

# Special Project: Husky Robot Navigation with ROS



Baylor University  
Spring 2020  
ELC-4V97  
Diego Alvaro



# Special Project:

# Husky Robot Navigation with ROS

ELC-4V97, Spring 2020

Baylor University

Diego Alvaro<sup>1</sup> and Dr. Scott Koziol<sup>2</sup>

**Abstract** – Robotics presence has been noticed more and more each day. Robotics quality and effectiveness has increased exponentially in the last decades. Nowadays, one of the most common and useful kind of robot is mobile robots due to the wide range of tasks they are able to develop. Moreover, autonomous mobile robots have been the main topic of research in the recent years. Some of their applications can be shown in recovering disaster efforts, research fields and even service robots.

This project aims to show and explain how to approach one of the main issues in autonomous robotics, robot navigation in an unknown environment. To accomplish that, this project will present a simulation of a Husky robot navigating through an environment where there is no prior acknowledge about it. To face this problem, the technique of SLAM will be used for localization and mapping. This simulation will be done by using ROS, currently one of the most popular tool to code robot software. Specifically, it will be the Indigo version.

In this report, there will be summarized a brief introduction to the main topics such as the Husky robot, ROS, navigation and SLAM. Then, the software used will be explained part by part. To continue, an implementation of the software and an explanation about how to execute it will be shown. Lastly, the conclusion, learnings and possible next steps will be presented at the end of this report.

Note: All the software cited in this report can be found in the Appendix section.

**Keywords** - ROS, navigation, Husky, mobile robot, Gazebo, RViz, SLAM

---

<sup>1</sup> Undergraduate Student, Industrial Technologies Engineering Major, Baylor University

<sup>2</sup> Professor, Department of Electrical & Computer Engineering, Baylor University

## ***Table of Contents***

I.	INTRODUCTION .....	4
I. I.	HUSKY ROBOT.....	4
I. II.	WHY ROS?.....	6
I. III.	GUIDANCE, NAVIGATION, AND CONTROL SYSTEMS.....	7
I. IV.	SLAM.....	8
II.	UNDERSTANDING ROS.....	9
III.	PACKAGES & NODES FOR NAVIGATION.....	12
III. I.	GAZEBO .....	12
III. II.	RVIZ.....	13
III. III.	NAVIGATION .....	15
III. III. I.	move_base.....	15
III. III. II.	gmapping .....	19
IV.	SIMULATION.....	23
V.	CONCLUSION.....	30
V. I.	WHAT'S NEXT? .....	31
	BIBLIOGRAPHY .....	32
	APPENDIX .....	32

## ***Figures & Tables***

Table 1. Husky Tech. Spec. ....	5
Figure 2 – Basic Launch Structure .....	10
Table 2. Main arguments in husky_gazebo package .....	13
Table 3. Main topics in husky_viz package .....	14
Table 4. Parameters of move_base .....	19
Table 5. Parameters of gmapping .....	22
Figure 3 – Commands to execute the required launch .....	23
Figure 4 – Husky and world launch in the Gazebo environment.....	24
Figure 5 – Husky launch in the RViz visualizer .....	24
Figure 6 – Husky and map launch by gmapping in Rviz .....	25
Figure 7 – Example of a goal and the global plan in Rviz .....	25
Figure 8 – Zoom of Figure 7 to see the local plan in red .....	26
Figure 9 – More examples of a goal and the global plan in Rviz .....	26
Figure 10 – Example of selecting a new goal in the middle of a path to another goal .....	27
Figure 11 – Example of how a small rotation and how it affects mapping .....	27
Figure 12 – Example of the Husky avoiding an obstacle .....	28
Figure 13 – Example of Gazebo and RViz at the same moment.....	28

## ***I. Introduction***

In the last decade, the number of robots and its quality have improved exponentially, in some way because of the used of those robots which make the work easier and faster. Nowadays, it is hard to think about one place in our daily life where robotics cannot be found. This happens due to the lots of advantages they have, such as helping us and making our life simpler. One of the biggest advantage they have is its effectiveness and capacity of doing things that humans cannot do.

Specially, mobile robots, which are the ones that are capable to move around in its environment and is not fixed to one physical location, are really useful in a lot of different fields such as research, manufacturing, transportation and many more. Mobile robots are now regularly used in many applications. One prominent application is aiding disaster recovery efforts in mines and after earthquakes. Military uses, such as for roadside bomb detection, form another broad category. Recently, products have been developed for consumer applications, such as the Roomba®. Finally, wheeled mobile robots are exploring Mars and are poised to return to the moon [1]. In this report, we are going to focus on the one called *Husky*, a four-wheel ground mobile robot.

The big goal of this project is to learn about robot Guidance, Navigation and Control systems. To accomplish that, this report shows the work done to simulate the Husky robot navigating to a series of waypoints in the *Gazebo* simulation environment using ROS Indigo. Moreover, the *RViz* visualizer will be used to see sensor data from the robot, and give it commands. In order to keep track of the localization and mapping of the robot and its environment, the technique of simultaneous localization and mapping (SLAM) will be used.

### ***I. I. Husky Robot***

Husky is a medium-sized robotic development platform. It is an unmanned ground vehicle (UGV) designed for robotics research in harsh outdoor environments. Its large payload capacity and power systems accommodate an extensive variety of payloads, customized to meet research need. Some of the features that can be added to the UGV are stereo cameras, GPS, LIDAR, IMUs and many more [2]. The first model was created in 2011 by *Clearpath*

*Robotics*, one of the most well-known Canadian robotics startups, and one of the “Top 50 Most Influential Companies in Robotics” according to *Robotics Business Review* [3].



Figure 1 – Husky Robot. Source: <https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/>

The choice to use Husky for this project is based on its efficiency to develop a wide range of different types researches in a relatively simple and not expensive way. Moreover, this robot is designed to change or improve its features quickly and easily. This flexible customization makes uncomplicated to face unexpected problems and adapt to changes saving a lot of time and head-breaks. Also, another reason for using Husky is its ease to be used. Husky is fully supported in ROS and uses an open source serial protocol. This is very helpful to get started producing research results faster by using existing researches and the growing knowledge base in the thriving ROS community [2].

In the next table, the main technical specifications of Husky robot are summarized.

Technical Specifications	
External Dimensions (L x W x H)	39 x 26.4 x 14.6 in
Internal Dimensions	11.7 x 16.2 x 6.1 in
Weight	110 lbs
Max Payload	165 lbs
Max Speed	2.2 mph
Run Time	3 hours
User Power	5V, 12V and 24V fused at 5V each

Table 1. Husky Tech. Spec.

## ***I. II. Why ROS?***

Robot Operating System, or ROS, is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms [4]. ROS was originally developed in 2007 by the Stanford Artificial Intelligence Laboratory (SAIL) with the support of the Stanford AI Robot project [5]. Nowadays, it is one of the most popular software platforms in the robotics community. The main reason of this, and what it really makes the difference, is that it makes the work easier saving time and effort. ROS allows to write a piece of software for a specific robot, but changing little parameters of the code can work for other robots. So the ROS community has shared a widespread number of packages that can be reused in new and different projects.

Following are some of the advantages of software that uses ROS [6]:

- ◆ **Distributed Computation.** Many modern robot systems rely on software that spans many processes and runs across several different computers. Some of them carry multiple computers for sensors or actuators. Also, when multiple robots work together, they usually need to communicate with one another. Another case that applies that is that operator can send commands to the robot from different devices such as a laptop or mobile phone.
- ◆ **Rapid Testing.** ROS provides a simple way to record and play back sensor data and other kinds of messages. In addition, well-designed ROS systems separate low-level direct control of the hardware and high level processing and decision making into separate programs. Making possible to work independently with either one. Thanks of that, testing is not as time consuming as usual. Also, this allows to work without physical robots which are not always available.
- ◆ **Software reuse.** As I said, developers can focus more time on experimenting with new ideas instead of spending that time on developing algorithms. This ability to be reused are based on two important ways. First, ROS's standard packages provide stable, debugged implementations of many important robotics algorithms. Secondly, ROS's message passing interfaces to both the

latest hardware and to implementations of cutting edge algorithms are available.

### ***I. III. Guidance, Navigation, and Control systems***

In order to be able to go from a set point to another point, a mobile robot should develop different operations to accomplish that successfully. The main competences required for a mobile robot are guidance, navigation and control. One single mistake in any of them could lead into an inefficient functioning and not to reach the purpose we want to achieve using our robot.

#### ◆ **Guidance.**

A guidance system is a device or group of devices used to navigate a ship, aircraft, missile, rocket, satellite, or other vehicle. It is in charge of controlling the craft's course. Typically, this refers to a system that navigates without direct or continuous human control. In this project, this part consists in choosing one unoccupied grid square which would be the robot goal destination.

#### ◆ **Navigation.**

Navigation is one of the most challenging competences required for a mobile robot. Success in navigation requires success at the four building blocks of navigation: *perception* (the robot must interpret its sensors to extract meaningful data), *localization* (the robot must determine its position in the environment), *cognition* (the robot must decide how to act to achieve its goals) and *motion control* (the robot must modulate its motor outputs to achieve the desired trajectory) [7]. In our project, planer laser sensors are going to be used as actuators which will provide the perception data. Then, the technique of SLAM is going to be used for localization as well as mapping.

#### ◆ **Control.**

Control is the key to create a system to behave in the desired manner. Control in the process of making a system variable adhere to a particular value, called reference value, by controlling the output. A control system is a set of



mechanical or electronic devices that regulates other devices or systems by way of control loops.

## ***I. IV. SLAM***

As I said, SLAM stands for Simultaneously Localization and Mapping. SLAM is a technique used in robotics to address how a body can navigate in a previously unknown environment while constantly building and updating a map of its workspace using feeds from on board sensors only. SLAM is regularly used when a robot needs to be truly autonomous, when there is no human input, when there is no prior knowledge about the environment, or when there is no GPS available. So, the problem of SLAM is twofold. An unbiased map is needed to know where the robot is localized, but at the same time, an accurate pose estimate is needed in order to build this map of environment. Since in SLAM, pure localization cannot be used neither known poses for mapping, that is why SLAM is one of the greatest challenges in probabilistic robotics.

The SLAM process consists of a number of steps. The goal of the process is to use the environment to update the position of the robot. Since the odometry of the robot (which gives the robots position) is often erroneous we cannot rely directly on the odometry. Lasers scans of the environment are used to correct the position of the robot. This is accomplished by extracting features from the environment and re-observing when the robot moves around. An *Extended Kalman Filter* (EKF) is the heart of the SLAM process. It is responsible for updating where the robot thinks it is based on these features. These features are commonly called *landmarks*. The EKF keeps track of an estimate of the uncertainty in the robot's position and also the uncertainty in these landmarks it has seen in the environment. When the odometry changes because the robot moves the uncertainty pertaining to the robot's new position is updated in the EKF using odometry update. Landmarks are then extracted from the environment from the robot's new position. The robot then attempts to associate these landmarks to observations of landmarks it previously has seen. Re-observed landmarks are then used to update the robot's position in the EKF. Landmarks which have not previously been seen are added to the EKF as new observations so they can be re-observed later. It should be noted that at any point in these steps the EKF will have an estimate of the robot's current position [8].

## II. Understanding ROS

Before getting to the point, it will be useful to understand some of the basic ideas and concepts that ROS uses. First, all ROS software is organized into *packages*. A ROS package is a coherent collection of files, generally including both executables and supporting files, that serves a specific purpose. Each package is defined by a manifest, which is a file called *package.xml*. This file defines some details about the package, including its name, version, maintainer, and dependencies. The directory containing *package.xml* is called the package directory. (In fact, this is the definition of a ROS package: Any directory that ROS can find that contains a file named *package.xml* is a package directory.) This directory stores most of the package's files. Thus, this is the basic idea about files and how they are organized into packages. Now, let's explain how to execute some ROS software.

One of the basic goals of ROS is to enable roboticists to design software as a collection of small, mostly independent programs called *nodes* that all run at the same time. For this to work, those nodes must be able to communicate with one another. The part of ROS that facilitates this communication is called the *ROS master*. To start the master, this command has to be used:

➤ *roscore*

The master should be allowed to continue running for the entire time that ROS is being used. One reasonable workflow is to start *roscore* in one terminal, then open other terminals for your “real” work. Once the master has been started, programs that use ROS can be run. A running instance of a ROS program is called a *node*. The basic command to create a node (also known as “running a ROS program”) is *roslaunch*. There are two required parameters to *roslaunch*. The first parameter is a package name. The second parameter is simply the name of an executable file within that package.

➤ *roslaunch package-name executable-name*

However, having to start multiple nodes can be annoying and hardworking. In order to work with more complex packages and goals, there is a type of file that allows the operator to start the master and a lot of different nodes all at once. This file is called *launch file*. The

use of launch files is widespread through many ROS packages. To execute a launch file, use the *roslaunch* command:

➤ *roslaunch package-name launch-file-name*

Before starting any nodes, *roslaunch* will determine whether *roscore* is already running and, if not, start it automatically. An important fact about *roslaunch* is that all of the nodes in a launch file are started at roughly the same time. As a result, the operator cannot be sure about the order in which the nodes will initialize themselves. Well-written ROS nodes do not care about the order in which they and their “siblings” start up. The basic idea of launch files is to list, in a specific XML format, a group of nodes that should be started at the same time. Launch files are XML documents, and every XML document must have exactly one root element. For ROS launch files, the root element is defined by a pair of launch tags (<launch>...</launch>). All of the other elements of each launch file should be enclosed between these tags. The heart of any launch file is a collection of node elements, each of which names a single node to launch. The simplest launch file would look like this:

```
<Launch>

  <node
    pkg="package-name"
    type="executable-name"
    name="node-name"
  />

</Launch>
```

Figure 2 – Basic Launch Structure

The *pkg* and *type* attributes identify which program ROS should run to start this node. These are the same as the two command line arguments to *roslaunch*, specifying the package name and the executable name, respectively. The *name* attribute assigns a name to the node. This overrides any name that the node would normally assign to itself.

To end this section, let’s explain how ROS nodes communicate. The primary mechanism that ROS nodes use to communicate is to send *messages*. Messages in ROS are organized into named *topics*. The idea is that a node that wants to share information will *publish* messages on the appropriate topic or topics; a node that wants to receive information will *subscribe* to the topic or topics that it’s interested in. The ROS master takes care of

ensuring that *publishers* and *subscribers* can find each other; the messages themselves are sent directly from publisher to subscriber. One really useful tool of ROS is a command used to see graphically and visualize more clearly the publish-subscribe relationships between ROS nodes.

➤ *rqt\_graph*

In this name, the “r” is for ROS, and the “qt” refers to the Qt GUI toolkit used to implement the program.

### **III. Packages & Nodes for Navigation**

In this section, the different package files as well as their nodes used in this project will be explained. Four different packages are going to be used. Although, only three of them are going to be called directly in the simulation. The other file is going to be called indirectly during these main files calling. First, one of the package will be display the Gazebo environment and the Husky. Gazebo is an open-source 3D high-fidelity robot simulator. Gazebo allows to define the characteristics of both the robot and the world, and interact with the robot via ROS in the same way the operator would interact with the real thing. Secondly, a package is used to display the RViz visualizer. RViz is a powerful 3D visualization tool for ROS. It is used to view wide variety of information, in this case, it is used to view the laser sensor data and our robot. Lastly, a package is used to compute the SLAM technique this file is called *husky\_navigation*. This package is based on the use of a planner laser sensor for SLAM. In addition, another package is required *husky\_description* provided by Clearpath Robotics which is the one that has the information needed to represent the Husky robot (URDF). This is the package that is going to be called indirectly.

#### **III. I. Gazebo**

First, a “world” is needed to be defined. This world is the environment where we are going to display our robot. There are a countless number on the internet that are open source, due to in this project creating a world is not the real purpose, we are going to choose one of them. The one we are going to choose is “koridor3.world” from Instabul Technical University.

This package *husky\_gazebo* contains two files “launch” and “worlds”. Inside worlds, it is the koridor3.world cited before. On the other one, it contains four different launch files: *husky\_empty\_world.launch*, *spawn\_husky.launch*, *koridor3.launch* and *husky\_koridor3.launch*. Basically, *husky\_koridor3.launch* is the main launch. It calls *koridor3.launch* and *spawn\_husky.launch*. This last launch uses the description package and display our robot in the Gazebo environment. *Koridor3.launch* is the one responsible of display the environment, the world cited before, in Gazebo. To accomplish that call the first launch cited which is the most basic launch that makes possible to use Gazebo. The most important arguments in these launches are shown next:

Arguments	
<b>paused</b>	Start Gazebo in a paused state (default false)
<b>used_sim_time</b>	Tells ROS nodes asking for time to get the Gazebo-published simulation time, published over the ROS topic /clock (default true)
<b>gui</b>	Launch the user interface window of Gazebo (default true)
<b>headless</b>	Enable gazebo state log recording
<b>debug</b>	Start gzserver (Gazebo Server) in debug mode using gdb (default false)
<b>verbose</b>	Run gzserver and gzclient with --verbose, printing errors and warnings to the terminal (default false)

Table 2. Main arguments in husky\_gazebo package

### III. II. RViz

The **husky\_viz** package contains two files. One is the launch file that will be used in the simulation. This launch file just call the node rviz with the robot.rviz in it. This file is the one that is in the other file called “rviz” of this package. Its main function is to display the rviz visualizer with all the required function and tools that we will need such as the robot itself, its laser sensor or different tools to set the navigation goals. This package is also provided by Clearpath Robotics, since the explanation of what these files contains is not the point in this project, they are not going to be explained. However, I would like to explain the principal topics used for navigation by RViz due to they are really interesting in order to understand how it is able to communicate its data and itself to another package like the navigation package. Hence, every topic listed next is very valuable at checking the navigation functionalities at some point.

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

Name	Topic	Message Type
Inflated Obstacles Layer	/local_costmap/inflacted_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

Table 3. Main topics in husky\_viz package

To understand better all of these topics, let's briefly explain each of them [9].

- Robot Footprint: These message is the displayed polygon that represents the footprint of the robot. Here the footprint is being taken 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 a layered approach is not being used, the local\_costmap in its whole will be displayed in this topic.
- Obstacles Layer: One of the main layers when a layered costmap is being used, containing the detected obstacles.
- Inflated Obstacles Layer: One of the main layers when a layered costmap is being used, 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. In order to see the goal that the robot is currently trying to achieve the `/move_base/current_goal` topic should be used.
- Planner Plan: Contains the complete global plan.
- Laser Scan: Contains the `laser_scan` data. Depending on the configuration this topic can be a real reading from the laser sensor or it can be a converted value from another type of sensor.

### **III. III. Navigation**

This last package is consisted in three different files. As usual, a launch file, then a *maps* file where when the robot maps the environment and the operator save the map, the map will be saved there, and finally a file called *config*. The main file inside the launch file is *gmapping\_demo.launch*. This launch file runs another two launch files, *gmapping.launch* which is the one in charge of carrying out the SLAM technique and *move\_base.launch* which is the one in charge of, given a goal in the world, making the robot attempt to reach it with a mobile base.

#### **III. III. I. move\_base**

The `move_base` launch provides an implementation of an action that, given a goal in the world, will attempt to reach it with a mobile base. The `move_base` node links together a global and local planner to accomplish its global navigation task. It supports any global planner adhering to the `nav_core::BaseGlobalPlanner` interface and any local planner adhering to the `nav_core::BaseLocalPlanner` interface. The `move_base` node also maintains two costmaps, one for the global planner, and one for a local planner that are used to accomplish navigation tasks [10]. To understand a bit more clearly, let's take a look of the different topics it uses and



its services. They can be divided in four groups: action subscribed topics, action published topics, subscribed topic and published topics.

For the action topics, the `move_base` node provides an implementation of the `SimpleActionServer`, that takes in goals containing `geometry_msgs/PoseStamped` messages. You can communicate with the `move_base` node over ROS directly, but the recommended way to send goals to `move_base` if you care about tracking their status is by using the `SimpleActionClient`.

➤ Action Subscribed Topics

- `move_base/goal` (`move_base_msgs/MoveBaseActionGoal`): A goal for `move_base` to pursue in the world.
- `move_base/cancel` (`actionlib_msgs/GoalID`): A request to cancel a specific goal.

➤ Action Published Topics

- `move_base/feedback` (`move_base_msgs/MoveBaseActionFeedback`): Feedback contains the current position of the base in the world.
- `move_base/status` (`move_base_msgs/GoalsStatusArray`): Provides status information on the goals that are sent to the `move_base` action.
- `move_base/result` (`move_base_msgs/MoveBaseActionResult`): Result is empty for the `move_base` action.

➤ Subscribed Topics

- `move_base_simple/goal` (`geometry_msgs/PoseStamped`): Provides a non-action interface to `move_base` for users that do not care about tracking the execution status of their goals.

➤ Published Topics

- `cmd_vel` (`geometry_msgs/Twist`): A stream of velocity commands meant for execution by a mobile base.

➤ Services

- `~make_plan (nav_msgs/GetPlan)`: Allows an external user to ask for a plan to a given pose from `move_base` without causing `move_base` to execute that plan.
- `~clear_unknown_space (std_srvs/Empty)`: Allows an external user to tell `move_base` to clear unknown space in the area directly around the robot. This is useful when `move_base` has its costmaps stopped for a long period of time and then started again in a new location in the environment.
- `~clear_costmaps (std_srvs/Empty)`: Allows an external user to tell `move_base` to clear obstacles in the costmaps used by `move_base`. This could cause a robot to hit things and should be used with caution

Lastly, a briefly explanation about the parameters used in this launch file is shown next. The *config* file cited before contains all the information and values needed to set up the costmap.

Parameters	
<code>~base_global_planner (string, default: "navfn/NavfnROS")</code>	The name of the plugin for the global planner to use with <code>move_base</code> . This plugin must adhere to the <code>nav_core::BaseGlobalPlanner</code> interface.
<code>~base_local_planner (string, default: "base_local_planner/TrajectoryPlannerROS")</code>	The name of the plugin for the local planner to use with <code>move_base</code> . This plugin must adhere to the <code>nav_core::BaseLocalPlanner</code> interface.
<code>~recovery_behaviors (list, default: [{name: conservative_reset, type: clear_costmap_recovery/ClearCostmapRecovery}, {name: rotate_recovery, type: rotate_recovery/RotateRecovery}, {name: aggressive_reset, type: clear_costmap_recovery/ClearCostmapRecovery}])</code>	A list of recovery behavior plugins to use with <code>move_base</code> . These behaviors will be run when <code>move_base</code> fails to find a valid plan in the order that they are specified. After each behavior completes, <code>move_base</code> will attempt to make a plan. If planning is successful, <code>move_base</code> will continue normal operation. Otherwise, the next recovery behavior in the list will be executed. These plugins must adhere to the <code>nav_core::RecoveryBehavior</code> interface.

Parameters	
<b>~controller_frequency (double, default: 20.0)</b>	The rate in Hz at which to run the control loop and send velocity commands to the base.
<b>~planner_patience (double, default: 5.0)</b>	How long the planner will wait in seconds in an attempt to find a valid plan before space-clearing operations are performed.
<b>~controller_patience (double, default: 15.0)</b>	How long the controller will wait in seconds without receiving a valid control before space-clearing operations are performed.
<b>~conservative_reset_dist (double, default: 3.0)</b>	The distance away from the robot in meters beyond which obstacles will be cleared from the costmap when attempting to clear space in the map. This parameter is only used when the default recovery behaviors are used for move_base.
<b>~recovery_behavior_enabled (bool, default: true)</b>	Whether or not to enable the move_base recovery behaviors to attempt to clear out space.
<b>~clearing_rotation_allowed (bool, default: true)</b>	Determines whether or not the robot will attempt an in-place rotation when attempting to clear out space. This parameter is only used when the default recovery behaviors are in use.
<b>~shutdown_costmaps (bool, default: false)</b>	Determines whether or not to shut down the costmaps of the node when move_base is in an inactive state.
<b>~oscillation_timeout (double, default: 0.0)</b>	How long in seconds to allow for oscillation before executing recovery behaviors. A value of 0.0 corresponds to an infinite timeout.
<b>~oscillation_distance (double, default: 0.5)</b>	How far in meters the robot must move to be considered not to be oscillating. Moving this far resets the timer counting up to the oscillation_timeout
<b>~planner_frequency (double, default: 0.0)</b>	The rate in Hz at which to run the global planning loop. If the frequency is set to 0.0, the global planner will only run when a new goal is received or the local planner reports that its path is blocked.

Parameters	
<code>~max_planning_retries (int32_t, default: -1)</code>	How many times to allow for planning retries before executing recovery behaviors. A value of -1.0 corresponds to an infinite retries.

Table 4. Parameters of `move_base`

### III. III. II. *gmapping*

This launch contains a ROS wrapper for OpenSlam's Gmapping. The gmapping launch provides laser-based SLAM. This launch file runs a node called `slam_gmapping`. Using `slam_gmapping`, a 2-D occupancy grid map (like a building floorplan) can be created from laser and pose data collected by a mobile robot [11]. To use `slam_gmapping`, it is needed a mobile robot that provides odometry data and is equipped with a horizontally-mounted, fixed, laser range-finder. The `slam_gmapping` node will attempt to transform each incoming scan into the `odom` (odometry) `tf` frame. The `tf` transforms used in this node are shown next:

- `<the frame attached to incoming scans> → base_link`  
usually a fixed value, broadcast periodically by a robot state publisher, or a tf static transform publisher.
- `base_link → odom`  
usually provided by the odometry system (e.g., the driver for the mobile base)
- `map → odom`  
the current estimate of the robot's pose within the map frame.

To keep getting into this launch file, let's explain its topics, services and parameters. The `slam_gmapping` node takes in `sensor_msgs/LaserScan` messages and builds a map (`nav_msgs/OccupancyGrid`). The map can be retrieved via a ROS topic or service.

- Subscribed Topics
  - `tf (tf/tfMessage)`: Transforms necessary to relate frames for laser, base, and odometry.
  - `scan (sensor_msgs/LaserScan)`: Laser scans to create the map from.

## ➤ Published Topics

- `map_metadata` (`nav_msgs/MapMetaData`): Get the map data from this topic, which is latched, and updated periodically.
- `map` (`nav_msgs/OccupancyGrid`): Get the map data from this topic, which is latched, and updated periodically.
- `~entropy` (`std_msgs/Float64`): Estimate of the entropy of the distribution over the robot's pose (a higher value indicates greater uncertainty)

## ➤ Services

- `dynamic_map` (`nav_msgs/GetMap`): Call this service to get the map data.

In the next table, the principle parameters used in this launch file and a brief explanation about each of them are shown.

Parameters	
<b><code>~throttle_scans</code> (int, default: 1)</b>	Process 1 out of every this many scans (set it to a higher number to skip more scans)
<b><code>~base_frame</code> (string, default: "base_link")</b>	. The frame attached to the mobile base.
<b><code>~map_frame</code> (string, default: "map")</b>	The frame attached to the map.
<b><code>~odom_frame</code> (string, default: "odom")</b>	The frame attached to the odometry system.
<b><code>~map_update_interval</code> (float, default: 5.0)</b>	How long (in seconds) between updates to the map. Lowering this number updates the occupancy grid more often, at the expense of greater computational load.
<b><code>~maxUrange</code> (float, default: 80.0)</b>	The maximum usable range of the laser. A beam is cropped to this value.
<b><code>~sigma</code> (float, default: 0.05)</b>	The sigma used by the greedy endpoint matching.
<b><code>~kernelSize</code> (int, default: 1)</b>	The kernel in which to look for a correspondence
<b><code>~lstep</code> (float, default: 0.05)</b>	The optimization step in translation.

Parameters	
<b>~astep (float, default: 0.05)</b>	The optimization step in rotation.
<b>~iterations (int, default: 5)</b>	The number of iterations of the scanmatcher.
<b>~lsigma (float, default: 0.075)</b>	The sigma of a beam used for likelihood computation.
<b>~ogain (float, default: 3.0)</b>	Gain to be used while evaluating the likelihood, for smoothing the resampling effects.
<b>~lskip (int, default: 0)</b>	Number of beams to skip in each scan. Take only every (n+1)th laser ray for computing a match (0 = take all rays)
<b>~minimumScore (float, default: 0.0)</b>	Minimum score for considering the outcome of the scan matching good. Can avoid jumping pose estimates in large open spaces when using laser scanners with limited range (e.g. 5m). Scores go up to 600+, try 50 for example when experiencing jumping estimate issues.
<b>~srr (float, default: 0.1)</b>	Odometry error in translation as a function of translation ( $\rho/\rho$ )
<b>~srt (float, default: 0.2)</b>	Odometry error in translation as a function of rotation ( $\rho/\theta$ )
<b>~str (float, default: 0.1)</b>	Odometry error in rotation as a function of translation ( $\theta/\rho$ )
<b>~stt (float, default: 0.2)</b>	Odometry error in rotation as a function of rotation ( $\theta/\theta$ )
<b>~linearUpdate (float, default: 1.0)</b>	Process a scan each time the robot translates this far.
<b>~angularUpdate (float, default: 0.5)</b>	Process a scan each time the robot rotates this far.
<b>~temporalUpdate (float, default: -1.0)</b>	Process a scan if the last scan processed is older than the update time in seconds. A value less than zero will turn time based updates off.
<b>~resampleThreshold (float, default: 0.5)</b>	The Neff based resampling threshold.

Parameters	
<b>~particles (int, default: 30)</b>	Number of particles in the filter.
<b>~xmin (float, default: -100.0)</b>	Initial map size (in metres)
<b>~ymin (float, default: -100.0)</b>	Initial map size (in metres)
<b>~xmax (float, default: 100.0)</b>	Initial map size (in metres)
<b>~ymax (float, default: 100.0)</b>	Initial map size (in metres)
<b>~delta (float, default: 0.05)</b>	Resolution of the map (in metres per occupancy grid block)
<b>~llsamplerange (float, default: 0.01)</b>	Translational sampling range for the likelihood.
<b>~llsamplestep (float, default: 0.01)</b>	Translational sampling step for the likelihood.
<b>~lasamplerange (float, default: 0.005)</b>	Angular sampling range for the likelihood.
<b>~lasamplestep (float, default: 0.005)</b>	Angular sampling step for the likelihood.
<b>~transform_publish_period (float, default: 0.05)</b>	How long (in seconds) between transform publications. To disable broadcasting transforms, set to 0.
<b>~occ_thresh (float, default: 0.25)</b>	Threshold on gmapping's occupancy values. Cells with greater occupancy are considered occupied (i.e., set to 100 in the resulting sensor_msgs/LaserScan).
<b>~maxRange (float)</b>	The maximum range of the sensor. If regions with no obstacles within the range of the sensor should appear as free space in the map, set maxUrange < maximum range of the real sensor <= maxRange.

Table 5. Parameters of gmapping

As we can see, this launch file is considerably more complex than the other ones. That is because the complexity of the SLAM technique. It has to manage all data from the sensors as well as computing the robot's localization and building the map at the same time. Although, it is fair to say that it is a really interesting and fascinating field of mobile robot and autonomous mobile robots.

## IV. Simulation

After understanding the most important information about each package and their nodes, it is time to test and simulate how to use *move\_base* with *gmapping* to perform autonomous planning and movement with simultaneous localization and mapping (SLAM), on a simulated Husky, or a factory-standard Husky with a laser scanner publishing on the scan topic.

The first thing to do is to start the Clearpath-configured Husky simulation environment, start the Clearpath-configured RViz visualizer and start the *gmapping* demo. To do that, the different launch files cited in the last section should be executed by writing the next commands in three separate terminal windows:

- `$ roslaunch husky_gazebo husky_koridor3.launch`
- `$ roslaunch husky_viz view_robot.launch`
- `$ roslaunch husky_navigation gmapping_demo.launch`

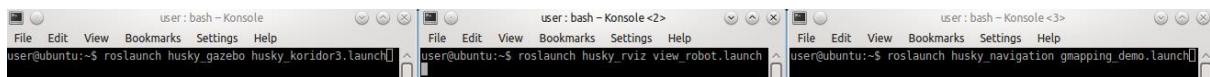


Figure 3—Commands to execute the required launch

The first command will start the Gazebo environment shown in *Figure 4*.



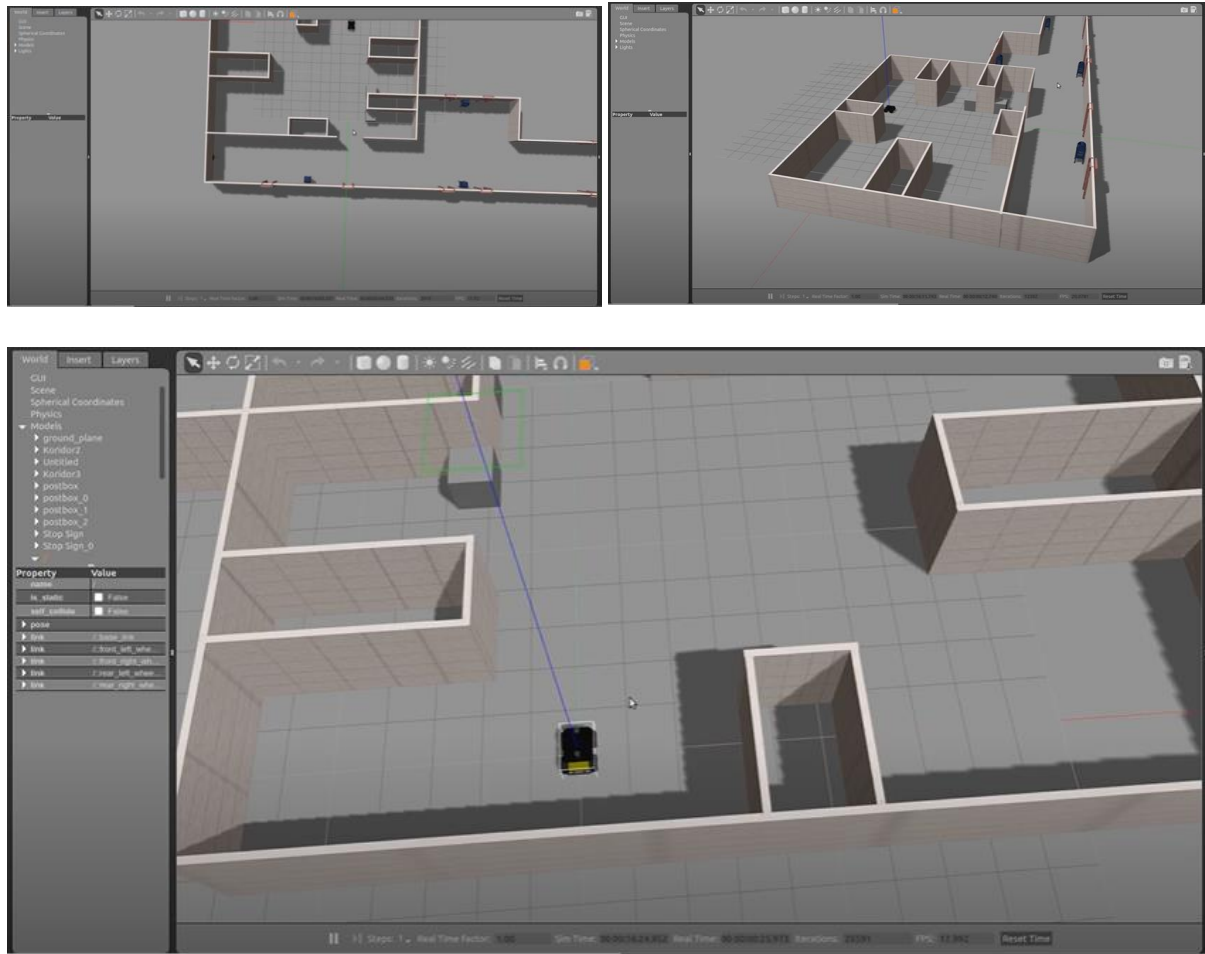


Figure 4 – Husky and world launch in the Gazebo environment

The second terminal window will open the RViz visualizer with our Husky robot in it.

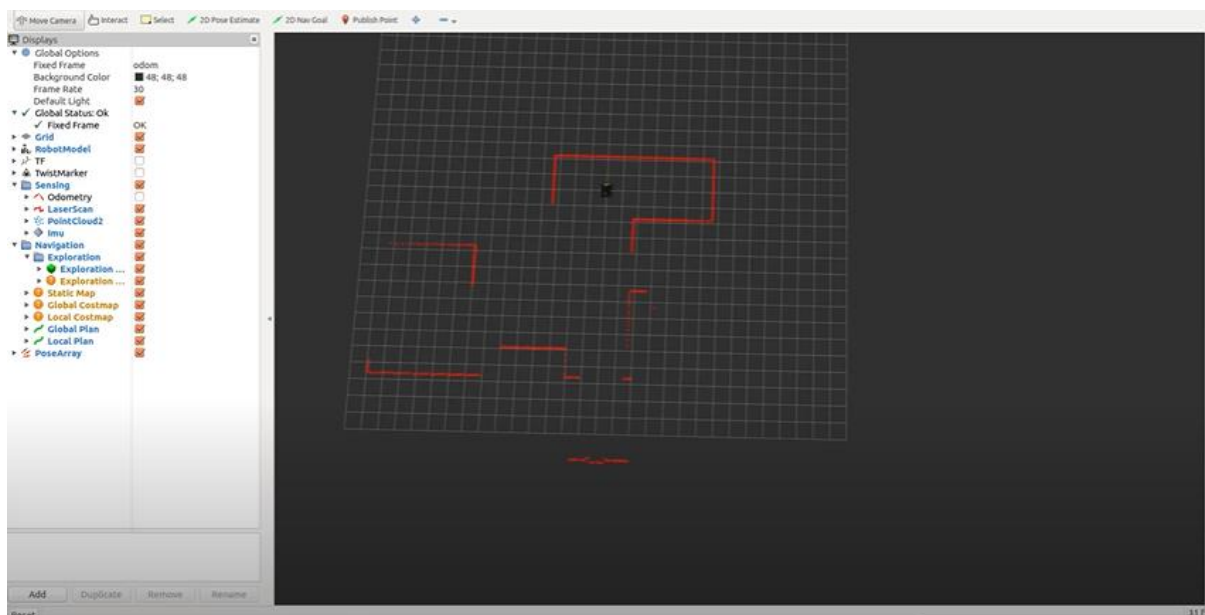


Figure 5 – Husky launch in the RViz visualizer

The third one will change the way we view the RViz window. We will see how the map, that the SLAM technique is creating, looks like.

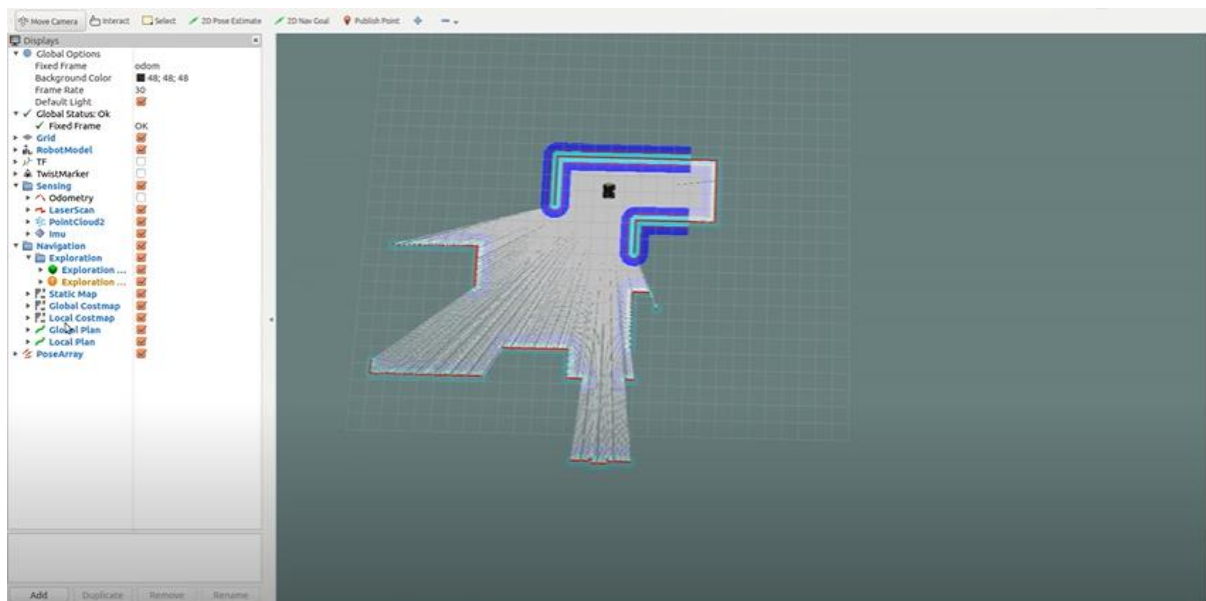
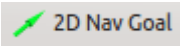


Figure 6 – Husky and map launch by gmapping in Rviz

In the Rviz visualizer, we have to make sure the visualizers in the *Navigation* group are enabled. Then, we will use the *2D Nav Goal* tool in the top toolbar  to select a movement goal in the visualizer, 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. The green arrow indicates the final orientation of the robot once it arrives to the goal location. It is important to be sure to select an unoccupied (dark grey) or unexplored (light grey) location.

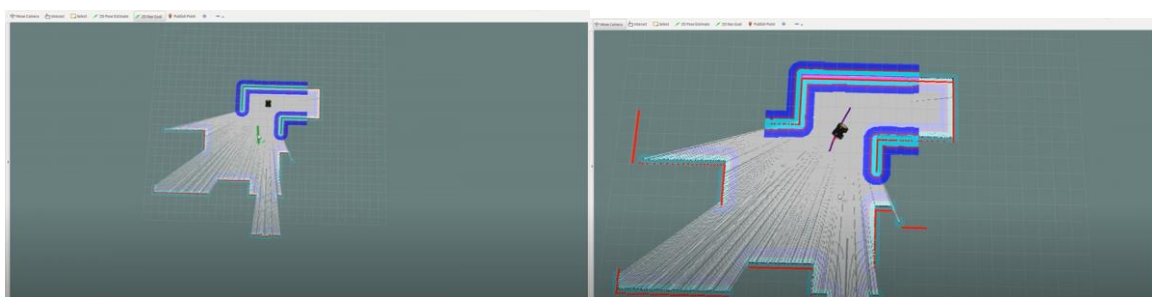


Figure 7 – Example of a goal and the global plan in Rviz



Figure 8—Zoom of Figure 7 to see the local plan in red

As the robot moves, you should see the grey static map (map topic) grow. Occasionally, the *gmapping* algorithm will relocalize the robot, causing a discrete jump in the map->odom transform. Furthermore, as you can see in *Figure 7*, *Figure 8* and *Figure 9*, a global plan is drawn from the start point to the finish point (purple color) and a local plan is being drawn along the way (red color), trying to follow the global path without crashing.

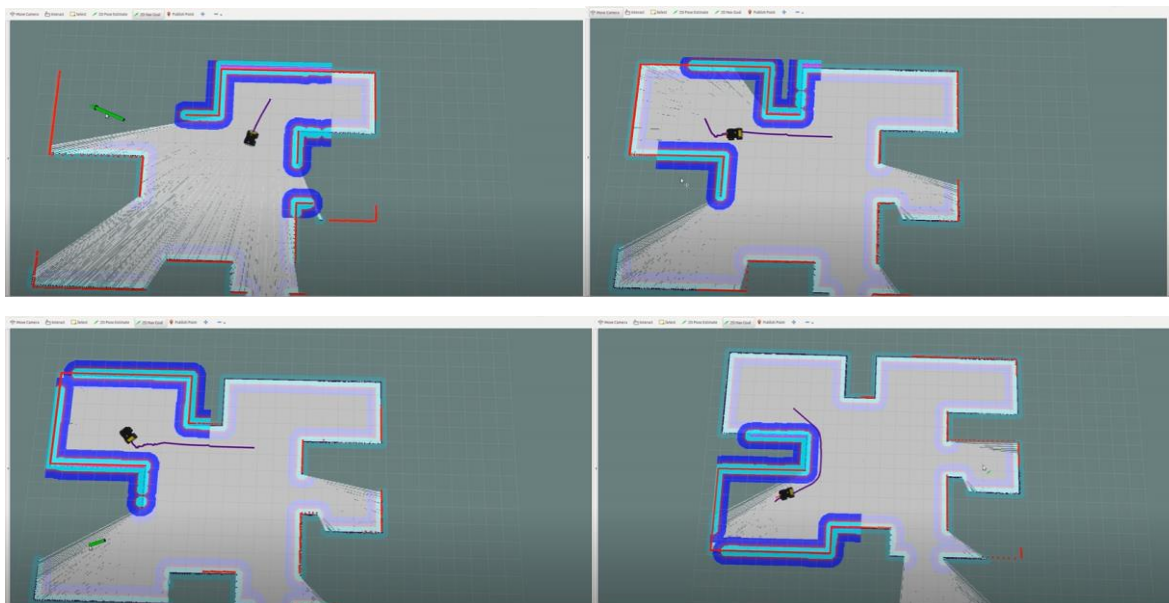


Figure 9—More examples of a goal and the global plan in Rviz

In *Figure 10*, we can see that when the robot is in the middle of a path to a goal location, if the operator selects a new goal, the robot will change immediately its path to the new goal and it will forget about achieving the last goal. Also, *Figure 11* shows pretty clearly that a little rotation of the robot will make its laser sensor to achieve and map previously unknown obstacles comparing it to the previous orientation in the second image.

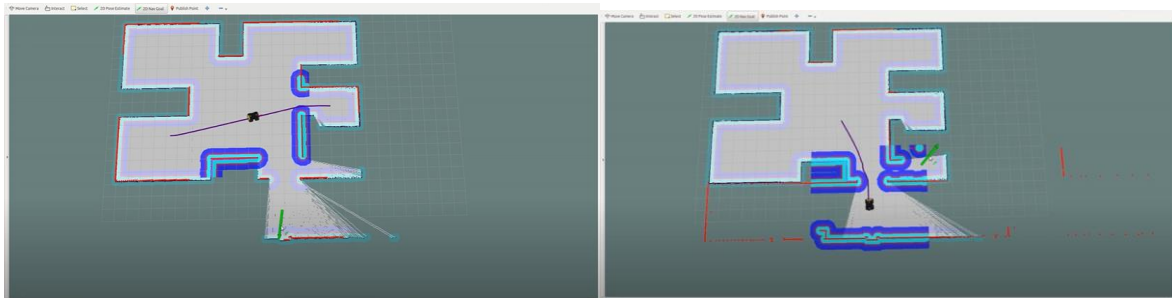


Figure 10— Example of selecting a new goal in the middle of a path to another goal

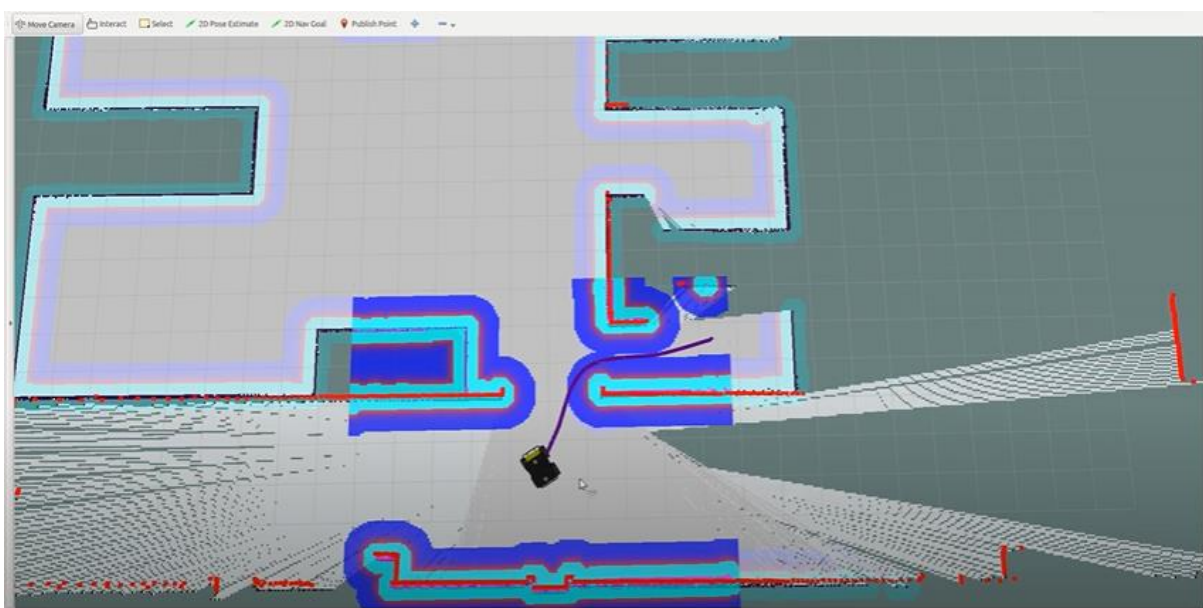


Figure 11 – Example of how a small rotation and how it affects mapping

In *Figure 12*, we can see, in the Gazebo environment, how accurate and careful is our robot to avoid any kind of obstacles.

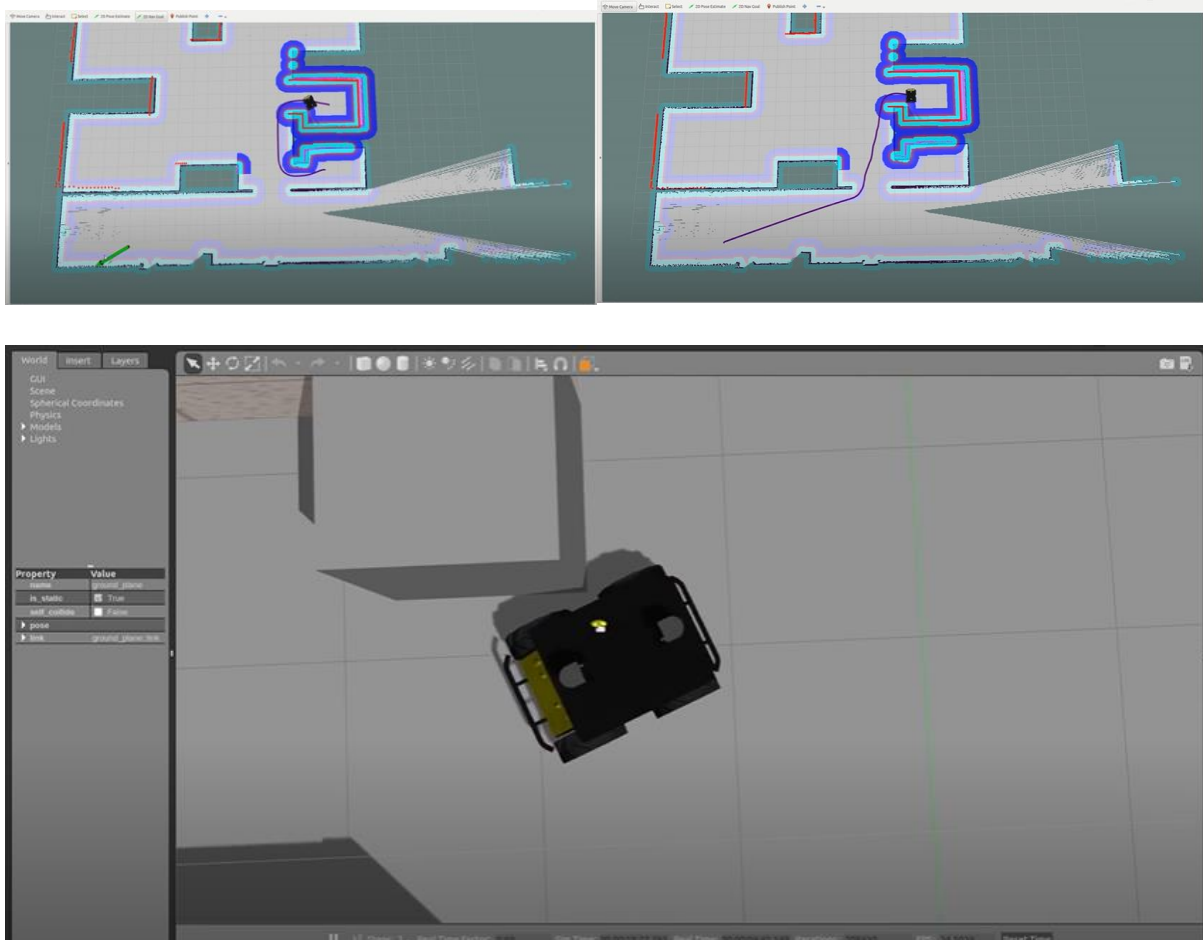


Figure 12 – Example of the Husky avoiding an obstacle

In the last figure, *Figure 13*, we can compare the position of our robot in the Gazebo environment and the RViz visualizer at the exactly same moment.

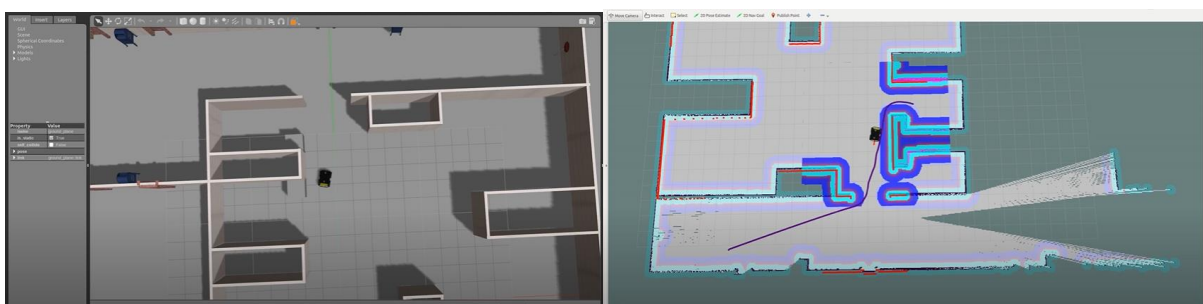


Figure 13 – Example of Gazebo and RViz at the same moment

We can continue moving our Husky robot all around the corridor until it has mapped everything. After we finished, in order to save the generated map, the `map_saver` utility can be run in a terminal window:

➤ `$ rosrun map_server map_server -f <filename>`

The map file is stored as two files: one is the YAML file, which contains the map metadata and the image name, and second is the image in a PGM file, which has the encoded data of the occupancy grid map [12].



## V. Conclusion

Robotics today is a much richer field than even a decade or two ago, with far-ranging applications. The continuous improvements sometimes make a bit difficult to catch up the latest advancements. Nevertheless, autonomous mobile robots seem to have a bright future in the modern world thanks to their wide range of applications and the solid base of researches and knowledge that they are supported to be developed and applied. The use of ROS also seem to have a bright future. The open source of packages and information allows people to start using ROS in a short amount of time as well as to help new researches to begin and see some examples that can be very useful to them.

This report has presented the information and basic concepts to understand and being able to simulate in ROS the navigation of a Husky robot using the technique of mapping and localization called SLAM. To sum up, I can conclude that this project has been quite a success. As I have shown in the *Simulation* section, our simulated Husky is able to navigate through different waypoints avoiding every kind of obstacles and mapping the previous unknown environment during this process. However, as I said, the *gmapping* algorithm sometimes will relocalize the robot, causing a discrete jump in the map->odom transform. This could be an important issue in running a robot in the real world. One possible measure to figure out this problem is to use the AMCL technique that will be discussed in the next section. Also, I hope this report can help people to understand and have a basic idea of ROS and one example of its countless implementations like navigating a Husky robot for an unknown environment and mapping that environment.

Personally, I have enjoyed and learned a lot in the process of doing this project. I find satisfying to have been able to learn about such an interesting and useful tool in robotics nowadays, like ROS is, that I had no idea about it before starting this project. In addition, not only I have learned about ROS but I could understand better and get it touch to different fields about the robotics industry such as guidance, navigation and control.

Lastly, I would like to say that robotics is a fascinating area of the world that can be shown in so many different applications from the simplest model and functionality to incredible complex examples. Also, the limits of robotics are being expanded each day in

unprecedented ways that we cannot even imagine. In my opinion, I cannot wait to continue exploring this amazing world.

### ***V.I. What's next?***

Even though, simulating robot navigation can be very useful and interesting at some points like when the robot is not available, or to ensure that the software works properly or in order to test the software more quickly, it would not be practical at all unless what you are simulating can be implemented in real world. Now, that we have been able to simulate and control successfully our Husky robot. The next interesting step is to implement our software to a real Husky robot and, then, test how well and accurate our robot performs. After that, if it is needed, the correspondent changes and improvements will be implemented to our code until our robot works as we desire. Moreover, although, in theory, the gmapping is self-sufficient, 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. In order to solve that, it is interesting to work as well with AMCL technique (Adaptive Monte Carlo Localization) applying an AMCL node to the navigation package. Even though AMCL is only used for static maps, it can be used to help gmapping. When the map is created with gmapping and send it to the AMCL node, it will trust the map and adapt the odometry, that will produce much better results.



## Bibliography

- [1] M. W. Spong and M. Fujita, "Control in Robotics," 2011.
- [2] Clearpath Robotics, "Husky. Unmanned Ground Vehicle.," [Online]. Available: <https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/>.
- [3] Robotics Business Review, "The 2017 RBR50 List Names Robotics Industry Leaders, Innovators," 2017.
- [4] Open Source Robotics Foundation, "About ROS," [Online]. Available: <https://www.ros.org/about-ros/>.
- [5] A. Martinez and E. Fernandez, Learning ROS for Robotics Programming, 2013.
- [6] J. M. O'Kane, A Gentle Introduction to ROS, Jason Matthew O'Kane, 2013.
- [7] R. Siegwart, I. Nourbakhsh and D. Scaramuzza, Introduction to Autonomous Mobile Robots, Segunda ed., The MIT Press, 2011.
- [8] S. Riisgaard and M. Blas, "SLAM for Dummies: A Tutorial Approach to Simultaneous Localization and Mapping," 2005. [Online]. Available: [http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-412j-cognitive-robotics-spring-2005/projects/1aslam\\_blas\\_repo.pdf](http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-412j-cognitive-robotics-spring-2005/projects/1aslam_blas_repo.pdf).
- [9] J. Fabro, R. Longhi, A. Scheneider and T. Becker, "ROS Navigation: Concepts and Tutorial," in *The ROS Multimaster Extension for Simplified Deployment of Multi-Robot Systems*, Springer International Publishing, 2016, pp. 121-160.
- [10] ROS, "ROS.org," [Online]. Available: [http://wiki.ros.org/move\\_base](http://wiki.ros.org/move_base).
- [11] ROS, "gmapping," ros.org, [Online]. Available: <http://wiki.ros.org/gmapping>.
- [12] L. Joseph and J. Cacace, "Building a map using SLAM," in *Mastering ROS for Robotics Programming*, Second ed., 2018.

## Appendix

- <https://github.com/DiegoAlvaro1/ELC4V97HuskyNavigation>

