

ROS Navigation: Concepts and Tutorial

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

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 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, establish 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 Stack · Tutorial · Real robots · Transformations

1 Introduction

Simultaneous Localization And Mapping (SLAM) is an important algorithm that allows the robot to acknowledge the obstacles around it and localize itself. When combined with some other methods, such as path planning, it is possible to allow robots to navigate unknown or partially known environments. ROS has a package

R.L. Guimarães (✉) · A.S. de Oliveira · J.A. Fabro · T. Becker · V.A. Brenner
LASER - Advanced Laboratory of Embedded Systems and Robotics, Federal University of Technology - Paraná, Av. Sete de Setembro, Curitiba 3165, Brazil
e-mail: rguimaraes@alunos.utfpr.edu.br

A.S. de Oliveira
e-mail: andreoliveira@utfpr.edu.br

J.A. Fabro
e-mail: fabro@dainf.ct.utfpr.edu.br

T. Becker
e-mail: beckerthiago@gmail.com

V.A. Brenner
e-mail: brenner@alunos.utfpr.edu.br

that performs SLAM and path planning along with other functionalities for navigation, 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 autonomous navigation in any robot. It will also explain how to use a virtual environment to use the Navigation Stack on virtual robots. These robots are designed to 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 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 indirectly 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 SLAM will be presented, along with some discussion about transformations.

In the second section, ROS 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 (do not worry, you will see that in detail later). Perception sensors are also discussed. The differences and advantages of each of the main kinds of perception sensors, like depth sensors, light detection 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 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, ensuring that the reader is apt to perceive a way to have his own robot navigate itself. 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 way in which all the navigation configuration files will be analyzed, examining the parameters and setting them to example values that correspond to pioneer (ARIA based) robots (e.g. Pioneer 3-AT, as seen on Fig. 1). The explanation of the whole process in addition to the demonstration of the effects of the parameter changes should be enough to clear up any doubts the reader might have.

Fig. 1 Side by side are two pioneer ARIA based robots: Pioneer 3-AT and Pioneer LX. These robots, owned by the Advanced Laboratory of Embedded Systems and Robotics of UTFPR, can use the Navigation Stack



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 experiments 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 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 known, partially known or unknown environment, in other words, the robot is capable of planning and tracking a path 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 SLAM systems (also called localization systems), that allow a robot to locate itself, whether there is a static map available or SLAM 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 entire area in which the robot could navigate would have to be mapped in a metrically correct way to use static maps, a big disadvantage of this resource. Depending on the environment that surrounds the robot, these static maps are capable of increasing or decreasing the confidence of the localization systems. To bypass the lack of flexibility of static maps, two other localization systems are offered by ROS to work with the Navigation Stack: `Gmapping` and `hector_mapping`.

Both Gmapping and hector_mapping are implementations of 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 its 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 proprioceptive sensors, which makes the usage of it hard for some robots (e.g. flying robots). The odometry information is interesting because they are able to aid on the generation of more precise maps, since understanding the robot kinematics we can estimate its pose.

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 wheel's positioning and the angle at which they are disposed. A more in-depth explanation of kinematics is done in Introduction to Mobile Robots [1].

However, as much useful as the odometry information can be, it is not immune to faults. The faults are caused by the lack of precision on the capture of data, friction, slip, drift and other factors. These accumulated factors can lead to inconsistent data and prejudice 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, suit the measures to the robot's navigation.

3 ROS Environment

Before we begin setting up the environment in which we will work, it is very important to be aware of the limitations of the Navigation Stack, so we are 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 will not be covered here.
- The Navigation Stack assumes that the robot receives a twist type message [2] 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 to 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. Do not worry with that for now, it is just good to know that if some node is not launching it may be because the package you are 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 know 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. Most of this chapter is structured in a “by example” approach from now on and if you like it, you should check out the ROS By Example—Volume 1 [3] book. In this book, there are a lot of useful codes for robots and a navigation section that complements very well what we discuss in this chapter. You also might find useful to check their example codes [4], where you can find a lot of useful codes, such as rviz configuration files and navigation examples.

Important Note: no matter what robot you're using, you will probably need to allow the access to the USB ports that have sensors or to the ports that communicate to your robots. For example, if you have the robot connected in `ttyUSB0` port, you can give the access permission by issuing the following command:

```
1 sudo chmod 777 -R /dev/ttyUSB0
```

This configuration will be valid only for the current terminal, so it is recommended to add this instruction to the `~/.bashrc` file. Edit this file by writing the following lines at the bottom of the file:

```
1 if [ -e /dev/ttyUSB0 ]; then
2 sudo chmod 777 -R /dev/ttyUSB0
3 echo "Robot found on USB0!"
4 fi
```

This command will check if there is something in the `/dev/ttyUSB0` port and, if the condition holds, it will change the permissions on the port and will print to the screen the message "Robot found on USB0!". It is noteworthy that, as said above, this command is assuming that the robot is on `/dev/ttyUSB0`. Also, this script will be executed every time a new terminal is opened and since a `sudo` command is issued a password will be prompted for non-root users.

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` [5]. 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). We will also need a ROS publisher to use the sensor data and that publisher is found in the `Openni_launch` [6] package, that can be obtained the exactly same ways. To do so, open your terminal window and type:

```
1 $ sudo apt-get install ros-<rostdistro>-openni-camera
2 $ sudo apt-get install ros-<rostdistro>-openni-launch
```

Remember to change "`<rostdistro>`" to your `ros` version (i.e. 'hydro' or 'indigo'). 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 (e.g. 100 V@60 Hz) to the DC voltage needed to power up the Kinect (12 V), therefore the solution is exchanging the AC adapter for a 12 V 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 12 V battery and the white wire to the negative(-). You can do this connection by soldering [7] or using connectors, such as crimp [8] or clamp [9] wire connectors.

Be aware that the USB connection is enough to blink the green LED in front of the Kinect and it does not indicate that the external 12 V 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 [10].

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:

```
1 $ sudo modprobe -r gspca_kinect
2 $ sudo modprobe -r gspca_main
```

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

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

Lastly, we need to convert the depth image data to a LaserScan message. That is needed because Gmapping, the localization system we are using, accepts as input a single LaserScan message. Fortunately, we have yet another packages to do this for us, the `Depthimage_to_laserscan` package and the `ira_laser_tools` package. You just need one of the two packages, and, although the final result is a LaserScan message for both, the data presented will be probably different. The `Depthimage_to_laserscan` package is easier to get and use, and as we have done with the other packages, we can check out the wiki page [11] 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
```

This package will take a horizontal line of the PointCloud and use it to produce the LaserScan. As you can imagine, it is possible that some obstacles aren't present at the selected height, the LaserScan message will not represent the PointCloud data so well and gmapping will struggle to localize the robot in the costmap generated by the Navigation Stack using the PointCloud directly.

The other option, `ira_laser_tools` package, produces a more representative result, but it is harder to use and consumes a lot more processing power. In this package you have two nodes: `laserscan_virtualizer` and `laserscan_multi_merger`. The first one, can be configured to convert multiple lines of your PointCloud into different LaserScan messages. The second one, the merger node, receives multiple LaserScan messages and merges them in a single LaserScan message. You can check the article found on [12] to find out more about the package.

To install this package, you have to first clone the git repository found on [13] to your catkin workspace source folder and then compile it. This is done with the following commands:

```
1 $ cd /home/user/catkin_ws/src
2 $ git clone https://github.com/iralabdisco/ira_laser_tools.git
3 $ cd /home/user/catkin_ws
4 $ catkin_make
```

Now that the nodes are compiled, you have to configure the launch files to suit your needs. Go to the launch folder, inside the `ira_laser_tools` package folder, and open `laserscan_virtualizer.launch` with your favorite editor. You should see something like this:

```
1 <!--
2 FROM: http://wiki.ros.org/tf#static\_transform\_publisher
3
4 <<static_transform_publisher x y z yaw pitch roll
   frame_id child_frame_id period_in_ms>>
5 Publish a static coordinate transform to tf using an x/
   y/z offset and yaw/pitch/roll. The period, in
   milliseconds, specifies how often to send a
   transform. 100ms (10hz) is a good value.
6 == OR ==
7 <<static_transform_publisher x y z qx qy qz qw frame_id
   child_frame_id period_in_ms>>
8 Publish a static coordinate transform to tf using an x/
   y/z offset and quaternion. The period, in
   milliseconds, specifies how often to send a
   transform. 100ms (10hz) is a good value.
9
10 -->
11
```



```

12 <launch>
13
14 <!-- DEFINE HERE THE STATIC TRANSFORMS, FROM
    BASE_FRAME (COMMON REFERENCE FRAME) TO THE
    VIRTUAL LASER FRAMES-->
15 <!-- WARNING: the virtual laser frame(s) *must*
    match the virtual laser name(s) listed in
    param: output_laser_scan -->
16 <node pkg="tf" type="static_transform_publisher"
    name="ira_static_broadcaster1" args="0 0 0
    0 0.3 0 laser_frame scan1 1000" />
17 <node pkg="tf" type="static_transform_publisher"
    name="ira_static_broadcaster2" args="0 0 0
    0 0.0 0 laser_frame scan2 1000" />
18 <node pkg="tf" type="static_transform_publisher"
    name="ira_static_broadcaster3" args="0 0 0
    0.0 0.0 0.3 laser_frame scan3 1000" />
19 <node pkg="tf" type="static_transform_publisher"
    name="ira_static_broadcaster4" args="0 0 0
    0.0 0.0 -0.3 laser_frame scan4 1000" />
20
21 <node pkg="ira_laser_tools" name="
    laserscan_virtualizer" type="
    laserscan_virtualizer" output="screen">
22 <param name="cloud_topic" value="/
    cloud_in"/> <!-- INPUT POINT CLOUD --
    >
23 <param name="base_frame" value="/
    laser_frame"/> <!-- REFERENCE FRAME
    WHICH LASER(S) ARE RELATED-->
24 <param name="output_laser_topic" value ="
    /scan" /> <!-- VIRTUAL LASER OUTPUT
    TOPIC, LEAVE VALUE EMPTY TO PUBLISH
    ON THE VIRTUAL LASER NAMES (param:
    output_laser_scan) -->
25 <param name="virtual_laser_scan" value ="
    scan1 scan2 scan3 scan4" /> <!-- LIST
    OF THE VIRTUAL LASER SCANS. YOU MUST
    PROVIDE THE STATIC TRANSFORMS TO TF,
    SEE ABOVE -->
26 </node>
27 </launch>

```

This launch file is responsible for converting some parts of the PointCloud in multiple LaserScans. As you can notice from the file, it is taking four lines from the Kinect PointCloud: two horizontal lines, one at the kinect level and another 30cm above the kinect level; two oblique lines, each of them with 0.3 rad of rotation to one side in relation to the Kinect and in the same horizontal level. You can have more information about the usage in the comments in the code itself and on this article [12], but here are the things you will most likely need to modify to correctly use the package:

- **The number of scans you want:** You have to find a number good enough to summarize a lot of information without using too much CPU. In our case, we tested with 8 horizontal lines, from 25 cm below the Kinect to 90 cm above of it. We used horizontal lines because that way the ground would not be represented in the LaserScan message. To do this, simply duplicate one of the node launching lines, changing the name of the node and the name of the scan to a unique name. For each line, you should configure the transform coordinates to whatever you see fit, although we recommend the horizontal lines approach. Do not forget to put the name of the new scan on the `virtual_laser_scan` parameter.
- **The tf frame of the Kinect:** Change `laser_frame` to the name of the frame of your Kinect. In our case, `camera_link`.
- **Input PointCloud:** Change this line to the name of the topic that contains the PointCloud. `Openni_launch` publishes it, by default, to `/camera/depth/points`.
- **The base_frame parameter:** Selects the frame to which the LaserScans will be related. It is possible to use the same frame as the Kinect.
- **The output_laser_topic parameter:** Selects the output topic for the LaserScan messages. We left it blank, so each of the LaserScans would go to the topic with their respective name.

Lastly, you have to configure the merger node. Start by opening up the `laser_scan_multi_merger.launch` file, on the same folder of the `virtualizer` launch file. You should see this:

```

1 <!--
2 DESCRITPION
3 -->
4
5
6 <launch>
7     <node pkg="ira_laser_tools" name="
8         laserscan_multi_merger" type="
9         laserscan_multi_merger" output="screen">
            <param name="destination_frame" value="/
                cart_frame"/>
            <param name="cloud_destination_topic"
                value="/merged_cloud"/>

```

```

10         <param name="scan_destination_topic"
           value="/scan_multi"/>
11         <param name="laserscan_topics" value =
           scandx scansx" /> <!-- LIST OF THE
           LASER SCAN TOPICS TO SUBSCRIBE -->
12     </node>
13 </launch>
    
```

This launch file is responsible for merging multiple LaserScans into a single one. As you can notice from the file, it is taking two LaserScan topics (`scandx` and `scansx`) and merging. The three things you will most likely modify are:

- **The destination_frame parameter:** Chooses the frame to which the merged LaserScan is related. Again, you can simply use the Kinect frame.
- **Cloud :** Change `laser_frame` to the name of the frame of your Kinect. In our case, `camera_link`.
- **The scan_destination_topic:** You will probably want gmapping to read this information, so, unless you have some special reason not to do that, use the `/scan` topic here.
- **The laserscan_topics paramter:** List all the topics that contain the LaserScan messages you want to merge. If you are using eight, you will most likely use `scan1` to `scan8` here.

Important Notes: When using this package, there are some important things you have to notice. First, it won't work correctly if you launch the merger before the virtualizer, and since ROS launchers do not guarantee the order, you may use a shell script to do that for you. Second, if the transformations take too long to be published, you may end up with some issues in the merger. To correct that, we modified the 149th line of the code, changing the `ros::Duration` parameter from 1 to 3 (setting this parameter too high will make the LaserScan publication less frequent). The line will then look like that:

```

1 tfListener_.waitForTransform(scan->header.frame_id.c_str(), destination_frame.c_str(),
  scan->header.stamp, ros::Duration(3));
    
```

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

- **kinect_aux [14]:** 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 is also installed with a simple `apt-get` command.
- **Natural Interaction—openni_tracker [15]:** 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 is kind of tough to install and the

process 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 listed below.

- 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 official `openni` website is no longer in the air, but you can get the files on `Openni.ru` [16] or on [17], where I made them available. The first two files (NITE and `Openni`) can be installed following the `cyphy_people_mapping` [18] 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 do not need to start a lot of packages manually when using the Kinect. Here is an example of a complete launch file for starting the Kinect and all the packages that make its data navigation-ready:

```
1 <launch>
2   <include file="$(find oppenni_launch)/launch/
3     oppenni.launch" />
4   <node respawn="true" pkg="
5     depthimage_to_laserscan" type="
6     depthimage_to_laserscan" name="laserscan">
7     <remap from="image" to="/camera/depth/
8       image" />
9   </node>
10 </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. Analyzing 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 is 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 [19]. The procedure for getting it to work is:

- Cloning the repository, by using the command `git clone https://github.com/bohlander/sicks300.git` (the URL will change depending on your version);
- Compile the files. For `roscmake` versions, use `roscmake sicks300 sicks300` and for `catkinized` versions, use `catkin_make`.
- Run the files by using the command `roscrun sicks300 sicks300_driver`. It should work with the default parameters, but, if it doesn’t, check if you have configured the baud rate 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`, `hokuyo_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 Transformations

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`). It is possible to understand a little more of transformations by examining the Fig. 2.

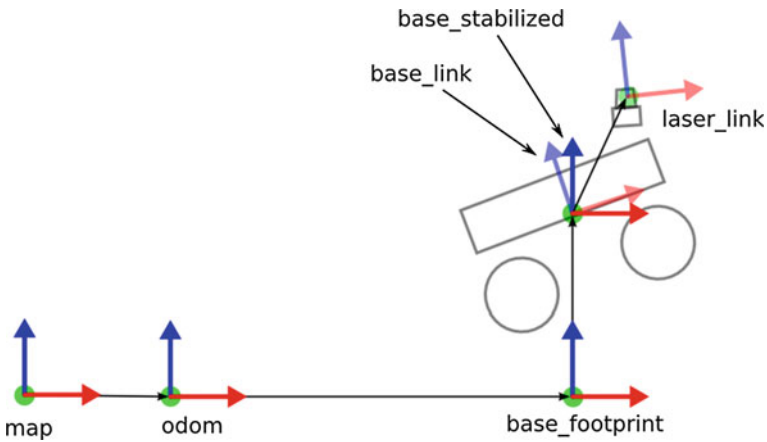


Fig. 2 Figure containing the most commonly used tf frames [20]

As you can see on the Fig. 2, we have several frames. A brief explanation of each of the topics is done below:

- ***map*** is the global reference frame and the robot pose should not drift much over time in relation to it [21].
- ***odom*** drifts and can cause discrete jumps when new sensor information is available.
- ***base_link*** is attached to the robot center.
- ***base_footprint*** is very straightforward: it is the `base_link` projection on the ground (zero height). Therefore, its transform is published in relation to the `base_link`.
- ***base_stabilized*** is the center position of the robot, not computing the roll and pitch angles. Therefore, its transform is also published in relation to the `base_link`.
- ***laser_link*** is the center position of the laser sensor, and its transform is published in relation to the `base_link`.

There is a clear hierarchy on these frames: `map` is the parent of `odom` and `odom` is the parent of `base_link`. The transform between `odom` and `base_link` has to be computed over the odometry sensor data and published. The transformation between `base_link` and `map` is computed by the localization system and the other transformation, between the `map` and the `odom` frames, uses this information to be computed.

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 will 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 35 cm (0.35 m) and our height distance (y) is 10 cm (0.1 m). 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. To understand the quaternion representation better, refer to Modeling and Control of Robot Manipulators [22]. 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 will 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 do not 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, 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 /home/user/catkin_ws/src
2 $ catkin_create_pkg packageName std_msgs 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 /home/user/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 Fig. 3. The items will be analyzed in the following sections block by block.

You can see on Fig. 3 that there are three types of nodes: provided nodes, optional provided nodes and platform specific nodes.

- The nodes inside the box, provided nodes, are the core of the Navigation Stack, and are responsible, mainly, by managing the costmaps and for the path planning functionalities.
- The optional provided nodes, `amcl` and `map_server`, are related to static map functions, as will be explained later, and since using a static map is optional, using these nodes is also optional.
- The platform specific nodes are the nodes related to your robot, such as sensor reading nodes and base controller nodes.

In addition, we have the localization systems, not shown in Fig. 3. If the odometry source was perfect and no errors were present on the odometry data, we would not need to have localization systems. However, in real applications that is not the case, and we have to account other types of data, such as IMU data, so we can correct odometry errors. The localization systems discussed on this chapter are:

amcl, gmapping and hector_mapping. Below, a table is available to relate a node to a keyword, so you can understand better the relationship of the nodes of the Navigation Stack.

Localization	Environment inter-action	Static maps	Trajectory planning	Mapping
amcl	sensor_sources	amcl	global_planner	local_costmap
gmapping	base_controller	map_server	local_planner	global_costmap
hector_mapping	odometry_source		recovery_behaviors	

3.5.1 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 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. It uses the base_footprint or base_link transformation to the map to work, therefore it needs a static map and it will only work after a map is created. This localization system is based on the Monte Carlo localization approach: it randomly distributes particles in a known map, representing the possible robot locations, and then uses a particle filter to determine the actual robot pose. To know more about this process, refer to Probabilistic Robotics [24].

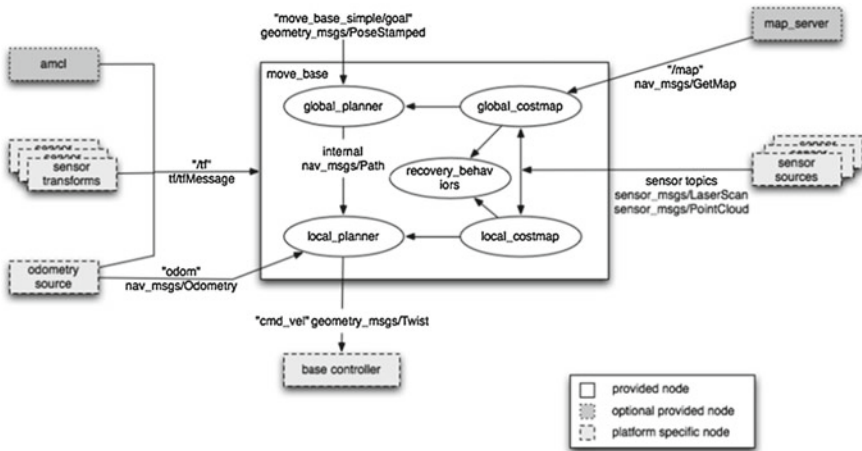


Fig. 3 Overview of a typical system running the Navigation Stack [23]

3.5.2 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). It works over the odom to map transformation, therefore it does not need the map nor IMU information, needing only the odometry.

3.5.3 Hector_mapping

As said in the background section, hector_mapping can be used instead of using gmapping. It uses the base_link to map transformation and it does not need the odometry nor the static map: it just uses the laser scan and the IMU information to localize itself.

3.5.4 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 are 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. It is noteworthy that the whole system will not work if the sensor transforms are not available, since they will be used to calculate the position of the sensor readings on the environment.

3.5.5 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. These projections are associated to a cost, and the robot objective is to achieve the navigation goal by creating a path with the least possible cost. 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.

3.5.6 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 while it deviates from newly inserted obstacles or obstacles that the global costmap definition was not able to detect. 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 navigation 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:

```

1 <launch>
2     <master auto="start" />
3
4     <!-- PLATFORM SPECIFIC -->
5     <node pkg="p2os_driver" type="p2os_driver" name=
6         "p2os_driver" >
7         <param name="port" value="/dev/ttyUSB0" /
8         >
9         <param name="pulse" value="1.0" />"
10    </node>
11
12    <node pkg="rostopic" type="rostopic" name="
13        enable_robot" args="pub /cmd_motor_state
14        p2os_driver/MotorState 1" respawn="true">
15    </node>
16
17    <!-- TRANSFORMS -->
18    <node pkg="tf" type="static_transform_publisher"
19        name="Pioneer3AT_laserscan_tf" args="0.1 0
20        0.35 0 pi/2 pi/2 base_link camera_link 100"
21        />
22
23    <!-- SENSORS CONFIGURATION -->

```

```

17 <arg name="kinect_camera_name" default="camera"
    />
18     <param name="/$(arg kinect_camera_name)/
        driver/data_skip" value="1" />
19     <param name="/$(arg kinect_camera_name)/
        driver/image_mode" value="5" />
20     <param name="/$(arg kinect_camera_name)/
        driver/depth_mode" value="5" />
21
22 <include file="$(find course_p3at_navigation)/
    myKinect.launch" />
23
24 <!-- NAVIGATION -->
25
26 <node pkg="gmapping" type="slam_gmapping"
    respawn="false" name="slam_gmapping" output=
    "screen">
27     <param name="map_update_interval" value="
        2.0"/>
28     <param name="maxUrange" value="6.0"/>
29     <param name="iterations" value="5"/>
30     <param name="linearUpdate" value="0.25"/>
31     <param name="angularUpdate" value="0.262"
        />
32     <param name="temporalUpdate" value="-1.0"
        />
33     <param name="particles" value="300"/>
34     <param name="xmin" value="-50.0"/>
35     <param name="ymin" value="-50.0"/>
36     <param name="xmax" value="50.0"/>
37     <param name="ymax" value="50.0"/>
38     <param name="base_frame" value="base_link"
        "/>
39     <param name="minimumScore" value="200.0"/
        >
40     <param name="srr" value="0.01"/>
41     <param name="str" value="0.01"/>
42     <param name="srt" value="0.02"/>
43     <param name="stt" value="0.02"/>
44 </node>
45
46 <node pkg="move_base" type="move_base" respawn="
    false" name="move_base" output="screen">
47     <roscparam file="$(find
        course_p3at_navigation)/

```

```

48         sg_costmap_common_params_p3at.yaml "
           command="load" ns="global_costmap" />
<rosparam file="$(find
           course_p3at_navigation)/
           sg_costmap_common_params_p3at.yaml "
49         command="load" ns="local_costmap" />
<rosparam file="$(find
           course_p3at_navigation)/
           sg_local_costmap_params.yaml" command
50         ="load" />
<rosparam file="$(find
           course_p3at_navigation)/
           sg_global_costmap_params.yaml"
51         command="load" />
<rosparam file="$(find
           course_p3at_navigation)/
           becker_base_local_planner_params.yaml
           " command="load" />
52 <param name="base_global_planner" type="
           string" value=" navfn/NavfnROS" />
53 <param name="controller_frequency" type="
           double" value="6.0" />
54 </node>
55
56
57 </launch>

```

As you can see the code is divided by commentaries in four sections as enumerated below:

3.6.1 PLATFORM SPECIFIC

Before explaining this section it is useful to clarify something: these PLATFORM SPECIFIC code is related to the platform specific nodes and it is not, by any means, the only platform specific sections of the navigation configuration, since almost all parameters presented on this chapter can be modified depending on your needs. 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` [25] and an instance of the `rostopic` [26] 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.

3.6.2 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.

3.6.3 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 1 frame of the Kinect for each valid one, outputting at approximately 15 Hz instead of 30 Hz;
- (3) The third and fourth lines are in respect to the resolution, where we have selected 320×240 QVGA 30 Hz for the image and 30 Hz QVGA for the depth image. The available options for image mode are:
 - 2: (640×480 VGA 30 Hz)
 - 5: (320×240 QVGA 30 Hz)
 - 8: (160×120 QQVGA 30 Hz)

And for the depth image mode:

- 2: (VGA 30 Hz)
- 5: (QVGA 30 Hz)
- 8: (QQVGA 30 Hz)

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.

3.6.4 NAVIGATION

Understanding which nodes you should run, why and what each of them does is the main focus of this chapter. To get our Pioneer 3-AT navigating, we are 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 navigation goal. Therefore, let us 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 analyzed 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 an interval, for which the default is 5 s.
- ***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.
- ***minimumScore***: its a threshold value for considering the outcome of the scan matching good. Here we set the parameter to 200, but you should test some values between 0 and 300 depending on your configuration. Keep in mind that scores go from about -3000 to more than 600.
- ***srr, srt, str and stt***: these parameters relate to the odometry errors. You should test your configuration and measure the four possible errors respectively: translation as a function of translation, translation as a function of rotation, rotation as a function of translation and rotation as a function of rotation. Here, `r` goes for *rho* (translation) and `t` goes for *theta* (rotation). The easiest way to do this is by sending goals to your robot and measuring the difference between the expected movement and the actual movement.

As to `move_base`, it bases its path planning techniques on the current location and the navigation 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:

- ***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. As you will see on the test section, we changed this parameter to use the `dwa_local_planner`, since it works better on our configuration.

- ***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 `rosparams`, 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, we begin looking at the `costmap_common_params` file, the one that contains parameters that apply for both the local and global costmaps:

```

1 obstacle_range: 5.0
2 raytrace_range: 6.0
3
4 max_obstacle_height: 1.0
5 min_obstacle_height: 0.05
6
7 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] ]
8 #robot_radius: 0.35
9 inflation_radius: 0.35
10 footprint_padding: 0
11
12 transform_tolerance: 1.0
13 map_type: costmap
14 cost_scaling_factor: 100
15
16
17 observation_sources: laser_scan_sensor
18 laser_scan_sensor: {sensor_frame: camera_link,
                      data_type: LaserScan, topic: scan, marking: true,
                      clearing: true}
19
20 #observation_sources: pointcloud_sensor
21 #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 us 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 do not use it here, so we set 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 behavior 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, it is very important to set the correct name of the subscribed topic, so the Navigation Stack takes the sensor data from the correct location. The marking and clearing values are self-explanatory, since they set if the sensor data from the observation source is allowed to mark and clear the costmap. This parameter raises yet some **very important** discussion:

- When selecting which observation source you are going to use, you have to consider that gmapping is using a LaserScan message as its observation source. If you are using a Kinect, you can choose the PointCloud message as your observation source here, and, although it will represent a lot more obstacles than the LaserScan would, that can bring a lot of problems to gmapping, that will struggle to get the current robot position.

Now that we have set all the costmap common parameters, we must set the parameter specific to the local and global costmaps. We will analyze them together, since

most of the parameters are very similar. First, take a look at the files. For the global costmap we have:

```

1 global_costmap:
2   global_frame: /map
3   robot_base_frame: base_link
4   update_frequency: 1.0
5   publish_frequency: 1.0 #0
6   static_map: false
7   width: 50 #3.4
8   height: 50 #3.4
9   origin_x: -25 #-1.20 is the actual position; -0.95 is
      the old one, for the frond wheel at the marker
10  origin_y: -25 #-1.91
11  resolution: 0.1

```

And for the local:

```

1 local_costmap:
2   global_frame: /odom
3   robot_base_frame: base_link
4   update_frequency: 5.0
5   publish_frequency: 10.0
6   static_map: false
7   rolling_window: true
8   width: 3.0
9   height: 3.0
10  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. They are set to different values because gmapping publishes the transform from odom to map, the global reference frame. If you set both of them to /odom you will be using the odometry data exclusively.
- ***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 is 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`. Take a look at the configuration for the Pioneer 3-AT:

```

1 TrajectoryPlannerROS:
2   max_vel_x: 0.5
3   min_vel_x: 0.1
4   max_rotational_vel: 0.5
5   max_vel_theta: 0.5
6   min_vel_theta: -0.5
7   min_in_place_rotational_vel: 0.5
8   min_in_place_vel_theta: 0.5
9   escape_vel: -0.1
10
11
12   acc_lim_th: 0.5
13   acc_lim_x: 0.5
14   acc_lim_y: 0.5
15
16   holonomic_robot: false

```

Again, we should analyze 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` [27] and `costmap_2d` [28] 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 is 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:

```

1 robot_base_frame: base_link
2
3 transform_tolerance: 1.0
4
5 robot_radius: 0.35
6
7 footprint: [ [0.3302, -0.0508], [0.254, -0.0508],
8             [0.254, -0.254], [-0.254, -0.254], [-0.254, 0.254],
9             [0.254, 0.254], [0.254, 0.0508], [0.3302, 0.0508] ]
10
11 inflater:
12   robot_radius: 0.35
13   inflation_radius: 0.35
14
15 obstacles:
16   observation_sources: pointcloud_sensor
17   pointcloud_sensor:
18     data_type: PointCloud2
19     topic: camera/depth/points
20     min_obstacle_height: 0.2
21     max_obstacle_height: 2.0
22     marking: true
23     clearing: true

```

```

22 z_voxels: 8
23 z_resolution: 0.25
24 max_obstacle_height: 2.0

```

As you can see in the file, the parameters do not 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 35 cm of radius, and, therefore, an inflation radius of 35 cm 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:

```

1 global_frame: map
2
3 robot_base_frame: base_link
4 update_frequency: 1.0
5 publish_frequency: 1.0
6 static_map: false
7 width: 50
8 height: 50
9 origin_x: -25
10 origin_y: -25
11 resolution: 0.1
12
13 plugins:
14 - {name: obstacles, type: "costmap_2d::VoxelLayer"}
15 - {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 $ rosrn 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 the Navigation 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, as an example, add the PointCloud 2 from the Kinect, as shown in Fig. 4.

As you can see in Fig. 4, 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;
- (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

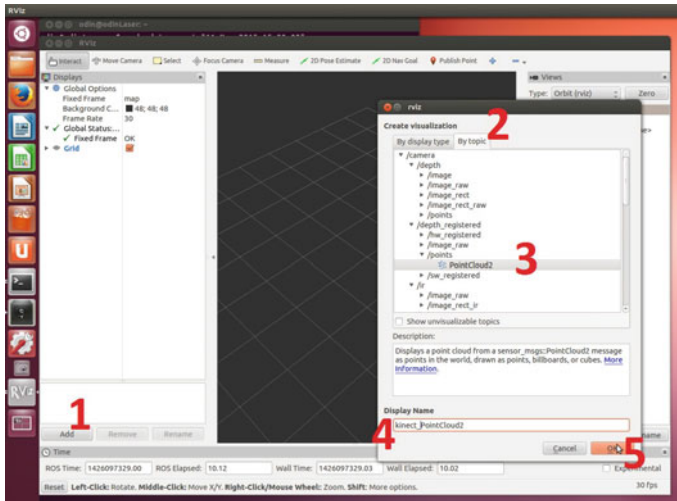


Fig. 4 Steps for adding new information for rviz to display

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 FOOT-PRINT	/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 POINT-CLOUD	/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*

Multiple Machines Communication is a rather complex topic and it is possible that you have to use more than one computer at the same time when navigating with a robot. Usually, in navigation scenarios, 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.

Although a ROS network can be very complex, if you are using just two machines as specified above, you will probably be able to communicate them with the following steps. If that is not your case, please refer to the Multiple Machines Communication ROS wiki page [29]. 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: 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

- 1 IPAddress Hostname
- 2 127.0.0.1 localhost
- 3 192.168.1.101 robot
- 4 192.168.1.100 masterpc

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:

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

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.

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 Figs. 5 and 6.

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. As for the costmaps, the yellow section indicates where the obstacle is, and is an infinite cost area. The radius around the obstacle, in our case almost entirely blue, is the inflation radius, where the cost is exponentially decreasing from the obstacle to the border of the inflation radius. Depending on your `inflation_radius` and `cost_scaling_factor` parameters, this coloring can be different.

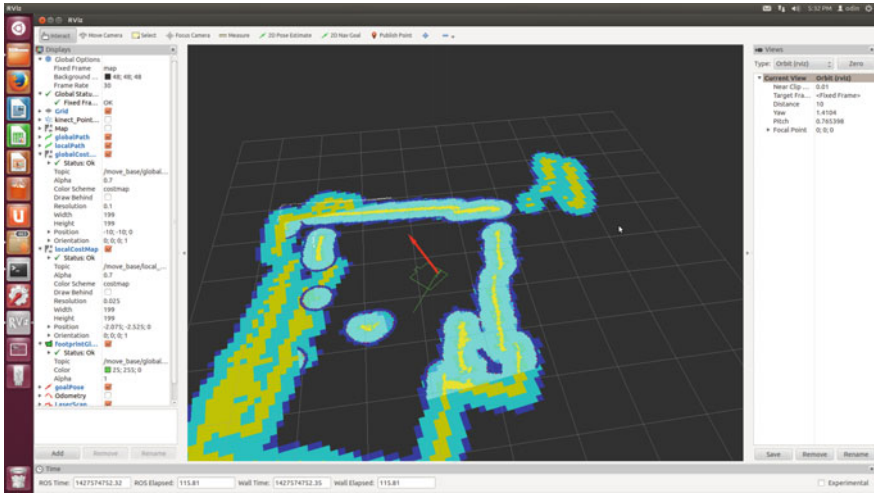


Fig. 5 Example costmaps for the Pioneer 3-AT navigating in a room

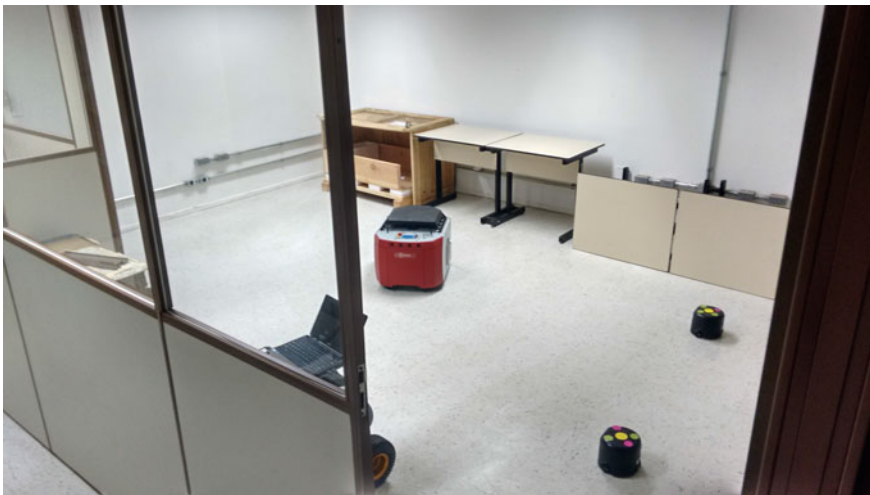


Fig. 6 Photograph of the robot navigating, at the same moment the print screen of Fig. 5 was taken

Conducting the tests we have found that the navigation does not work so great with the default planner. Therefore, we tested the `dwa_local_planner` and the results were way better. The `dwa_local_planner` is, in general, better for slow robots, and to use it you should modify your `move_base` launcher as follows:

```

1 <node pkg="move_base" type="move_base" respawn="false"
  name="move_base" output="screen" clear_params="true"
  >
2     <param name="controller_frequency" value="
      10.0" />
3     <param name="controller_patience" value="15.0
      " />
4     <param name="planner_frequency" value="2.0" /
      >
5     <param name="clearing_rotation_allowed" value
      ="false" />
6     <rosparam file="$(find course_p3at_navigation
      )/sg_costmap_common_params_p3at.yaml"
      command="load" ns="global_costmap" />
7     <rosparam file="$(find course_p3at_navigation)/
      sg_costmap_common_params_p3at.yaml" command="
      load" ns="local_costmap" />
8     <rosparam file="$(find course_p3at_navigation)/
      sg_local_costmap_params.yaml" command="load"
      />
9     <rosparam file="$(find course_p3at_navigation)/
      sg_global_costmap_params.yaml" command="load"
      />
10    <param name="base_local_planner" value="
      dwa_local_planner/DWAPlanerROS" />
11    <rosparam file="$(find course_p3at_navigation
      )/dwa_planner_ros.yaml" command="load" />
12 </node>

```

As you can see, a new param file is added, and this file is very similar to the base_local_planner parameters. You can find the original version of the file, taken from the scitos_2d_navigation github [30] on their URL or you can see our modified version here:

```

1 DWAPlanerROS:
2   acc_lim_x: 1.0
3   acc_lim_y: 0.0
4
5   acc_lim_th: 2.0
6
7   min_vel_x: -0.55
8   min_vel_y: 0.0
9   max_vel_y: 0.0
10  max_rot_vel: 1.0
11  min_rot_vel: 0.4

```

```
12
13 yaw_goal_tolerance: 0.1
14
15 xy_goal_tolerance: 0.3
16
17 latch_xy_goal_tolerance: true
18
19 sim_time: 1.7
20
21 path_distance_bias: 5.0
22 goal_distance_bias: 9.0
23 occdist_scale: 0.01
24
25 oscillation_reset_dist: 0.05
26
27 prune_plan: true
28
29 holonomic_robot: false
```

If you are having some troubles, you may verify the `view_frames` functionality of the `tf` ROS package. It is very useful for spotting problems on your transformation tree and it also helps to learn better some of the concepts about transformations that we have presented. To generate a “frames.pdf” file containing the current `tf` tree, insert the following lines on the shell:

```
1 $ rosrun tf view_frames
```

As a result, we got the `tf` tree seen on Fig. 7, where you can clearly see the hierarchy between the frames and understand a little better the relation between them. Each transform on the `tf` tree (represented by the arrows) has some related data, such as the average rate and the node that is broadcasting the transform. If you don't have a tree structure like that, where some frame is not linked to the tree or the map, `odom` and `base_link` are not presented on this order, please check your `tf` configuration. Please note that this is not the only possible configuration, as you can have `base_footprint` when you are using `hector_mapping`, for example. Another thing that you may have noticed is that we do not have only one frame for the Kinect: the `openni_launcher` node publishes 4 transformations between 5 distinct frames, and the only transformation that we have manually launched, in this case, is the transformation between the `base_link` and the `camera_link`.

After seeing the system was working as expected, we tried to do some more significant tests. First we arranged our test room in two ways: a single wood object on the middle of the room and the robot at one side; a corridor formed by some objects. You can see the two scenarios and the results of the navigation in `rviz` in Figs. 8 and 9.

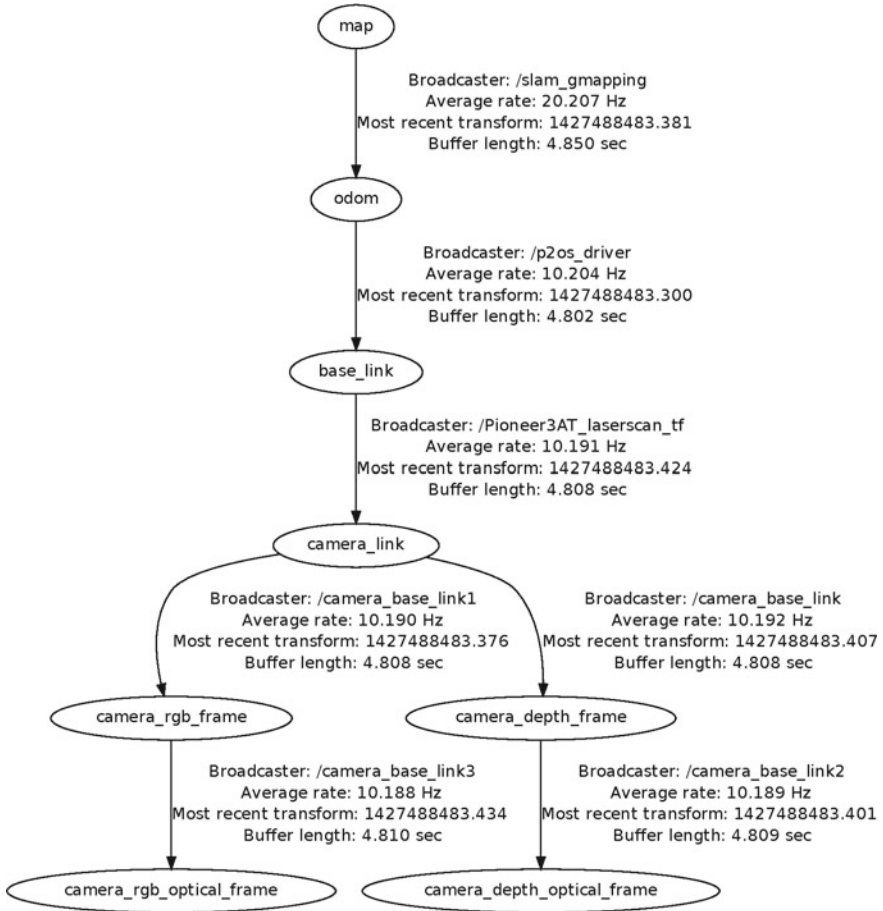


Fig. 7 tf Tree generated by the view_frames command

To do these experiments, a initial and final position on the room were chosen for each one. After that the distance and the difference in angle were measured, and then we published the goal with these values. We have done that with a simple node, very similar to the node you can find here [31]. You can also publish a goal by issuing the following command (**CHANGE THE VALUES TO SUIT YOUR NEEDS**):

```
1 rostopic pub /move_base_simple/goal geometry_msgs/PoseStamped 'header: frame_id: "/base_link", pose: position: x: 0.2, y: 0, orientation: x: 0, y: 0, z: 0, w: 1 '
```

It is possible to see on Fig. 8 that the robot would get to the goal by following a straight line if no object was present, but it successfully avoids the obstacle and gets to the exact point it intended to. It is also possible to see on the images that it doesn't just avoid the obstacle, but a region around it, the inflation radius, since there is a low

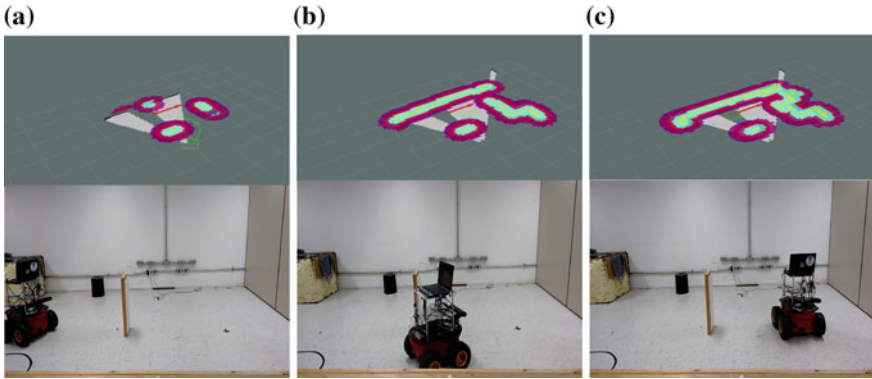


Fig. 8 **a** Robot moving right after it receives a goal. **b** Robot avoiding the obstacle. **c** Robot getting to the goal

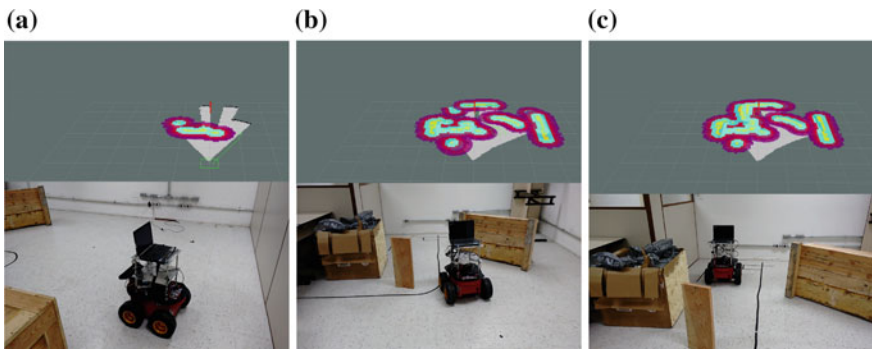


Fig. 9 **a** Robot moving right after it receives a goal. **b** Robot entering the corridor. **c** Robot getting to the goal

cost area right around of it. When it gets to the goal, it rotates until it gets the right angle. In the way, you can see that the local and global CostMaps do not perfectly match and start to drift, but the gmapping transformation is able to adjust that with very little errors.

On Fig. 9 the task of the robot is harder: it has to avoid some obstacles and get to a point that is after a corridor. At first, it tries to avoid the obstacles by going around them, but it sees the wall. After that, it decides to enter the inflation radius, a high cost area, but trying to stay as close to the border as possible. At the end, it successfully gets to the goal and adjusts itself to the right direction.

In `rviz` it is possible to see some details about the navigation:

- Obstacles on the gmapping map and on the CostMaps should overlap each other perfectly, and that is almost the case in most of the situations;
- Obstacles on local and global CostMap should also overlap each other, and gmapping does a good job in correcting the odometry errors to get that;

- The inflation radius is avoided by the robot, that usually translates right at its border to get to the goal. This behavior can be different depending on the cost factors you chose at the planner parameters;
- The global path is not perfectly followed by the local path in almost every situation, because the parameters to calculate the smallest cost of them are usually different. The frequency of update and cost parameters should alter this behavior as needed.

4.3.1 Conclusion

Navigating through unknown environments is a very hard task, however we had great results doing it while using the Navigation Stack. This set of tools helped us to achieve great results, allowing the robot to create a dynamically generated map and achieve goals without crashing. We hope that this tutorial chapter, along with some background reading of the references we have presented, is enough for you to also get your robot navigating.

References

1. I.R.S.D. Siegwart, *Introduction to Autonomous Mobile Robots*, Intelligent Robotics and Autonomous Agent series (The MIT Press, Cambridge, 2011)
2. ROS.org, Ros package geometry_msgs (2015), http://wiki.ros.org/geometry_msgs. Accessed 03 Jan 2015
3. R.P. Goebel, Ros by example, vol. 1 (2014), <http://www.lulu.com/shop/r-patrick-goebel/ros-by-example-hydro-volume-1/ebook/product-21393108.html>. Accessed 13 June 2015
4. R.P. Goebel, Ros metapackage rbx1 (2014), <https://github.com/pirobot/rbx1/tree/hydro-devel>. Accessed 13 June 2015
5. ROS.org, Ros package openni_camera (2015), http://wiki.ros.org/openni_camera Accessed 03 Jan 2015
6. ROS.org, Ros package openni_launch (2015), http://wiki.ros.org/openni_launch. Accessed 03 Jan 2015
7. AnthonyJ350, How to solder / soldering basics tutorial (2011), <https://www.youtube.com/watch?v=BxASFu19bLU>. Accessed 03 Jan 2015
8. mjlorton, Tutorial: How to crimp connectors, strip wire and use heat shrink. (2013), <https://www.youtube.com/watch?v=kjSGCSwNuAg>. Accessed 03 Jan 2015
9. E.C. Wireconnector, E-clamp quick wire connector (2013). <https://www.youtube.com/watch?v=GI8IRSQQbJk>. Accessed 03 Jan 2015
10. ROS.org, Adding a kinect to an irobot create/roomba (2015), <http://wiki.ros.org/kinect/Tutorials/Adding>. Accessed 03 Jan 2015
11. ROS.org, Ros package depthimage_laserscan (2015), http://wiki.ros.org/depthimage_to_laserscan. Accessed 03 Jan 2015
12. S.F.A.S D.G. Ballardini, A.L. Fontana, ira_tools: a ros laserscan manipulation toolbox (2014), [arXiv:1411.1086](https://arxiv.org/abs/1411.1086). Accessed 12 Jan 2015
13. S.F.A.S.D.G. Ballardini, A.L. Fontana, Ros package ira_tools (2014), https://github.com/iralabdisco/ira_laser_tools. Accessed 12 Jan 2015
14. ROS.org, Ros package kinect_aux (2015), http://wiki.ros.org/kinect_aux. Accessed 03 Jan 2015

15. ROS.org, Ros package `openni_tracker` (2015), http://wiki.ros.org/openni_tracker. Accessed 03 Jan 2015
16. O.T.S. for 3D Sensing, Openni sdk history (2014), <http://www.openni.ru/openni-sdk/openni-sdk-history-2/index.html>. Accessed 12 June 2015
17. R.L. Guimaraes, Complementary files for using more kinect functions (2015), https://github.com/rosnavigation/KINECT_FILES. Accessed 12 June 2015
18. ROS.org, Ros package `cyphy_mapping` (2015), http://wiki.ros.org/cyphy_people_mapping. Accessed 03 Jan 2015
19. STRANDS, Ros package `sicks300` (2014), <https://github.com/strands-project-releases/sicks300/tree/release/hydro/sicks300>. Accessed 03 Jan 2015
20. ROS.org, How to set up `hector_slam` for your robot (2015), http://wiki.ros.org/hector_slam/Tutorials/SettingUpForYourRobot. Accessed 03 Jan 2015
21. W. Meeussen, Coordinate frames for mobile platforms (2015), <http://www.ros.org/repos/rep-0105.html>. Accessed 03 Jan 2015
22. B. Sciacivco, L. Siciliano, *Modelling and Control of Robot Manipulators*, Advanced Textbooks in Control and Signal Processing (CreateSpace Independent Publishing Platform, Charleston, 2000)
23. ROS.org, Navigation tutorials: Robotsetup (2015), <http://wiki.ros.org/navigation/Tutorials/RobotSetup>. Accessed 03 Jan 2015
24. W.F.D. Thrun, S. Buggard, *Probabilistic Robotics*, Intelligent Robotics and Autonomous Agent Series (MIT Press, Cambridge, 2005)
25. ROS.org, Ros package `p2os_driver` (2015), http://wiki.ros.org/p2os_driver. Accessed 03 Jan 2015
26. ROS.org, Ros package `rostopic` (2015), <http://wiki.ros.org/rostopic>. Accessed 03 Jan 2015
27. ROS.org, Ros package `base_planner` (2015), http://wiki.ros.org/base_local_planner. Accessed 03 Jan 2015
28. ROS.org, Ros package `costmap_2d/layered` (2015), http://wiki.ros.org/costmap_2d/layered. Accessed 03 Jan 2015
29. ROS.org, Running ros across multiple machines (2015), <http://wiki.ros.org/ROS/Tutorials/MultipleMachines>. Accessed 19 June 2015
30. N. Bore, Ros `scitos_navigation` package github (2013), https://github.com/nilsbore/scitos_2d_navigation. Accessed 20 June 2015
31. R. Yehoshua, `Sendgoals.cpp` (2013), <http://u.cs.biu.ac.il/~yehoshr1/89-685/Fall2013/demos/lesson7/SendGoals.cpp>. Accessed 20 June 2015