Navigation of a mobile robot using hand gestures

Konstantina Gerasimopoulou

Department of Computer Science and Engineering

University of Ioannina

Abstract

The aim of this thesis paper is the navigation of a wheeled differential robot within a specific environment, after the commands given by the recognition of gestures. The navigation issue consists of three main parts, the mapping, the robot localization and the path planning. The purpose of the path planning is to find a sequence of free points, in order to reach successfully and without conflicts, from the start point to the goal position. The path planning we have implemented is divided into two parts, global planning, which computes the path according to its prior knowledge of the environment, and local planning, which is essentially responsible for robot's motion and dynamic obstacle avoidance that may arise. To communicate with the robot, we have used the ROS environment and the implementation has been tested both on simulation level and on our real robot.

Keywords: navigation, gesture recognition, path planning, ROS

Table of contents

Chapter 1. Introduction

The logic of the implementation followed in this work for both the gesture recognition by the robot and the navigation has been followed in other published papers.

Initially an implementation that has much in common with ours is the one of Michael Van den Bergh, Daniel Carton, Roderick De Nijs, Nikos Mitsou, Christian Landsiedel, Kolja Kuehnlenz, Dirk Wollherr, Luc Van Gool and Martin Buss, who dealt with real-time gesture recognition by a robot that perceives the given orientation and is directed respectively in space. As soon as the robot detects a human, it approaches them at a specific distance waiting to receive direction commands from the orientation of the fingers. Then the robot moves in space with the navigation stack packages.[1]

The next logical step was to study the navigation stack packages. From Kaiyu Zheng's detailed research work, we got sufficient information on the logic of Global and Local Planner in

navigation stack, as well as for the mapping and localization methods that are available as packages in ROS.[2]

Finally, for the theory of navigation algorithms that we implemented we relied on the book “Principles of Robot Motion: Theory, Algorithms, and Implementation”, where many important navigation algorithms and optimal path finding algorithms are mentioned, and also how to use the configuration space.[3]

Chapter 2. Hardware

The robot that was used is the Pioneer 3-DX of our university lab, as well as the Nvidia Jetson TX2 and the Sick LMS 200 laser, which are integrated on it, as shown in the image below.

Image 2.1: Pioneer 3-dx of our lab

2.1 Robotic base

The Pioneer 3-DX is a differential drive robot, which consists of 2 wheels with encoders, for measuring rotations and hence the angular velocity, as well as the caster wheel, which is not involved in movement and it is used for balance. It provides a surface, the deck, which is located above the internal components and in its surface other components can be incorporated, and front and rear SONAR sensors. [4]

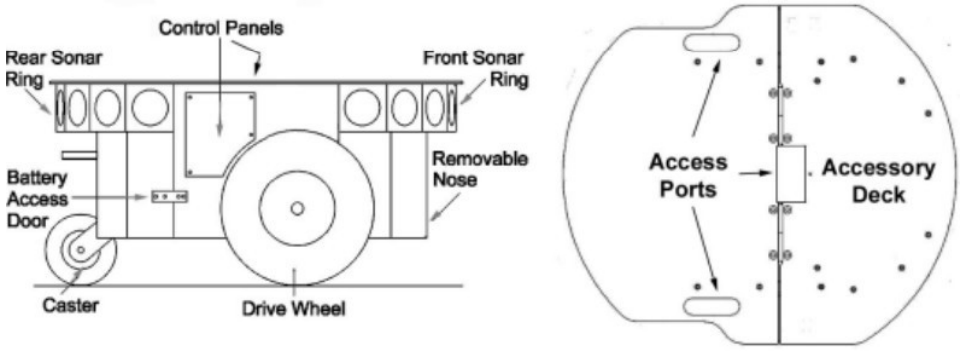

Image 2.2: The robotic base of the Pioneer 3-DX

Also on the left side is the user control panel, from which we have access to the microcontroller. This particular robot of our lab

also has front and rear bumper sensors for the physical contact detection with objects.

The differential drive system consists of two wheels mounted on a common axis and each wheel moves independently forward or forward back. Differential vehicles can not move along the axis that connects the wheels. They move straight forward when given the same velocity on both wheels, while if one wheel rotates more quickly, the robot moves towards the slowest wheel. In case of opposite velocities, the robot rotates in the same relative position and in the case that the velocity of only the one wheel is zero, the robot rotates around this wheel.

2.2 Nvidia Jetson TX2

The Jetson TX2 is a complete computer system with Quad-Core ARM® Cortex®-A57 processor, Pascal architecture GPU with 256CUDA cores, 8GB LPDDR4 memory and 32GB eMMC 5.1 storage. The plastic base on which the computer system is screwed, is

mounted on a 2 DOF robotic rotary joint (PTU) to achieve optimal use of its fixed camera.

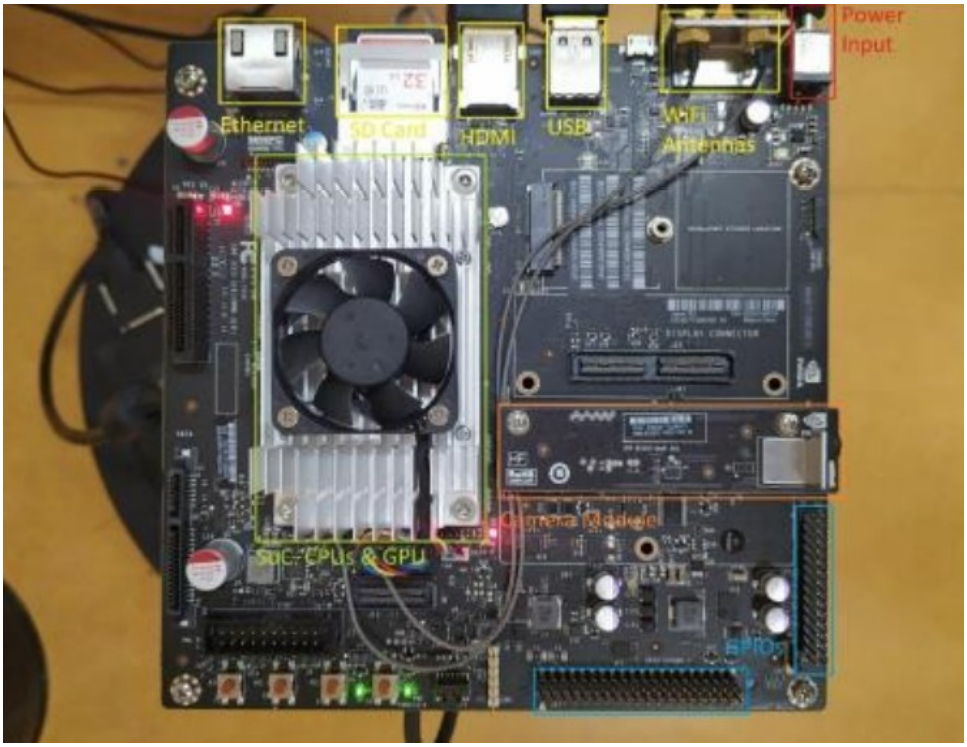

Image 2.3: Nvidia Jetson

This robotic joint rotates up to + -159 degrees with respect to vertical axis, while for the horizontal axis the rotation can be from -47degrees to +31, forming a range of 318 and 78 degrees for both respectively axes.

Image 2.4: The PTU on top of the Sick LMS laser

2.3 Sick LMS 200 laser

The PTU on which the Jetson is mounted on, is located at the top of the laser device. The LMS 200 uses LIDAR technology(Light Detection And Ranging) and this process results in 3Drepresentation of the environment. More specifically, in this technology, the laser beam is emitted on the surface of a mirror, which forms an angle of 45 degrees with the plane and it rotates. In this way, the resulting radius of 90 degrees, rotates, and after the return of the reflected light and its analysis by a sensor, depending on the time needed, the distance of the object is calculated.(Distance = (Speed of light x Time elapsed) / 2).

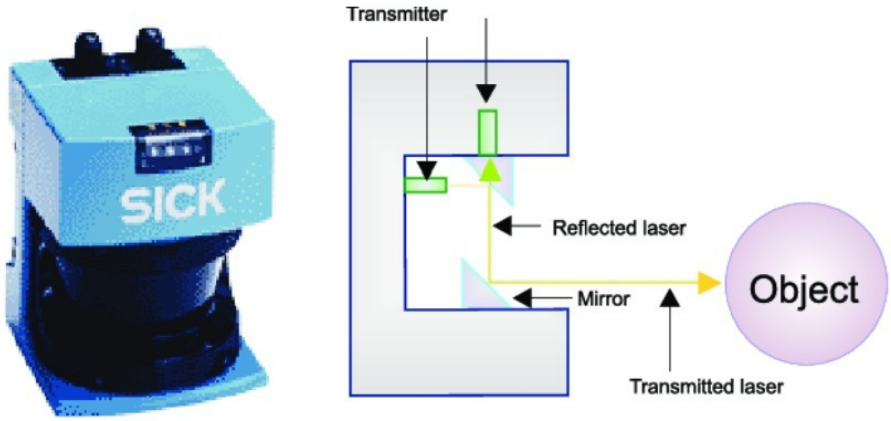

Image 2.5: The PTU on top of the Sick LMS laser

The scanning range of the LMS 200 is a total of 180 degrees, performed per1 degree and is performed from right to left. The result of each scan is an array with 180 values. The first place in the array corresponds to0 degrees, the last position corresponds to 180 degrees and all intermediate positions correspond similarly to the other degrees. Their values are the distance measured in each individual measurement, with a maximum of 10 meters.

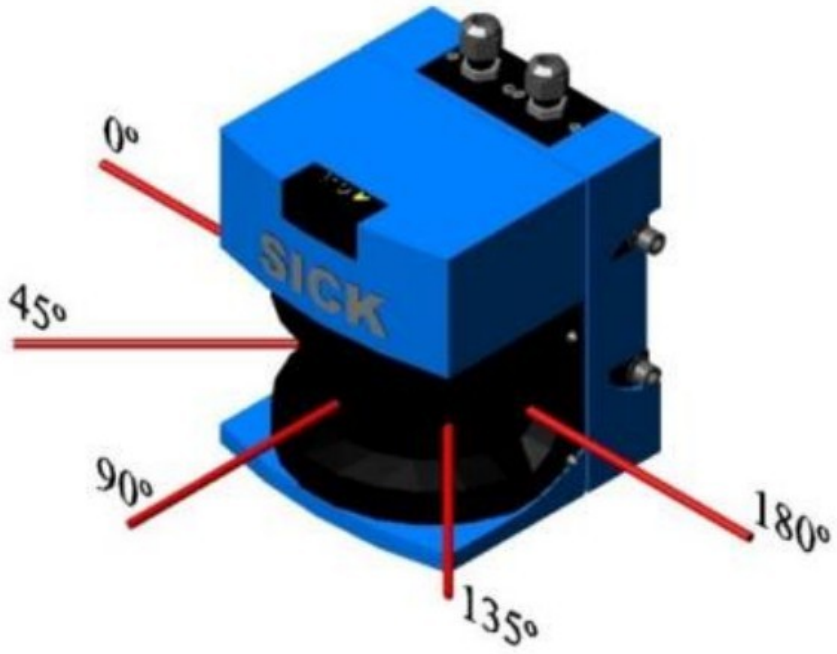

Image 2.6: Sick LMS 200

Chapter 3. Software

3.1 Robot Operating System (ROS)

For the complete operation of a robot we need both the hardware and the software through which we will give the appropriate commands. Robot software platforms include tools for robotic applications development, such as low-level control devices, navigation, use of sensors, SLAM (Simultaneous Localization And Mapping), tools for debugging purposes, as well as management of packages and libraries. Among many platforms, one that stands out is the ROS (Robot Operating System), mainly for the possibility to transfer the information between processes, even in different machines, and organizing executable code into packages. ROS is mainly for ubuntu systems, but now ROS2 allows use both on macOS and Windows systems.[7]

Image 3.1: Robot Operating System

The latest version was used for the simulation (ROS Noetic), which requires the latest version of ubuntu respectively (20.04 Facal), while the older release (melodic) was used for the real robot, because Nvidia does not currently provide an official JetPack package for installing the latest version of ubuntu.

ROS is considered an meta-operating open source system and it supports a variety of programming languages, popular in robotics, such as Python, C ++, and Lisp. In this thesis only Python was used in conjunction with the client library rospy.

To use ROS you must first run the ROS master. This is achieved by running the "roscore" command in the terminal. After running the master, we can now run the programs via ROS and get information through terminal commands.

The ROS also allows programs to run from different computers wirelessly. This can be done with several ways, one of which is through IP addresses. More specifically, it is enough to add in the bashrc file of the computer that will work as the master the command “export ROS_HOSTNAME = xxx.xxx.xxx.xxx” and “export ROS_IP = xxx.xxx.xxx.xxx” with the IP address we will get from the command “Ifconfig”, while on the other computers we add in the bashrc file the command “Export ROS_IP = xxx.xxx.xxx.xxx” with its own IP address that we find through “Ifconfig” and the export command “ROS_MASTER_URI = http://yyy.yyy.yyy.yyy: 11311” entering the address of the computer-master we got earlier.

Communication in ROS is based on the idea that executable programs(nodes) are executed and communicate with each other by exchanging data through messages. There are 3 different message exchanging methods, topics, services and actions, although, only services and actions provide a two-way communication.

In this thesis only the topics method was used, and each node can publish a message on a topic and subscribe to another topic waiting to receive a message from another publisher node.

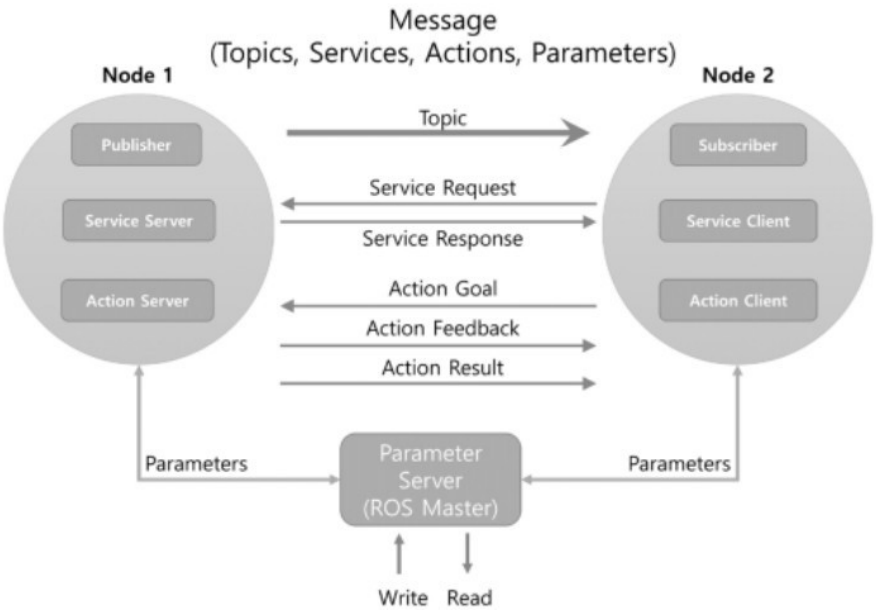

Image 3.2: Node communication in ROS environment

ROS also has many commands that we can execute in the shell. The most important ones are the program execution commands, rosrun and roslaunch. The difference is that rosrun is used to run a single program-node, while with roslaunch we can

run more nodes as well as change several parameters, through the required launch file.

Also important are the commands that give us information, about a node, a topic, or a message, and these are the commands rosnode, rostopic and rosmsg respectively. With these commands we can see what is running at the moment, information about a node, what is published at any moment on a topic, and much more. [5]

With the installation of ROS, some extra programs are installed, Gazebo and Rviz.

Gazebo is a 3D simulator and it provides some robot models ,sensors, environments for developing robotic applications and offers a realistic simulation. In general, it is a common technique, even if there is a real robot, to develop a ROS program firstly at the simulation and then apply it to the real robot. This makes the process easier, as if a program works on the simulation, with little or no changes will work the same effectively in the real world.

Image 3.3: Gazebo environment

Rviz, on the other hand, is a helpful - but often necessary - debugging tool, which can be used both at the simulation level and when we run programs on the real robot. For example, it can be used to visualize the distance from an object as perceived by a distance laser sensor (LSD), for having the image of the robot's camera, and many other features for which in this way does not need to create separate software. Also, Rviz is required at the stage of mapping as well as the robot's localization.

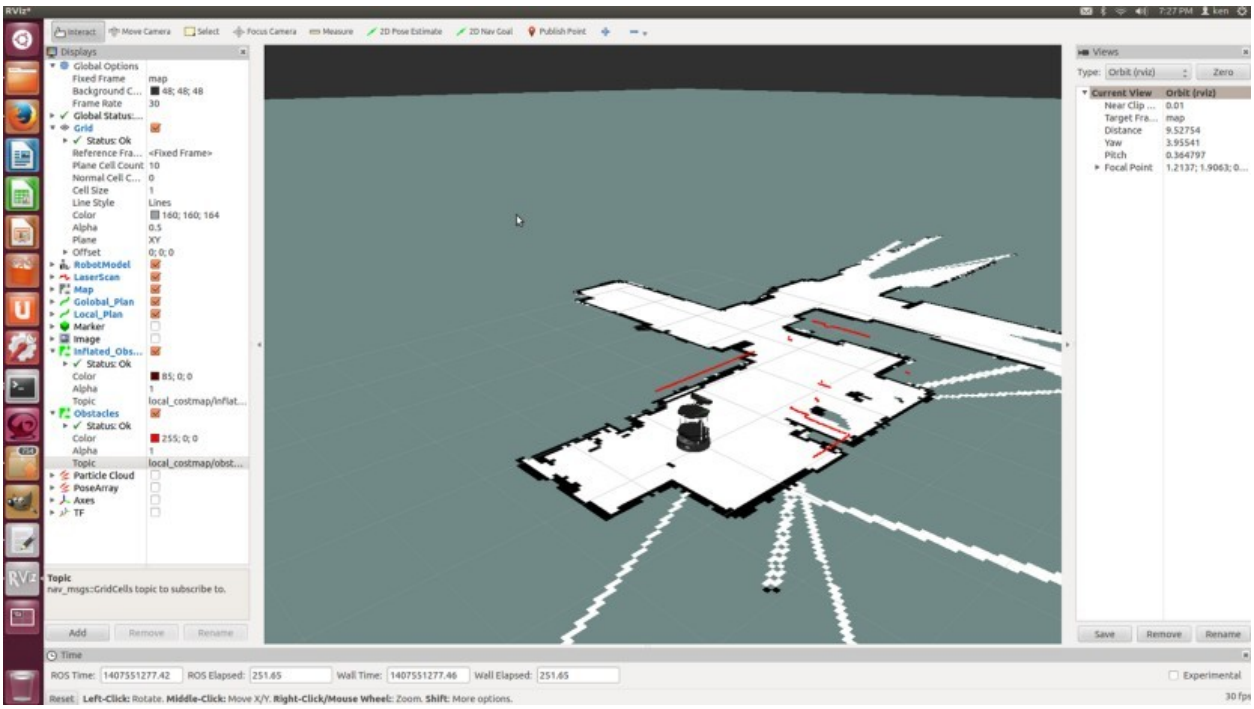

Image 3.3: Gazebo environment

3.2 Communication with hardware

To communicate with the robotic base and the sensors that have been added to it, we need some programs which will perform the interface. These programs run internally on the ROS and have the form of nodes.

3.2.1 RosAria

For the control of the sensors and wheels of the pioneer 3-DX there are some already implemented packages written with ROS convention. The most popular are p2os and RoSAria. P2os was used at the simulation level, while RosAria was installed on the robot. Must note that RosAria[8] requires the Aria[9] Library to be installed, which executes commands at a low level by controlling the robotic base.

In more detail, RosAria gives us the ability to obtain information from important topics related to sonar sensors, bumper sensors and battery information. In addition, it receives commands from the

topic that is responsible for the velocity of the robot and gives information to another topic about odometry, ie its veocity and position.

More specifically, the topics and the corresponding types of messages that are published on them are:

- Topic: /rosaria/cmd_vel msg: geometry_msgs/Twist

This is the topic associated with velocity commands given to the robotic base and it is the only topic which the node RosAria subscribes to. Twist type messages consist of two-dimension vectors. The first vector is responsible for linear velocities, while the second vector gives the values for the angular velocities, around the axes. But because this robot is a differential two-wheeled model, the only values that affect the velocity of the robot are the linear velocity on the X axis and the angular velocity on the Z axis.

- Topic: /rosaria/pose msg: nav_msgs/Odometry

This topic contains information about the trajectory of the robotic base. This information is derived from the calculations of the

RosAria package based on the values from the motor encoders. The Odometry is one of the ways we take the position of the robot, but it is not quite reliable for the real robot as it contains errors. This is due to the fact that the ideal conditions do not exist in the theoretical application as well as in the simulation.

• Topic: /rosaria/bumper_state msg: rosaria/BumperState

The status of the bumpers is published on this topic.

• Topic: /rosaria/sonar msg: sensor_msgs/PointCloud Topic: /rosaria / sonar_pointcloud2 msg: sensor_msgs/PointCloud2

In this topic the information from the sonar sensors is published

• Topic: /rosaria/state_of_charge msg: std_msgs/Float32

Information about the battery level of the robotic base, withvalues from 0.0 to 1.0.

• Topic: /rosaria/battery_voltage msg: std_msgs/Float64

Current battery voltage in DC Volts.

• Topic: /rosaria/battery_recharge_state msg: std_msgs/Int8Information about the battery status of the robotic base battery.

• Topic: /rosaria/motors_state msg: stf_msgs/Bool

Information related to the condition of the motors of the robotic base. True value if it is enabled and the robot will respond to one velocity command, otherwise False value.

To execute the RosAria node, we just execute the command "Rosrun rosaria RosAria" in the shell. This command will execute the node "RosAria" from the "rosaria" folder.[10]

3.2.2 Sicktoolbox

To use the laser, we need another program-node. The sicktoolbox is the package responsible for the use and control of the Sick Lidar laser. In particular, the two packages which are used are sicktoolbox[11], which contains the drivers and header files, and

sicktoolbox_wrapper[12] which applies the above specifically in the ROS environment.

Information about the two packages can be found on the official page of ROS.[6]

To execute the node, we just execute *rosrun sicktoolbox_wrapper sicklms*

in the shell. This command will run the sicklms node of the sicktoolbox_wrapper folder. The data collected by the Lidar laser are published on the topic /scan and the messages are of type sensor_msgs/LaserScan. Writing the command in the shell "rostopic echo /scan", we can get information about the laser.

Specifically, the structure of the LaserScan message consists of:

1. Header.

The header of the message which contains its timestamp, ie the exact time of the first scan.

2. angle_min

The minimum scanning angle in radians.

3. angle_max

The maximum scanning angle in radians.

4. angle_increment

Angular scan step, ie the angle formed between two successive measurements, in radians.

5. time_increment

The time required between two consecutive measurements.

6. scan_time

The time required for the scan of this particular message.

7. range_min

The minimum distance from an obstacle that can be measured.

8. range_max

The maximum distance from an obstacle that can be measured.

9. ranges []

An array with the values of the specific scan, in the order thattook place, from right to left.

10. intensities []

The array with the values of the intensities of the reflected rays.

3.2.3 Jetson Camera

To use the camera in the ROS environment, we use the launch file jetson_csi_cam.launch of the jetson_csi_cam folder and run it by executing in the shell the command "roslaunch jetson_csi_cam jetson_csi_cam.launch"with default resolution 1920x1080 pixels and 30 fps. During its execution command it is possible to change these parameters, as long as we add at the end of the command "width: = x height: = x fps: x", where x are the desired values.[13]

3.3 Software for gesture recognition

Regarding the part of recognizing gestures, the libraries used are Tensorflow and OpenCv. Tensorflow is an open source library used mainly for machine learning, ie it enables training and execution of deep neural networks for many uses, one of which is image recognition. In addition, Tensorflow enables GPU execution, which the training and execution process easier.

OpenCv is also an open source library, which is used for computer vision applications. In the present work it was used to process the image from the robot camera while Tensorflow was needed for the detection and recognition of gestures. It should be noted that on the jetson board the older version of Tensorflow 1.11.0 has been installed, as it does not support, for now, the official package for Tensorflow 2.

Chapter 4. Implementation of navigation algorithms and simulation results

4.1 Gazebo and Rviz visualization

To be able to develop and control the navigation algorithms at the simulation level we must first have the appropriate visualization in both Gazebo and Rviz. More specifically, we need the visualization of the robot, the sensors and the camera as well as their full functionality. We also need the visualization of a world (Gazebo world), ie an environment, which we modify as we wish, in which it the robot can be navigated. Finally, we need the mapping of this world in order to properly implement the navigation algorithms, and the map appearance in Rviz for the debugging process.

The files of the world and the robot used are from an already implemented project to which the navigation stack is used.[14]

In particular, 2 launch files were used. The first launch file, pioneer_world.launch, consists of nodes and execution parameters

for the display in the Gazebo, while the second, pioneer_rviz.launch, implements the visualization in Rviz.

Regarding the display of the robot in Gazebo but also in RViz, initially we need the robot and sensor URDF files. URDF (universal robot description format) is an XML format for the representation of robotic models. It includes relevant information about the physical components of the robot (<links>) and how they relate to each other (<joints>).

<links> have <visual> properties about how they appear in Gazebo and Rviz, <collision>, about the space they occupy, and <intertial>, for mass and inertia.

The <joints> fields refer to the links between parent and child link as well the type of joint. The URDF also identifies transformations between links. The URDF file used in this case is a package based on p2os, one of the already implemented packages-drivers of the robot, with the difference that it has a Gazebo plugin for camera and laser added.

The world used in this project is a simple world, bounded by cubes and the map of this world in Rviz, has been implemented with slam algorithm of the ROS gmapping package.

4.2 Path planning

Initially, the goal of path planning is to find the optimal path, that is the one with the minimum time and the minimum distance, avoiding static and dynamic obstacles. In all cases the path planning is divided into global path planning and local path planning. The navigation algorithms implemented in this work are based on this logic. With similar logic the already implemented ROS navigation packages on the navigation stack are implemented.

Global path planning uses its prior information of the environment to create the optimal path, if it exists, while local path planning recalculates that path to avoid dynamic obstacles.

4.2.1 Global Planner

Global planning needs the map of the environment to calculate the optimal path. Among the implementation methods is the division of the map grid into small pieces, the cells, where each cell has a specific weight and then the execution of a minimal path finding algorithm, such as Dijksta or A *.

For this work it was chosen to implement the Dijkstra algorithm. This algorithm is for undirected, linked graphs with weighs at the nodes. Graphs consist of nodes and the edges that connect them to each other. The undirected graphs are those that the edges do not have a specific direction, ie are bidirectional, while the connected graphs are those that each node has a path to every other node. Dijkstra finds the minimal path from one source node to any other node in the graph. Its complexity is O (V ^ 2), where V is the number of nodes, but with binary stack usage can be reduced to O (E log V). where E is the number of the edges.

The basic idea of the algorithm is that each time we find the nearest node from the source node, which has not been discovered, and is either directly connected to the source node or is connected to a

node that has already been discovered. Below is the Dijkstra's algorithm as shown on wikipedia.[15]

Dijkstra's algorithm

```
1  function Dijkstra(Graph, source):
2
3      create vertex set Q
4
5      for each vertex v in Graph.Vertices:
6          dist[v] ← INFINITY
7          prev[v] ← UNDEFINED
8          add v to Q
9      dist[source] ← 0
10
11     while Q is not empty:
12         u ← vertex in Q with min dist[u]
13
14         remove u from Q
```

```
15
16      for each neighbor v of u still in Q:
17          alt ← dist[u] + Graph.Edges(u, v)
18          if alt < dist[v]:
19              dist[v] ← alt
20              prev[v] ← u
21
22  return dist[], prev[]
```

To be able to run the algorithm and find the optimal path, we first need to have the environment information in the form of a graph, ie with nodes and their edges.

The mapping of the environment and therefore the map that has as a result, it gives us the information we need about the environment. In ROS a 2D Occupancy Grid Map (OGM) is commonly used, which was used in our case as well. In this map the free area in which the robot can move is shown in white, in black the occupied-inaccessible area and in gray the area which has not been explored and there is no knowledge about it.

This information is displayed in pixel grayscale values from 0 to 255, which results from the Bayes theorem, which measures the probable capacity (occupancy probability).

ROS provides this information in a different form, as a message type “nav_msgs / OccupancyGrid” which is published on the topic “map” from the “map_server” node. Specifically, this message has a data field []containing a value for each region. Values can be 0, 100

or -1, if the area is free, reserved or unknown respectively. These areas are called cells. [16]

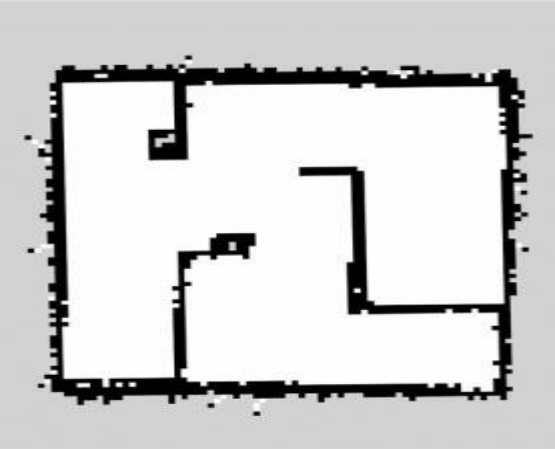

Image 4.1: Occupancy Grid in ROS

The logic was that we only need those cells for the graph that are accessible by the robot, because we want to use the Dijkstra's algorithm on the nodes we can go to. For that reason the GridController class was created. In this class we subscribe in the map topic, where the map_server node publishes, and in the callback function we get from the message the data field [], and specifically, from this array only the available cells, with a value of 0. It should be noted that this array is a one-dimensional array. To

find the value of one(x, y) cell in the array we just have to look in the index *index = x + info.width * y*, whereinfo.width is the width of the grid, which we get from its corresponding field of the OccupancyGrid message.

However, to make velocity calculations and position control easier in local path planning, we had to convert the cells to points as to the coordinate system of the world, in this case with respect to the /map frame coordinate system. That is done with me the following types:

*map_x_coordinate_in_meters = (i * self.map_resolution) + self.map_origin.position.x + self.map_resolution / 2*

*map_y_coordinate_in_meters = (j * self.map_resolution) + self.map_origin.position.y + self.map_resolution / 2*

In that way we now have all the accessible points by the robot in world coordinates.

Then another problem was the very large number of points due to the many decimal digits. That way, neither would Dijkstra run

efficiently, but it would also not make sense to manage points of such great decimal precision in local planning.

It was decided, therefore, to round off the points. After proper study, we saw that it would not affect the accuracy of the navigation if the points are either integers or with decimal digits equal to 0.5 and therefore this was used.

In this way, in both coordinates of the point, if the number is positive and the decimal part is less than or equal to 0.3 then we get the threshold of the number of this, while if the decimal part is greater than or equal to 0.7, we get the ceiling of the number, otherwise we get the whole part of the coordinate and we add 0.5. If the coordinate is a negative number then we perform the ceiling and the threshold inversely.

Now that we have the set of nodes we can use Dijkstra's algorithm. Source node is the initial position of the robot, in which we apply the same rounding we applied to the grid points. This initial position can be taken in various ways.

One of them is to take the position with respect to the /odom frame. However, the/odom frame does not have a fixed position on the

map, but it is where the robot is when we execute the program. Therefore, initially, the robot should always be in the same position, where the /map frame is located, so that the points resulted from the analysis of the grid are the same with the points that perceives the robot from odometry.

That, in addition to not being convenient, will show deviations due to odometry errors when the robot moves after a while. The next idea was to get the initial position from the amcl_pose topic. In this topic the node amcl publishes the position of the robot with respect to the /map frame. This node subscribes to the following topics: scan, tf, initialpose and map. Based on the information from the laser, the initial position of the robot, map and transforms, it estimates the position of the robot (amcl_pose)as well as the transform between the /odom frame and the /map frame.

For our implementation of the Dijkstra's algorithm we relied on the idea of 4-point connectivity. This means that it is considered that the robot can only move at the points that are just above, below, right and left, as shown in the image below. For our implementation, it should be noted that neighbors are considered

to be points whose coordinates have a +-0,5 or +-1 distance from the current node, due to how we handled the grid cells.

When we execute the algorithm,we explore all the nodes of the graph until we reach the target node. Every time we discover a node, we inform the weightand the parent node, ie the last node we visited to get there from the source node. In this way, after thealgorithm is terminated, starting from the target node and iteratively, we find through the parent-nodes, the minimum path we are looking for. In case that the target point either does not exist on the map or exists but it is occupied, the algorithm informs that no path was found. The points of the minimum path then pass to local planning so the robot can start moving.

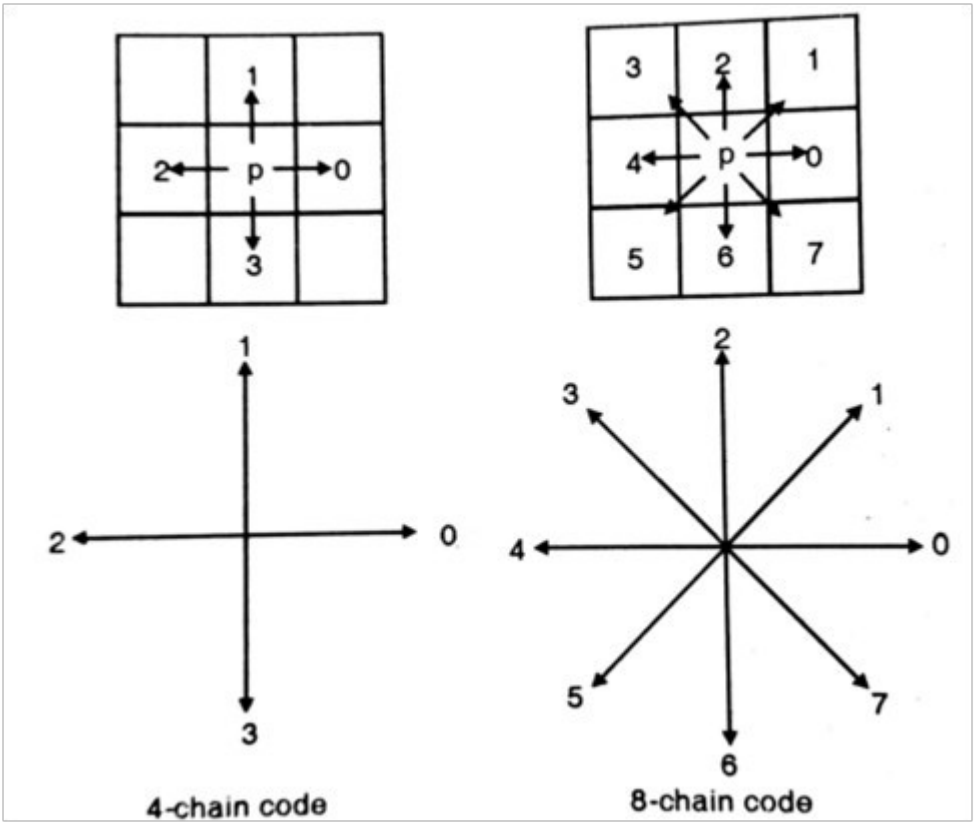

Image 4.2: 4-point and 8-point connectivity

4.2.2 Local Planner

The idea of the local path planner is that depending on the local goal position and the dynamic obstacles that may occur, velocity

values are calculated. At this point we can now get real-time information from the sensors unlike the global planning where we handled the a priori information that we have saved as a map.

If we now need the messages from the sensors, it is required initially to subscribe to the respective topics. So we subscribe to the topic where the laser publishes the LaserScan messages and on one of the two topics that give us information about the position of the robot.

rospy.Subscriber("/base_scan", LaserScan, self.clbk_laser),

rospy.Subscriber("/odom", Odometry, self.pose_callback)

or

rospy.Subscriber('/amcl_pose', PoseWithCovarianceStamped, self.pose_callback)

Assuming there are no dynamic obstacles, that is, obstacles that the robot does not already know their existence, the situation is very simple. Manipulating the points of the path in order, we implement velocity calculations individually, and therefore motion, to get to

each of them. So, for each point, we repeat the following process: the robot turns(if necessary) to look towards the target and moves forward to it. This process is done repeatedly and with small check values, until the target has been approached to the accuracy we want. The angular velocity given for the purpose of rotating the robot towards the target is calculated by the formula

*angular_speed = (desired_angle_goal - yaw) * K_angular*

where yaw is the orientation of the robot and desired_angle_goal is the desired angle of rotation resulting from the relation

desired_angle_goal = math.atan2 (self.y_goal - y_robot, self.x_goal - x_robot)

and K_angular is a constant that for our implementation is set to 0.2

As for the case that there is a new obstacle on the path, the robot must be able to avoid it effectively and continue moving on the path normally. To achieve this we must properly use the information we receive from the sensor laser. The LaserScan message, as discussed in the previous chapter, contains, has as a field an array with the distance values of objects the laser scan

perceives. By analyzing this information we can assess where the obstacle is and avoid it efficiently.

It was decided, therefore, to implement a common logic which is to split the scan spectrum into regions as shown in the figure, and to handle each one separately. So we have the regions right, fright, front, fleft, left from right to left. From all these regions we are not interested in the values in the left and right region as they do not affect motion of the robot in our implementation.

One issue was we did not use the same laser in simulation and on the real robot. In the simulation we used a laser with a minimum scan angle of -3.14 rad, a maximum angle of 3.14 rad and 720 values in the ranges range. For this reason we divided by 5 the total 180-539,that is, those rays that correspond to the upper half of a circle, so that to take only the regions we are interested in. This is how the areas came about:

regions = { 'right': min(min(msg.ranges[180:251), 10), 'fright': min(min(msg.ranges[252:323), 10), 'front': min(min(msg.ranges[324:395), 10), 'fleft': min(min(msg.ranges[396:467),

In each region, from all the values of the distances of each ray scan, we take the minimum distance. Also, distance values can be all infinity, so to handle this, we put each time to take the minimum between the nearest distance and 10.

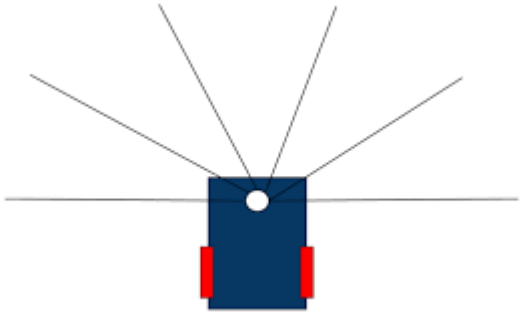

Image 4.3: Scanning spectrum divided into regions

Because there was a confusion in how ROS handles laser information, it should be noted that the laser perceives as the 0 angle the line that is right in front of it, ie theoretically the y axis if the robot placed at (0,0).

In this way, we added to the movement of the robot towards each target point, a check whether it an obstacle was found. This variable becomes True if an obstacle is found in one of the three regions that interest us, fleft, front, fright, at a shorter distance of 0.3. This value is an efficient value in the simulation related to the position of the laser on the robot.

In case an obstacle is found on the path, then firstly the robot stops and then it avoids the obstacle. Then the implementation reminds of the bug algorithms that the robot moves around an obstacle until it can continue its journey. The difference is that we are not interested in the robot to pass just above the intermediate points, it is enough to approach them so that it can stay as precisely as possible on the path that has been calculated, as long as it reaches the end point of the path. For that reason, as soon as the robot detects an obstacle, it initially rotates so much so that the

obstacle is now to its left or right depending on the region in which the obstacle is located. If an obstacle is located in the regions front, fright, front and fright, fleft and fright, front and fleft and fright, then the robot rotates counterclockwise, while if the obstacle is in the regions fleft, front and fleft, it rotates clockwise. When the robot no longer sees the obstacle, depending on whether the obstacle is now to its right or left respectively, the robot starts following the obstacle until it changes its orientation and its position by a constant 0.5. That was needed because many times it had not gone far enough, and it ended up trying to approach the inaccessible point continuously in the same way. So we change enough its orientation to ensure that it will try to approach the point in a different way. Just when this process is completed, a check is made on how much the robot has moved. Because, as mentioned, it is not necessary to pass exactly above the points if there are obstacles nearby or even just above the points, we want in these cases to consider that we are close enough and continue to the next point of the path. But because the robot can approach many times the same obstacle, some extra check was needed whether it should start moving to the next point and not to be done uncontrollably every

time. The uncontrollable way had significant errors and it could may not even get close to the final goal. So the following check has been added: it is considered that it is quite close to the current point and starts going to the next one on the path if the Euclidean distance of the robot from the previous point is greater than the Euclidean distance from the next point. About the first point of the path, but also the last one, in the case of a dynamic obstacle, we do a separate check. In the first case we just start moving to the next point, while in the second one we approach the goal as much efficiently as possible.

The values of the linear and angular velocities derived from the calculations, are published on the topic of velocity:

self.velocity_publisher = rospy.Publisher ('/ pioneer / cmd_vel', Twist, queue_size = 10)

Obstacle avoidance is implemented based on closed shapes and works more efficiently on the convex ones.

4.3 Simulation results

The program we implemented for the robot navigation is a ROS node. The logic of this work is when the robot recognizes a specific hand gesture to perform certain functionalities. In simulation, we tested only the robot's navigation.

To execute the program in Gazebo and in Rviz we firstly execute the two launch files on different terminals with the commands: *roslaunch pioneer_gazebo pioneer_world.launch* and *roslaunch pioneer_description pioneer_rviz.launch*Note

Note that since these are launch files, we do not need to execute the roscore command to run the ROS master firstly.

If performed successfully and there are no runtime connection issues the world and the robot will be displayed in the Gazebo as well as the display in Rviz. To see which nodes and which topics are running, we just execute the commands "Rostopic list" and "rosnode list" and the following results will appear.

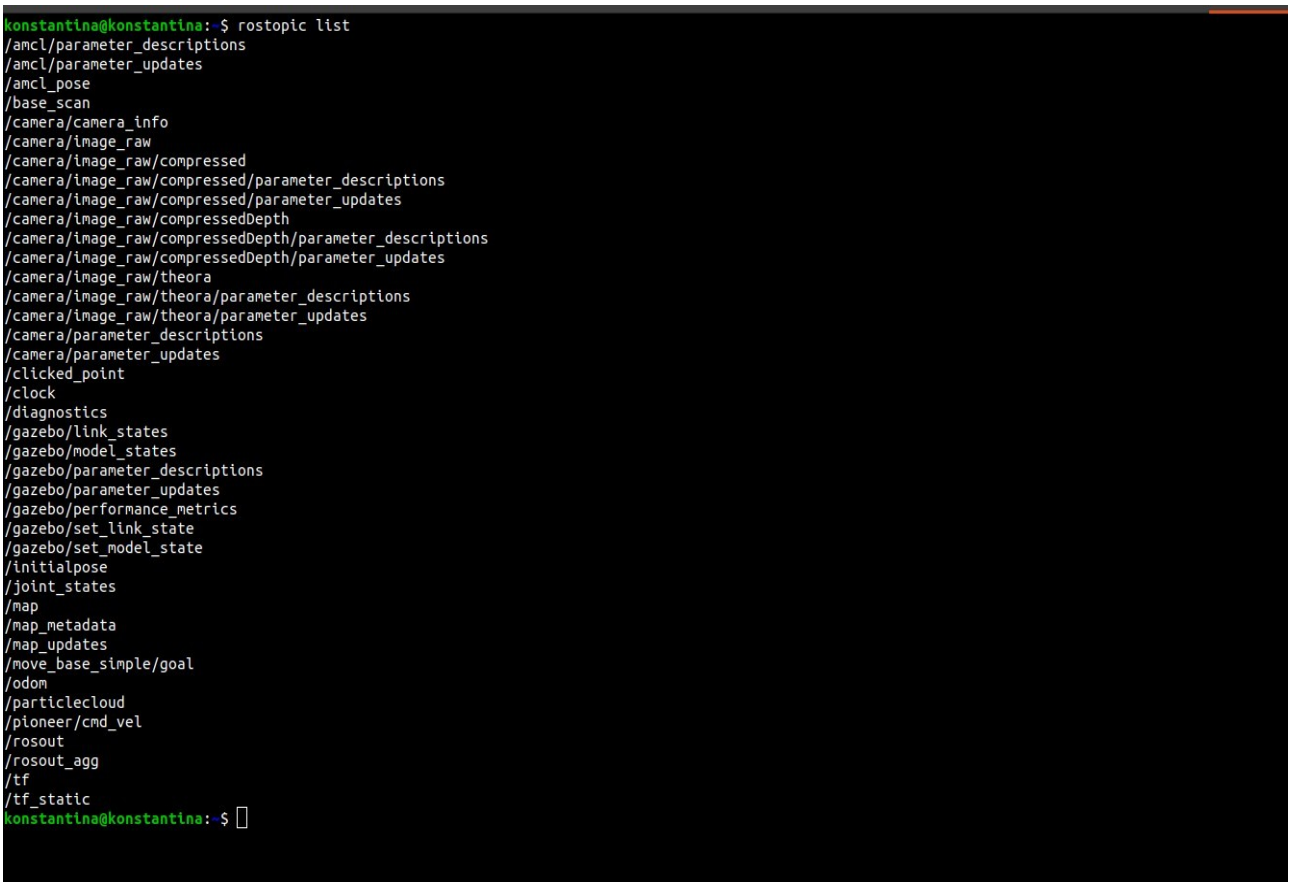

Image 4.4: “rostopic list” command results

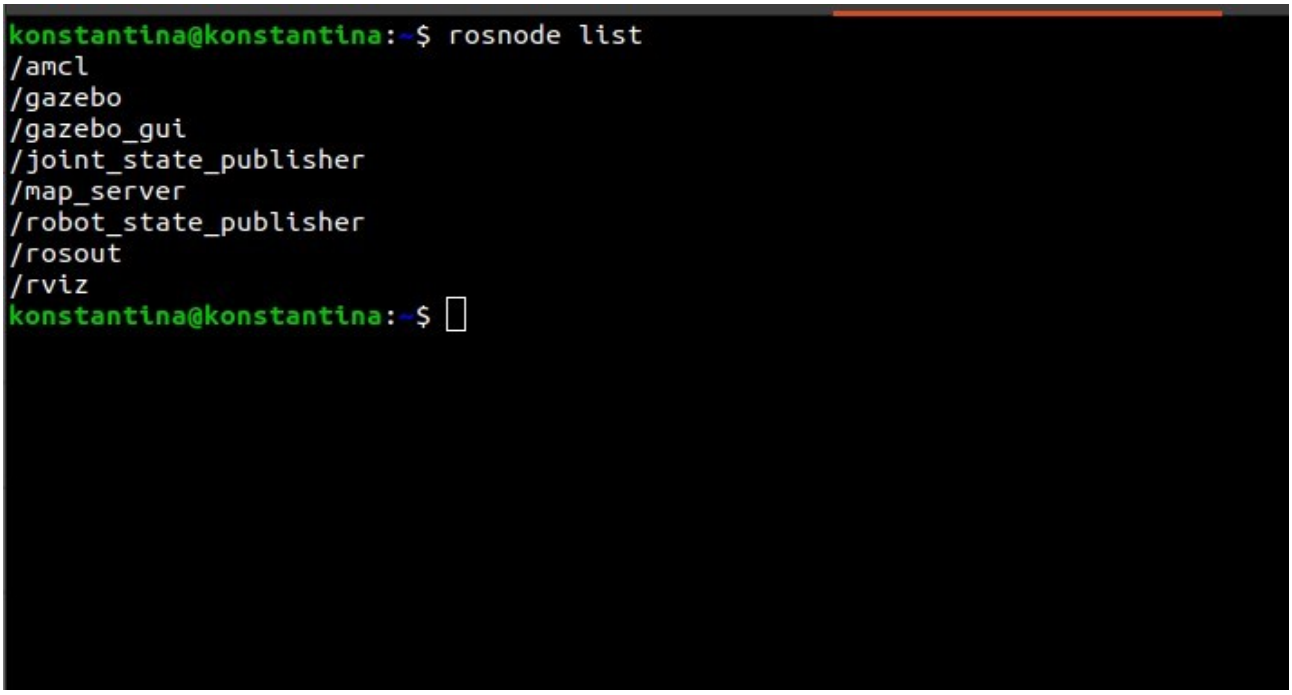

Image 4.5: “rosnode list” command results

The main nodes, in terms of functionality, are the amcl node, which estimates robot's position with respect to the map and map_server node which loads the files for the map information. Indicatively some important topics are "/base_scan" which includes information from each laser scan, the “/amcl_pose” which is the position and the orientation of the robot with respect to the map coordinate system

and the topic “/map” which includes information about which cells on the map are accessible by the robot.

It is worth noting that initially the laser scan in Rviz did not really coincide with the objects in space, because the robot in Rviz is not in the real place as it is in its "real" position in Gazebo. To avoid errors, we must always correct its initial position perceived by the robot so that the amcl node can be operated efficiently. This can be done in two ways, either setting as a parameter the initial position of the robot, if we already know it, at the execution command of the amcl node, either via Rviz with the 2D Pose Estimate option.

There is no problem with a slight deviation from the actual position, because the AMCL algorithm will correct the position estimate after the robot moves a little .Then we run our own node with the command *rosrun my_path_planningRobotNavigation.py.* Once the path to the target is calculated with GlobalPlanner, the robot goes through the Local Planner process and starts moving.

To display the points of the path that has resulted from Global Planning, Marker message is used and in blue color, while to see the trajectory that the robot finally performed, the Path message

was used, in green color, and its constant values are robot's position every moment.

For the position of the robot in the simulation both ways mentioned worked successfully. That is, the position and orientation from the /amcl_pose topic, with respect to the /map frame, and position and orientation from /odom topic, with respect to the /odom frame. However, it was observed that the position estimation from the amcl node sometimes when there was an obstacle near the wall deviated from the real position. Therefore, as long as the odometry does not have significant errors in the simulation level, it was finally used. The two frames are identical in the simulation and are in the (0,0) position.

A part of the implementation of the algorithm executed by LocalPlanning that was difficult was setting the right check values, since it was designed in theory and it was difficult to be implemented immediately correctly in action. For example, it was necessary to consider which check values work best for the laser distance checks, or the values given at velocities so the logic of avoiding obstacles works properly.

At the end after the required number of experiments, the suitable, for our implementation, values were defined. In all the examples we ran we had the robot in the starting position (-3,0), and before our path planning and navigation program is executed, what we first see in Gazebo and Rviz is what we see in Figures 4.6 and 4.7 respectively.

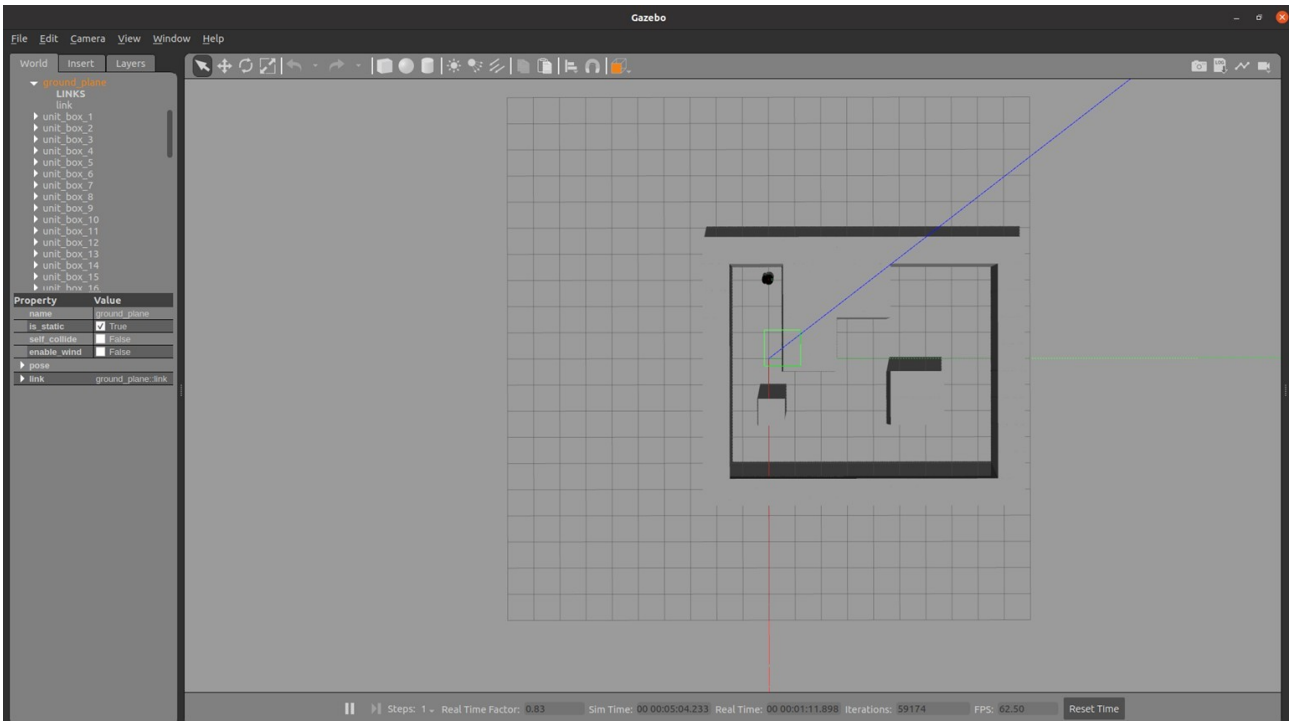

Image 4.6: The robot at its initial position in Gazebo

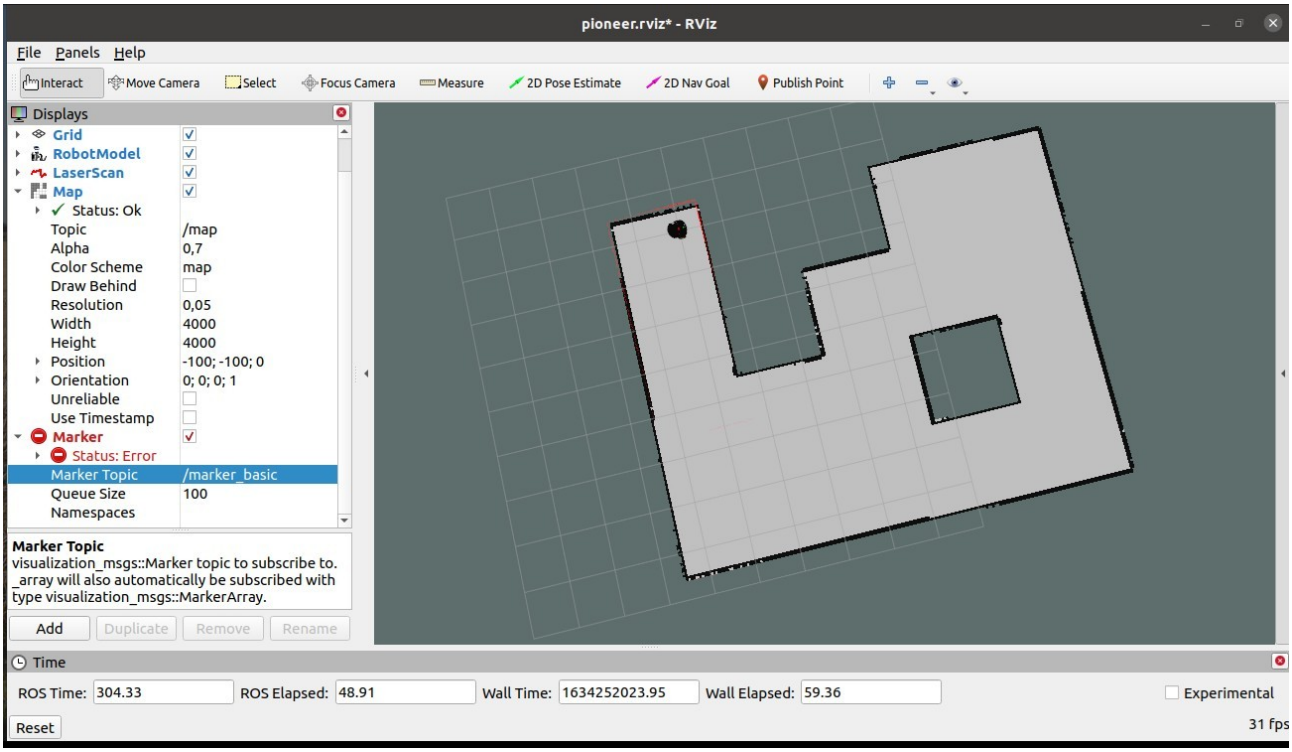

Image 4.7: The robot at its initial position in RViz

In Figure 4.8 a simple example of path planning and navigation is presented, without dynamic obstacles, and target point the point (4, 5). As it seems the robot has correctly followed the points of the path. The discrepancies are due to the tolerance to deviation from the points, to avoid possible errors.

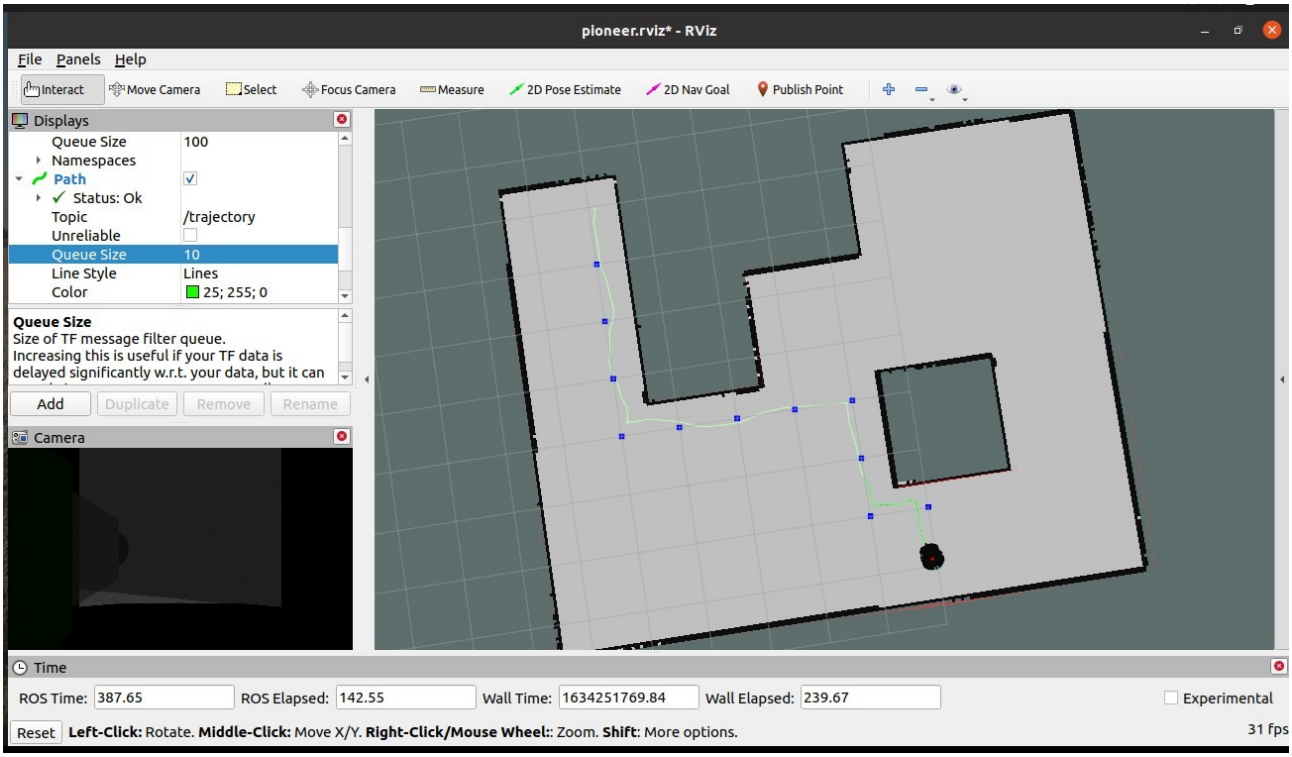

Image 4.8: Path planning results without dynamic obstacles

In Figures 4.9 and 4.10 we see another example of execution with a dynamic obstacle we have added to the robot path to point (4.0) and which was effectively avoided.

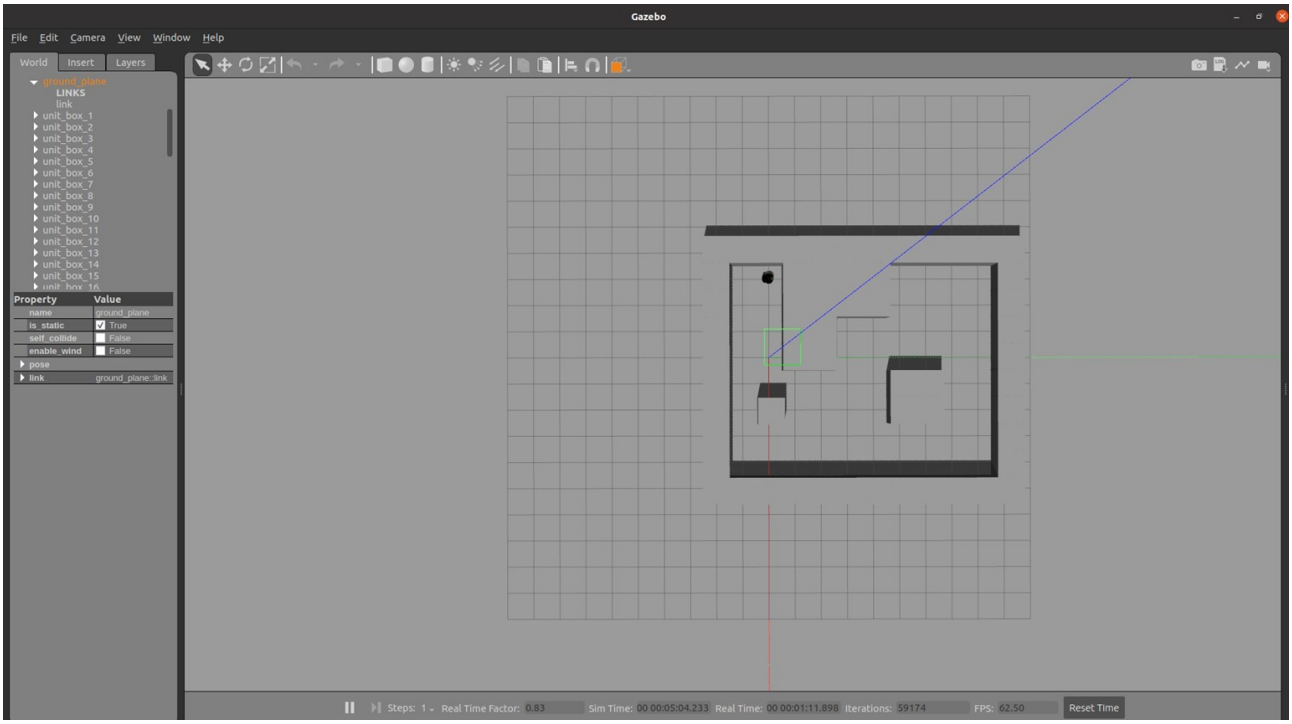

Image 4.9: Path planning with dynamic obstacle

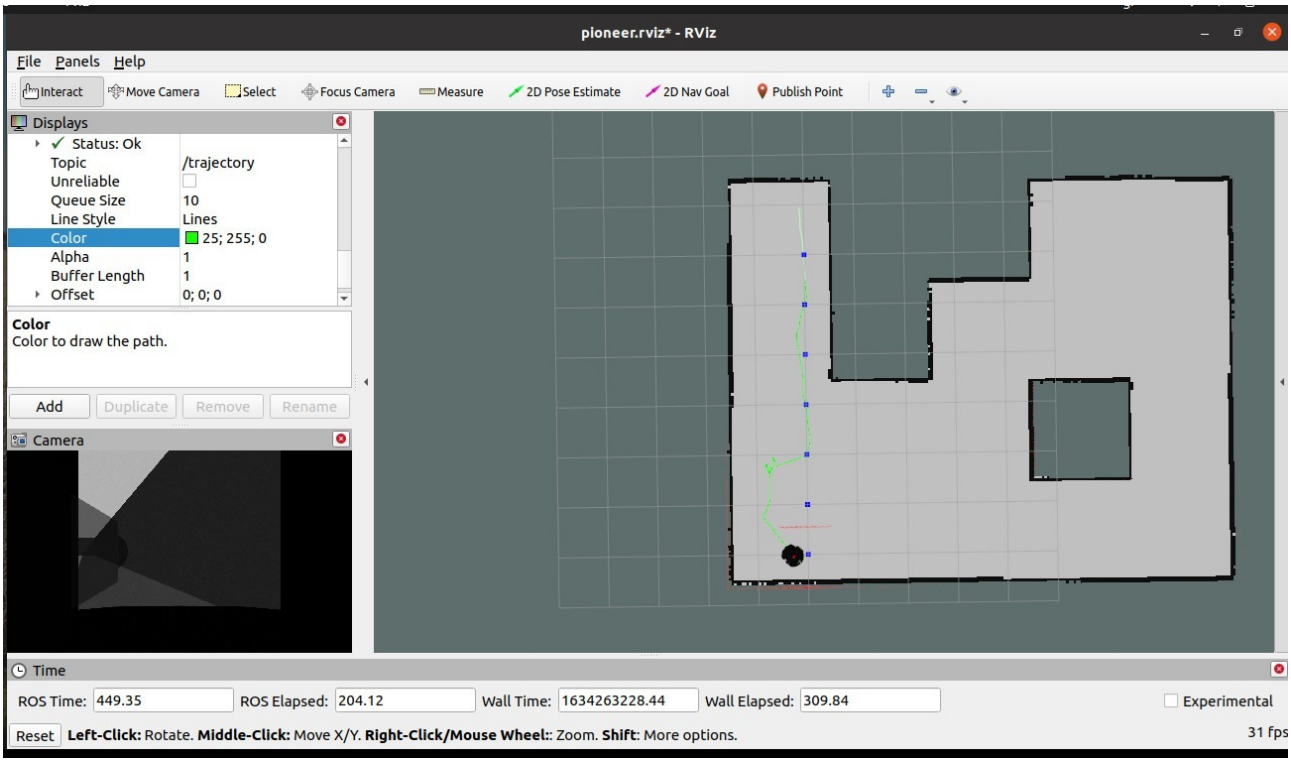

Image 4.10: Path planning results with dynamic obstacle

In Figure 4.10 we performed an example with the path points not actually accessible by the robot. These path points are located at the boundaries due to the process of rounding the digits of the accessible grid cells. However, there is no problem and the robot reaches the final goal correctly. This occurs because of the handling

we have on the part of avoiding dynamic obstacles in local planning. The robot remains close to its path points, avoiding collisions and reaching the target point (-2, 6).

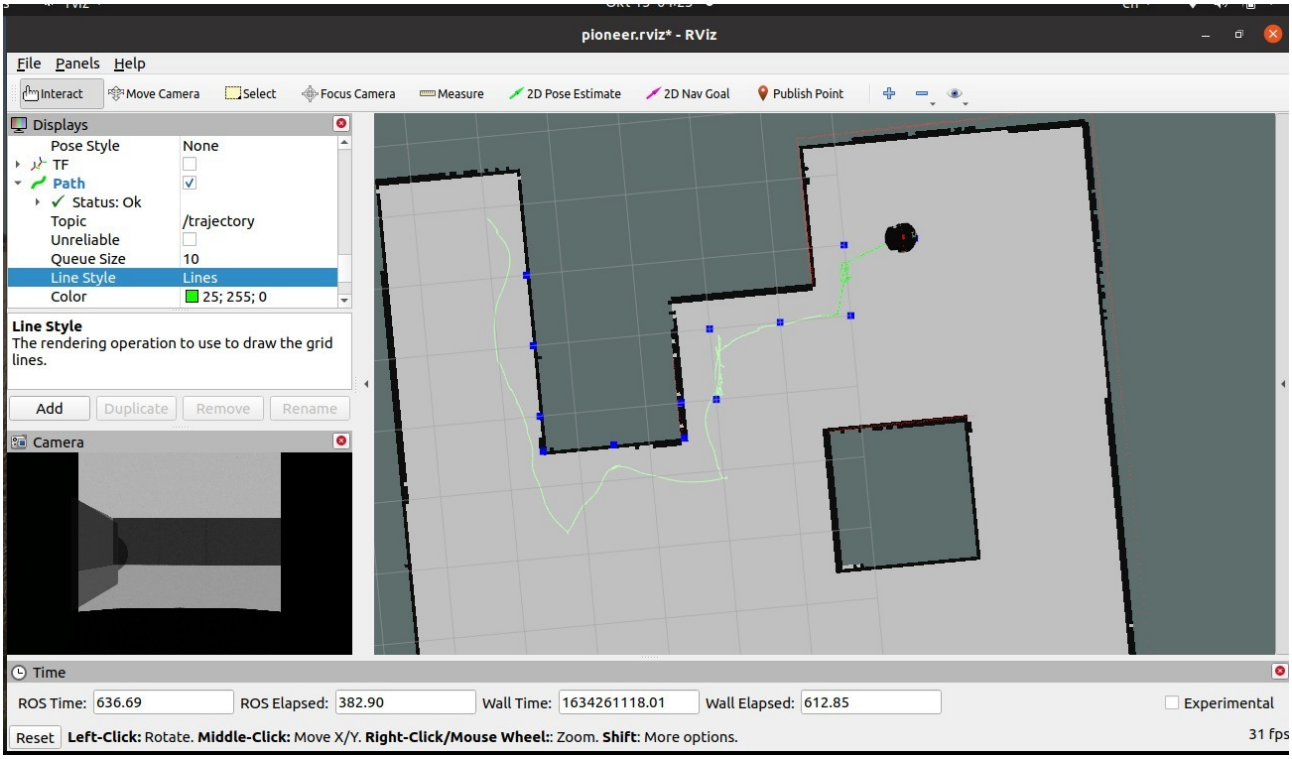

Image 4.11: Path planning results with path points near the boundaries

Chapter 5. Implementation on the real robot

5.1 Gesture recognition

For the gesture recognition part we used a project that uses the tensorflow and openCv libraries. We came to use it as it does not use a specific ROI, but it performs hand detection. This was necessary for our implementation, as when the robot moves we do not have the image display of the camera, unless an additional monitor is used. It also offers the ability to detect more than one human hand and it runs on the GPU. Transfer learning was applied to the project, ie the values of the neural network, which was used for the desired purpose, have arisen from an already trained network of a similar purpose.

The trained neural network used follows the ssd architecture with backbone mobilenet for image features extraction. It has been trained with a training set of gesture images and a csv file, which

contains the labels of these images, after they have been converted to an appropriate format, tfrecord, recognized by tensorflow.

One issue was how our program-node which is responsible for navigation would communicate with the gesture recognition program which works outside the ROS environment. There were two options, either to convert with the appropriate code the program so it can be executed as node and communicate via a topic with the navigation node, or to be executed outside of ROS and communicates with the navigation node via TCP socket. Finally, the first way was applied and the recognition program can now communicate with the navigation node based on ROS rules.

In the ROS environment, openCv does not have direct access to the image of the camera and we can get the image information only through the topic that the node of the camera driver publishes on. Therefore, we must first take the image from the topic and then give this information in the openCv functions. Nevertheless, the format of the image is different and is not compatible to do this directly. ROS, therefore, provides the node cv_bridge that converts openCv image format to ROS image format.

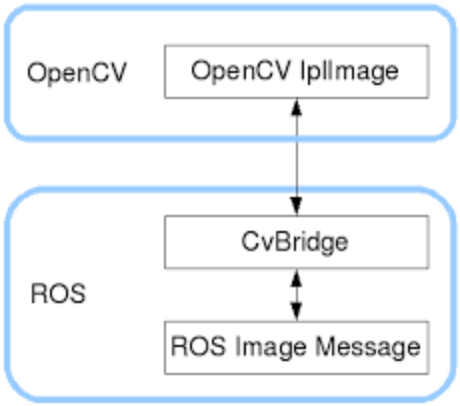

Image 5.1: Communication via CvBridge

The changes the code needed to be executed inside ROS were, at first, to create a node. This was easily done with the code line:

rospy.init_node ('gesture_recognition_node', anonymous = True).

In addition, this node had to subscribe to the topic of the image/csi_cam_0/image_raw. In this way every time a message is published on the topic /csi_cam_0/image_raw, in the callback function, image_callback, by using the cv_bridge library, we convert the ROS image to an openCv format image, with the following code line:

cv_image = bridge.imgmsg_to_cv2 (ros_image, "rgb8")

Then the gesture recognition program continues normally its functionality by taking the openCv image and giving it as input in the "frozen graph inference", ie in the neural network that has been loaded from memory with its stored values. The program performs both hand detection and gesture recognition. The gestures which are recognized are 5 and these are: "OK", "six", "thumb", "fist",“Other”.

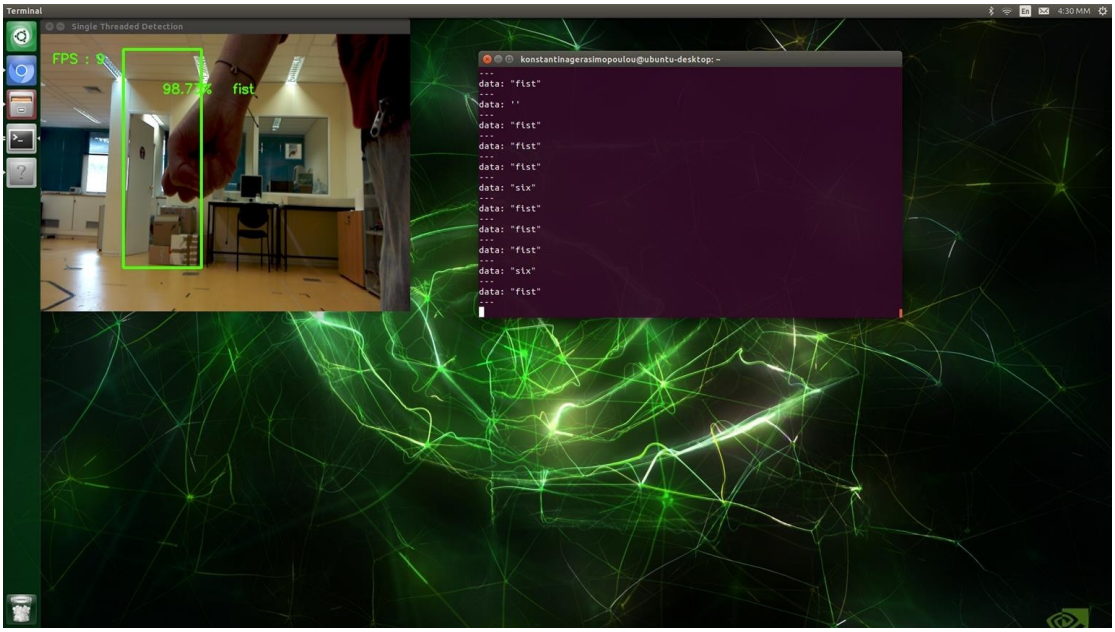

Image 5.2: Recognition of the “fist” gesture

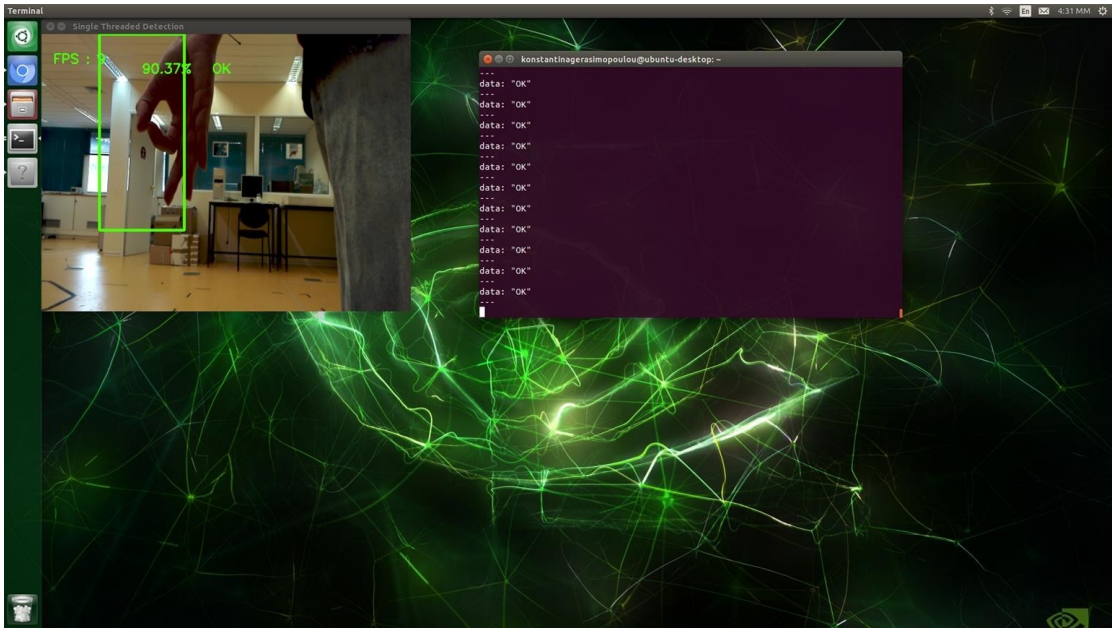

Image 5.3: Recognition of the “OK” gesture

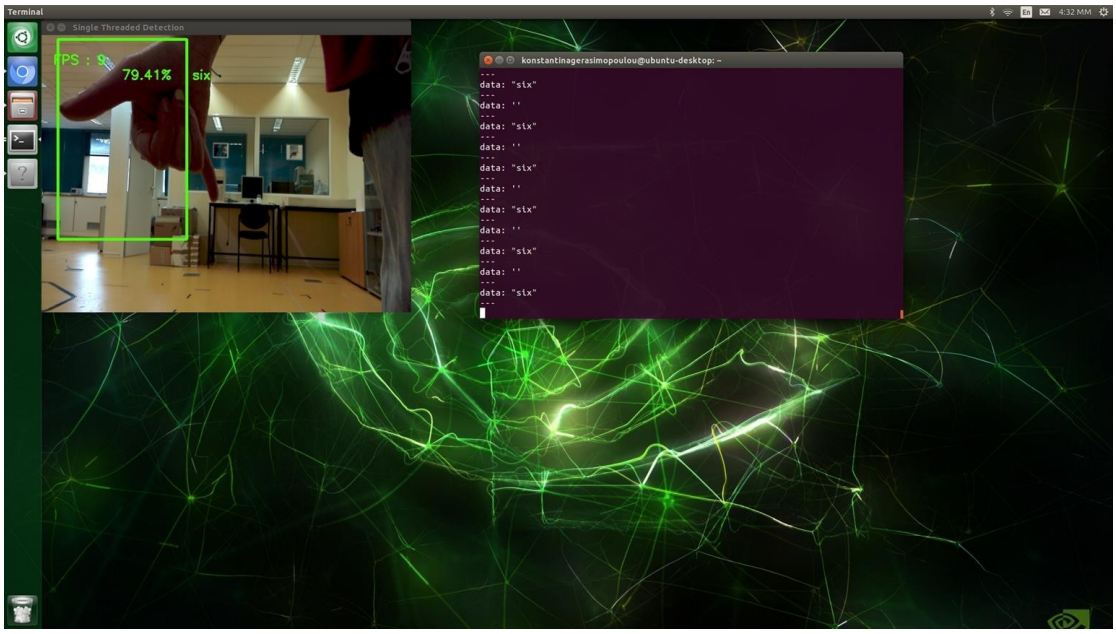

Image 5.4: Recognition of the “six” gesture

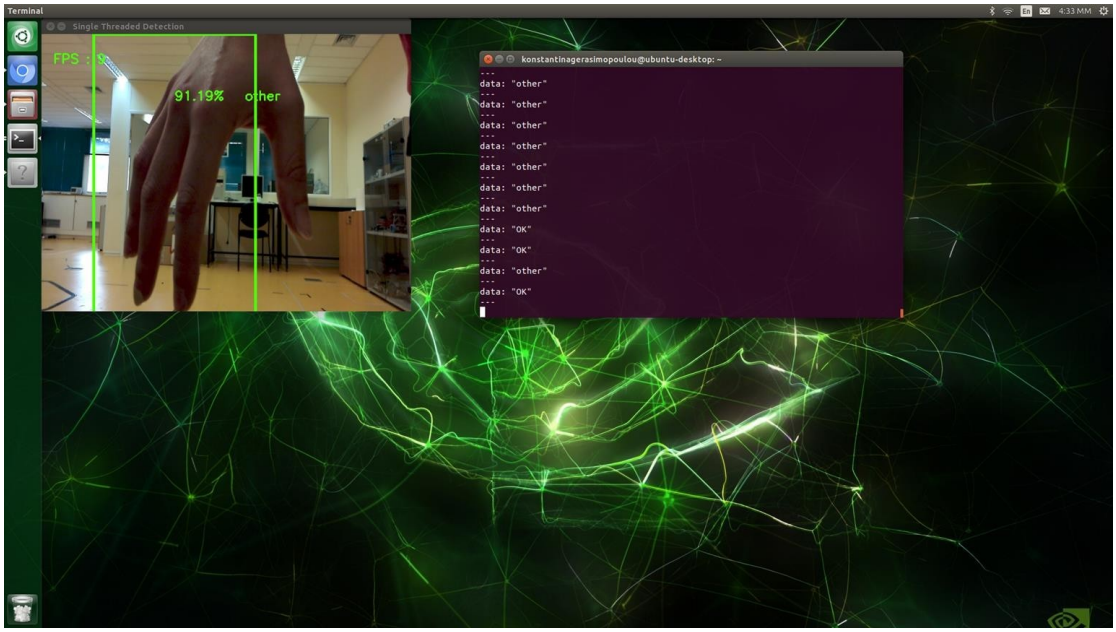

Image 5.5: Hand detection as “other” gesture

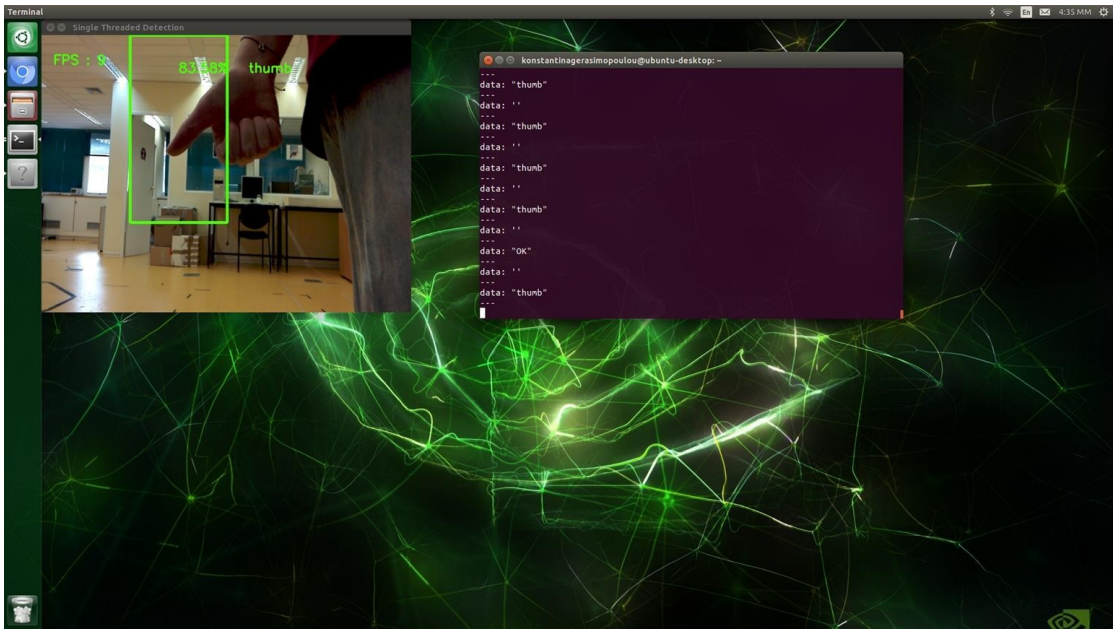

Image 5.6: Recognition of the “thumb” gesture

This result is published on the topic /gesture_class, which we have created, as a String message. At any time in the terminal we can see the publishing information with the command:

rostopic echo /gesture_class.

We get this information from this topic then from navigation node. Specifically for our experiments, we used the “OK” gesture and the “six” gesture. So after we have subscribed to the gesture_class topic from the navigation node, to the callback function, which is constantly running as a thread, we make the suitable checks for the gestures we have used. In the “OK” case we set as a goal for the navigation the point (3, 3), while for the gesture "six" robot’s moving is terminated.

We run this node that we created with the command *“rosrungesture_recognition gesture_recognition_in_video.py”* since we have, firstly, turn on the camera with the command *“roslaunch jetson_csi_camjetson_csi_cam.launch width: = x height: = x fps: 20 ”*. For our work it was observed that in the part of gesture

recognition, the fps value with the best results were 20 fps, so that was actually used.

5.2 Experimental results

Since information about the environment is required for the implementation of the navigation we needed the map of an our choice space. This way, we have the knowledge of the points accessible by the robot but it is also required for the amcl node if it has been decided to be used to get the robot's position at any given time. For this reason, the main and primary requirement was the mapping of our university lab where the robot is located.

There are many methods for implementing mapping. One of them is the SLAM (Simultaneous localization and mapping) algorithm during whose the robot moves to fully discover the space, and at the same time it estimates its position with respect to the map frame. In SLAM algorithm both the camera and the laser can be used, however the latter has a greater accuracy of results.

We used the SICK LMS 200 laser for the mapping process, as previously described, and from the ROS gmapping package the

slam_gmapping node [17], from which the map is represented as a two-dimensional Occupancy Grid.

More specifically for the procedure we followed, we executed the nodes RosAria to start the communication with the robotic base, which is waiting for velocity commands, and the node sicklms to use the Sick LMS laser. Next, we used an already implemented ROS package, theteleop_twist_keyboard, from which we can give velocity commands in the /cmd_vel topic of velocities, directly from our keyboard. This is required as the robot must move enough to create the map effectively.

To be able to see the results of the mapping and to decide whether the trajectory performed by the robot is sufficient to create the map or if it needs to move more to complete scan the space, we use Rviz, as shown in Figure 5.7

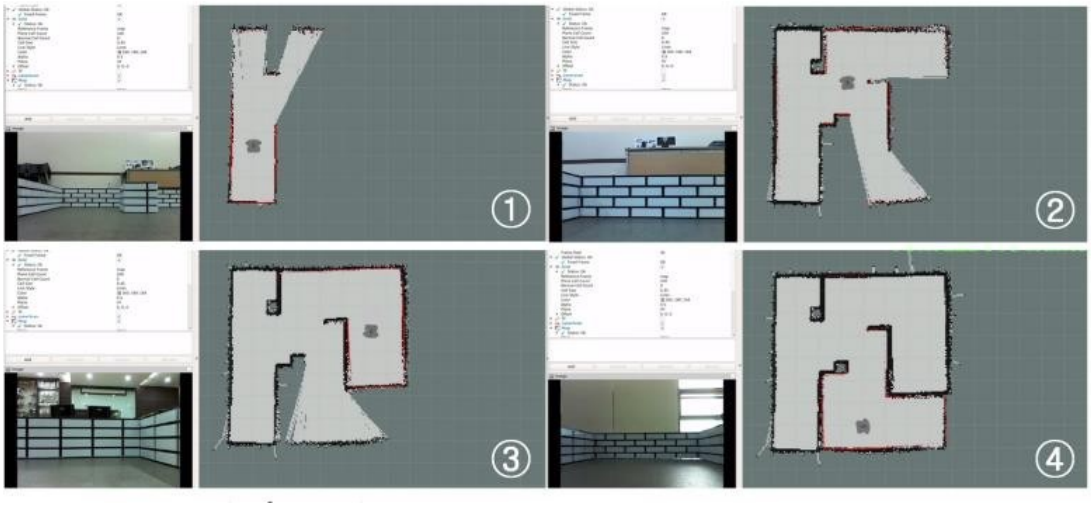

Image 5.7: Mapping process in ROS

Executing the corresponding rviz node, we select the TF display type, so that we can see the position of the robot in the space and the LaserScan display type, with the topic /scan selected, which the laser node publishes on. When the SLAM is executed, the map frame will be established at the current robot position in the environment.

The position of the map frame is now considered to be at the point (0, 0). After we have moved the robot enough and the map is represented correctly, the next step is to save it for its future use.

This will be done with the map_server package whichprovides the map _saver command-line

rosrun map_server map_saver -f mymap

where mymap is the name of the map. It is also possible to change some parameters for the mapping results. We used the default values.

The map is now stored on our computer in the form of two files. The "* .pgm" file is the image of the map, ie what it will look like in Rviz, and the "* .yaml" file that contains information about the parameters we used for the mapping process. To use it, we just need to run the map_server node with the “* .yaml” map file as follows:

rosrun map_server map_server mymap.yaml.

This node will then publish an OccupancyGrid message on the /map topic. As previously analyzed in the part of our algorithms implementation, by subscribing from our node to this topic, we can use this message to create points in the manner required to be handled by our algorithms.

In addition, as previously mentioned, a 360 degree laser was used in the simulation, while the SICK LMS 200 has a 180 degree scanning range and a ranges array with 180 values. To have the regions right, fright, front, fleft, left we divided the 180 values into 5 regions of 36 values as follows:

regions = { 'right': min(min(msg.ranges[0:35]), 10), 'fright': min(min(msg.ranges[36:71]), 10), 'front': min(min(msg.ranges[72:107]), 10), 'fleft': min(min(msg.ranges[108:143]), 10), 'left': min(min(msg.ranges[144:179]), 10), }

The next issue was how we would eventually get each moment the position of the robot. At the level of simulation it was observed that both the position from odometry, with respect to the /odom frame, but also the position from the amcl node, with respect to the map frame, worked just as efficiently. The position from odometry had no errors and since the two frames were identical, we had no problem executing our algorithms. The position from the /amcl_pose topic sometimes showed deviations in cases of dynamic obstacles, however after small movements the errors were

corrected .As for the real robot, it was observed that there are significant errors in calculation of the position by the odometry, a fact that made our path planning algorithms to not work properly. Once we run the basic nodes of the robot and before we do anything else, we must correct the initial position estimation of the robot. This is what we do from Rviz with the 2D Pose Estimate option. As the robot will move and it is not functional and easy to have a monitor connected to it, we use our computer helpfully to run the rviz node and do the debugging. Our computer communicates with the robot via the IP addresses as, previously, described. The other nodes can be run on the robot. Note that if we know that the robot always has a fixed starting position and we know exactly what this is, we can set it, also, at the amcl node execution.

We observe, then, as shown in Figure 5.8, that in the beginning the odom frame is where the robot is, ie the base_link frame. However, when the robot had moved in space, the odom frame changed its position. This happened because of the errors in odometry, so the robot can not determine exactly how much it moved.

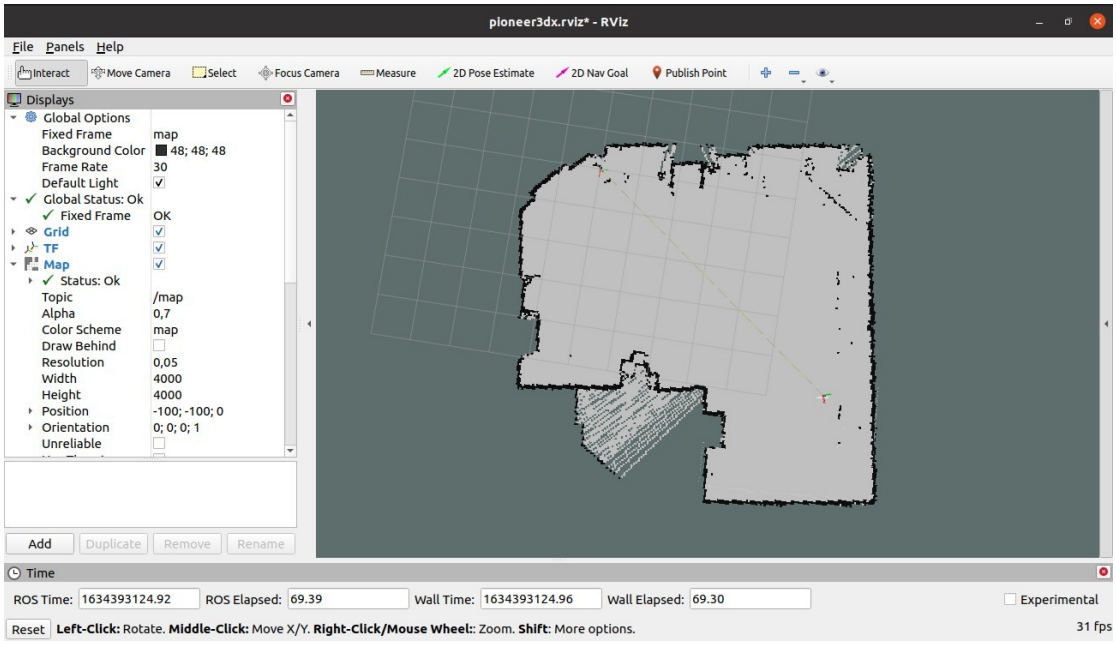

Image 5.8:Initially the odom and base_link frames have the same position

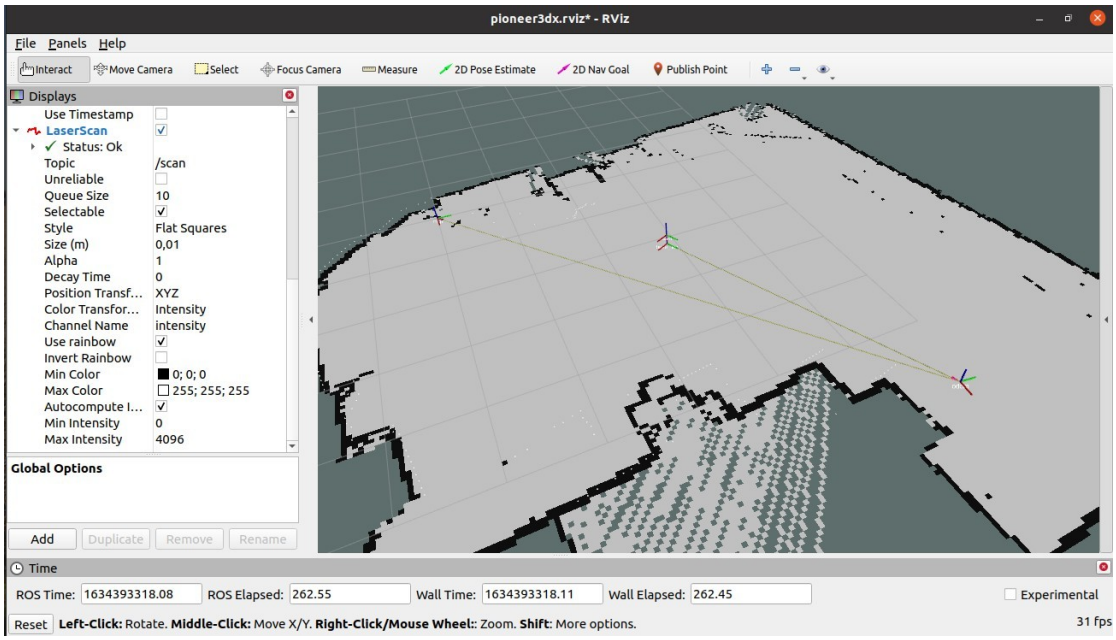

Image 5.9: Odom frame has changed its position due to odometry errors

Nevertheless, even if there were no errors, the robot should always be in the initial position (0, 0), ie the odom frame should be exactly where the map frame is. This is required as the points of the space, which we have found through the map_server node, are calculated with respect to the map frame.

So we went straight to the solution that the amcl node gives us. In this case, although the results were quite satisfying, in terms of the position accuracy, due to the fact that amcl_node publishes on the topic/amcl_pose with a fairly low frequency, in contrast to the requirements of our local planning algorithm, made the whole process quite slow. After searching for solutions to this problem, we found two solutions which are suggested. One solution was to modify the laser scanning frequency. It was quite successful for a short time but generally it is considered to be a not appropriate way as it affects the quality of the position estimation. The next solution, and the best one, was to get the position from the messages in the tf topic, ie from the transforms between the map frame and the base_link frame. However, on the tf topic the

transform odom → base_link is published, from the RosAria node. Respectively, th eamcl node calculates the map→ odom transform and then publishes it on the tf topic. Since we need ~~m~~map base_link transform, we created a tf.TransformListener that at all times calculates the map →base_link transform. That way, we can get the exact position and orientation of the robot.

With initial robot position away from the map frame, as shown in Figure 5.8 from previously, all three ways of finding the position were examined in detail. We moved the robot like shown in Figure 5.9, and we obtained the results of its position estimation for each of the three methods as shown in the following pictures:

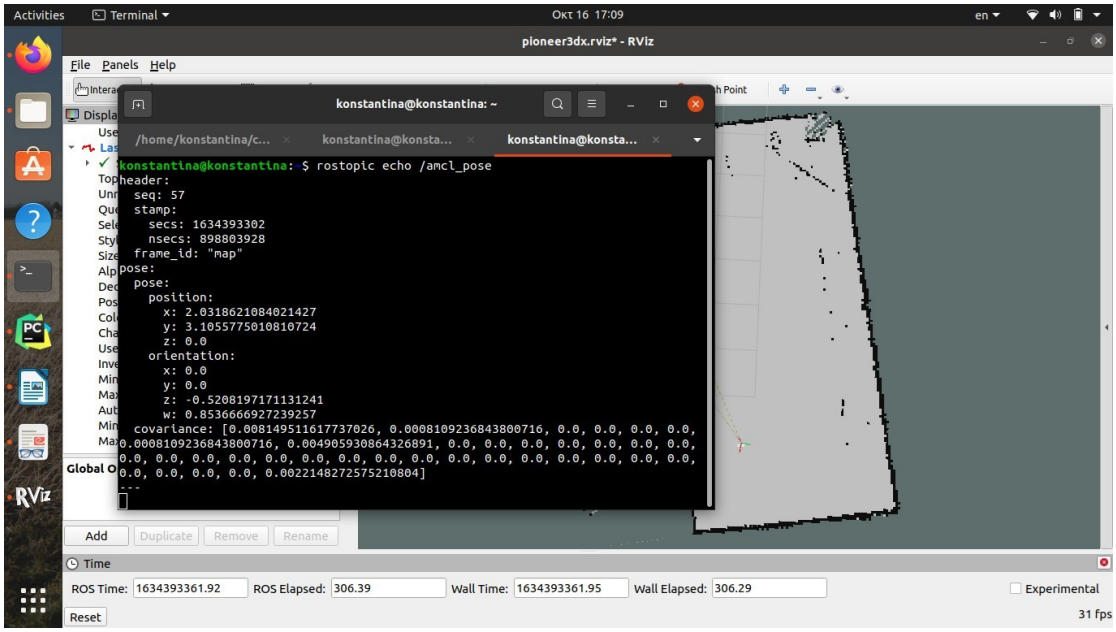

Image 5.10: Robot's position according to amcl node

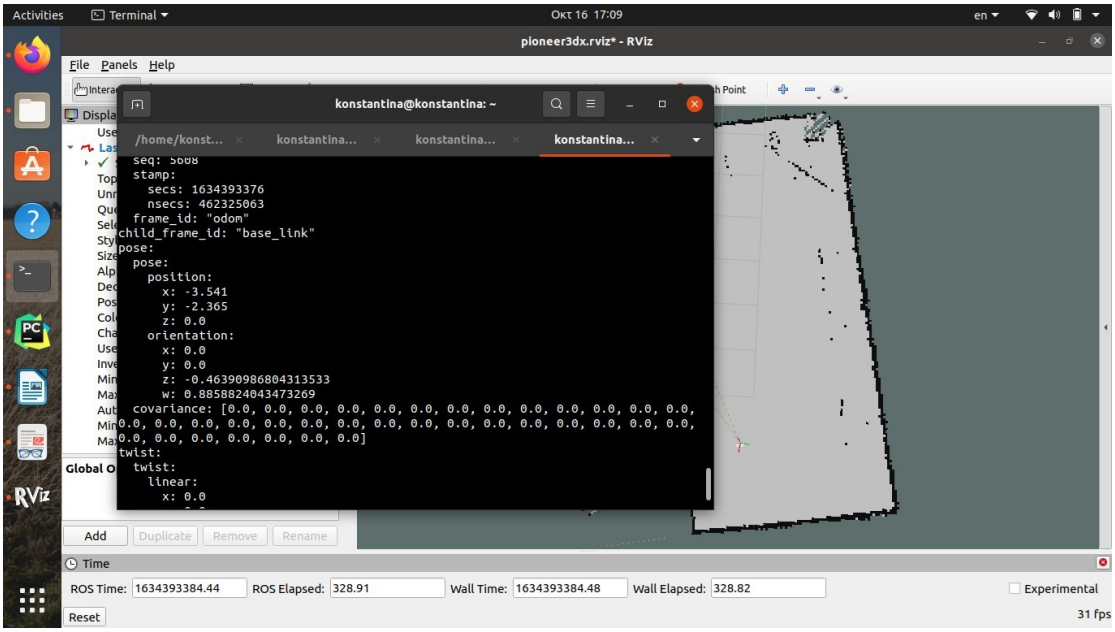

Image 5.11: Robot's position according to odometry

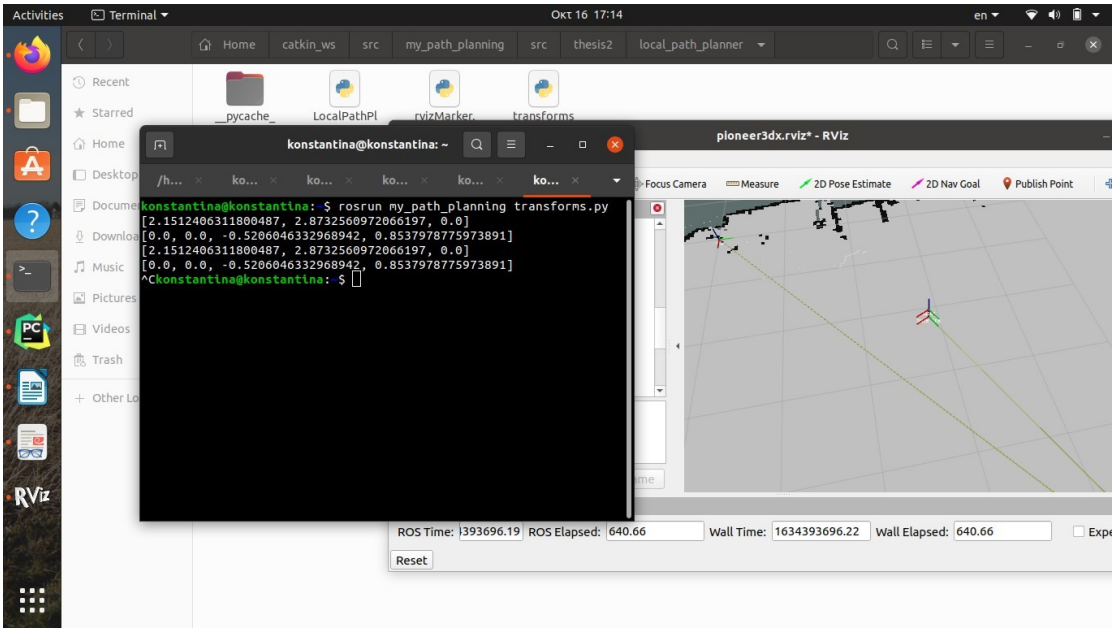

Image 5.12: Robot's position according to *map* → *base_link* transform

To get the exact position of the robot in space we used the Rviz “publish point” option. In one terminal we executed the command: *rostopic echo /clicked_point* and then we clicked on Rviz right on the robot frame, base_link. The result is shown in the image below.

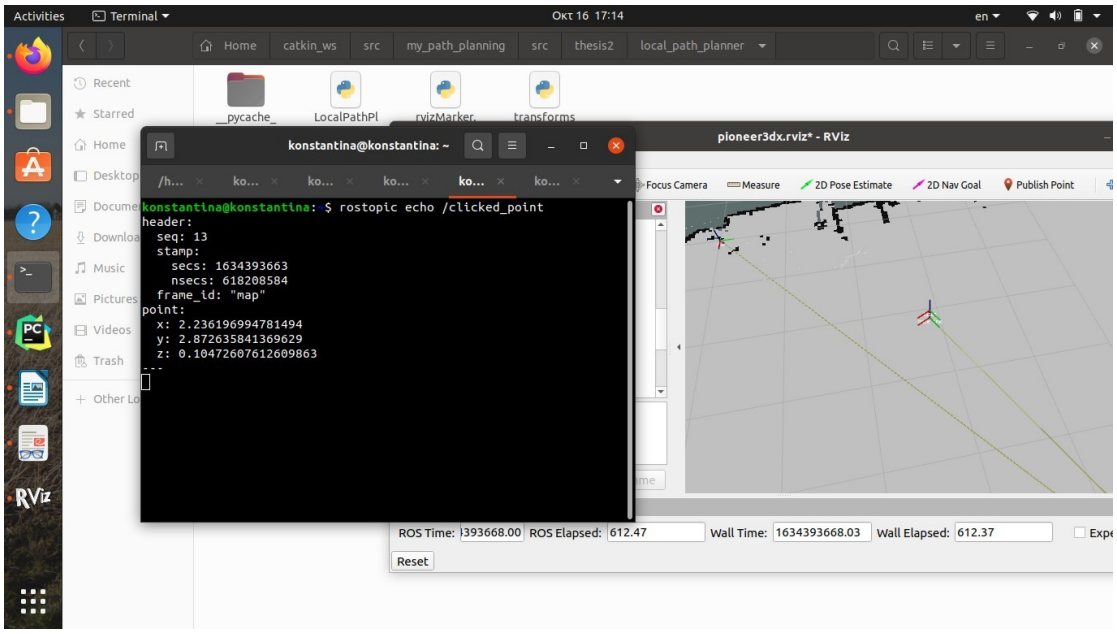

Image 5.13: base_link frame position with respect to the map frame

We observe that the most accurate method is to get the position from the transform of the frames. It is also worth noting that the results from odometry are completely different from the real

position, not only because of the odometry errors during the robot motion, but mainly because of the initial position of the robot, and therefore the odom frame was not at the same position as the map frame.

Now we can start the path planning process with our algorithms, waiting for a specific gesture from the topic on which the gesture recognition node publishes. The robot starts moving towards to a point of our choice as soon as the “OK” gesture is recognized and it gets to the point, unless it recognizes the "six" gesture, where it stops.

Specifically, we chose the point (4, 5) as the goal point.In Figure 5.14, the "OK" gesture has been recognized by the robot and the path points have been calculated from global path planning. The points of the pathare (-1, 2), (0, 2), (1, 2), (2, 2), (3, 2), (4, 2), (5, 2), (5, 3), (5, 4) and are displayed in RViz via a Marker message, in blue color.

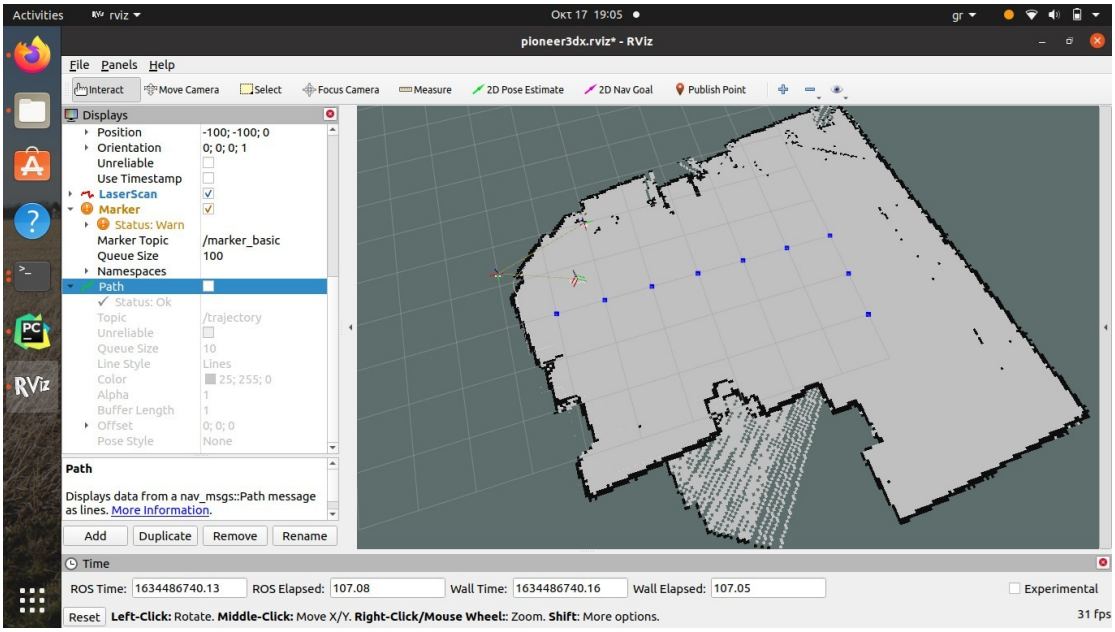

Image 5.14: Path points generated by global path planning

Then the robot starts moving with local path planning. In figure 5.15 the navigation is completed, while the trajectory performed by the robot is shown in green color, via a Path message.

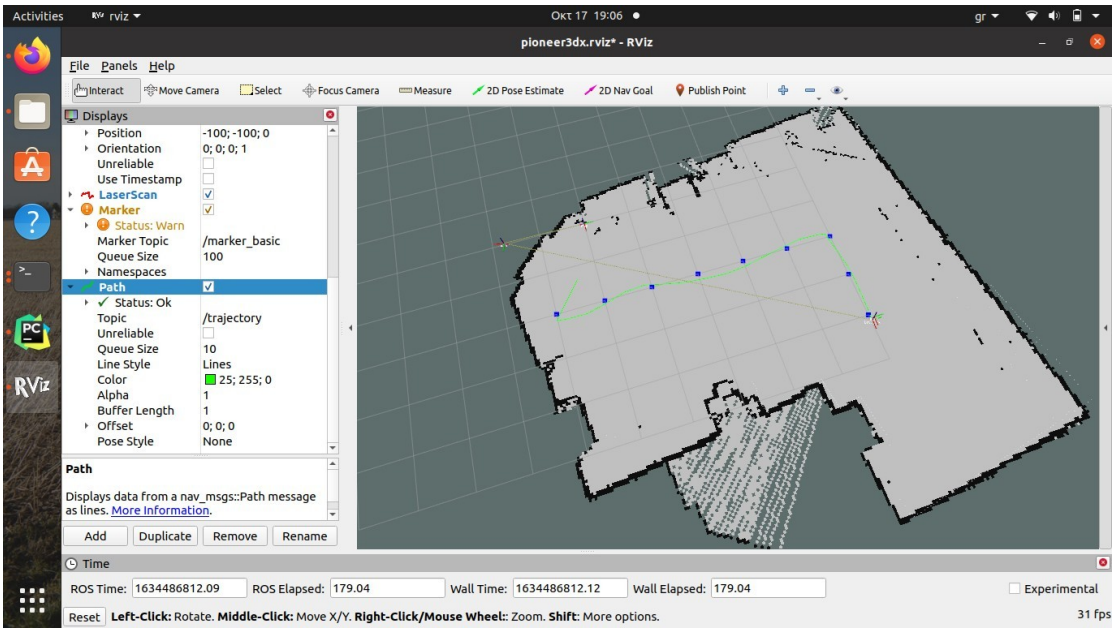

Image 5.15: Local path planning results

Below is the execution of the navigation with a dynamic obstacle. To the left of RViz we echo the information of /gesture_class topic where the recognized gestured is published. Our node-program subscribes to this topic and waits until it receives a specific message to start the navigation process. In figure 5.16 the "OK" gesture has been recognized and the navigation starts. In figure 5.17 global path planning has calculated the points of the path and local path planning starts moving the robot. After the robot moves a little we stop it by showing to the camera the “six” gesture, as shown in

Figure 5.18. Then showing again the "OK" gesture, the robot starts again its navigation, running fromagain the global path planning and recalculating the path points.

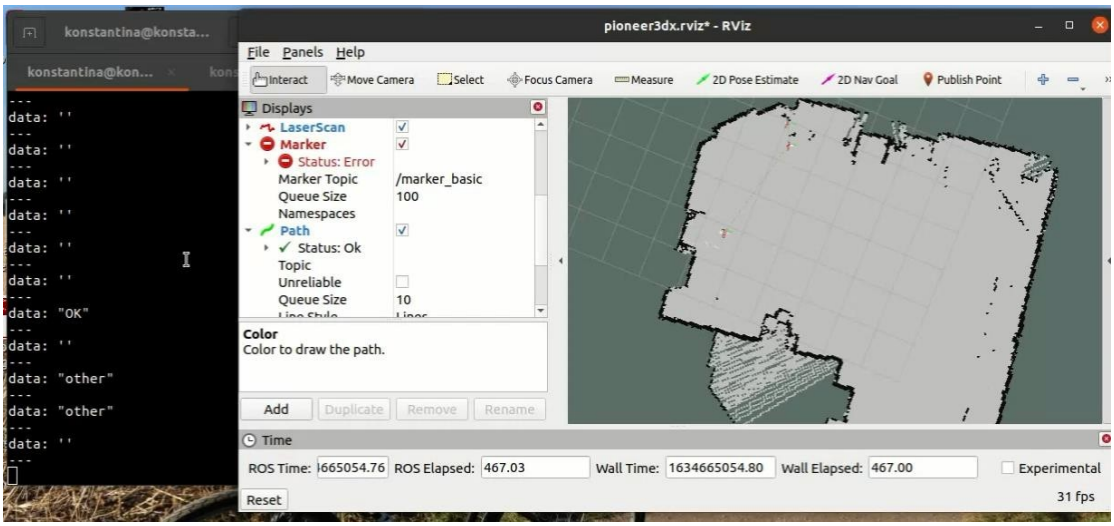

Image 5.16: “OK” gesture has been recognized and navigation to (4,4) starts

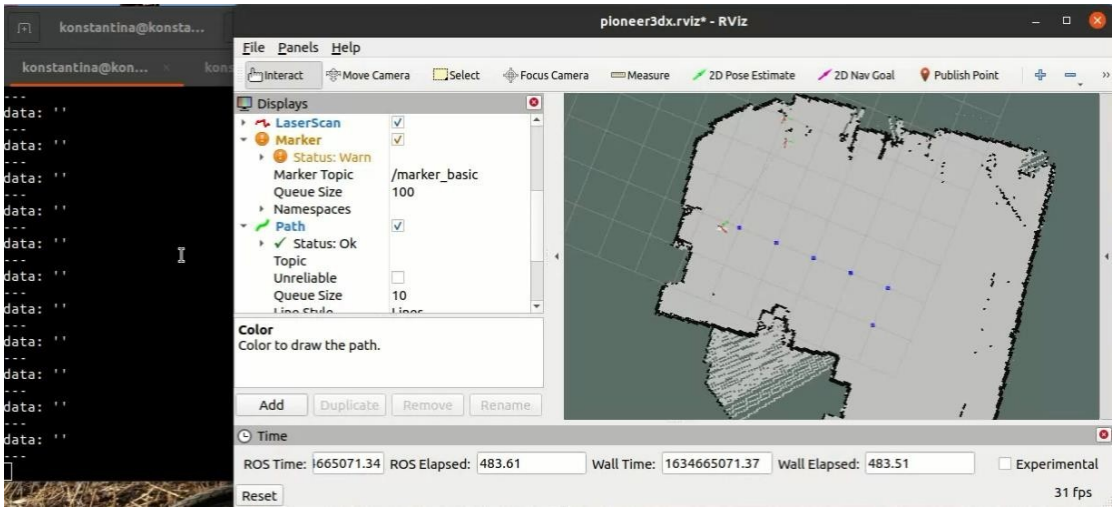

Image 5.17: Path points generated by global planner

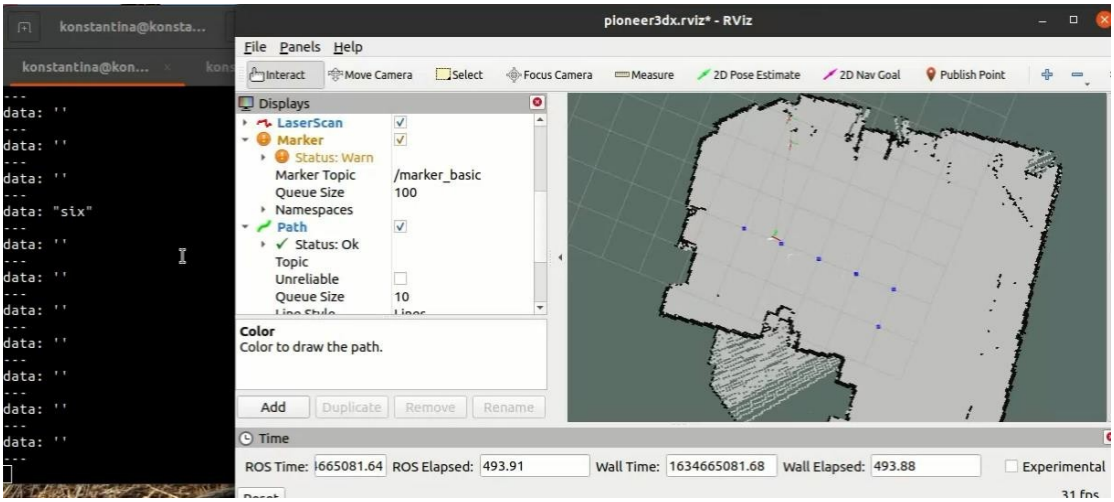

Image 5.18: Robot stops after recognizing the “six” gesture

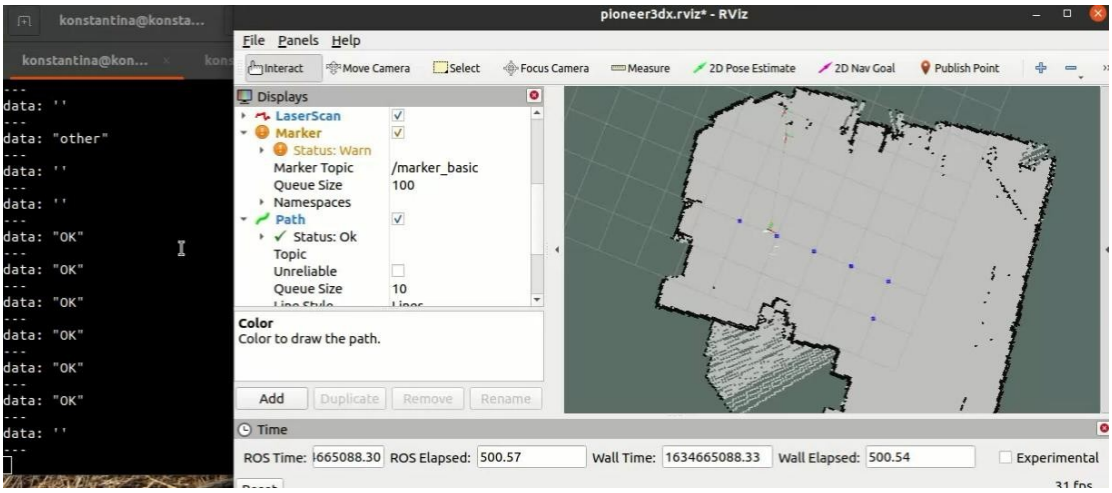

Image 5.19: Robot starts again its navigation after recognizing the “OK” gesture

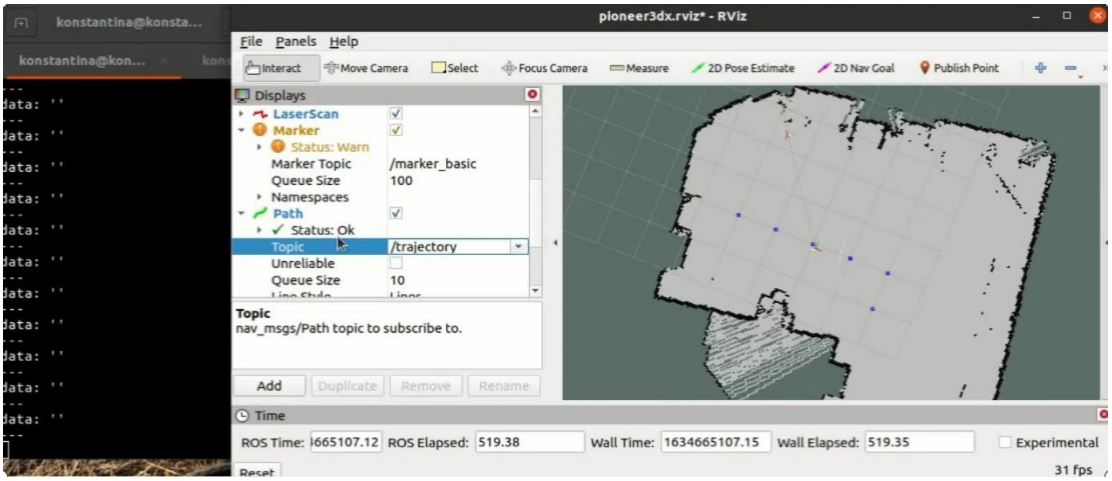

Image 5.20: Robot detects the obstacle and starts the avoidance process

The robot while moving to the point (3, 3) detects the obstacle and begins the avoidance process. The final trajectory performed by the robot is shown in the image 5.21.

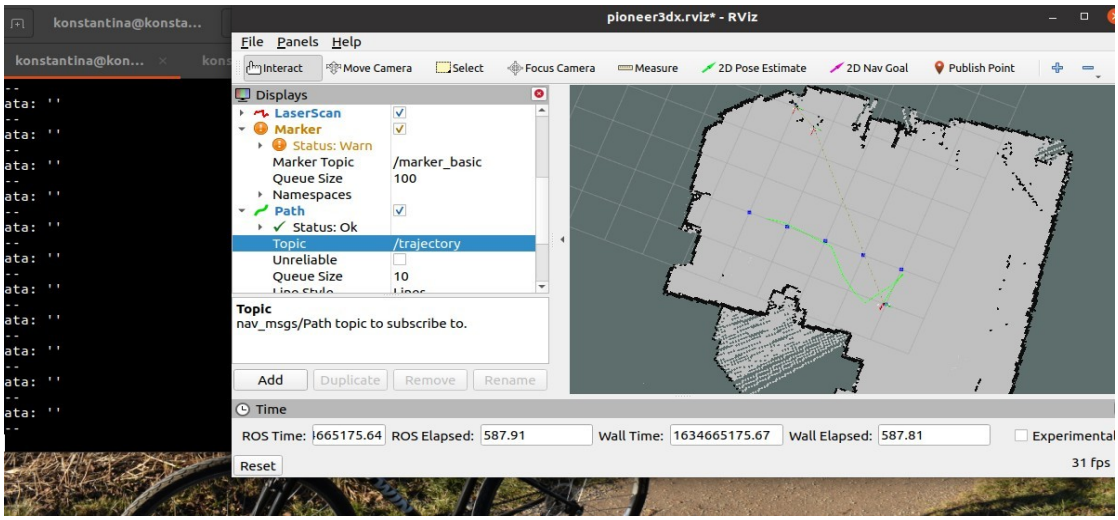

Image 5.21: Robot's trajectory to the (4,4) point

Image 5.22: “OK” is recognized and navigation to (4,4) starts

Image 5.23: “six” is recognized and robot stops its navigation

Image 5.24: “OK” is recognized and robot starts again the navigation process

Image 5.25: Robot detects the obstacle and starts its avoidance process

Image 5.26: Robot moves near the obstacle while avoiding it and continues to its path

Chapter 6. Summary and future extensions

During this thesis, a complete navigation system was implemented and tested, where a differential two-wheeled robot interacts with people and moves according to commands received through gestures. This implementation can be adjusted successfully to any indoor environment after the mapping process with the method we described. The functionality can be extended with appropriate positioning methods in outdoor spaces .In addition, the way the robot communicates with the person can be modified in various ways. One of them could be to perceive natural language commands using a microphone sensor. Finally, with the implementation of a comprehensive and detailed communication with humans, the robot can be a companion robot and perform specific tasks that will be defined from the commands it receives.

References

[1] ROS Navigation Tuning Guide https://kaiyuzheng.me/documents/navguide.pdf

[2] Real-time 3D hand gesture interaction with a robot for understanding directions from humans https://www.researchgate.net/publication/224256248_Realtime_3D_hand_gesture_interaction_with_a_robot_for_understanding_directions_from_humans

[3] Principles of Robot Motion: Theory, Algorithms, and Implementation https://ieeexplore.ieee.org/book/6267238

[4] Pioneer P3-DX https://www.generationrobots.com/media/Pioneer3DXP3DX-RevA.pdf

[5] Nvidia Jetson TX2 https://www.nvidia.com/en-us/autonomousmachines/embedded-systems/jetson-tx2/

[6] LiDAR 73 https://en.wikipedia.org/wiki/Lidar

[7] ROS Official Webpage https://wiki.ros.org

[8] ROSARIA Package Summary https://wiki.ros.org/ROSARIA

[9] ARIA Library Source https://github.com/srmq/ARIA

[10] ROSARIA Package Source https://github.com/amor-ros-pkg/rosaria

[11] Sicktoolbox Package Summary https://wiki.ros.org/sicktoolbox

[12] Sicktoolbox_wrapper Package Summary https://wiki.ros.org/sicktoolbox_wrapper

[13] Jetson_csi_cam Package Source https://github.com/peter-moran/jetson_csi_cam

[14] Pioneer3dx ROS kinetic and Gazebo7 simulation http://jenjenchung.github.io/anthropomorphic/code.html

[15] Dijkstra Algorithm https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

[16] map_server ROS http://wiki.ros.org/map_server

[17] SLAM Gmapping Package Summary https://wiki.ros.org/gmapping