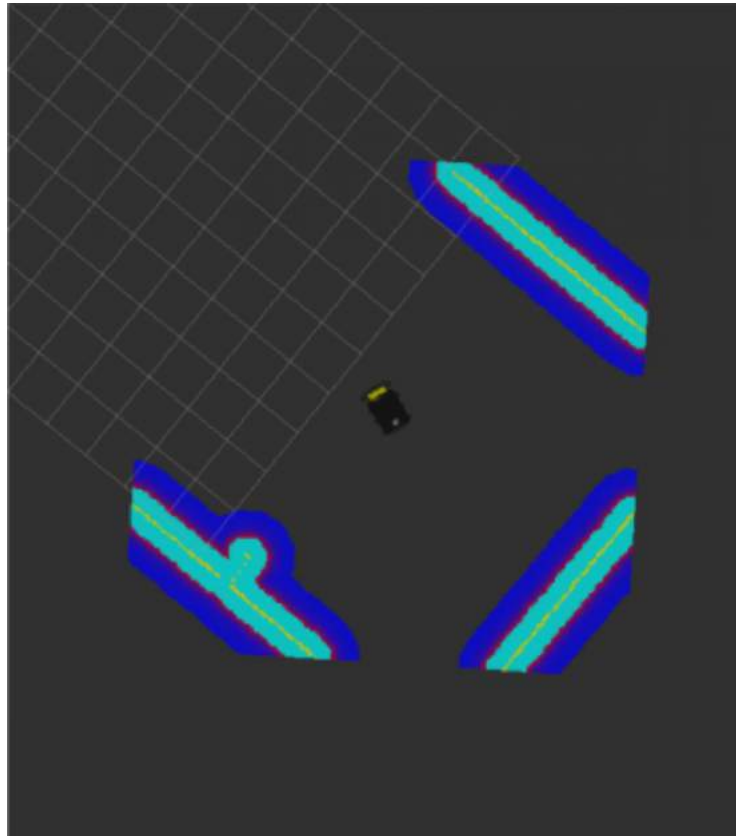
Expected Result for Exercise 5.7

10x10 costmap (map view):



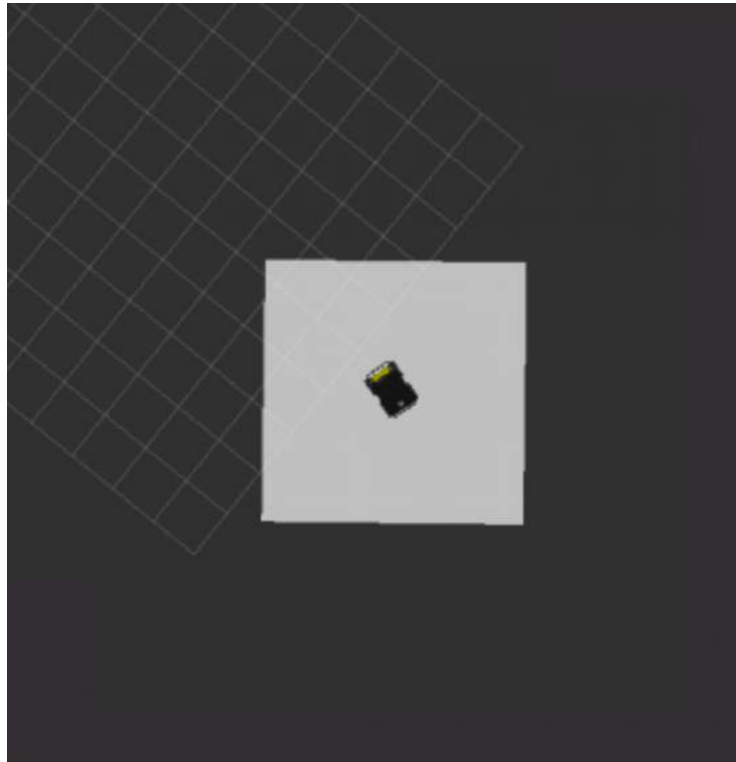
Local Costmap 10 by 10 area

10x10 costmap (costmap view):



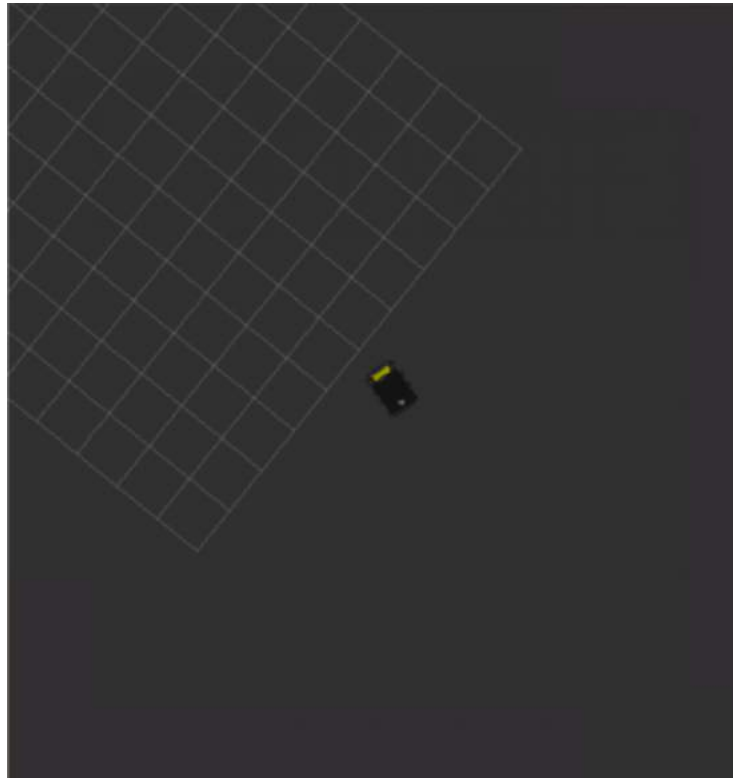
CostMap 10 by 10 view in RVIZ

5x5 costmap (map view):



Local Costmap 5 by 5 area

5x5 costmap (costmap view):



Local Costmap 5 by 5 View

As you've seen in the previous exercise, it's very important to set a correct width and height for your costmap. Depending on the environment you want to navigate, you will have set one value or another in order to properly visualize the obstacles.

Just as we saw for the global costmap, layers can also be added to the local costmap. In the case of the local costmap, you will usually add these 2 layers:

- **costmap_2d::ObstacleLayer**: Used for obstacle avoidance.
- **costmap_2d::InflationLayer**: Used to inflate obstacles.

So, you will end up with something like this:

```
[ ]: plugins:
  - {name: obstacle_layer,      type: "costmap_2d::ObstacleLayer"}
  - {name: inflation_layer,    type: "costmap_2d::InflationLayer"}
```

VERY IMPORTANT: Note that the **obstacle layer** uses different plugins for the **local costmap** and the **global costmap**. For the local costmap, it uses the **costmap_2d::ObstacleLayer**, and for the global costmap it uses the **costmap_2d::VoxelLayer**. This is very important because it is a common error in Navigation to use the wrong plugin for the obstacle layers.

As you've already seen through the exercises, the local costmap keeps updating itself. These update cycles are made at a rate specified by the **update_frequency** parameter. Each cycle

works as follows:

- Sensor data comes in.
- Marking and clearing operations are performed.
- The appropriate cost values are assigned to each cell.
- Obstacle inflation is performed on each cell with an obstacle. This consists of propagating cost values outwards from each occupied cell out to a specified inflation radius.

Exercise 5.8

- In the local costmap parameters file, change the **update_frequency** parameter of the map to be slower.
- Repeat Exercise 5.6 again, and see what happens now.

Expected Result for Exercise 5.8

The object in the costmap is spawned with a little delay

Now, you may be wondering... what are the marking and clearing operations you mentioned above?

As you already know, the costmap automatically subscribes to the sensor topics and updates itself according to the data it receives from them. Each sensor is used to either **mark** (insert obstacle information into the costmap), **clear** (remove obstacle information from the costmap), or both.

A **marking** operation is just an index into an array to change the cost of a cell. A **clearing** operation, however, consists of raytracing through a grid from the origin of the sensor outwards for each observation reported.

The marking and clearing operations can be defined in the **obstacle layer**.

At this point, we can almost say that you already know how to configure both global and local costmaps. But if you remember, there's still a parameters file we haven't talked about. That's the **common costmap parameters file**. These parameters will affect both the global and the local costmap.

Basically, the parameters you'll have to set in this file are the following:

- **footprint**: Footprint is the contour of the mobile base. In ROS, it is represented by a two-dimensional array of the form $[x_0, y_0], [x_1, y_1], [x_2, y_2], \dots]$. This footprint will be used to compute the radius of inscribed circles and circumscribed circles, which are used to inflate obstacles in a way that fits this robot. Usually, for safety, we want to have the footprint be slightly larger than the robot's real contour.
- **robot_radius**: In case the robot is circular, we will specify this parameter instead of the footprint.

- **layers parameters:** Here we will define the parameters for each layer.

Each layer has its own parameters.

Obstacle Layer The obstacle layer is in charge of the **marking and clearing operations**.

As you already know, the costmap automatically subscribes to the sensor topics and updates itself according to the data it receives from them. Each sensor is used to either mark (insert obstacle information into the costmap), clear (remove obstacle information from the costmap), or both.

A marking operation is just an index into an array to change the cost of a cell. A clearing operation, however, consists of raytracing through a grid from the origin of the sensor outwards for each observation reported.

The marking and clearing operations can be defined in the obstacle layer.

- **max_obstacle_height (default: 2.0):** The maximum height of any obstacle to be inserted into the costmap, in meters. This parameter should be set to be slightly higher than the height of your robot.
- **obstacle_range (default: 2.5):** The default maximum distance from the robot at which an obstacle will be inserted into the cost map, in meters. This can be overridden on a per-sensor basis.
- **raytrace_range (default: 3.0):** The default range in meters at which to raytrace out obstacles from the map using sensor data. This can be overridden on a per-sensor basis.
- **observation_sources (default: “”):** A list of observation source names separated by spaces. This defines each of the source_name namespaces defined below.

Each source_name in observation_sources defines a namespace in which parameters can be set:

- **/source_name/topic (default: source_name):** The topic on which sensor data comes in for this source. Defaults to the name of the source.
- **/source_name/data_type (default: “PointCloud”):** The data type associated with the topic, right now only “PointCloud,” “PointCloud2,” and “LaserScan” are supported.
- **/source_name/clearing (default: false):** Whether or not this observation should be used to clear out freespace.
- **/source_name/markings (default: true):** Whether or not this observation should be used to mark obstacles.
- **/source_name/inf_is_valid (default: false):** Allows for Inf values in “LaserScan” observation messages. The Inf values are converted to the laser’s maximum range.

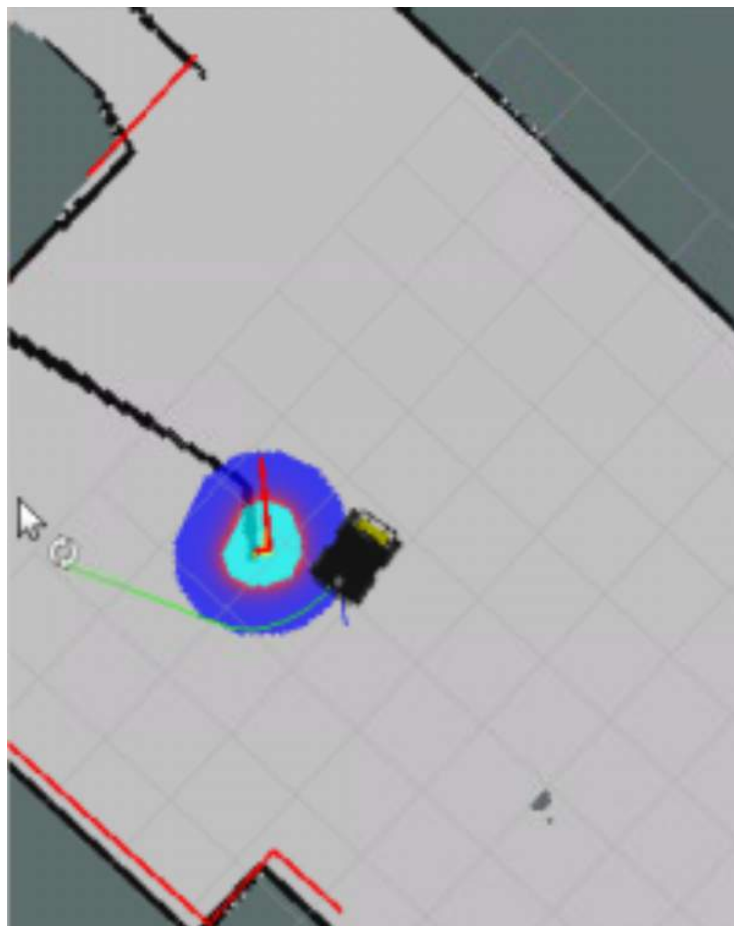
VERY IMPORTANT: A very important thing to keep in mind is that the **obstacle layer** uses different plugins for the **local costmap** and the **global costmap**. For the local costmap, it uses the

costmap_2d::ObstacleLayer, and for the global costmap it uses the **costmap_2d::VoxelLayer**. This is very important because it is a common error in Navigation to use the wrong plugin for the obstacle layers.

Exercise 5.9

- a) Add a file named **my_common_costmap_params.yaml** to the params directory of the package you created in Exercise 4.5.
- b) Copy the contents of the **costmap_common.yaml** file of the **husky_navigation** package into this new file.
- a) Now, modify the **obstacle_range** parameter and set it to 1.
- b) Move the robot close to an obstacle and see what happens.

Expected Result for Exercise 5.9



Obstacle range changed

Inflation Layer The inflation layer is in charge of performing inflation in each cell with an obstacle.

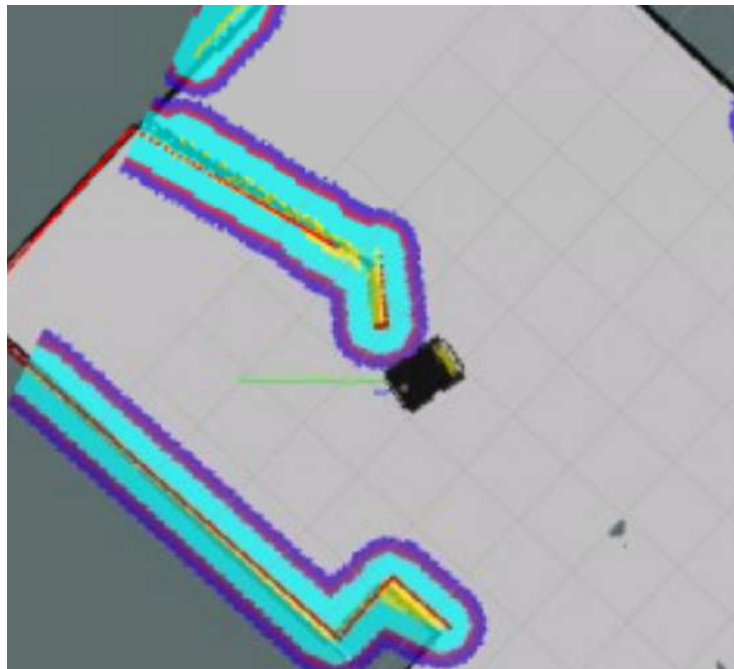
- **inflation_radius (default: 0.55):** The radius in meters to which the map inflates obstacle cost values.
- **cost_scaling_factor (default: 10.0):** A scaling factor to apply to cost values during inflation.

Exercise 5.10

- Now, modify the **inflation_radius** parameter of the costmap to be slower.
- Move close to an object and check the difference.

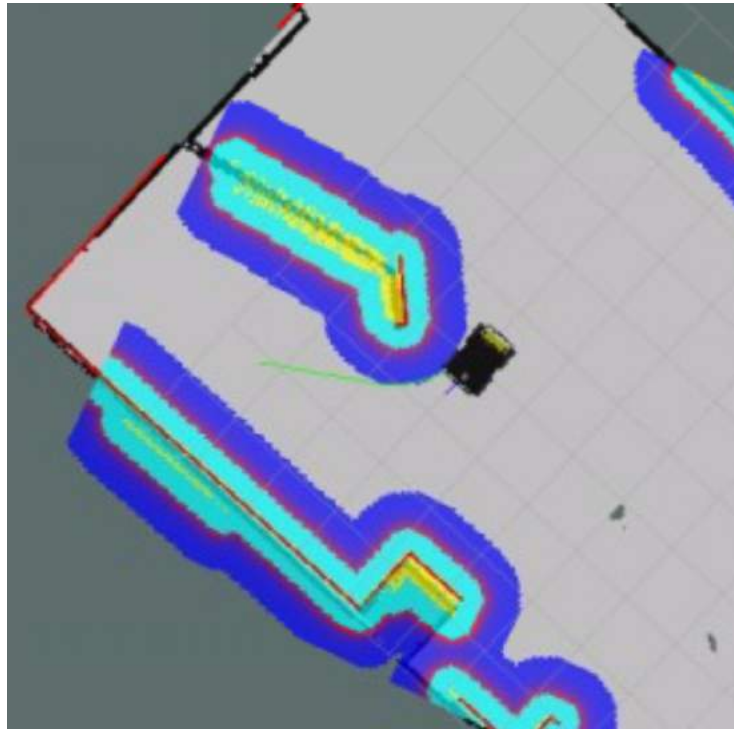
Expected Result for Exercise 5.10

Low inflation:



Low Infiltration

High inflation:



High infiltration value

Static Layer The static layer is in charge of providing the static map to the costmaps that require it (global costmap).

- **map_topic (string, default: "map"):** The topic that the costmap subscribes to for the static map.

Recovery Behaviors

It could happen that while trying to perform a trajectory, the robot gets stuck for some reason. Fortunately, if this happens, the ROS Navigation Stack provides methods that can help your robot to get unstuck and continue navigating. These are the **recovery behaviors**.

The ROS Navigation Stack provides 2 recovery behaviors: **clear costmap** and **rotate recovery**.

In order to enable the recovery behaviors, we need to set the following parameter in the move_base parameters file:

- **recovery_behavior_enabled (default: true):** Enables or disables the recovery behaviors.

Rotate Recovery

Basically, the rotate recovery behavior is a simple recovery behavior that attempts to clear out space by rotating the robot 360 degrees. This way, the robot may be able to find an obstacle-free path to continue navigating.

It has some parameters that you can customize in order to change or improve its behavior:

Rotate Recovery Parameters

- **/sim_granularity (double, default: 0.017):** The distance, in radians, between checks for obstacles when checking if an in-place rotation is safe. Defaults to 1 degree.
- **/frequency (double, default: 20.0):** The frequency, in HZ, at which to send velocity commands to the mobile base.

Other Parameters

IMPORTANT: These parameters are already set when using the `base_local_planner` local planner; they only need to be set explicitly for the recovery behavior if a different local planner is used.**

- **/yaw_goal_tolerance (double, default: 0.05):** The tolerance, in radians, for the controller in yaw/rotation when achieving its goal
- **/acc_lim_th (double, default: 3.2):** The rotational acceleration limit of the robot, in radians/sec²
- **/max_rotational_vel (double, default: 1.0):** The maximum rotational velocity allowed for the base, in radians/sec
- **/min_in_place_rotational_vel (double, default: 0.4):** The minimum rotational velocity allowed for the base while performing in-place rotations, in radians/sec

These parameters are set in the `move_base` parameters file.

Exercise 5.11

Force the robot to move to an obstacle and check if the Rotate Recovery behavior launches.

Clear Costmap

The clear costmap recovery is a simple recovery behavior that clears out space by clearing obstacles outside of a specified region from the robot's map. Basically, the local costmap reverts to the same state as the global costmap.

The `move_base` node also provides a service in order to clear out obstacles from a costmap. This service is called `/move_base/clear_cotmaps`.

Bear in mind that by clearing obstacles from a costmap, you will make these obstacles invisible to the robot. So, be careful when calling this service since it could cause the robot to start

hitting obstacles.

Exercise 5.12

- If there's not one yet, add an object to the scene that doesn't appear in the global costmap. For instance, the object in Exercise 5.6.
- Move the robot so that it detects this new obstacle in the local costmap.
- Turn the robot so that it doesn't see anymore the obstacle (the laser beams don't detect it).
- Perform a call to the **/clear_costmaps** service through the WebShell, and check what happens.

Execute in WebShell #1

```
[ ]: rosservice call /move_base/clear_costmaps "{}"
```

Expected Result for Exercise 5.12

Object detected by the laser and placed into the local Costmap:

Husky turns and laser doesn't detect the object anymore, but it still appears in the local costmap.

After calling the **/move_base/clear_costmaps** service, the object is cleared from the local costmap:

Oscillation Suppression

Oscillation occurs when, in any of the x, y, or theta dimensions, positive and negative values are chosen consecutively.

To prevent oscillations, when the robot moves in any direction, the opposite direction is marked invalid for the next cycle, until the robot has moved beyond a certain distance from the position where the flag was set.

In order to manage this issue, 2 parameters exist that you can set in the move_base parameters file.

- **oscillation_timeout (double, default: 0.0)**: How long, in seconds, to allow for oscillation before executing recovery behaviors. A value of 0.0 corresponds to an infinite timeout.
- **oscillation_distance (double, default: 0.5)**: How far, in meters, the robot must move to not be considered oscillating. Moving this far resets the timer counting up to the `~oscillation_timeout`

Recap

Congratulations! At this point, you've already seen almost all of the important parts that this chapter covers. And since this is the last chapter of the course, this means that you are very close to knowing how to deal with ROS Navigation in its entirety!

Anyways, you may be overwhelmed with all of the information that you've received about Path Planning. That's why I think this is a good moment to do a summary of all that you've seen in this chapter up until now. Let's begin!

The move_base node

The move_base node is, basically, the node that coordinates all of the Path Planning System. It takes a goal pose as input, and outputs the necessary velocity commands in order to move the robot from an initial pose to the specified goal pose. In order to achieve this, the move_base node manages a whole internal process where it takes place for different parts:

- global planner
- local planner
- costmaps
- recovery behaviors

The global planner

When a new goal is received by the move_base node, it is immediately sent to the global planner. The global planner, then, will calculate a safe path for the robot to use to arrive to the specified goal. The global planner uses the global costmap data in order to calculate this path.

There are different types of global planners. Depending on your setup, you will use one or another.

The local planner

Once the global planner has calculated a path for the robot, this is sent to the local planner. The local planner, then, will execute this path, breaking it into smaller (local) parts. So, given a plan to follow and a map, the local planner will provide velocity commands in order to move the robot. The local planner operates over a local costmap.

There are different types of local planners. Depending on the kind of performance you require, you will use one or another.

Costmaps

Costmaps are, basically, maps that represent which points of the map are safe for the robot to be in, and which ones are not. There are 2 types of costmaps:

- global costmap
- local costmap

Basically, the difference between them is that the global costmap is built using the data from a previously built static map, while the local costmap is built from the robot's sensor readings.

Recovery Behaviors

The recovery behaviors provide methods for the robot in case it gets stuck. The Navigation Stack provides 2 different recovery behaviors:

- rotate recovery
- clear costmap

Configuration

Since there are lots of different nodes working together, the number of parameters available to configure the different nodes is also very high. I think it would be a great idea if we summarize the different parameter files that we will need to set for Path Planning. The parameter files you'll need are the following:

- **move_base_params.yaml**
- **global_planner_params.yaml**
- **local_planner_params.yaml**
- **common_costmap_params.yaml**
- **global_costmap_params.yaml**
- **local_costmap_params.yaml**

Besides the parameter files shown above, we will also need to have a launch file in order to launch the whole system and load the different parameters.

Overall

Summarizing, this is how the whole path planning method goes:

After getting the current position of the robot, we can send a goal position to the **move_base** node. This node will then send this goal position to a **global planner** which will plan a path from the current robot position to the goal position. This plan is in respect to the **global costmap**, which is feeding from the **map server**.

The **global planner** will then send this path to the **local planner**, which executes each segment of the global plan. The **local planner** gets the odometry and the laser data values and finds a collision-free local plan for the robot. The **local planner** is associated with the **local costmap**, which can monitor the obstacle(s) around the robot. The **local planner** generates the velocity commands and sends them to the base controller. The robot base controller will then convert these commands into real robot movement.

If the robot is stuck somewhere, the recovery behavior nodes, such as the **clear costmap recovery** or **rotate recovery**, will be called.

Now everything makes more sense, right?

Dynamic Reconfigure

Until now, we've seen how to change parameters by modifying them in the parameters files. But, guess what... this is not the only way that you can change parameters! You can also change dynamic parameters by using the `rqt_reconfigure` tool. Follow the next steps:

Exercise 5.14

a) Run the next command in order to open the `rqt_reconfigure` tool.

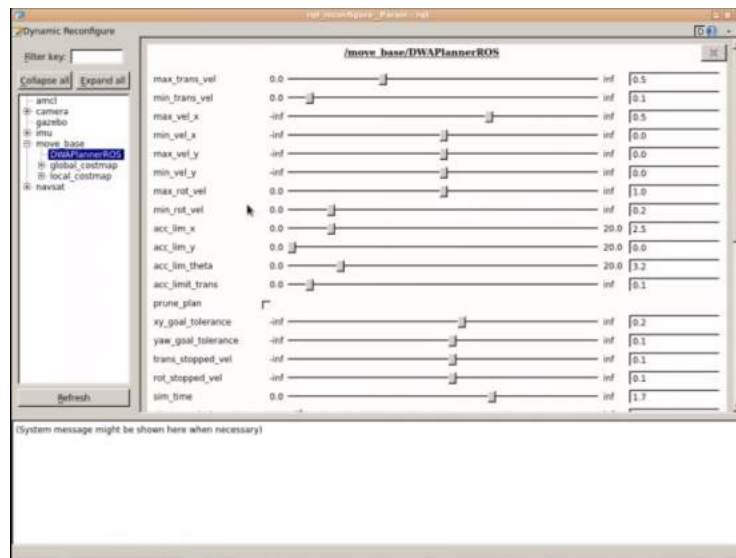
Execute in WebShell #2

```
[ ]: rosrn rqt_reconfigure rqt_reconfigure
```

- Open the `move_base` group.
- Select the `DWAPlanerROS` node.
- Play a little bit with the following 3 parameters:
 - **path_distance_bias**
 - **goal_distance_bias**
 - **occdist_scale**
- The above parameters are the ones involved in calculating the cost function, which is used to score each trajectory. More in detail, they define the following:

- **path_distance_bias**: The weighting for how much the controller should stay close to the path it was given.
 - **goal_distance_bias**: The weighting for how much the controller should attempt to reach its local goal, also controls speed.
 - **occdist_scale**: The weighting for how much the controller should attempt to avoid obstacles.
- Open Rviz and visualize how the global and local plans change depending on the values set.

Expected Result for Exercise 5.14



RQT Reconfigure

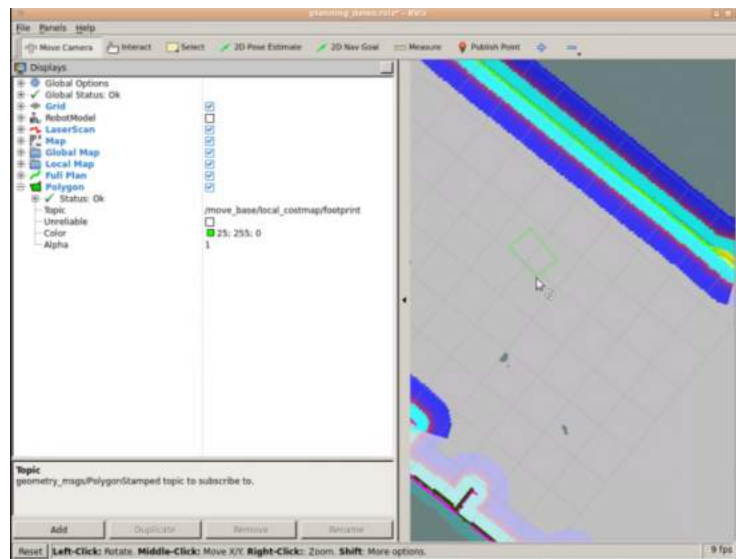
Other Useful Visualizations in Rviz

Until now, we've seen some ways of visualizing different parts of the `move_base` node process through Rviz. But, there are a couple more that may be interesting to know:

Robot Footprint

It shows the footprint of the robot.

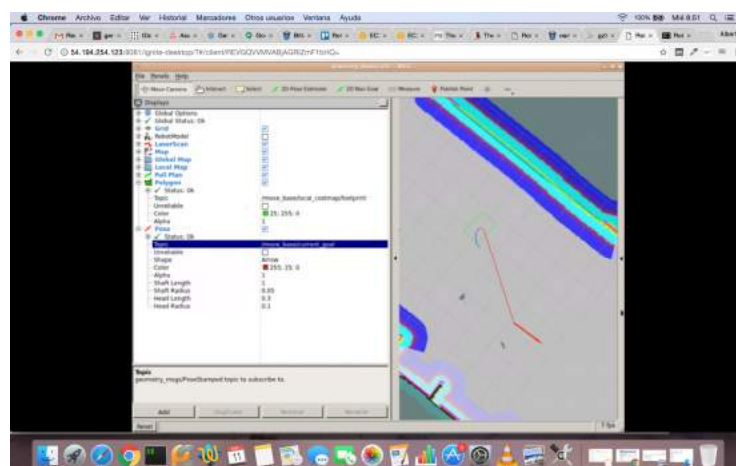
5 - Path Planning Part 2 (Obstacle Avoidance)



Robot footprint seen in RVIZ as a green square

Current Goal

To show the goal pose that the navigation stack is attempting to achieve, add a Pose Display and set its topic to `/move_base_simple/goal`. You will now be able to see the goal pose as a red arrow. It can be used to learn the final position of the robot.



Current goal and path to it

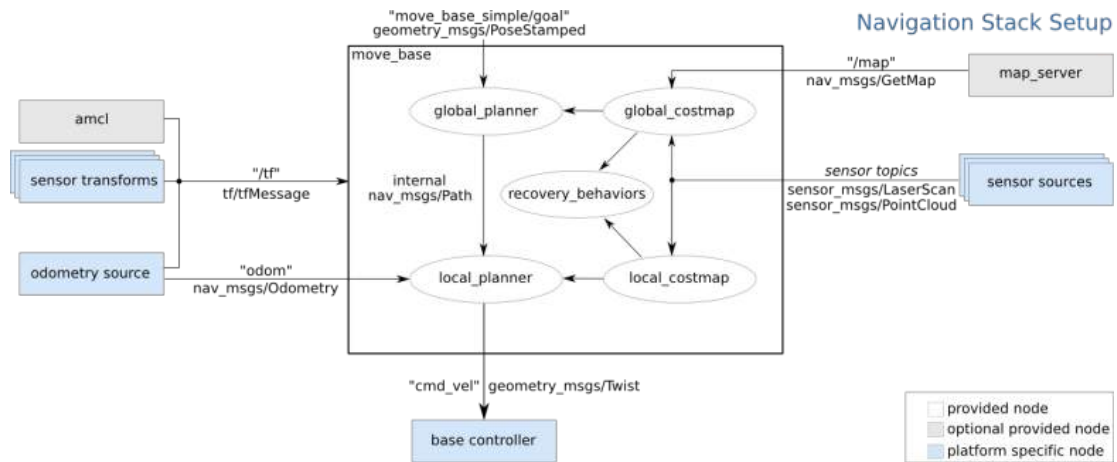
Exercise 5.15

Open Rviz and try to visualize the elements described above.

Expected Result for Exercise 5.15

Images similar to those shown above in RViz.

CONCLUSIONS



Move base internal structure graph. Image from <http://wiki.ros.org>

Summarizing, the ROS Path Planning system is basically managed by the **move_base** node.

##

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 Path Planning Part 2: [Path Planning Part 2 Solutions](#)

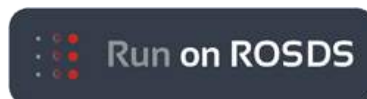
Unit 6. Project

ROS Navigation

Course Project



Navigation Project Environment



- ROSJect Link: <http://bit.ly/2n5zEOW>
- Package Name: **summit_xl_gazebo**
- Launch File: **main.launch**

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
-

Navigate the Summit Robot!

In this project, you will have to make a Summit robot 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 official 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, 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.

- Step 1: Generate a Map of the environment (Dedicate 2 hours)
- Step 2: Make the Robot Localize itself in the Map that you've created (Dedicate 2 hours)
- Step 3: Set Up a Path Planning System (Dedicate 3 hours)
- Step 4: Create a ROS program that interacts with the Navigation Stack (Dedicate 3 hours)

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

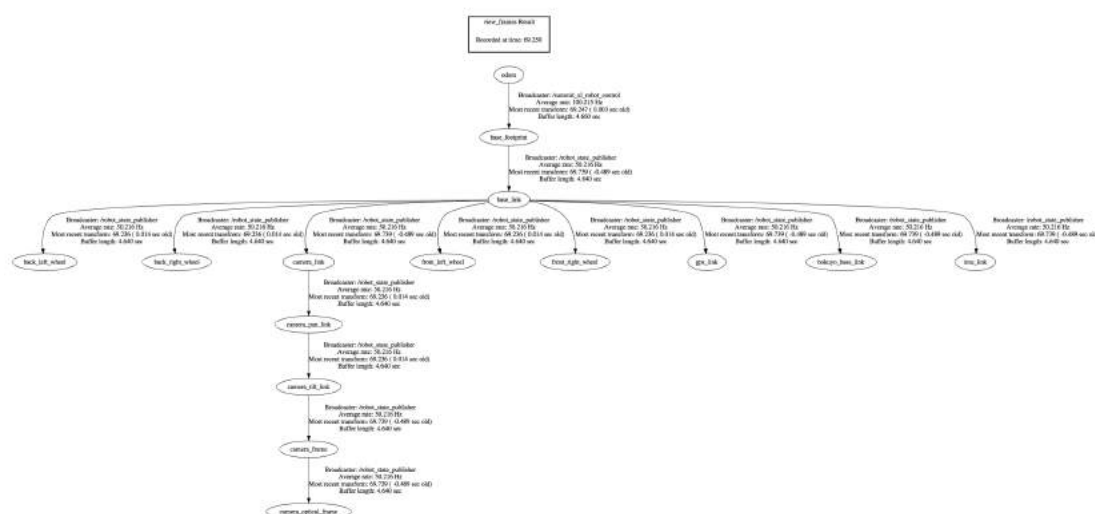
As you've learned in the first 2 chapters of the course, the first thing you need in order to navigate autonomously with a robot is a map of the environment. Where else would this project start? Therefore, the first step would not be anything other than creating a map of the environment that we want 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:



Summit Frame Tree

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 (Chapter 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/>

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

```
[ ]: <launch>
  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping"
output="screen">
  <!-- 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_udate_interval" value="5.0"/ -->
  <param name="base_frame" value="base_footprint"/>
  <param name="odom_frame" value="odom"/>
  <param name="map_udate_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>

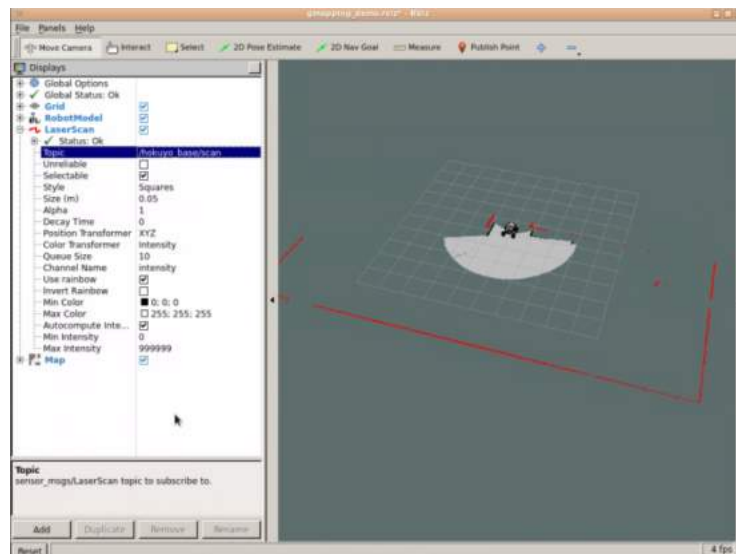
```

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:

```
[ ]: 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:



Difference between the real laser length and the length that is actually taken into account to build the map

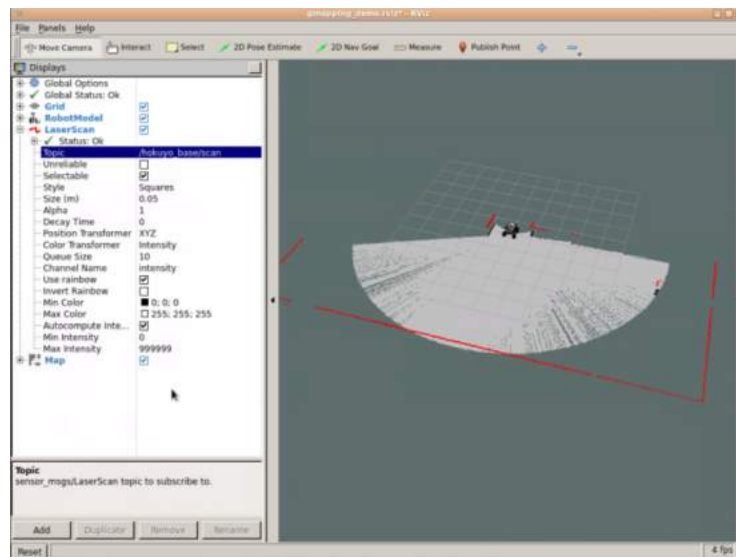
Now, start moving the robot around the room to create the map.

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 environment, 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:

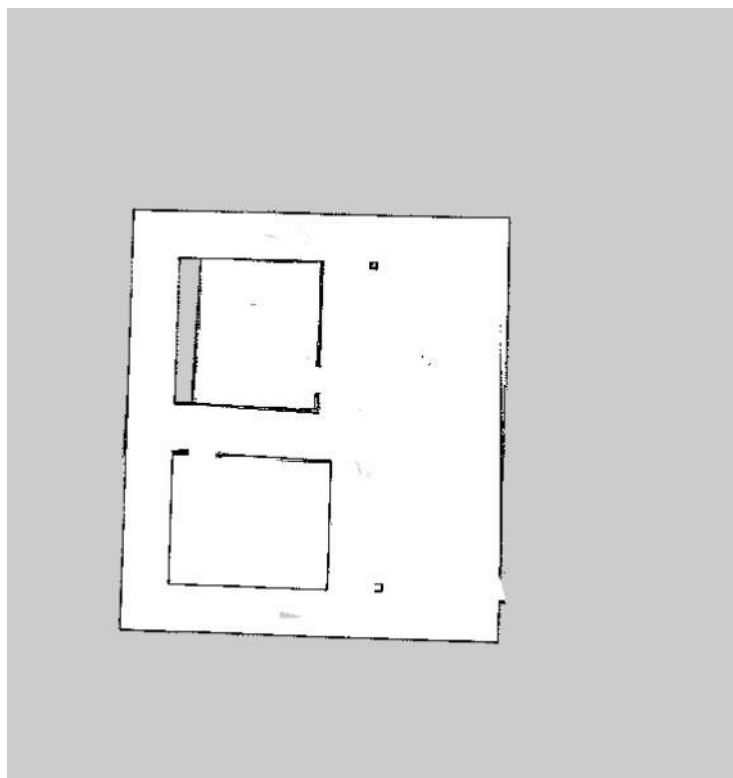


Effect of changing the urange parameter value

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:



Map of the environment

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

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.

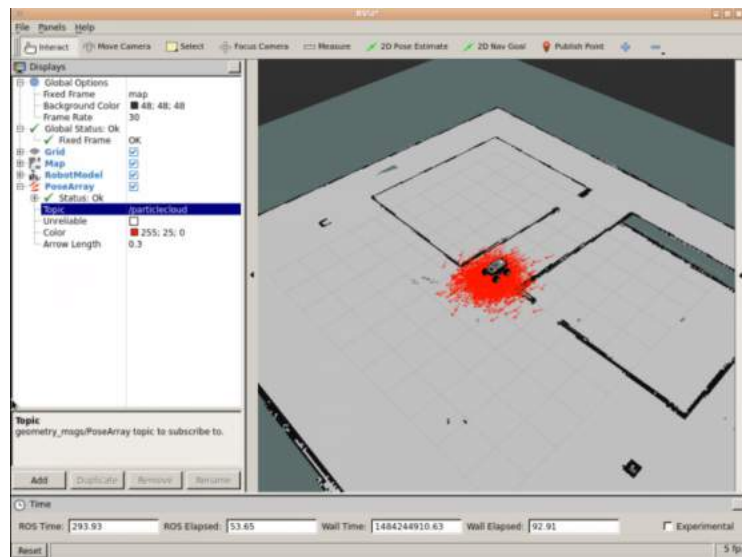
Here you can see a full list of parameters that you can set to configure the amcl node:
<http://wiki.ros.org/amcl>

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 itself in the map.

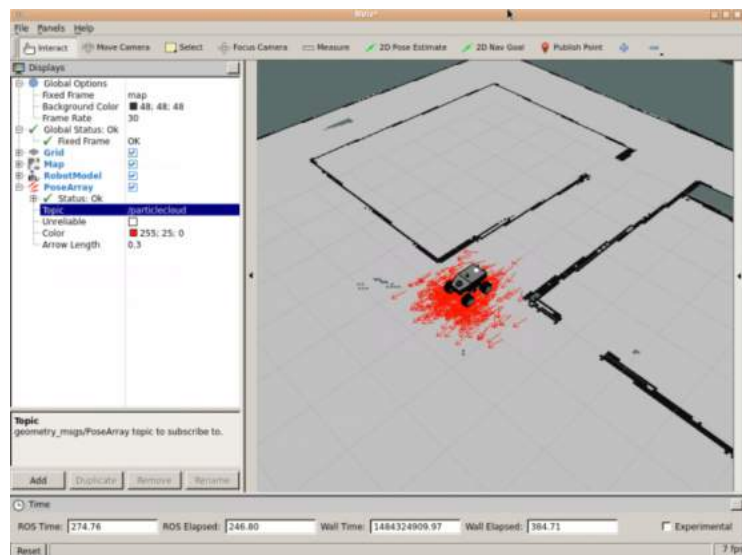
If you've done all of the previous steps correctly, you should see something like this in Rviz:



The robot initial localization

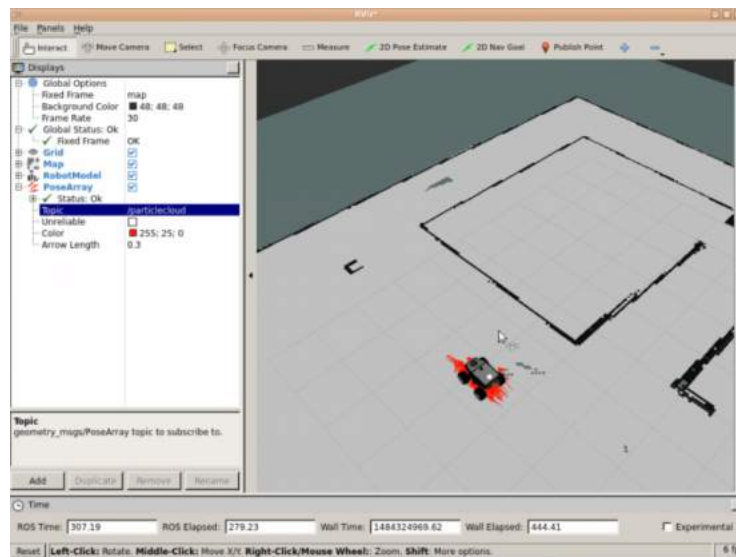
In order to check that the localization works fine, move the robot around the environment and check that the particle cloud keeps getting smaller the more you move the robot.

Dispersed particles:



Particles should get closer as the robot moves

Closer particles:



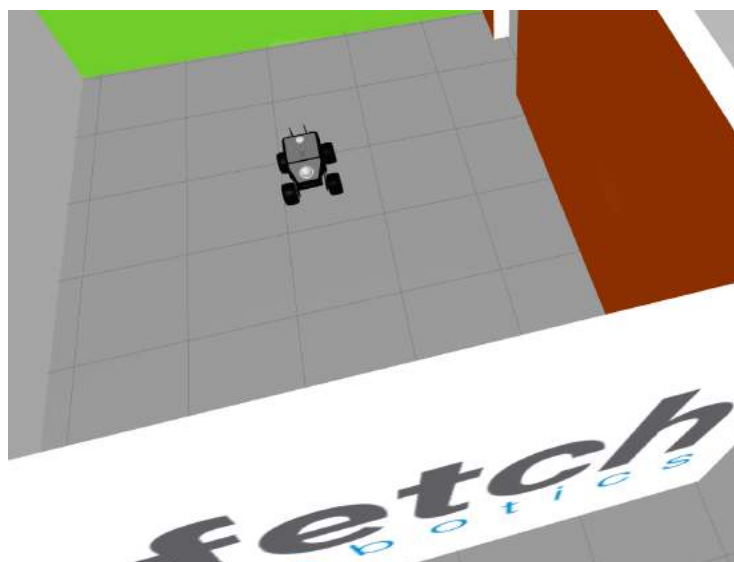
Most of the particles are together when the localization is correct

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 environment. For each point, assign a tag (with the name of the spot) alongside its coordinates in the map.

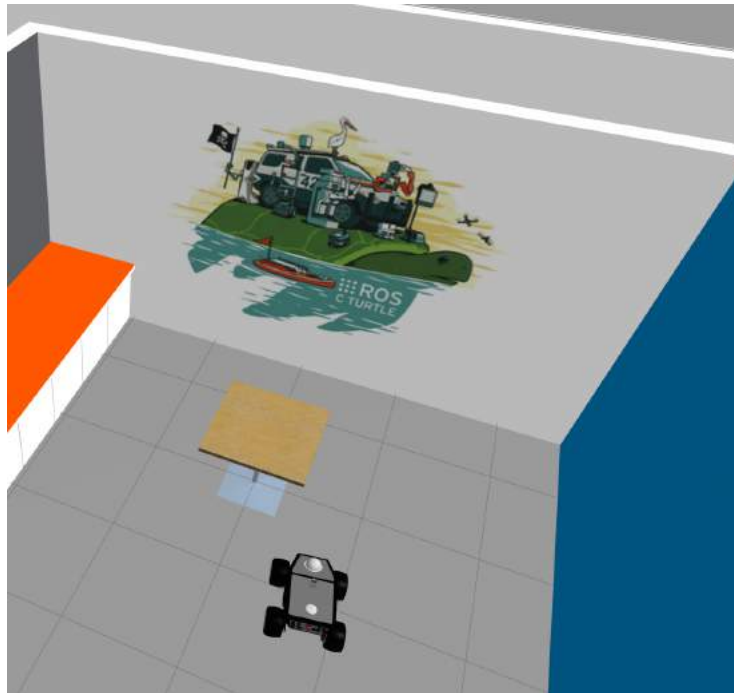
These are the 3 spots that you'll have to register into the table:

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



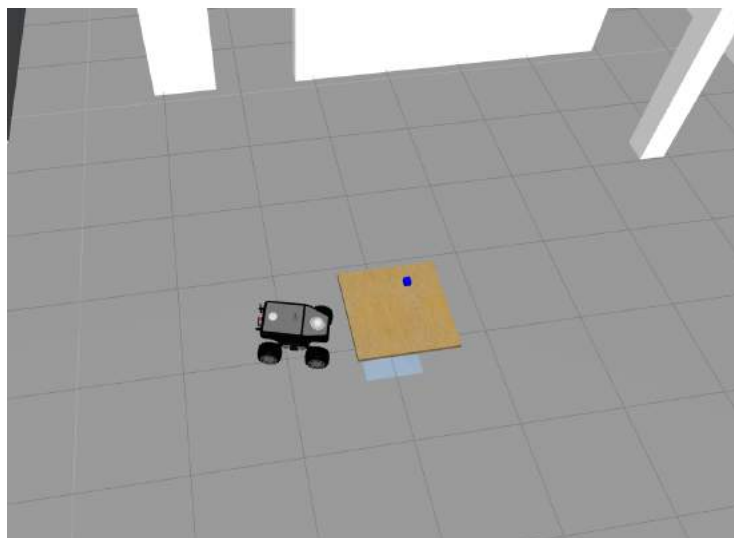
Robot inside a room

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



Robot facing the poster

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



Robot facing the 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.

If this is the case, then do the following. Inside the package 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:

```
[ ]: # 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()`

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

[ ]: add_service_files(
    FILES
    MyCustomServiceMessage.srv
)

[ ]: generate_messages(
    std_msgs
)

[ ]: catkin_package(
    rospy
)
```

In package.xml: Add these 2 lines to the file:

```
[ ]: <build_depend>message_generation</build_depend>
    <build_export_depend>message_runtime</build_export_depend>
    <exec_depend>message_runtime</exec_depend>
```

When you've finished 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:

```
Scanning dependencies of target std_msgs_generate_messages_cpp
Scanning dependencies of target _my_summit_navigation_generate_messages_check_deps_MyServiceMessage
[ 0%] Built target std_msgs_generate_messages_cpp
Scanning dependencies of target std_msgs_generate_messages_py
[ 0%] Built target std_msgs_generate_messages_py
Scanning dependencies of target std_msgs_generate_messages_lisp
[ 0%] Built target std_msgs_generate_messages_lisp
[ 0%] Built target _my_summit_navigation_generate_messages_check_deps_MyServiceMessage
Scanning dependencies of target my_summit_navigation_generate_messages_cpp
Scanning dependencies of target my_summit_navigation_generate_messages_py
[ 50%] [ 50%] Generating Python code from SRV my_summit_navigation/MyServiceMessage
Generating C++ code from my_summit_navigation/MyServiceMessage.srv
[ 75%] Generating Python srv __init__.py for my_summit_navigation
[ 75%] Built target my_summit_navigation_generate_messages_py
Scanning dependencies of target my_summit_navigation_generate_messages_lisp
[100%] Generating lisp code from my_summit_navigation/MyServiceMessage.srv
[100%] Built target my_summit_navigation_generate_messages_lisp
[100%] Built target my_summit_navigation_generate_messages_cpp
Scanning dependencies of target my_summit_navigation_generate_messages
[100%] Built target my_summit_navigation_generate_messages
```

Output while compiling

rossrv show output:

```
ubuntu@ip-172-31-44-229:/home/user/catkin_ws$ rossrv show MyServiceMessage
[my_summit_navigation/MyServiceMessage]:
string label
---
bool navigation_successfull
string message
```

Showing the structure of a service message

2 Service Code 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:

```
[ ]: 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:

```
[ ]: <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" output="screen">
    <rosparam file="$(find
my_summit_path_planning)/config/move_base_params.yaml" command="load"
```

```

/>
  <rosparam file="$(find
my_summit_path_planning)/config/costmap_common_params.yaml"
command="load" ns="global_costmap" />
  <rosparam file="$(find
my_summit_path_planning)/config/costmap_common_params.yaml"
command="load" ns="local_costmap" />
  <rosparam file="$(find
my_summit_path_planning)/config/local_costmap_params.yaml"
command="load" />
  <rosparam file="$(find
my_summit_path_planning)/config/global_costmap_params_map.yaml"
command="load" />
  <rosparam file="$(find
my_summit_path_planning)/config/dwa_local_planner_params.yaml"
command="load" />
  <rosparam file="$(find
my_summit_path_planning)/config/global_planner_params.yaml"
command="load" />

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

</launch>

```

3. Create the move_base_params.yaml file

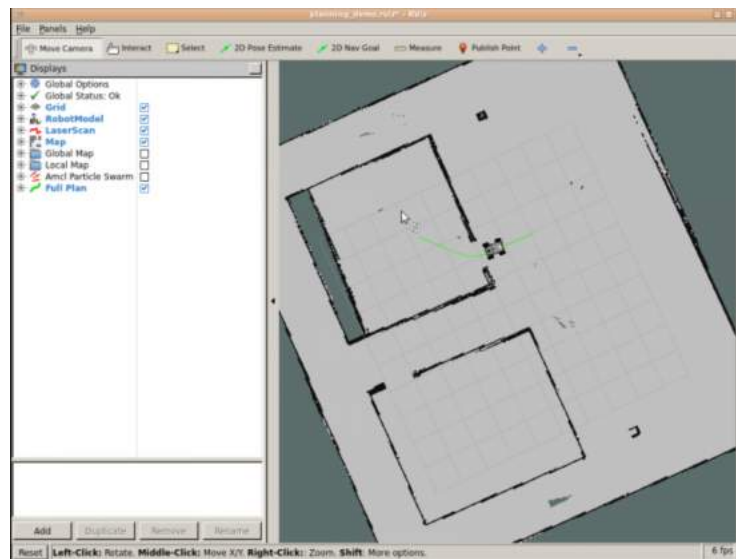
4. Create the costmap_common_params.yaml, the global_costmap_params.yaml, and the local_costmap_params.yaml files

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

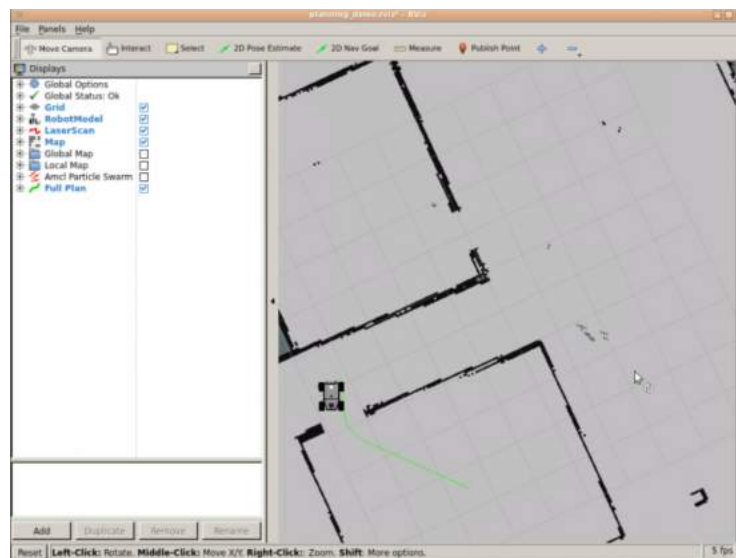
Once you think your parameter files are correct, launch the node and test that the path planning works fine. Check that the robot can plan trajectories and navigate to different points in the environment.

Room 1:



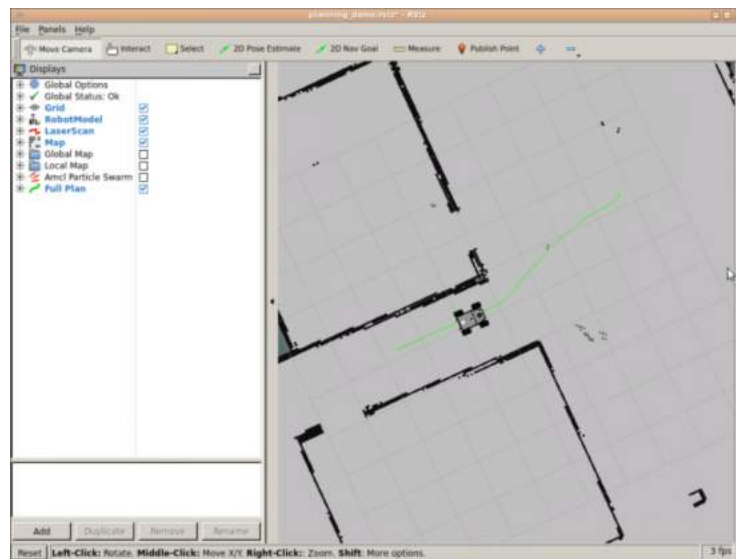
Example of the Summit planning in the map

Room 2:



Another example of planning

Main Hall:



Another example of planning

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:

```
[ ]: # 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()

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

[ ]: add_service_files(
    FILES
    MyCustomServiceMessage.srv
)

[ ]: generate_messages(
    std_msgs
)
```

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

In package.xml: Add these 2 lines to the file:

```
[ ]: <build_depend>message_generation</build_depend>
    <build_export_depend>message_runtime</build_export_depend>
    <exec_depend>message_runtime</exec_depend>
```

When you've finished 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 writing 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:

```
[ ]: 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](#)

Final Recommendations

I'm finished, now what?

ROS Development Studio (ROSDS)



ROSDS logo

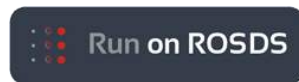
ROSDS is the The Construct web based tool to program ROS robots online. It requires no installation in your computer. Hence, you can use any type of computer to work on it (Windows, Linux or Mac). Additionally, free accounts are available. **Create a free ROSDS account here:** <http://rosds.online>

You can use any of the many ROSjects available in order to apply all the things you've learned during the Course. You just need to paste the ROSject link to your browser's URL, and you will automatically have the simulation prepared in your ROSDS workspace.

Down below you can check some examples of the Public Rosjects we provide:

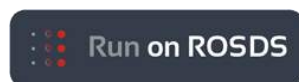
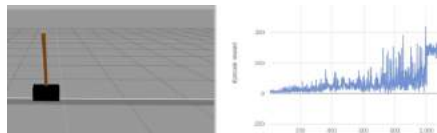
ARIAC Competition





- ROSject Link: <https://bit.ly/2t2px0t>

Cartpole Reinforcement Learning



- ROSject Link: <https://bit.ly/2t2uGWr>

ARIAC Competition



- ROSject Link: <https://bit.ly/2Tt4lw8>

Want to learn more?



Robot Ignite Logo

Once you have finished the course, you can still learn a lot of interesting ROS subjects.

- Take more advanced courses that we offer at the Robot Ignite Academy, like Perception or Navigation. Access the Academy here: <http://www.robotigniteacademy.com>
- Or, you can go to the ROS Wiki and check the official documentation of ROS, now with new eyes.

Thank You and hope to see you soon!

