Intelligent Systems and Robotics Laboratory course

Cerone Luigi 261212

D'Ascenzo Andrea 261123

University of L'Aquila
Department of Computer Science
DISIM

1.0 Final Version

L'Aquila, 2019/20

Timeline

Activity Name	Description	Start time	End time
(A) Documentation	Project documentation.	10/11/2019	24/11/2019
(B) Robot	Robot design.	10/11/2019	24/11/2019
(C) Simulation	Simulation of the robot with Gazebo.	20/11/2019	31/1/2020
(D) Review	Review of the design and code.	24/11/2019	31/12/2019
(E) Materials	Bill of materials	24/11/2019	31/12/2019
(F) Assembling	Robot assembling	1/1/20	31/1/20
(G) Testing	Robot testing	31/1/20	17/02/20
(H) Project presentation	Delivery of the project in the exam date	18/02/20	18/02/20

Table of Contents

1. Robot Design

1.1 Description of the problem

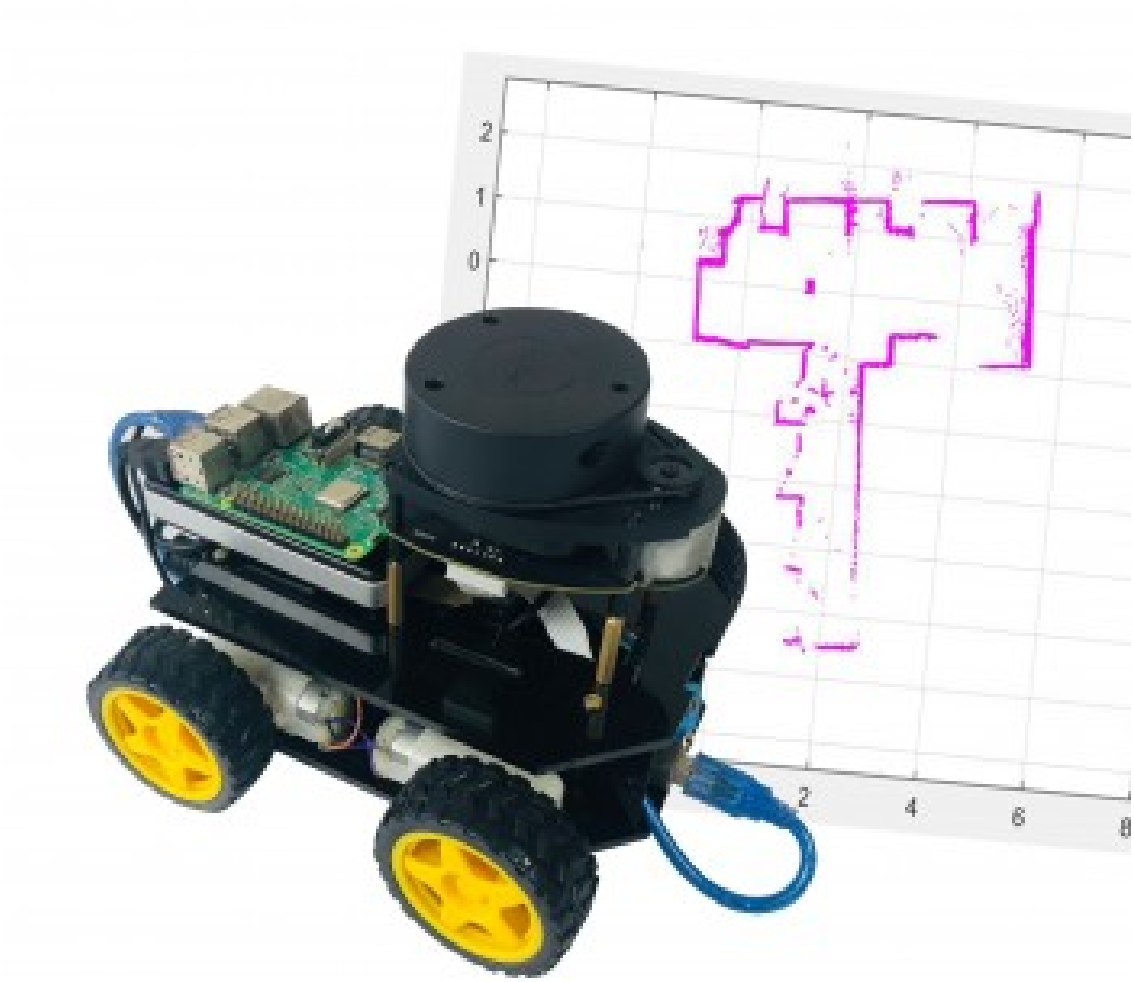
The purpose of the project is to develop a mobile robot able to move in an unknown environment. The robot is equipped with a series of sensors that allow it to autonomously explore its surroundings while avoiding obstacles and, at the same time, building a 2-dimensional map. Ideally we would like to design a robot that could be used in different scenarios: even though the roaming logic is similar to the ones used by cleaning robots, we could equip our robot with specific wheels and use it to explore area where humans, for example, can't access. Simplifying, we would like to create a robot that could be used in all those scenarios where there is the need to explore an area and the dispositions of different objects within it but a human, for any kind of reason, can't do or is not able to do such task.

1.2 Abstract model

Our robot should have the following components in order to operate:

- Controller:
  - Raspberry Pi: The robot is equipped with a Raspberry Pi board, which is a single-board computer with a linux-like OS mounted on.
- Sensors:
  - Laser scanner: This sensor allows our robot to perform Lidar scanning, thanks to the laser readings our robot can measure the reflected light and, with it, compute the distance from a specific object.
  - Odometry encoders: They measure the rotations of the robots' wheels.
- Actuators:
  - Wheels: By their rotations, our robot can move in the working environment.

Figure 1.1: Example of a robot model.



1.3 Robot tasks and behaviour

- Once turned on, start exploring the surroundings area.
- Avoid obstacles (i.e. wall, objects, etc.) by circumnavigating them.
- Create a live map of the known environments, backing up it every once in a while.
- Apply a logic-based and a meaningful navigation, avoiding to go back in the same area, instead prefer the exploration of new places.
- When a user decides to, go back to the starting point.

1.4 Robot requirements

1.4.1 Functional requirements

- The robot has to properly navigate in an unknown environment.

- The robot has to avoid obstacles in the environment.
- The robot has to generate a consistent map of the working environment.
- The robot has to store its starting position.
- The robot has to move autonomously in the environment.
- The robot has to go back home when called.

1.4.2 Non-functional requirements

- **Reliability**: The readings of our robot (hence the generated map) has to be consistent and reliable. Also, when called, our robot has to perform the path planning back home.

  Portability: Design a robot that is able to work in different domains where the exploration of the area is required.

2. Robot architecture

2.1 Used technologies

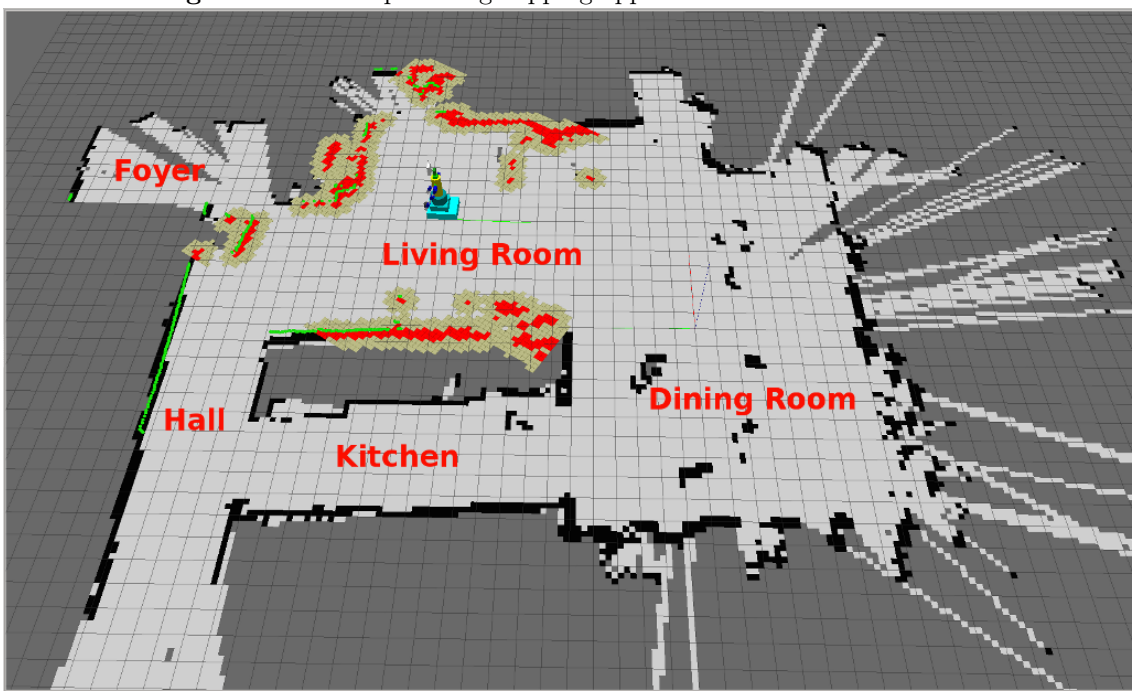
2.1.1 ROS

The main component on which our robot relies is ROS, which 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. The main tools that we have decided to use are listed in the following. In our project we make a deep usage of the Navigation Stack provided by ROS.

SLAM

SLAM, short for Simulatenous Localization and Mapping, is the computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location within it. The ROS framework provides different implementation of algorithm that solve such problem. In our case we've decided to use the http://wiki.ros.org/gmapping that is based on a LASER sensor mounted on the robot. GMapping employs a Particle Filter (PF), which is a technique for model-based estimation. In SLAM, we are estimating two things: the map and the robot's pose within this map. Each particle in the PF can be seen as a candidate solution to the problem. Together, the set of particles approximates the true probability distribution, i.e. the probability of the map and the robot's pose given the control inputs and sensor readings (e.g. the laser).

Figure 2.1: Example of a gmapping application with LASER sensor.



Odometry

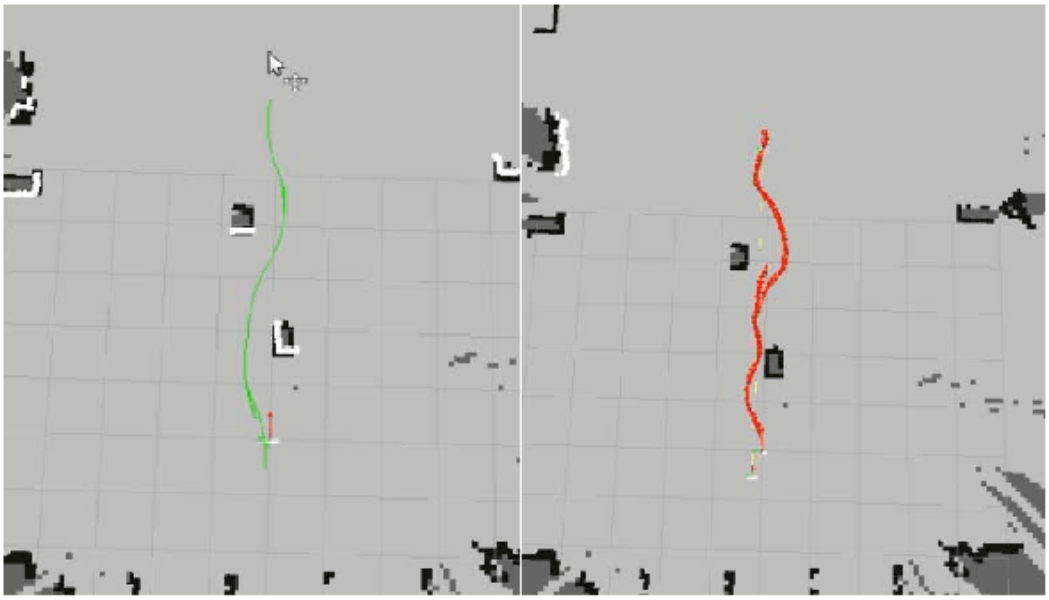
Odometry is the use of data from motion sensors to estimate change in position over time. It is used in robotics by some legged or wheeled robots to estimate their position relative to a starting location. This method is sensitive to errors due to the integration of velocity measurements over time to give position estimates. In the ROS context, this technique has been implemented in

different libraries, in our case we use Odom implementation provided by the Navigation Stack of ROS. The navigation stack uses tf to determine the robot's location in the world and relate sensor data to a static map. In our specific case we use this tool to turn the robot on itself. In order to work with Quaternion and angular velocity the tf2 library was introduced.

Path Planning

Path planning is a technique used in robotics to find a sequence of valid configurations that moves the robot from the source to destination. In ROS this topic has been deeply studied and there are some libraries that implements different path planning algorithm. In our project we've decided to use move base library which provides an implementation of an action (see the actionlib package) 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. In our case the map is run-time built. We use path planning to make the robot return to the starting position when the user has decided to. Move_base tries to match the laser scans to the map thus detecting if there is any drift occurring in the pose estimate based on the odometry (dead reckoning). This drift is then compensated by publishing a transform between the map frame and the odom frame such that at the end the transform map to base_frame corresponds to the real pose of the robot in the world.

Figure 2.2: Example of a move_base path planning application with LASER sensor.

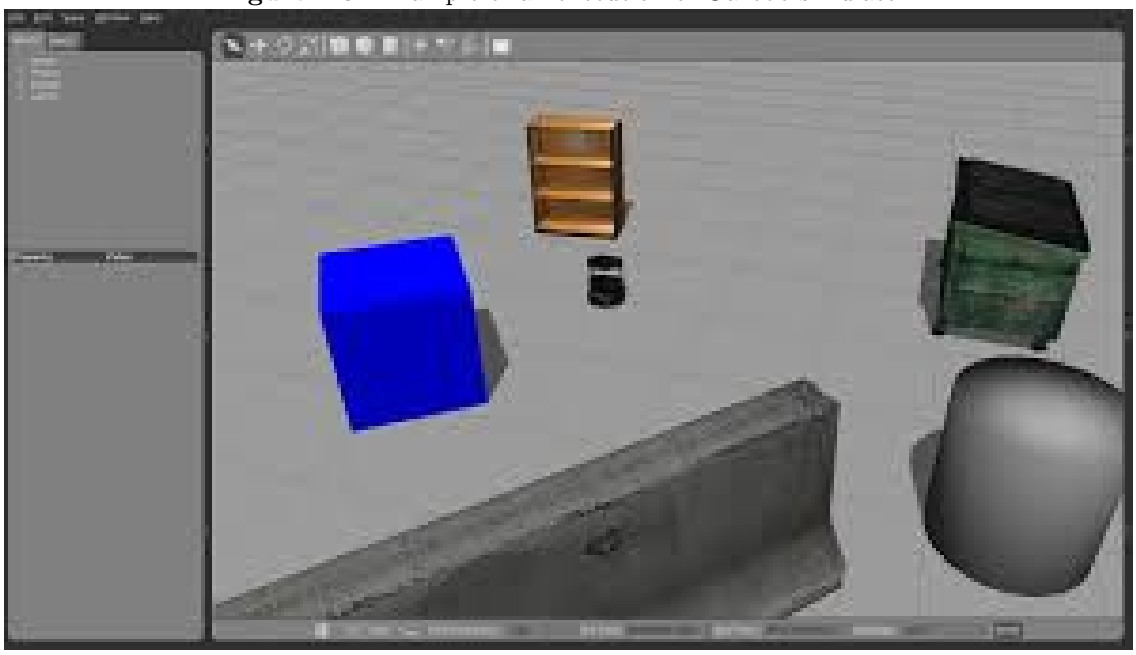


2.1.2 Gazebo

Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. It is a robust physics engine with high-quality graphics, convenient programmatic and graphical interfaces. Best of all, Gazebo is free with a vibrant community. We've introduced a simulator to ease the development of the project and to avoid the purchase of expensive sensors and robot models. We've decided to use Gazebo instead of others (like V-REP) for its ease of usage and its perfect integration with ROS. In fact we've downloaded the

ros-melonic-desktop-full package which is bundled with Gazebo. It comes with different prebuilt worlds and models, of course it allows for the building of custom world in which we can test and use our robot model.

Figure 2.3: Example of an execution of Gazebo simulator.



2.1.3 RViz

rviz is a ROS graphical interface that allows you to visualize a lot of information, using plugins for many kinds of available topics. It is different from Gazebo. Gazebo is the actual real world physics simulator with which you can set up a world and simulate your robot moving around. Rviz is the visualization software, that will allow you to view that gazebo data (if you are simulating) or real world data (if you are not using gazebo, but a real robot).

With rviz we can subscribe to specific ROS topic and we can visualize the Laser sensor reading, the path that was planned by the path planner module, and so on.

2.1.4 Two-wheeled car model

A car with two wheels model is used. This is a really basic model kit which is used in testing environment for differential driving car. It is equipped with two DC motors. We will mount on this model a Raspberry Pi model 2 version B+ in order to control the motors by using an L293D chip. In the following we'll refer to this car as Raspberry Car to underline the fact that the Raspberry Board is controlling this car model. More information on this part are provided in the following sections.

2.2 Robot conceptual architecture

The robot is built on a Sense-Plan-Act paradigm, in which:

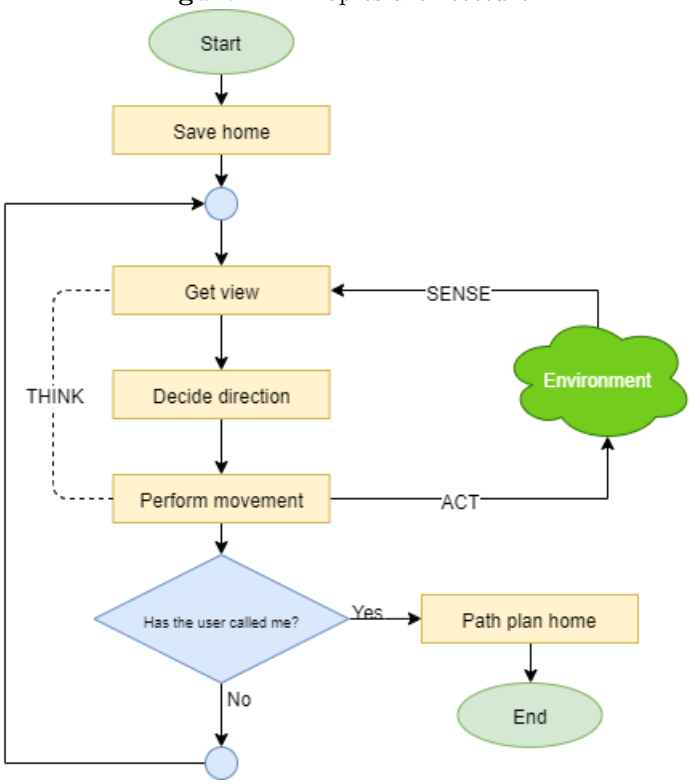
- **Sense**: We gather information about the operating environment by using the laser sensor readings. One scan of the surroundings give us information about the context and the distance between our robot and the obstacles. Each time a scan is performed the map is updated, so if the context is dynamic (i.e. moving obstacles) the map is updated and we immediately receive this information. According to the sensor readings we infer some sort of knowledge about the environment.

- **Plan**: A logic-based planner is used, specifically one based on Prolog logical programming language. The robot decides the next move by querying its knowledge base. The algorithm tries to avoid "map loop", i.e. it does not navigate always the same part of the map, insted it prefers to explore new unseen and unkown zone.

- **Act**: In our project the final act is to move the robot by using its wheels. After a move decision has been made by the "brain" of the robot, it is also performed by physically rotate the wheels.

Graphically we have:

Figure 2.4: Topics architecture.



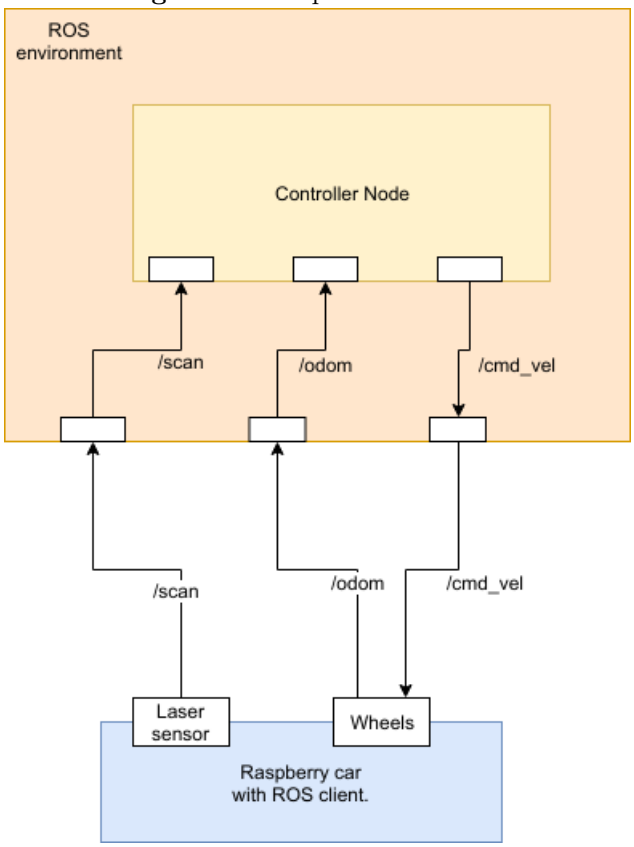
N.B. The "*Decide direction*" box will be further described in a next version of the current document, we are currenlty working on it.

2.3 Topics architecture

In this section we illustrate technically how the SENSE-THINK-ACT paradigm is implemented by using the ROS topics.

- The **SENSE** phase in which we gather information about the operating environment is performed by using the laser sensor equipped on the robot. As stated in the official documentation, the result of the sensor's readings are published by the Raspberry Car on the */scan* topic. The format of these messages is described in the official doc. In details, the ROS client (running on the Raspberry Car) performs a reading from the laser sensor and send a message by using the specific topic, to the ROS server. In the message, among different information, there is a *ranges* array composed of 360 elements, where the i-th item represents the distance in meter registered by the sensor in i-th degree. Recall that the sensor is a 360 degrees scanner, that's why the array has this size. Another important topic for the SENSE phase is the */odom*. As the name suggest, it's the place where the wheels encoders publish their information, like actual rotation. This topic is important if we want to estimate change in position over time. By combining the information provided by these two topic our controller node can obtain a clear **view** of the robot's surroundings in a specific instant of time.
- In the **THINK** phase, given a view, obtained by the precedent phase, the controller node has to decide the direction in which the Raspberry Car has to move. In order to fulfill the functional requirements, there is the need to perform some kind of logical inference. There needs to be some logic behind a direction, the robot can't and shouldn't be moved without any logic. That's where **Prolog** comes into play. This topic is further detailed in the next section.
- In the **ACT** phase, the Raspberry Car has to be moved in the direction obtained by the the previous phase. In order to do so, ROS is used. Specifically, the move_base package provide a topic, called *cmd_vel*. If we publish a message with a certain format in this topic, it will be received and parsed by the ROS instance running on the Raspberry Car, thus the robot will move in that direction. The message format is described in the official documentation.

Figure 2.5: Topics architecture.



2.4 Logic-based planner

To put it simply: Given a view, composed of four element (one for each cardinal direction i.e. North, South, etc.) where should the robot go? That's the question to which our logic-based planner has to answer.
Obviously, the goal is to span the entire space bounded by edges, most of the time represented by walls. Obstacles are the only elements that our robot has to be aware of, indeed the planner should solve the issue of circumnavigating them. In case of ties regarding the direction to take - there could be more than one cardinal point which is obstacle-free -, they would be broken randomly. When the user calls back the robot, our planner will have to compute a path of minimum length in order to go back home, meanwhile staying vigilant about obstacle avoidance.

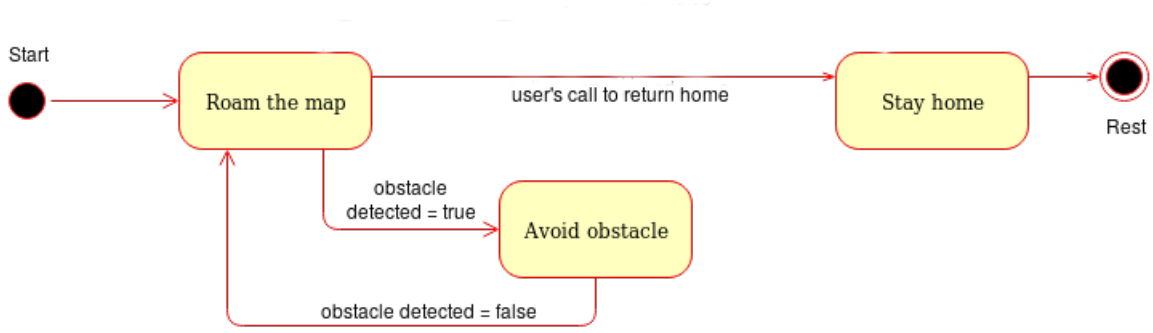
A succint state diagram represents the main tasks of the robot based on the planner

2.4.1 Design decision about Prolog

The team had to decide between the usage of SWI-Prolog with the prothonics library or Teleor-/Qulog with Qu-Prolog and Pedro. Before taking the decision the following considerations have been made:

- The robot is a purely reactive agent, in fact it has (or at least should have) the ability to perceive changes in the environment and response to it in a timely fashion. It is a direct stimulus-response agent, once it performs a scan of its surroundings, it take an action i.e. it decides a direction to follow. "Agent responses to changes in the environment in a stimulus-response based. The reactive architecture is realized through a set of sensors and

Figure 2.6: Basic state diagram.



effectors, where perceptual input is mapped to the effectors to changes in the environment."[1] Prothonics is a perfect match, in fact it's a "Python 3 library to create proactive agents, that perform reasoning by using SWI-Prolog." Mentioning SWI-Prolog, it's key features relies on an open-source philosophy and an extended and active community which enhance the feasibility of the choice.

- As stated in the official website, Pedro "is a subscription/notification communications system that also provides support for peer-to-peer communication". On the other hand we have that one of the core concept of ROS Computation Graph level are topics. We have extensively used this word in the previous of this document. They "are named buses over which nodes exchange messages". So ROS has built-in functionalities for messages handling and the capabilities to be a message broker. We can't replace ROS with Pedro because the former offers all the libraries illustrated in in the previous chapter for performing SLAM, path planning and odometry encoding. Thus, the only possible thing is to introduce Pedro while keeping ROS, but this would be an overhead for our system because we would have two different publish/subscribe tools for communication.

Therefore, the team has decided to use the prothonics library in this project, admitting that it has to be adapted and improved in order to fulfill both the functional and the non-functional requirements.

2.4.2 Implemented behaviours

The team has developed two different behaviours. These are Prolog files that we give as input to prothonics engine and that will be used to decide the next direction to take. One thing to notice is that there are two different type of coordinates. One is with respect to the map, and the other is with respect to the robot. When the robot starts rotating, the two doesn't always match. In order to **remap** one system with/versus the other, we use Prolog dynamic fact to store the current degree or rotation of the robot. it could be one of 0, 90, 180, 270 degrees. Of course when the robot rotates, we have to update this value by assertion and retract of facts. The two implemented behaviours are:

- **Least visited then empty:** When the execution starts, the robot create an empty array. This structure is used to maintain an history of all the directions taken, in this way we can count how many times each direction (such as *North*) has been taken. During the think phase, prothonics will perform the query **takeDecision(D)** and the result (i.e. variable D) is the direction that should be chosen. When such a query is performed, this behaviour will count the number of occurrences of each direction in the history, and will select the

[1] https://www.researchgate.net/publication/275643980_Agent_Architecture_An_Overview

least visited one. This direction is then **remapped** as explained in the initial note. If the final direction is empty it is sent as result of think phase, otherwise another predicate is called **emptyDirection(D)**. As the name suggest, the result of this call is the first direction which is currently empty. To summarize, with this behaviour the robot prefers to visit least explored directions but falls back to empty ones when they are busy.

- **Least visited then random:** When the execution starts, the robot create an empty array. This structure is used to maintain an history of all the directions taken, in this way we can count how many times each direction (such as *North*) has been taken. During the think phase, prothonics will perform the query **takeDecision(D)** and the result (i.e. variable D) is the direction that should be chosen. When such a query is performed, this behaviour will count the number of occurrences of each direction in the history, and will select the **least visited one**. This direction is then **remapped** as explained in the initial note. If the final direction is empty it is sent as result of think phase, otherwise another predicate is called **randomDirection(D)**. As the name suggest, the result of this call is a randomly chosen direction between the 4 possible ones. If this random is not empty the predicate will keep picking random one, until an empty one is found (at least one always exists). To summarize, with this behaviour the robot prefers to visit least explored directions but falls back to random ones when they are busy.

In order to use one or the other, we have created two different *launch files* for the node called **controller**.

3. Materials

3.1 List of materials

In order to build the robot car we need the following elements:

Component name	Shop	Price
Lidar 360 laser sensor	Link	$\approx 100e$
Raspberry kit	Link	$\approx 60e$
Powerbank 5V with charger	Link	$\approx 20e$
Robot structure with wheels	Link	$\approx 30e$
Optical Odometry Encoders	Link	$\approx 10e$
Total		$\approx 220e$

4. SysML

4.1 Sequence diagram

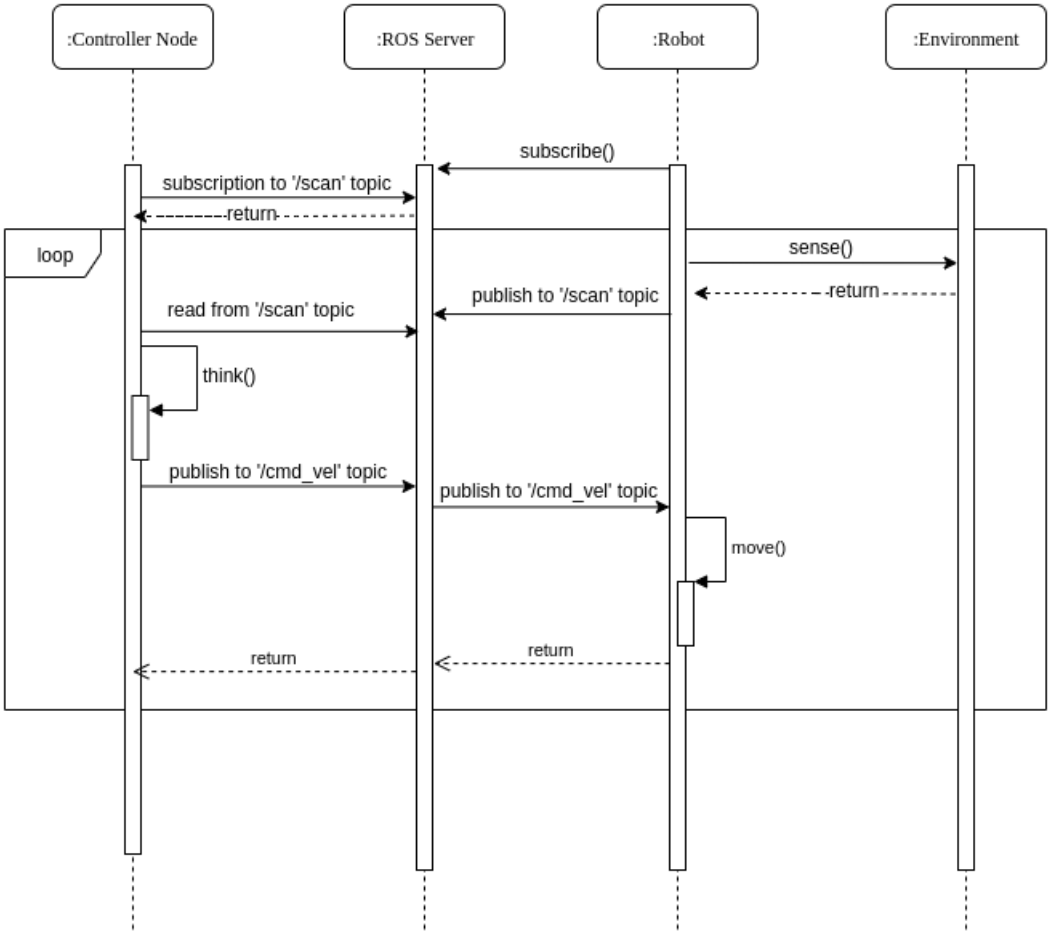


Figure 4.1: Robot sequence diagram.

As stated in the section 5.3 during the physical implementation of the robot, the team had the need to include two different plugin in order to interact with /cmd_vel and /odom topic.

5. Assembling

The team has available most of the elements stated in chapter 3, the only element that we needed to buy is the laser sensor.

5.1 Electronic circuit

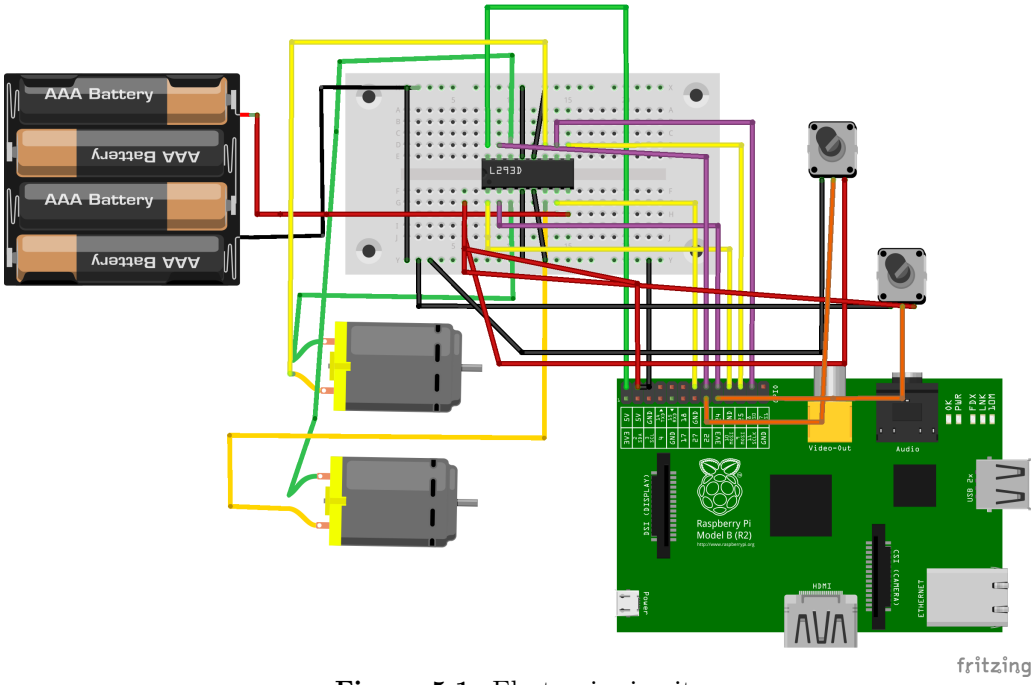


Figure 5.1: Electronic circuit.

In the above picture are represented the electronic circuit realized on the robot car model. From left to right there are:

- 1 6V battery pack used to power the motor.
- 2 brushless DC motors (one for each wheel).
- 1 L239D chip used to control the rotation and the speed for each one of the motor.
- 2 optical wheel encoders used to measure how far the wheels have rotated, and, given the fact that we know the circumference of its wheels, compute the distance travelled.
- 1 Raspberry Pi model 2B+.

5.2 Physical model structure

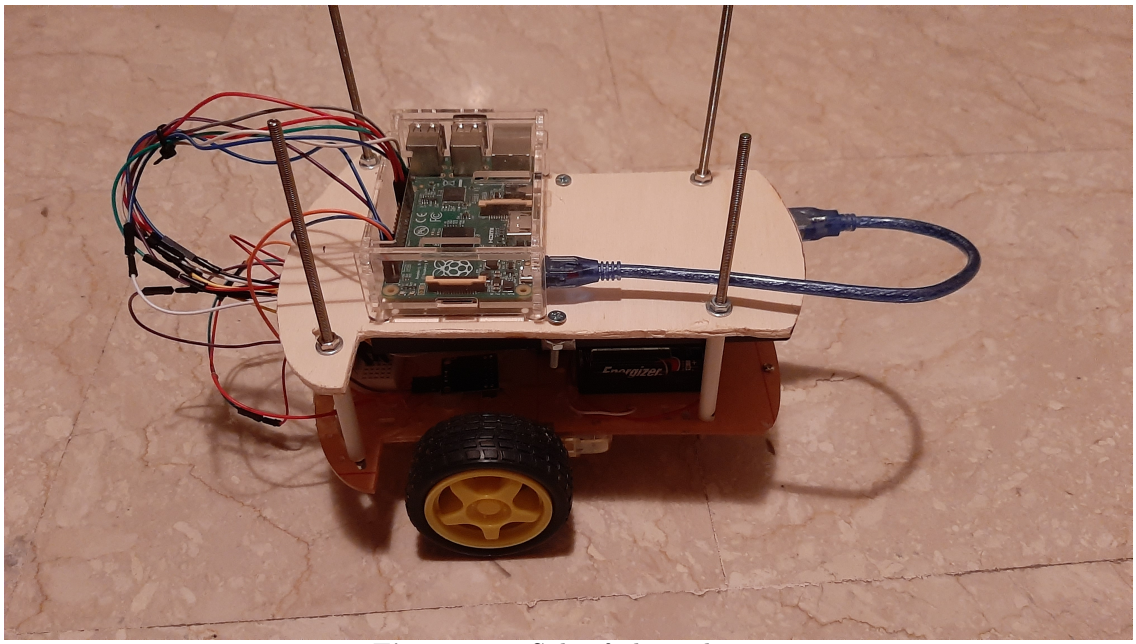


Figure 5.2: Side of physical car.

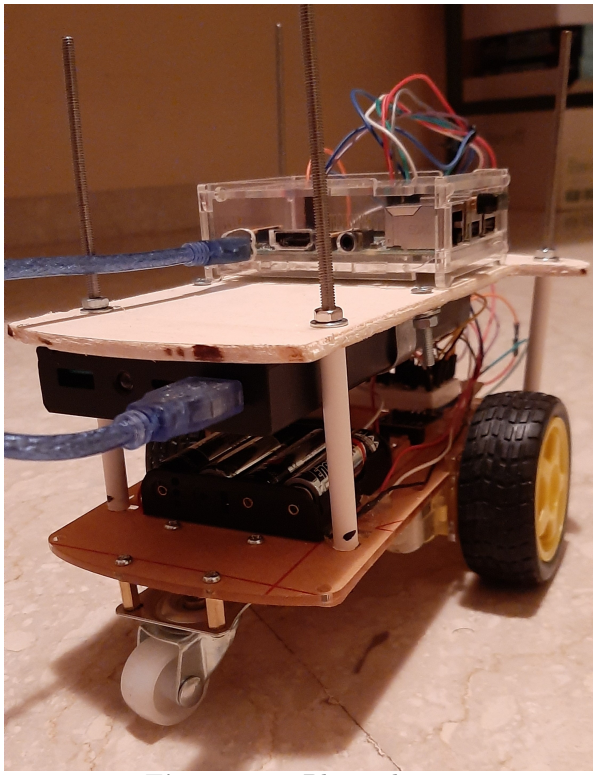


Figure 5.3: Physical car.

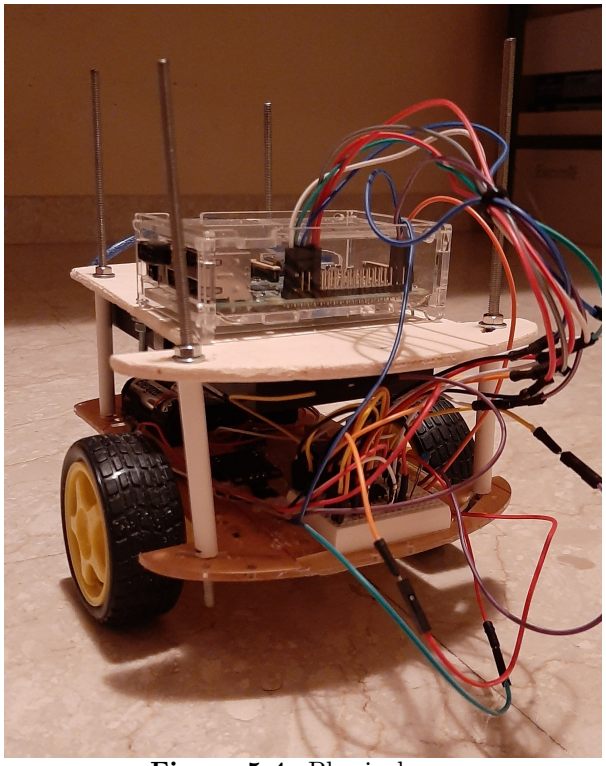


Figure 5.4: Physical car.

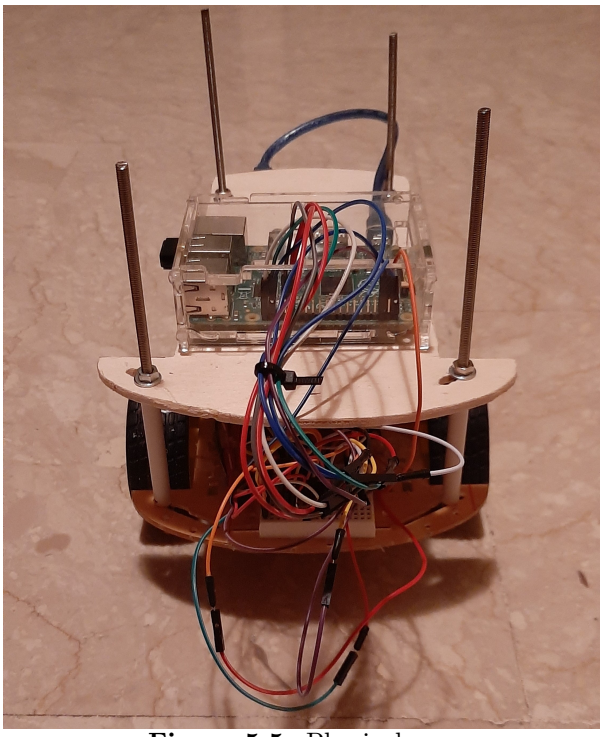


Figure 5.5: Physical car.

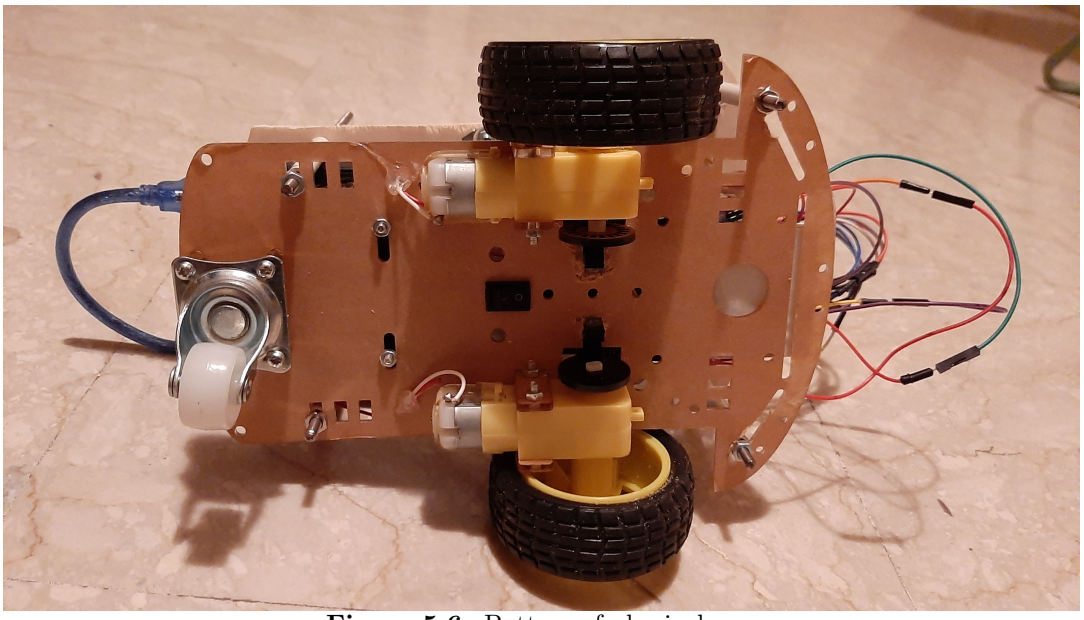


Figure 5.6: Bottom of physical car.

Figure 5.7: Physical car with lidar.

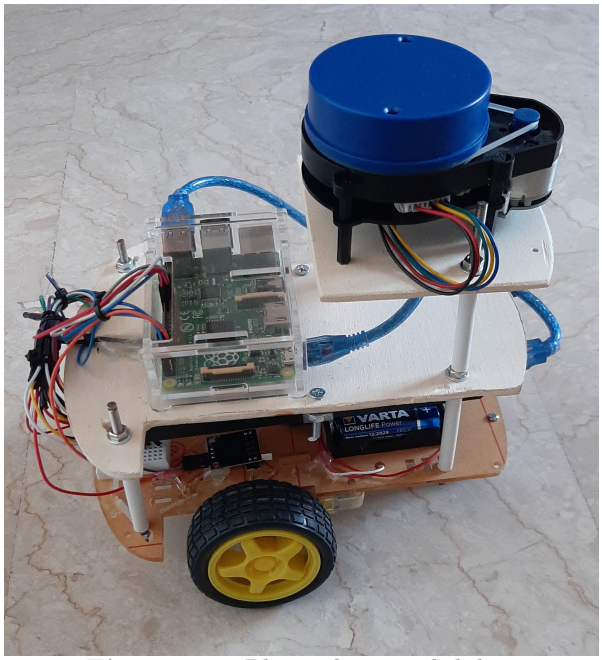


Figure 5.8: Physical car with lidar.

5.3 Necessary plugins

During the implementation of the robot we've realized that we need a plugin to interact with ROS topic. In particular, on the topic /cmd_vel ROS expects message of type Twist and on topic /odom ROS expects message of type Odometry. Our Raspberry Car doesn't have built-in neither in the motor driver nor in the wheel encoders this functionality, so we added a new plugin called differential_drive. It provides "some basic tools for interfacing a differential-drive robot with the ROS navigation stack". In order to do so, it uses some external topic detailed in the following. Specifically we used 2 scripts:

- diff_tf which converts the output from the wheel encoders into Odometry object.

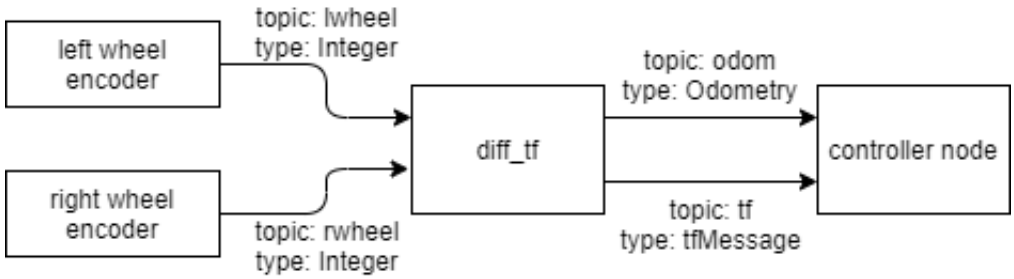


Figure 5.9: diff_tf plugin.

- twist_to_motors which translates a twist message into velocity target messages for the two motors.

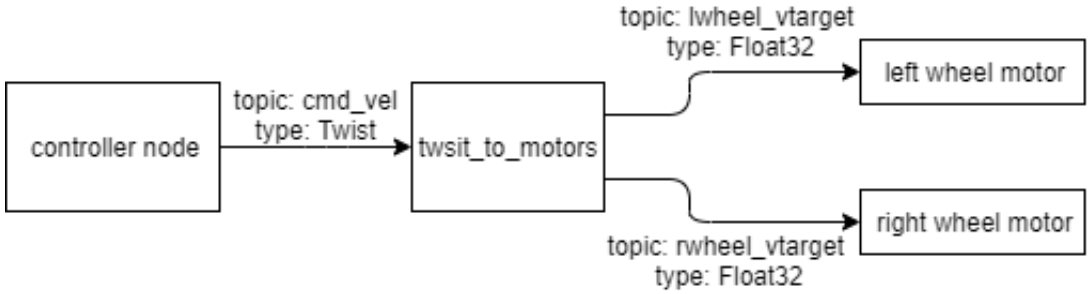


Figure 5.10: twist_to_motor plugin.

A. Project code

In this section we will explore all the developed code. Given the fact that we are working with ROS system we have organized all the robot functionalities in different **nodes**. In the following we'll delve into each one of them. All the code is available on Github in this repository: https://github.com/LuigiCerone/ISRL.

A.1 Node mio_robot

This node is a virtual representation of our robot schema and functionality. It is executed over the pc through the ROS software.

```
mio_robot
├── launch
│   ├── mio_robot_simulator_rviz.launch
│   └── mio_robot_world_gazebo.launch
├── maps
│   ├── closed_room_map.pgm
│   ├── closed_room_map.yaml
│   └── closed_room.world
├── param
│   ├── base_local_planner_params.yaml
│   ├── costmap_common_params.yaml
│   ├── dwa_local_planner_params.yaml
│   ├── global_costmap_params.yaml
│   ├── local_costmap_params.yaml
│   └── move_base_params.yaml
├── urdf
│   ├── mio_robot_gazebo_plugins.xacro
│   └── mio_robot.xacro
├── CMakeLists.txt
└── package.xml
```

Generally speaking (check the attached code for further information):

- **launch**: This folder contains all the launch files used to start this node. There are specific ad hoc xml-files that are used to start the execution of the code.
  - **mio_robot_world_gazebo.launch**: this launch file open the Gazebo simulator in which the defined map and the specified robot are spawned. The entire map is clearly shown.

  - **mio_robot_simulator_rviz.launch**: this launch file runs the rviz simulator in which the robot and the maps that it detects are visualized.

- **maps**: This folder contains the files that create a non-standard map (maps definied by ROS). It is used for further testing.
- **param**: This folder contains the configuration files in yaml extension used to set up the ROS package move_base, in which different params have to be set. Just to present some of them, we mention: *robot_base_frame* param that in our case we set equal to base_footprint topic (responsible of locating the robot on the map), the *base_local_planner* which contains the name of the planner used to plan the trajectory back home.
- **urdf**: This folder contains the URDF description of out simulated robot:
  - **mio_robot_gazebo_plugins.xacro**: this xacro file contains the plugin used in Gazebo, such as differential drive and sensor laser. It is importend in the mio_robot xacro file.
  - **mio_robot.xacro**: this xacro file contains the material description of our robot. You can find specified the sizes of the body, wheels and laser and their position.

A.2 Node controller

This node is executed on the laptop and is responsible for the planning part. It interacts with **prothonics library** and **Prolog** in order to compute the direction the robot should take, avoiding possible obstacles.

The prothonics part is used "as-is" and is taken from the official site. We added a new *behaviour.pl* Prolog file in which we are currently working with the aim of implementing a logic based planner that satisfies the functional requirements.

```
controller
├─ launch
│  └─ controller.launch
├─ src
│  ├─ prothonics library
│  ├─ behaviour.pl
│  └─ robot.py
├─ CMakeLists.txt
└─ package.xml
```

The main file is *robot.py* in which we define a ROS subscriber to the */scan* topic. By using the LaserScan inner structure we compute the distances between our robot and the obstacles. This allows the system to compute the current view of its surroundings that are given to the prothonics library. The result of the logic program execution is a direction that is then published on the *cmd_vel* topic to which the plugin twist_to_motor is subscribed.

A.3 Node raspberry_car

This node is exectued on the phisycal car by the Raspberry equipped on it. During the developing we have used SSH protocol in order to sync and execute files.

This is the structure of the node files:

```
raspberry_car
├── launch
│   ├── raspberry_car.launch
│   ├── raspberry_car_encoder.launch
│   └── raspberry_car_twist.launch
├── src
│   ├── diff_tf.py
│   ├── twist_to_motors.py
│   ├── encoders.py
│   ├── motors.py
│   └── raspberry_car.py
├── CMakeLists.txt
└── package.xml
```

Without going too much in detail (for this we refer to the code) we have that:

- **launch**: This folder contains all the launch files used to start this node. There are specific ad hoc xml-files that are used to start the execution of the code.
- **src**:
  - diff_tf and twist_to_motors are the plugin introduced in 5.3.
  - encoders.py is Python script that interact with the Raspberry GPIO pins where the optical encoders are connected. These encoders have an IR sensor that act as a switch, when there is a state change (0/1) means that the wheel is turning and that a tick in the encoder has been read. The script keeps counting on the number of ticks processed and publish (via ROS API) these values (one for each wheel) on two specific topics (those required by the diff_tf plugin to work).
  - motors.py is a Python script that interact with the Raspberry GPIO pins and, by using the L293D chip, to the motors. This script is responsible of turning/stopping the wheels of the physical car. This is possible due to the twist_to_motors plugin that publish on two topic to which the motors.py script subscribe. When there is a new message on the topic, this script implement all the logic for physical movement.

Figure A.1: One wheel with its motor, the encoder and one LM393 IR-based optical encoder.

A.4 Node simulator_teleop

This node is for testing purpose only, it is executed on the laptop. As the name suggests, it is used to teleop, i.e. for controlling the robot by using the keyboard of the laptop (keys W, S, A, D, X). We use it to check that the physical car is properly working. It publishes information on the /cmd_vel topic to which the node *raspberry_car* is subscribed and the car should move in that direction.

Briefly:

- launch contains the .launch file that we use to execute this node on the command line interface of the laptop.
- node contains the logic for the node written in Python language. The code is a porting of the official TurtleBot3 teleop node with some adjustments specific to our use cases.

```
simulator_teleop
├── launch
│   └── simulator_teleop.launch
├── node
│   └── simulator_teleop
├── CMakeLists.txt
└── package.xml
```

B. Diagrams

B.1 Nodes and Topics on physical car

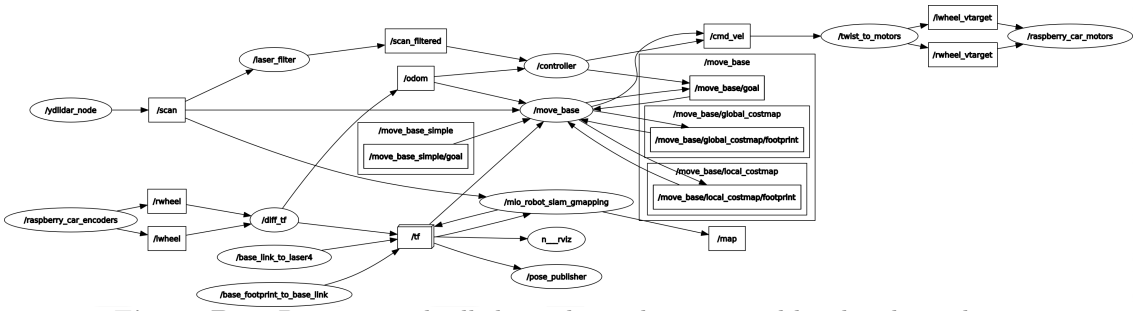


Figure B.1: Diagram with all the nodes and topics used by the physical car.

B.2 TF tree on physical car

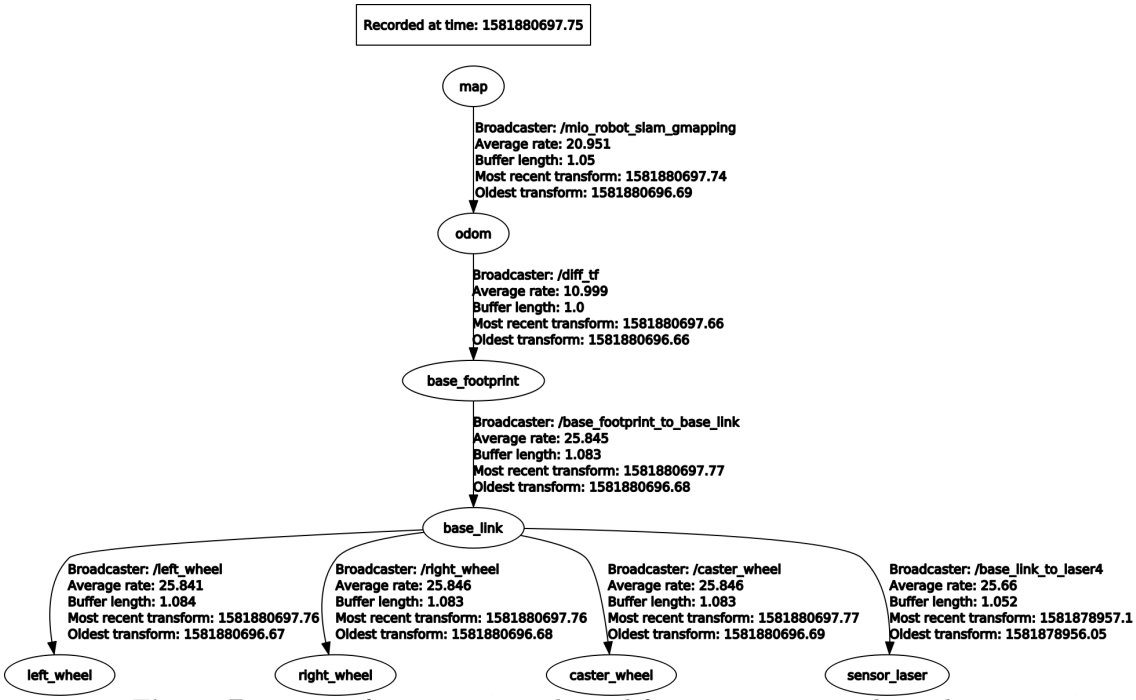


Figure B.2: Transformations tree derived from execution on physical car.

C. Formulas physical robot

C.1 Angular velocity equation

In order to derive the needed angular velocity to let the robot rotate, the following equation has been applied.

$$\frac{V_{wheel} * r_{wheel} * t}{C} = \frac{\phi}{2\pi}$$

where V_{wheel} stands for the wheel velocity specified in radians per second; r_{wheel} indicates the wheel's radius; t represents the time need to complete the rotation of ϕ radians; lastly, C measures the wheel's circumference.