ROS Navigation: Concepts and Tutorial

Chapter *in* Studies in Computational Intelligence · February 2016

DOI: 10.1007/978-3-319-26054-9_6

CITATIONS
8

READS
5,701

5 authors, including:

João A Fabro
Federal University of Technology - Paraná/Brazil (UTFPR)
47 PUBLICATIONS **92** CITATIONS
SEE PROFILE

Rodrigo Longhi Guimarães
Federal University of Technology - Paraná/Brazil (UTFPR)
2 PUBLICATIONS **8** CITATIONS
SEE PROFILE

Thiago Becker
4 PUBLICATIONS **10** CITATIONS
SEE PROFILE

ROS Navigation: concepts and tutorial

Rodrigo Longhi Guimarães, André Schneider de Oliveira, João Fabro, Thiago Becker, and Vinícius Amilgar Brenner

Federal University of Technology - Parana
Av. Sete de Setembro, 3165, Curitiba, Brazil
rguimaraes@alunos.utfpr.edu.br,andreoliveira@utfpr.edu.br,
fabro@dainf.ct.utfpr.edu.br,beckerthiago@gmail.com,
brenner@alunos.utfpr.edu.br

Abstract. This tutorial chapter aims to teach the main theoretical concepts and explain the use of ROS Navigation Stack. This is a powerful toolbox to path planning and Simultaneous localization and mapping (SLAM) but its application is not trivial due to lack of comprehension of the related concepts. This chapter will present the theory inside this stack and explain in an easy way how to perform SLAM in any robot. Step by step guides, example codes explained (line by line) and also both simulated and real robot testing will be available. We will present the requisites and the how-to's that will make the readers able to set the odometry, stablish reference frames and its transformations, configure perception sensors, tune the navigation controllers and plan the path on their own virtual or real robots.

Keywords: ROS, navigation, tutorial, real robots, virtual robots

1 Introduction

Simultaneous localization and mapping (SLAM) is an important approach that allows the robot to acknowledge the obstacles around it and plan a path to avoid these restrictions. This method is a merge of several approaches that allow robots to navigate on unknown or partially known environments. ROS has a package that performs SLAM, named Navigation Stack, however, some details of its application are hidden, considering that the programmer has some expertise. The unclear explanation and the many subjective aspects within the package can lead the user to fail using the technique or, at least, consume extra effort.

This chapter aims to present the theory inside ROS Navigation Stack and explain in an easy way how to perform SLAM in any robot. It will also explain how to use a virtual environment to apply the SLAM on virtual robots. These robots are designed to both publish and subscribe the same information in real and virtual environments, where all sensors and actuators of real world are functional in virtual environment.

We begin the chapter explaining what the navigation stack is with some simple and straight-forward definitions and examples where the functionalities of

the package are being explained in conjunction with reminders of some basic concepts of ROS. Moreover, a global view of a system running the navigation stack at its full potential is shown, where the core subset of this system is explained bit by bit. In addition, most of the not directly related components of the system, like multiple machine communication, will be indicated and briefly explained, with some tips about its implementation.

The chapter will be structured in four main topics. Firstly, an introduction of mobile robot navigation and a discussion about simultaneous localization and mapping will be presented, along with some discussion about transformations.

In the second section, environment, the main purpose is to let the reader know everything he needs to configure his own virtual or real robot. Firstly, the limitations of the navigation stack are listed together with the expected hardware and software that the reader should have to follow the tutorial. Secondly, an explanation about odometry and kinematics is given, focusing on topics like precision of the odometry and the impact of the lack of it in the construction of costmaps for navigation. Perception sensors are also discussed. The differences and advantages of each of the main kinds of perception sensors, like depth sensors, light detection sensors and ranging (LIDAR) sensors are also emphasized. In addition, reference frames and its transformations are discussed, showing how to achieve the correct merge of the sensors information. Still on the second section, a dense and relevant subsection about the navigation stack is presented. It is in respect to the configuration of the navigation stack to work with as many robots as possible, trying to organize the tutorial in a way that both reach high level of detail and generalization, granting that the reader is apt to perceive a way to make his own robot perform SLAM. An in-depth discussion of map generation and map's occupancy is performed. To do that, the tutorial is structured in a step by step fashion in which all the navigation configuration files will be analysed, examining the parameters one by one and setting them to example values that correspond to the robots we're using in the chapter (the Pioneer 3-AT and LX and their simulated models, Fig.1). The explanation of the whole process in addition to the demonstration of the effects of the parameter changes is more than enough to clear up any doubts the reader might have.

In the third section we will discuss the experiments in real and virtual environments to prove the accuracy of the SLAM method. All steps to use a virtual environment where the reader should be able to test his own configuration for the navigation stack are demonstrated. Lastly, some trials are run in virtual and real robots, to illustrate some more capabilities of the package. In this section we will also take a glance over rviz and vrep usage.

Lastly, a brief biography of the authors will be presented, showing why this team is able to write the tutorial chapter here presented.

2 Background

ROS has a set of resources that are useful so a robot is able to navigate through a medium, in other words, the robot is capable of planning and following a path

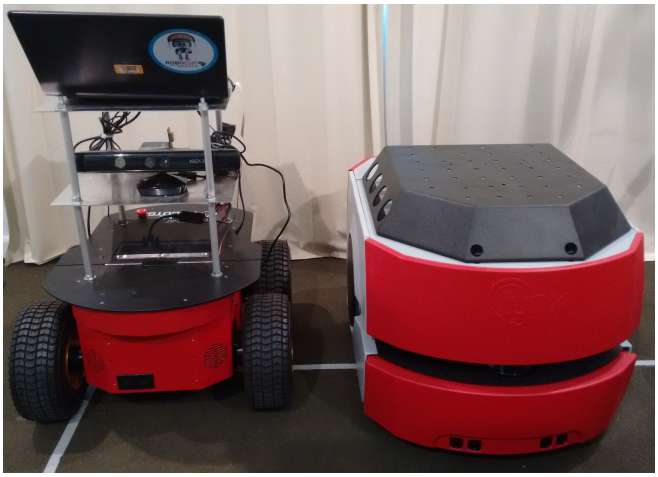

Fig. 1. Side by side are the two robots used to test the navigation stack: The Pioneer 3-AT, widely used in many research centres all over the world, is a great value choice, whereas the Pioneer LX is a high-end option.

while it deviates from obstacles that appear on its path throughout the course. These resources are found on the navigation stack.

One of the many resources needed for completing this task and that is present on the navigation stack are the localization systems, that allow a robot to locate itself, whether there is a static map available or simultaneous localization and mapping is required. AMCL is a tool that allows the robot to locate itself in an environment through a static map, a previously created map. The disadvantage of this resource is that, because of using a static map, the environment that surrounds the robot can not suffer any modification, because a new map would have to be generated for each modification and this task would consume computational time and effort. Being able to navigate only in modification-free environments is not enough, since the robots should be able to operate in places like industries and schools, where there is constant movement. To bypass the lack of flexibility of static maps, two other localization systems are offered by the navigation stack: gmapping and hector_mapping.

Both gmapping and hector_mapping are based on SLAM, a technique that consists on mapping an environment at the same time that the robot is moving, in other words, while the robot navigates through an environment, it gathers information from the environment through his sensors and generates a map. This way you have a mobile base able not only to generate a map of an unknown environment as well as updating the existent map, thus enabling the use of the device in more generic environments, not immune to changes.

The difference between gmapping and hector_mapping is that the first one takes in account the odometry information to generate and update the map and the robot's pose, however, the robot needs to have encoders, preventing some

robots(e.g. flying robots) of using it. The odometry information is interesting because they are able to aid on the generation of more precise maps, since understanding the robot dynamics we can estimate its pose.

The dynamic behaviour of the robot is also known as kinematics. Kinematics is influenced, basically, by the way that the devices that guarantee the robot's movement are assembled. Some examples of mechanic features that influence the kinematics are: the wheel type, the number of wheels, the wheels positioning and the angle at which they are disposed.

However, as much useful as the odometry information can be, it isn't immune to faults. The faults are caused by the lack of precision on the capitation, friction, slip, drift and other factors, and, with time, they may accumulate, making inconsistent data and prejudicing the maps formation, that tend to be distorted under these circumstances.

Other indispensable data to generate a map are the sensors‘ distance readings, for the reason that they are responsible in detecting the external world and, this way, serve as reference to the robot. Nonetheless, the data gathered by the sensors must be adjusted before being used by the device. These adjustments are needed because the sensors measure the environment in relation to themselves, not in relation to the robot, in other words, a geometric conversion is needed. To make this conversion simpler, ROS offers the TF tool, which makes it possible to adjust the sensors positions in relation to the robot and, this way, adequate the measures to the robot‘s navigation.

3 ROS Environment

Before we begin setting up the environment in what we will work, it's very important to be aware of the limitations of the navigation stack, so we're able to adapt our hardware. There are four main limitations in respect to the hardware:

- The stack was built aiming to address only differential drive and holonomic robots, although it is possible to use some features with another types of robots, which won't be covered here.
- The navigation stack assumes that the robot receives a twist type message [1] with X,Y and Theta velocities and is able to control the mobile base to achieve these velocities. If your robot is not able to do so, you can adapt your hardware or just create a ros node that converts the twist message provided by the navigation stack to the type of message that best suits your needs.
- The environment information is gathered from a LaserScan message type topic. If you have a planar laser, such as a Hokuyo URG or a SICK Laser, it should be very easy to get them to publish their data, all you need to do is install the hokuyo_node, sicktoolbox or similar packages, depending on your sensor. Moreover, it is possible to use other sensors, as long as you can convert their data to the LaserScan type. In this chapter, we will use converted data from Microsoft Kinect's depth sensor.
- The navigation stack will perform better with square or circular robots, whereas it is possible to use it with arbitrary shapes and sizes. Unique sizes

and shapes may cause the robot to have some issues in restricted spaces. In this chapter we will be using a custom footprint that is an approximation of the robot.

If your system complies with all the requirements, it is time to move for the software requirements, which are very few and easy to get.

- You should have ROS Hydro or Indigo to get all the features, such as layered costmaps, that are a Hydro+ feature. From here, we assume you are using a Ubuntu 12.04 with ROS Hydro.
- You should have the navigation stack installed. In the full desktop version of ROS it is bundled, but depending on your installation it might be not included. Don't worry with that for now, it's just good to know that if some node is not launching it may be because the package you're trying to use is not installed.
- As stated on the hardware requirements, you might need some software to get your system navigation ready:
  - If your robot is not able to receive a twist message and control its velocity as demanded, one possible solution is to use a custom ROS node to transform the data to a more suitable mode.
  - You need to have drivers able to read the sensor data and publish it in a ROS topic. If you're using a sensor different from a planar laser sensor, such as a depth sensor, you'll most likely also need to change the data type to LaserScan through a ROS node.

Now that you now all you need for navigation, it is time to begin getting those things. In this tutorial we'll be using Pioneer 3-AT and Pioneer LX and each of them will have some particularities in the configuration that will help us to generalize the settings as much as possible. We'll be using Microsoft's Kinect depth sensor in the Pioneer 3-AT and the Sick S300 laser rangefinder that comes built-in in the Pioneer LX.

3.1 Configuring the Kinect Sensor

The Kinect is a multiple sensor equipment, equipped with a depth sensor, an rgb camera and microphones. First of all, we need to get those features to work by installing the drivers, which can be found in the ros package openni_camera[2]. You can get openni_camera source code from its git repository, available on the wiki page, or install the stack with a linux terminal command using apt-get(RECOMMENDED). To do so, open your terminal window and type:

```
1 $ sudo apt-get install ros-<rosdistro>-openni-camera
```

Remember to change "<rosdistro>" to your ros version(i.e. 'hydro' or 'indigo'). We'll also need a ROS publisher to use the sensor data, and that publisher is found in the openni_launch[3] package. Again, you can get the source

code from their git repository at the package wiki page or you can get the built package by opening the Ubuntu terminal and typing the following command(RECOMMENDED):

```
1 $ sudo apt-get install ros-<rosdistro>-openni-launch
```

In the same way, not forgetting to change "<rosdistro>" to your ros version. With all the software installed, it's time to power on the hardware.

Depending on your application plugging Kinect's USB Cable to the computer you're using and the AC Adapter to the wall socket can be enough, however needing a wall socket to use your robot is not a great idea. The AC Adapter converts the AC voltage (i.e. 100V@60Hz) to the DC voltage needed to power up the kinect (12V), therefore the solution is exchanging the AC adapter for a 12V battery. The procedure for doing this is explained briefly in the following topics:

- Cut the AC adapter off, preferably near the end of the cable.
- Strip a small end to each of the two wires(white and brown) inside of the cable.
- Connect the brown wire to the positive(+) side of the 12V battery and the white wire to the negative(-). You can do this connection by soldering [4] or using connectors, such as crimp[5] or clamp[6] wire connectors.

Be aware that the USB connection is enough to blink the green LED in front of the Kinect and it doesn't indicate that the external 12V voltage is there. You can also learn a little more about this procedure by reading the "Adding a Kinect to an iRobot Create/Roomba" wiki page[7].

Now that we have the software and the hardware prepared, some testing is required. With the kinect powered on, execute the following command on the Ubuntu terminal:

```
1 $ roslaunch openni_launch openni.launch
```

The software will come up and a lot of processes will be started along with the creation of some topics. If your hardware is not found for some reason, you may see the message "No devices connected.... waiting for devices to be connected". If this happens, please verify your hardware connections(USB and power). If that does not solve it, you may try to remove some modules from the Linux Kernel that may be the cause of the problems and try again. The commands for removing the modules are:

Once the message of no devices connected disappears, you can check some of the data supplied by the Kinect in another Terminal window(you may open

```
1 $ sudo modprobe -r gspca_kinect $ sudo modprobe -r gspca_main
```

multiple tabs of the terminal by pressing CTRL+SHIFT+T) by using one or more of these commands:

```
1 $ rosrun image_view disparity_view image:=/camera/depth/disparity
2 $ rosrun image_view image_view image:=/camera/rgb/image_color
3 $ rosrun image_view image_view image:=/camera/rgb/image_mono
```

The first one will show a disparity image, while the second and third commands will show the RGB camera image in color and in black and white respectively.

At last, we need to convert the depth image data to a LaserScan message. Fortunately, we have yet another package to do this for us, the depthimage_to_laserscan package. As we've done with the other packages, we can check out the wiki page[8] and get the source code from the git repository or we can simply get the package with apt-get:

```
1 $ sudo apt-get install ros-<rosversion>-depthimage-to-laserscan
```

With this last installation, you should have all the software you need for Kinect utilization with navigation purposes, although there's a lot of other software you can use with it. We'd like to point out two of these packages that can be very valuable at your own projects:

- **kinect_aux[9]:** this package allows to use some more features of the kinect, such as the accelerometer, tilt, and LED. It can be used along with the openni_camera package and it's also installed with a simple apt-get command.
- **Natural Interaction - openni_tracker[10]**: One of the most valuable packages for using with the kinect, this package is able to do skeleton tracking functionalities and opens a huge number of possibilities. It's kind of tough to install and the installation can lead to problems sometimes, so we really recommend you to do a full system backup before trying to get it to work. First of all, install the openni_tracker package with an apt-get, as stated on the wiki page. After that, you have to get these recommended versions of the files(the official openni website is no longer in the air, so it can be a tough job):
  - NITE-Bin-Linux-x86-v1.5.2.23.tar.zip
  - OpenNI-Bin-Dev-Linux-x86-v1.5.7.10.tar.bz2
  - SensorKinect093-Bin-Linux-x86-v5.1.2.1.tar.bz2

The first two files (NITE and Openni) can be installed following the cyphy_people_mapping[11] tutorial and the last file should be installed by:

- Unpacking the file

```
1 $ tar -jxvf SensorKinect093-Bin-Linux-x86-v5.1.2.1.tar.bz2
```

- Changing the permission of the setup script to allow executing.

```
1 $ sudo chmod a+x install.sh
```

- Installing.

```
1 $ sudo ./install.sh
```

Now that we have all software installed we should pack it all together in a single launch file, to make things more independent and don't need to start a lot of packages manually when using the kinect. Here's an example of a complete launch file for starting the kinect and all the packages that make its data navigation-ready:

```
<launch>
        <include file="$(find openni_launch)/launch/openni.launch"
             />
        <node respawn="true" pkg="depthimage_to_laserscan" type="
            depthimage_to_laserscan" name="laserscan">
               <remap from="image" to="/camera/depth/image" />
        </node>
</launch>
```

As you can see and you might be already used to because of the previous chapters, the ROS launch file is running one node and importing another launch file: openni.launch, imported from the openni_launch package. Analysing the code in a little more depth:

– The first and sixth lines are the launch tag, that delimits the content of the launch file;
– The second line includes the openni.launch file from the openni_launch package, responsible for loading the drivers of the kinect, getting the data, and publishing it to ROS topics;

- The third line starts the package depthimage_to_laserscan with the "laserscan" name. It also sets the respawn parameter to true, in case of failures. This package is responsible for getting a depth image from a ROS topic, converting it to a LaserScan message and publishing it to another topic;
- The fourth line is a parameter for the depthimage_to_laserscan. By default, the package gets the depth image from the /image topic, but the openni_launch publishes it in the /camera/depth/image topic, and that is what we are saying to the package.

There's still the transform(tf) missing, but we will discuss that later, because the configuration is very similar to all sensors.

3.2 SICK S300 Laser Sensor

The pioneer LX comes bundled with a SICK S300 laser sensor and we'll describe here how to get its data, since the process should be very similar to other laser rangefinder sensors. The package that supports this laser model is sicks300, that is currently only supported by the fuerte and groovy versions of ROS. We are using a fuerte installation of ROS in this robot, so it's no problem for us, but it must be adapted if you wish to use it with hydro or indigo. For our luck, it was adapted and it is available at STRANDS git repository[12]. The procedure for getting it to work is:

- Cloning the repository, by using the command git clone https://github.com /bohlender/sicks300.git (the URL will change depending on your version);
- Compile the files. For rosbuild versions, use rosmake sicks300 sicks300 and for catkinized versions, use catkin_make.
- Run the files by using the command rosrun sicks300 sicks300_driver. It should work with the default parameters, but, if it doesn't, check if you have configured the baudrate parameter correctly(the default is 500000 and for the pioneer LX, for example, it is 230400).

The procedure should be very similar for other laser sensors and the most used packages, sicktoolbox_wrapper, hokyuo_node and urg_node, are very well documented on the ROS wiki. Its noteworthy that there is another option for reading pioneer LX laser sensor data: cob_sick_s300 package.

3.3 Transforms

As explained on the background section, the transforms are a necessity for the navigation stack to understand where the sensors are located in relation to the center of the robot(base_link). We will publish a transform between the Pioneer 3-AT base_link and the kinect sensor laser_link(the center of the kinect sensor) as example, given that this procedure is equal for any type of sensor. The first thing in order to use transforms is getting the distances you need. You'll have to measure three-dimensional coordinates in respect to the center of the robot

and will also need to get the angles between the robot pointing direction and the sensor pointing direction. Our kinect sensor is aligned with the robot pointing direction on the yaw and its z coordinate is also aligned with the robot center(and that's the usual case for most of the robots), therefore we have to measure only the x and y distances between the sensor and the robot centers. The x distance for our robot is 35cm(0.35m) and our height distance(y) is 10cm(0.1m). There are two standard ways to publish the angle values on transforms: using quaternion or yaw/pitch/roll. The first one, using quaternion, will respect the following order:

```
1 static_transform_publisher x y z qx qy qz qw frame_id child_frame_id
  period_in_ms
```

Where qx, qy, qz and qw are the versors in the quaternion representation of orientations and rotations. The second way of publishing angles, using yaw/roll/pitch and the one we will be using, is published in the following order:

```
1 static_transform_publisher x y z yaw pitch roll frame_id child_frame_id
  period_in_ms
```

The common parameters for both quaternion and yaw/row/pitch representations are:

- x, y and z are the offset representation, in meters, for the three-dimensional distance between the two objects;
- frame_id and child_frame_id are the unique names that will bound the transformations to the object to which they relate. In our case, frame_id is the base_link of the robot and child_frame_id is the laser_link of the sensor.
- period_in_ms is the time between two publications of the tf. It is possible to calculate the publishing frequency by calculating the reciprocal of the period.

In our example, for the Pioneer 3-AT and the Kinect, we have to include, in the XML launcher(just write this line down at a new launch file, we'll indicate further in the text where to use it), the following code to launch the tf node:

```
1 <node pkg="tf" type="static_transform_publisher"
  name="Pioneer3AT_laserscan_tf" args="0.1 0 0.35 0 pi/2 pi/2 base_link
  camera_link 100" />
```

If you have some doubts on how to verify the angles you measured, you can use rviz to check them, by including the laser_scan topic and verifying the

rotation of the obtained data. If you don't know how to do that yet, check the tests section of the chapter, which includes the rviz configuration.

3.4 Creating a package

At this point, you probably already have some XML files for launching nodes containing your sensors initialization and for initializing some packages related to your platform(robot powering up, odometry reading, etc...). To organize these files and make your launchers easy to find with ros commands, let's create a package containing all your launchers. To do that, go to your src folder, inside your catkin workspace, and create a package with the following commands(commands valid for catkinized versions of ROS):

```
1 $ cd ~/catkin_ws/src
2 $ catkin_create_pkg packageName std_msgs rospy roscpp move_base_msgs
```

These two commands are sufficient for creating a folder containing all the files that will make ros find the package by its name. Copy all your launch files to the new folder, namesake to your package, and then compile the package, by going to your workspace folder and issuing the compile command:

```
1 $ cd ~/catkin_ws
2 $ catkin_make
```

That's all you need to do. From now on, remember to put your launch and config files inside this folder. You may also create some folder, such as launch and config, provided that you specify these sub-paths when using find for including launchers in other launchers.

3.5 The Navigation Stack - System Overview

Finally, all the pre-requisites for using the navigation stack are met. Thus, it is time to begin studying the core concepts of the navigation stack, as well as their usage. Since the overview of the Navigation stack concepts was already done in the Background section, we can jump straight to the system overview, which is done in Figure 2. Let's analyse the items block by block in the following sections.

AMCL and map_server The first two blocks that we can focus on are the optional ones, responsible for the static map usage: amcl and map_server. map_server contains two nodes: map_server and map_saver. The first one, namesake to the package, as the name indicates, is a ROS node that provides static map data as

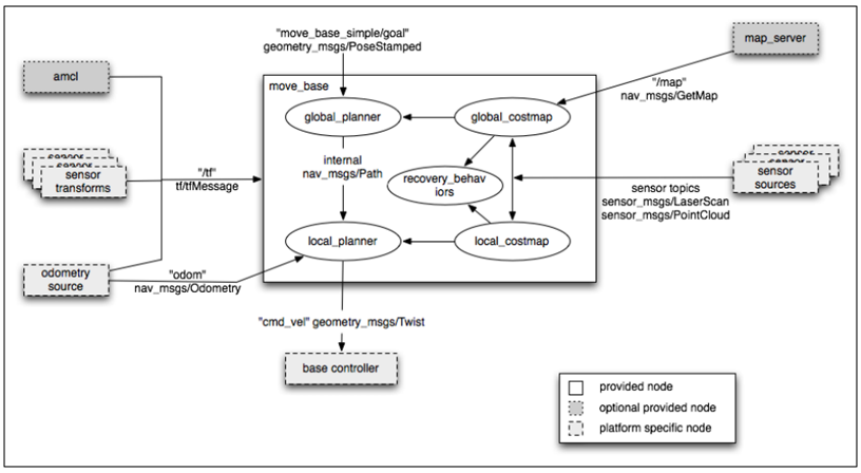

Fig. 2. Overview of a typical system running the navigation stack. Figure Source: ROS WIKI.

a ROS Service, while the second one, map_saver, saves a dynamically generated map to a file. AMCL does not manage the maps, it is actually a localization system that runs on a known map. This localization system is based on the Monte Carlo localization approach: it randomly distributes the particles in a known map, representing the possible robot locations, and then uses a particle filter to determine the actual robot pose.

gmapping Gmapping, as well as amcl, is a localization system, but unlike amcl, it runs on an unknown environment, performing Simultaneous Localization and Mapping(SLAM). It creates a 2D occupancy grid map using the robot pose and the laser data(or converted data, i.e. Kinect data).

Sensors and controller These blocks of the system overview are in respect to the hardware-software interaction and, as indicated, are platform specific nodes. The odometry source and the base controller blocks are specific to the robot you're using, since the first one is usually published using the wheel encoders data and the second one is the responsible for taking the velocity data from the cmd_vel topic and assuring that the robot reproduces these velocities.

Local and Global costmaps The local and global 2D costmaps are the topics containing the information that represents the projection of the obstacles in a 2D plane(the floor), as well as a security inflation radius, an area around the obstacles that guarantee that the robot will not collide with any objects, no matter what is its orientation. While the global costmap represents the whole environment(or a huge portion of it), the local costmap is, in general, a scrolling window that moves in the global costmap in relation to the robot current position.

Local and Global planners The local and global planners do not work the same way. The global planner takes the current robot position and the goal and traces the trajectory of lower cost in respect to the global costmap. However, the local planner has a more interesting task: it works over the local costmap, and, since the local costmap is smaller, it usually has more definition, and therefore is able to detect more obstacles than the global costmap. Thus, the local planner is responsible for creating a trajectory rollout over the global trajectory, that is able to return to the original trajectory with the fewer cost. Just to make it clear, move_base is a package that contains the local and global planners and is responsible for linking them to achieve the nav goal.

3.6 The Navigation Stack - Getting it to work

At last, it is time to write the code for the full launcher. It is easier to do this with an already written code, such as the one that follows:

```
<launch>
        <master auto="start"/>

        <!-- PLATFORM SPECIFIC -->
        <node pkg="p2os_driver" type="p2os_driver" name="
            p2os_driver" >
                <param name="port" value="/dev/ttyUSB0" />
                <param name="pulse" value="1.0" />"
        </node>

        <node pkg="rostopic" type="rostopic" name="enable_robot"
            args="pub /cmd_motor_state p2os_driver/MotorState 1"
            respawn="true">
        </node>

        <!-- TRANSFORMS -->
        <node pkg="tf" type="static_transform_publisher" name="
            Pioneer3AT_laserscan_tf" args="0.1 0 0.35 0 pi/2 pi/2
            base_link camera_link 100" />

        <!-- SENSORS CONFIGURATION -->
        <arg name="kinect_camera_name" default="camera" />
                <param name="/$(arg kinect_camera_name)/driver/
                    data_skip" value="10" />
                <param name="/$(arg kinect_camera_name)/driver/
                    image_mode" value="5" />
                <param name="/$(arg kinect_camera_name)/driver/
                    depth_mode" value="5" />

        <include file="$(find course_p3at_navigation)/myKinect.
            launch" />
```

```
    <!-- NAVIGATION -->

    <node pkg="gmapping" type="slam_gmapping" respawn="false"
        name="slam_gmapping" output="screen">
            <param name="map_update_interval" value="2.0"/>
            <param name="maxUrange" value="6.0"/>
            <param name="iterations" value="1"/>
            <param name="linearUpdate" value="0.25"/>
            <param name="angularUpdate" value="0.262"/>
            <param name="temporalUpdate" value="-1.0"/>
            <param name="particles" value="300"/>
            <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="base_frame" value="base_link"/>
    </node>

    <node pkg="move_base" type="move_base" respawn="false"
        name="move_base" output="screen">
            <rosparam file="$(find course_p3at_navigation)/
                sg_costmap_common_params_p3at.yaml" command="
                load" ns="global_costmap" />
            <rosparam file="$(find course_p3at_navigation)/
                sg_costmap_common_params_p3at.yaml" command="
                load" ns="local_costmap" />
            <rosparam file="$(find course_p3at_navigation)/
                sg_local_costmap_params.yaml" command="load" />
            <rosparam file="$(find course_p3at_navigation)/
                sg_global_costmap_params.yaml" command="load" /
                >
            <rosparam file="$(find course_p3at_navigation)/
                becker_base_local_planner_params.yaml" command=
                "load" />
            <param name="base_global_planner" type="string"
                value=" navfn/NavfnROS" />
            <param name="controller_frequency" type="double"
                value="6.0" />
    </node>


</launch>
```

As you can see the code is divided by commentaries in four sections: PLATFORM SPECIFIC, TRANSFORMS, SENSORS CONFIGURATION and NAVIGATION.

PLATFORM SPECIFIC This first section of the code is relative to the nodes you have to run so your robot is able to read the information in the /cmd_vel topic and translate this information into the respective real velocities to the robot's wheels. In the case here represented, for the Pioneer 3-AT, two nodes are run: p2os_driver[13] and an instance of the rostopic[14] node. The first one, p2os_driver, is a ROS node specific for some Pioneer robots, including the Pioneer 3-AT, able to control the motors in accordance to the information it receives from the ros topics /cmd_vel and /cmd_motor_state. /cmd_vel has the velocities information and /cmd_motor_state tells the package if the motors are enabled. That is the reason the rostopic node should be run: it publishes a true value to the /cmd_motor_state topic so the p2os_driver knows that the motor should be enabled. p2os_driver also publishes some useful data, like sonar sensor data, transformations, battery_state and more.

TRANSFORMS As discussed in the transforms section, you should create a transform between the robot's base_link and the sensor laser_link. Here is the place we recommend it to be put, although it can be launched in a separate launch file or in any particular order in this launch file. Any other transforms you may have should be put here too.

SENSORS CONFIGURATION This section of the code was left to initialize all the nodes regarding the sensors powering up and configuration. The first four lines of this section contain some configurations of the kinect sensor:

- 1)The first line changes the camera name to kinect_camera_name;
- 2)The second lines sets it to drop 10 frames of the kinect for each valid one, outputting at 2 or 3Hz instead of 30Hz;
- 3)The third and fourth lines are in respect to the resolution, where we have selected 320x240 QVGA 30Hz for the image and 30Hz QVGA for the depth image. The available options for image mode are:
  - **2**: (640x480 VGA 30Hz)
  - **5**: (320x240 QVGA 30Hz)
  - **8**: (160x120 QQVGA 30Hz)

  And for the depth image mode:
  - **2**: (VGA 30Hz)
  - **5**: (QVGA 30Hz)
  - **8**: (QQVGA 30Hz)

It is noteworthy that these parameters configurations for the kinect, although recommended, are optional, since the default values will work. Besides the parameter configuration, there is also the line including the kinect launcher that we

wrote at the "Configuring the Kinect Sensor" section, which powers the sensor up and gets its data converted to laser data. If you're using any other kind of sensor, like a laser sensor, you should have your own launcher include here. Finally, if you odometry sensors aren't configured yet(in our case, the p2os_driver is responsible for this) you may do this here.

NAVIGATION Understanding which nodes you should run, why and what each of them do is the main focus of this chapter. To get our Pioneer 3-AT navigating, we're using two navigation nodes(gmapping and move_base) and a lot of parameters. As explained before, gmapping is a localization system, while move_base is a package that contains the local and global planners and is responsible for linking them to achieve the nav goal. Therefore, let's start explaining the gmapping launcher, the localization system we are using, since we have odometry available and our map is unknown(we will not use any static maps). For that, each parameter will be analysed at once, as follows:

- map_update_interval: time(in seconds) between two updates of the map. Ideally, the update would be instantaneous, however, it would cost too much for the CPU to do that. Therefore, we use a interval, for which the default is 5 seconds.
- maxUrange: the maximum range for which the laser issues valid data. Data farther from this distance will be discarded.
- iterations: the number of iterations of the scanmatcher.
- linearUpdate, angularUpdate and temporalUpdate: thresholds for a scan request. temporalUpdate asks for a new scan whenever the time passed since the last scan exceeds the time indicated in the parameter, while linearUpdate and angularUpdate ask for scan when the robot translates or rotates(respectively) the amount specified in the parameters.
- particles: sets the number of particles used in the filter.
- xmin, ymin, xmax and ymax: these four coordinates form, together, the map size.
- base_frame: indicates the frame that corresponds to the mobile base in the transform tree.

As to move_base, it bases its path planning techniques on the current location and the nav goal. In the node launcher code, we have the usual syntax to launch a node, followed by a list of seven parameters, five of which are rosparams. The params are two:

- 1) base_global_planner is a parameter for selecting the plugin(dynamically loadable classes). The plugin we use is the default for 1.1+ series, so we put this statement here just to ensure we've selected the correct one.
- 2) controller_frequency is a parameter that fixes the rate(in Hz) at which the control loop will run and velocity commands will be issued.

The rosparam, in turn, are files that contain more parameters for the move_base, and which are done this way to keep the files organized and easy to read. Thus, we

will take advantage of this fact and analyze the parameter files separately. First, let's begin looking at the costmap_common_params file, the one that contains parameters that apply for both the local and global costmaps:

```
obstacle_range: 5.0
raytrace_range: 6.0

max_obstacle_height: 1.0
min_obstacle_height: 0.05

footprint: [ [0.3302, -0.0508], [0.254, -0.0508], [0.254, -0.254],
     [-0.254, -0.254], [-0.254, 0.254], [0.254, 0.254], [0.254,
    0.0508], [0.3302, 0.0508] ]
#robot_radius: 0.35
inflation_radius: 0.35 #0.2
footprint_padding: 0

transform_tolerance: 1.0
map_type: costmap
cost_scaling_factor: 100 #10.0


#observation_sources: laser_scan_sensor
#laser_scan_sensor: {sensor_frame: camera_link, data_type:
    LaserScan, topic: scan, marking: true, clearing: true}

observation_sources: pointcloud_sensor
pointcloud_sensor: {sensor_frame: camera_link, data_type:
    PointCloud2, topic: /camera/depth/points, marking: true,
    clearing: true}
```

As you may know, the sharp(#) represents a commented line or value, and does not affect the results. This way, let's present the meaning of each of the params used in the costmap common parameters file:

- obstacle_range and raytrace_range: obstacle_range relates to the maximum distance(in meters) that will be considered when taking the obstacle data and putting it to the costmap, while raytrace_range is the maximum distance(also in meters) that will be considered when taking the free space around the robot and putting it to the costmap.
- max_obstacle_height and min_obstacle_height: these parameters set the area that will consider the sensor data as valid data. The most common is setting the min height near the ground height and the max height slightly greater than the robot's height.
- robot_radius and inflation_radius: when you're considering your robot as circular, you can just set the robot_radius parameter to the radius(in meters) of your robot and you get a circular footprint. Although, even if you don't have

a circular robot, it is important to set the inflation_radius to the "maximum radius" of your robot, so the costmap creates a inflation around obstacles and the robot doesn't collide, no matter what is it direction when getting close to obstacles.
- footprint and footprint_padding: when you want a most precise representation of your robot, you have to comment the robot_radius parameter and create a custom footprint, as we did, considering [0,0] as the center of your robot. footprint_padding is summed at each of the footprint points, both at the x and y coordinates, and we don not use it here, so we setted it to zero.
- transform_tolerance: sets the maximum latency accepted so the system knows that no link in the transform tree is missing. This parameter must be set in an interval that allows certain tolerable delays in the transform publication and detects missing transforms, so the navigation stack stops in case of flaws in the system.
- map_type: just here to enforce we are using a costmap.
- cost_scaling_factor: this parameter sets the scaling factor that applies over the inflation. This parameter can be adjusted so the robot has a more aggressive or conservative behaviour near obstacles.
  $e^{-cost\_scaling\_factor \times (distance\_from\_obstacle - inscribed\_radius) \times (costmap\_2d::INSCRIBED\_INFLATED\_OBSTACLE - 1)}$
- observation_sources: This last parameter is responsible for choosing the source of the sensor data. We can both use here point_cloud, as we're using for the kinect, or laser_scan, as the commented lines suggest and as may be used for a hokuyo or sick laser sensor. Along with the laser type, is very important to set the correct name of the subscribed topic, so the navigation stack takes the sensor data from the correct location.

Now that we have set all the costmap common parameters, we must set the parameter specific to the local and global costmaps. We will analyse them together, since most of the parameters are very similar. First, let's take a look at the files. For the global costmap we have:

```
global_costmap:
  global_frame: /map
  robot_base_frame: base_link
  update_frequency: 1.0
  publish_frequency: 1.0 #0
  static_map: false
  width: 50 #3.4
  height: 50 #3.4
  origin_x: -25 #-1.20 is the actual position; -0.95 is the old
      one, for the frond wheel at the marker
  origin_y: -25 #-1.91
  resolution: 0.1
```

And for the local:

```
local_costmap:
```

```
  global_frame: /odom
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 10.0
  static_map: false
  rolling_window: true
  width: 3.0
  height: 3.0
  resolution: 0.025
```

As you can see, both of them start with a tag specifying the costmap to which they relate. Then, we have the following common parameters:

- global_frame: indicates the frame for the costmap to operate in.
- robot_base_frame: indicates the transformation frame of the robot's base_link.
- update_frequency and publish_frequency: The frequency(in Hz) for map update and for publication of the display data.
- static_map: indicates whether the system uses or not a static map.
- width and height: width and height of the map, in meters.
- resolution: resolution of the map in meters per cell. This parameter is usually higher in smaller maps(local).

Aside from these common parameters, there's the definition of the map size along with the choosing between rolling window map or not. For the global map, we adopted the fixed map(there's no need to set rolling_windows to false, since it is the default), therefore we need to declare the x and y initial positions of the robots in respect to the map window. For the local_costmap, we use a rolling window map and the only parameter we have to set is the rolling_window to true.

Lastly, we have the base_local_planner parameters file. The base_local_planner treats the velocity data according to its parameters so the base_controller receives coherent data. Thus, the base_local_planner parameters are platform_specific. Let's take a look at the configuration for the Pioneer 3-AT:

```
TrajectoryPlannerROS:
  max_vel_x: 0.5
  min_vel_x: 0.1
  max_rotational_vel: 0.5
  max_vel_theta: 0.5
  min_vel_theta: -0.5
  min_in_place_rotational_vel: 0.5
  min_in_place_vel_theta: 0.5
  escape_vel: -0.1


  acc_lim_th: 0.5
  acc_lim_x: 0.5
```

```
acc_lim_y: 0.5

holonomic_robot: false
```

Again, we should analyse the most important parameters separately.

- min_vel_x and max_vel_x: The minimum and maximum velocities(in meters/second) allowed when sending data to the mobile base. The minimum velocity should be great enough to overcome friction. The maximum velocity adjust is good for limiting the robot's velocity in narrow environments.
- max_rotational_vel and min_in_place_rotational_vel: limits for the rotational velocities, the difference is that rotational_vel is the maximum rotation velocity when the mobile base is also moving forward or backward, while in_place_rotational_vel is the minimum rotation vel so the robot can overcome friction and turn without having to move forward or backward.
- min_vel_theta and max_vel_theta: the minimum and maximum rotational velocities(in radians/second).
- min_in_place_vel_theta: alike min_in_place_rotational_vel, but in radians per second.
- escape_vel: this speed delimits the driving speed during escapes(in meters per second). Its noteworthy that this value should be negative for the robot to reverse.
- acc_lim_x, acc_lim_y and acc_lim_theta: accelerations limits. They are the x,y and rotational acceleration limits respectively, wherein the first two are in meters per squared second and the last is radians per squared second.
- holomic_robot: this is a boolean responsible to choose between holonomic and non-holonomic robots, so the base_local_planner can issue velocity commands as expected.

Finally, we have a basic set up, contemplating all the usual parameters that you have to configure and some more. There is a small chance that some parameter is missing for your configuration, therefore it is a good idea to do a quick check in the base_local_planner[15] and costmap_2d [16] wiki pages.

The way that we have presented does not use layers, although ROS Hydro+ supports this feature. Porting these files to this new approach of costmaps is not a hard task, and that's what we will cover now.

3.7 Layered Costmaps

For this approach, we use the launchers and the configuration files from the previous package. First, we create a package named p3at_layer_navigation, as stated on the creating a package section. Then, we copy all files from the previous package but the package.xml and CMakelists.txt files to the folder of the newly created package. For the base_local_planner, nothing should be modified, since the planning will not be affected in any way when exploding the maps in layers. The common costmaps file is the one that will be affected the most, and here is one example costmap_common_params.yaml file that illustrates this:

```
robot_base_frame: base_link

transform_tolerance: 1.0

robot_radius: 0.35

footprint: [ [0.3302, -0.0508], [0.254, -0.0508], [0.254, -0.254],
     [-0.254, -0.254], [-0.254, 0.254], [0.254, 0.254], [0.254,
    0.0508], [0.3302, 0.0508] ]

inflater:
  robot_radius: 0.35
  inflation_radius: 0.35

obstacles:
  observation_sources: pointcloud_sensor
  pointcloud_sensor:
    data_type: PointCloud2
    topic: camera/depth/points
    min_obstacle_height: 0.2
    max_obstacle_height: 2.0
    marking: true
    clearing: true
  z_voxels: 8
  z_resolution: 0.25
  max_obstacle_height: 2.0
```

As you can see in the file, the parameters don't change much, the difference is that they are organized in a different way: there are some parameters that are common for all costmaps and there are some parameters that are common between layers. In this example, we create two layers: a inflater layer, that considers a circular robot with 35cm of radius, and, therefore, an inflation radius of 35cm so it doesn't collide with anything; a obstacles layer, that takes the pointcloud data(if you are using a laser, please change that here) and passes this data to the costmap.

The two other files have a slight modification: you should specify the layers they are using by using the plugins mnemonic, as shown for the global_costmap configuration file:

```
global_frame: map

robot_base_frame: base_link
update_frequency: 1.0
publish_frequency: 1.0
static_map: false
width: 50
height: 50
```

```
origin_x: -25 #-1.20 is the actual position; -0.95 is the old one,
     for the frond wheel at the marker
origin_y: -25 #-1.91
resolution: 0.1

plugins:
 - {name: obstacles, type: "costmap_2d::VoxelLayer"}
 - {name: inflater, type: "costmap_2d::InflationLayer"}
```

The local_costmap should have the same plugins statement at the end. Moreover, you can add any extra layers you want. The structure of the topics will change a little bit, since the footprint is now inside the layers and the costmaps are divided in multiple topics. You can get to know a little more about this organization in the Using rviz section.

4 Starting with a Test

Before we begin the testing, we must find a way to visualize the navigation in action. That can be done through the software rviz, that allows us, amongst other things, to visualize the sensor data in a comprehensive way and to check the planned paths as they are generated. The execution of rviz is often in a computer different from the one that operates on the robot, so you may use multiple machines that share the same ROS topics and communicate.

4.1 Using rviz

To run rviz, simply issue the following command:

```
1 $ rosrun rviz rviz
```

The interface of rviz depends on your version, but the operation should be very similar. It is way easier to configure rviz with navigations stack up and running, although it is possible to do so without it. In this tutorial we will only cover the configuration steps when the navigation stack launcher is already implemented, so make sure you have launched all the nodes needed for navigation and just then launched rviz. After launching rviz, you should add the topics you wish to display. First, let's do an example by adding the PointCloud 2 from the kinect, as shown in figure 3.

As you can see in figure 3, there are four steps for adding a new topic when navigation stack is already running:

– 1) Click on the button "add" at the bottom left-hand side of the screen;

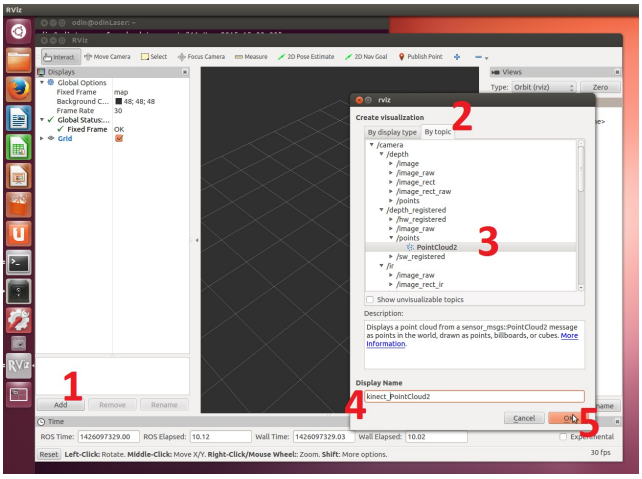

Fig. 3. Steps for adding new information for rviz to display.

– 2) Choose the tab "By topic" on the windows that appears. This is only possible when the topics are available, so if you don't have the navigation stack running you will have to choose the info in the tab "By display type" and manually insert the topic names and types.

– 3) Select the topic and its message type on the central frame of the window. In this example, we are selecting the PointCloud2 data that the kinect provides on the /camera/depth_registered/points topic.

– 4) Write a meaningful display name in the textbox, so you don't forget what the data is representing in the future.

– 5) Confirm the addition by pressing "Ok".

The process is equal for all kinds of topics, so a list of the most common topics(note: depending if you changed some topic names, some things on the list may differ) should be enough to understand and add all the topics you need.

NAME	TOPIC	MESSAGE TYPE
ROBOT FOOTPRINT	/local_costmap/robot_footprint	geometry_msgs/PolygonStamped
LOCAL COSTMAP	/move_base/local_costmap/costmap	nav_msgs/GridCells
OBSTACLES LAYER	/local_costmap/obstacles	nav_msgs/GridCells
INFLATED OBSTACLES LAYER	/local_costmap/inflated_obstacles	nav_msgs/GridCells
STATIC MAP	/map	nav_msgs/GetMap or nav_msgs/OccupancyGrid
GLOBAL PLAN	/move_base/TrajectoryPlannerROS/global_plan	nav_msgs/Path
LOCAL PLAN	/move_base/TrajectoryPlannerROS/local_plan	nav_msgs/Path
2D NAV GOAL	/move_base_simple/goal	geometry_msgs/PoseStamped
PLANNER PLAN	/move_base/NavfnROS/plan	nav_msgs/Path
LASER SCAN	/scan	sensor_msgs/LaserScan
KINECT POINTCLOUD	/camera/depth_registered/points	sensor_msgs/PointCloud2

It is interesting to know a little more about topics that you haven't heard about, because every topic listed here is very valuable at checking the navigation functionalities at some point. Therefore, lets do a brief explanation at each of the topics:

- **ROBOT FOOTPRINT:** These message is the displayed polygon that represents the footprint of the robot. Here we are taking the footprint from the local_costmap, but it is possible to use the footprint from the global_costmap and it is also possible to take the footprint from a layer, for example, the footprint may be available at the /move_base/global_costmap/obstacle_layer_footprint/footprint_stamped topic.
- **LOCAL COSTMAP:** If you're not using a layered approach, your local_costmap in its whole will be displayed in this topic.
- **OBSTACLES LAYER:** One of the main layers when you're using a layered costmap, containing the detected obstacles.
- **INFLATED OBSTACLES LAYER:** One of the main layers when you're using a layered costmap, containing areas around detected obstacles that prevent the robot from crashing with the obstacles.
- **STATIC MAP:** When using a pre-built static map it will be made available at this topic by the map_server.
- **GLOBAL PLAN:** This topic contains the portion of the global plan that the local plan is considering at the moment.
- **LOCAL PLAN:** Display the real trajectory that the robot is doing at the moment, the one that will imply in commands to the mobile base through the /cmd_vel topic.

- **2D NAV GOAL:** Topic that receives navigation goals for the robot to achieve. If you want to see the goal that the robot is currently trying to achieve you should use the /move_base/current_goal topic.
- **PLANNER PLAN:** Contains the complete global plan.
- **LASER SCAN:** Contains the laser_scan data. Depending on your configuration this topic can be a real reading from your laser sensor or it can be a converted value from another type of sensor.
- **KINECT POINTCLOUD:** This topic, shown in the example, is a cloud of points, as the name suggests, that forms, in space, the depthimage captured by the kinect. If you are using a laser sensor, this topic will not be available.

These are the most used topics, however you may have a lot more depending on your setup and in what you want to see. Besides, just the local_costmap and the most used layers of it were presented, but you may want to see the global costmap and its layers, in addition to another layers that you may use. Explore the topics you have running with the rviz and you may find more useful info.

4.2 Multiple Machines Communication

It is possible that you have to use more than one computer at the same time when navigating with a robot. Usually, two computers are used: one is mounted on the mobile base and is responsible for getting sensor data and passing velocities commands to the robot, while the other is responsible for heavy processing and monitoring.

To get the machines working together you must specify names for both machines in the /etc/hosts file on your system. The usual hosts file is similar to:

IPAddress Hostname
127.0.0.1 localhost
192.168.1.101 robot
192.168.1.100 masterpc

You have to choose a name for both the machines(here we chose robot and masterpc) and add to both the /etc/hosts files the entries for them. The entry on the /etc/hosts must have the IP of the machine in the wireless lan they share and the name you picked(two new entries per file, one for itself and other for the other PC). After that, you should set the ROS_MASTER_URI variables. In the master machine, it should be:

```
1 $ export ROS_MASTER_URI=http://localhost:11311
```

In the other machine, you should type:

Test your configuration this way, and if the configurations work, add the export lines to the end of the /.bashrc file of both computers, so every time a terminal window is opened these commands are issued.

```
1 $ export ROS_MASTER_URI=http://othermachine:11311
```

4.3 Real tests on Pioneer 3-AT

Finally, it is time to see the robot navigating. Launch your navigation file and then run rviz. If you made all the correct configuration for the navigation stack and for rviz you should be able to see your robot footprint, pose and the costmaps. Try selecting the 2D Nav goal at the top of the screen in rviz, click and hold at some point on the map and then choose the direction, so the robot knows where to go and in what position it should stop. A global path would be generated, as well as a local, and you should see them indicated by green lines. You can see an example of the robot navigating on figures 4 and 5.

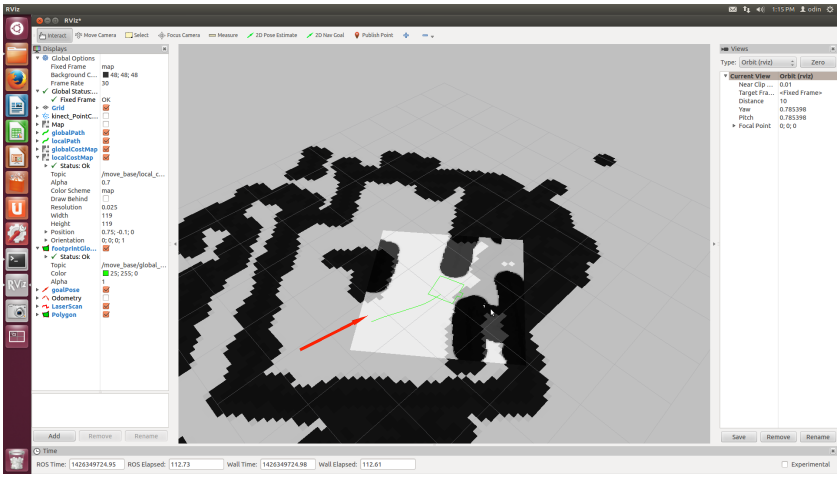

Fig. 4. Example of the Pioneer 3-AT navigating in a room.

As you can see in the pictures, a global plan is drawn from the start point to the finish point and a local plan is being drawn along the way, trying to follow the global path without crashing. Conducting the tests we have found that the robot does not works so great without AMCL working. Although, in theory, the gmapping is self-sufficient and AMCL is only used for static maps, we have found that, in practice, the robots get lost easily when using gmapping only, that being because the odometry errors. Gmapping bases its mapping and localization on the odometry and the odometry errors make it confused. When you create the map with gmapping and send it to the AMCL node, it trusts the map and adapts the odometry, and we have found that this use produces much better results. In order to do that, you have to modify your launcher file, adding the AMCL launcher in the navigation section. The code snippet will look like this:

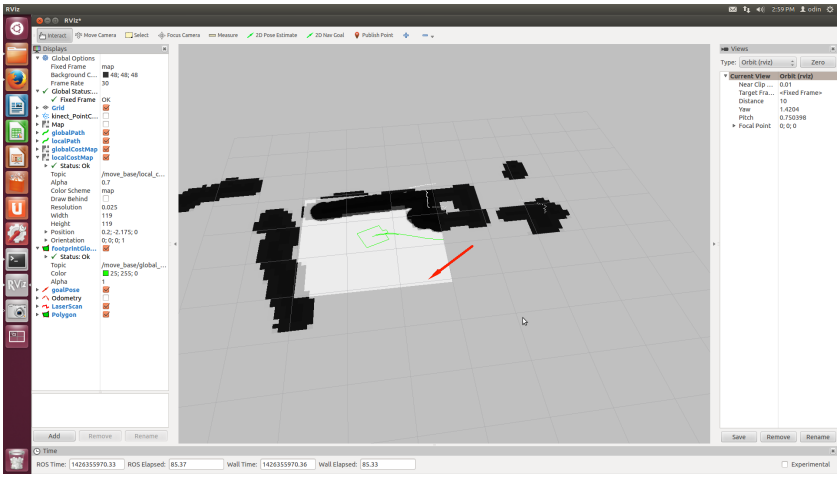

Fig. 5. Example of the Pioneer 3-AT navigating in another room.

```
<launch>
        <node name="amcl" pkg="amcl" type="amcl">
            <param name="odom_frame_id" value="odom"/>
            <param name="base_frame_id" value="base_link"/>
            <param name="global_frame_id" value="map"/>
        </node>
</launch>
```

Pay attention to the attribution of the right topic names and frame_ids. AMCL has a lot of other parameters, but, for our use here, the default parameters were enough.

5 Authors Biographies

5.1 Rodrigo Longhi Guimarães

Studied to be an electronic technician at Technological Federal University of Paran(UTFPR) and worked in the field, as a full flight simulator maintainer for circa three years. Nowadays, he is a student of computer engineering at UTFPR and works as a research intern at LASER(advanced laboratory of embedded systems and robotics). At LASER, works with a Pioneer 3-AT to compete in the Robocup's @Home category, where the robot should autonomously navigate, follow people, identify objects, interact with people through natural speaking and perform some other tasks that show the robot's ability to operate in a domestic environment. In the last edition of the Latin American Robots Competition, the robot was placed third in its category.

5.2 André Schneider de Oliveira

5.3 João Fabro

5.4 Thiago Becker

5.5 Vinícius Amilgar Brenner

Formed electronic technician at UTFPR and now student of computer engineering on the same institution, divides his time studying and being a fellow of LASER, the advanced laboratory of embedded systems and robotics, where he studies robotics and ROS and builds soccer robots to dispute on the Robocup's F-180 category.

References

1. ROSwiki, "geometry_msgs." http://wiki.ros.org/geometry_msgs, 2015.
2. ROSwiki, "openni_camera." http://wiki.ros.org/openni_camera, 2015.
3. ROSwiki, "openni_launch." http://wiki.ros.org/openni_launch, 2015.
4. AnthonyJ350, "How to solder / soldering basics tutorial." https://www.youtube.com/watch?v=BxASFu19bLU, 2011.
5. mjlorton, "How to solder / soldering basics tutorial." https://www.youtube.com/watch?v=kjSGCSwNuAg, 2013.
6. E.-C. Wireconnector, "How to solder / soldering basics tutorial." https://www.youtube.com/watch?v=5_47xaUZQwk, 2013.
7. ROSwiki, "Adding a kinect to an irobot create/roomba." http://wiki.ros.org/http://wiki.ros.org/kinect/Tutorials/Adding2015.
8. ROSwiki, "depthimage_to_laserscan." http://wiki.ros.org/depthimage_to_laserscan, 2015.
9. ROSwiki, "kinect_aux." http://wiki.ros.org/kinect_aux, 2015.
10. ROSwiki, "openni_tracker." http://wiki.ros.org/openni_tracker, 2015.
11. ROSwiki, "cyphy_people_mapping." http://wiki.ros.org/cyphy_people_mapping, 2015.
12. STRANDS, "sicks300 git repository." https://github.com/strands-project-releases/ sicks300/tree/release/hydro/sicks300, 2015.
13. ROSwiki, "p2os_driver." http://wiki.ros.org/p2os_driver, 2015.
14. ROSwiki, "rostopic." http://wiki.ros.org/rostopic, 2015.
15. ROSwiki, "base_local_planner." http://wiki.ros.org/base_local_planner, 2015.
16. ROSwiki, "costmap_2d/layered." http://wiki.ros.org/costmap_2d/layered, 2015.